Temporal Monitors for TinyOS

Doina Bucur

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

Abstract. Networked embedded systems generally have extremely low visibility of system faults. In this paper, we report on experimenting with online, node-local temporal monitors for networked embedded nodes running the TinyOS operating system and programmed in the nesC language. We instrument the original node software to signal asynchronous atomic events to a local nesC component running a runtime verification algorithm; this checks LTL properties automatically translated into deterministic state-machine monitors and encoded in nesC. We focus on quantifying the added (i) memory and (ii) computational overhead of this embedded checker and identify practical upper bounds with runtime checking on mainstream embedded platforms.

Keywords: Runtime verification, embedded software, LTL, automata, TinyOS, nesC

1 Introduction

Embedded systems have become ubiquitous outside static, controlled industrial settings. They include both mobile embedded communication systems such as smartphones, and either static or mobile sensors and actuators operating in highly dynamic environments, such as *networks of wireless sensors* monitoring a natural or urban setting. Three features are common to such systems: (i) extreme *reactiveness*, in that system operation is driven by asynchronous external events—the inherent difficulty of writing correct software for asynchronous operation effectively increasing the probability of system failure; (ii) a tight bound on computational and memory *resources* of the hardware platform, necessary to ensure energy efficiency in operation; (iii) a need for *autonomous* operation. In this, we look at wireless sensor systems (WSNs), "the volatility of [which] is always in tension with ambitious application goals, including long-term deployments of several years, large scale networks of thousands of nodes, and highly reliable data delivery" [15]. We take the application case of the mainstream cross-platform operating system for wireless sensor nodes, TinyOS [14], its programming language, the event-based network embedded systems C (nesC) [12, 11], and Telos [19]—an ultra-low power, highly ROM- and RAM-constrained wireless sensor module developed at the University of California, Berkeley.

Many failures of a WSN reportedly are rooted in faults at a single node. These faults are then rarely recovered from, due to the impracticability of reaching a

faulty node at its deployment location, and to the scarcity of mechanisms for self-diagnosis and repair of WSN nodes, e.g., generic built-in error checkers. In general, node-local failure modes include buffer and stack overflows, deadlocked, livelocked software and data races [16], no next-hop destination for data routing in a multihop network, unexpected outliers or gradient in sensed data, degradation of the battery [15], incorrect temporal use of the OS kernel's API [1], and any number of programmer-written invariants and qualitative or quantitative temporal properties.

We contribute an automata-theoretic runtime checker for node-local properties, running natively on embedded nodes running TinyOS. We support future-time LTL properties over system events. Relatively few efforts were made with regard to formal verification against failures in embedded software for WSNs. *Static checking* methods targeted at nesC or C for TinyOS, both for a single networked node [2] and a network [22, 17, 21, 18, 25] have met with mixed success in what regards the degree of *coverage* feasibly achieved, particularly with regard to the external *context* the node reacts to: the main difficulty of statically verifying a WSN is in modelling and checking a given node's software exhaustively against all possible networking and environmental settings.

On the other hand, *runtime checking* has become moderately accepted; naturally, checking at runtime precludes the need to exhaustively model the software's context. This is the case of SafeTinyOS [3], a node-local program-analysis-based checker for memory safety, now part of TinyOS; this has demonstrated that TinyOS kernel memory safety at runtime incurs a 13% ROM (code) and 5.2% CPU overhead. While other runtime-monitoring tools were also contributed, few fulfill, like our method, the crucial autonomy requirement that the runtime checking reside on the embedded nodes themselves. None of these checkers allow properties other than invariants (in the case of SafeTinyOS) and small handcrafted state machines as interface contracts for nesC software components in [1].

Standard temporal property languages such as LTL are suitable to express specifications for nesC software. For this, we construct the set of boolean atomic propositions to include (i) boolean conditions over program variables, e.g., `(data > 0x10)`, and (ii) program checkpoints, e.g., the entry point of the nesC function `Timer.fired()`. Using these additions to encode system events, memory safety properties may be written as LTL invariants, and interface contracts as LTL *Precedence* patterns. We allow all LTL properties which can be violated in finite time.

To generate runtime monitors for such temporal properties, we use a formal automata-theoretic algorithm to translate LTL into deterministic Büchi automata over finite words. The question whether a property currently holds thus translates into the question whether the current finite trace of execution events leads the monitor on an accepting transition; each verification step is done on-the-fly. We give a native nesC template implementation of a runtime monitor, together with an automatic translation from a deterministic automaton over finite words into this template. We then evaluate the overhead introduced

by the monitoring and verification in terms of code (stored on-board in ROM memory), volatile memory (RAM) and CPU load. We find that absolute overhead per monitor is negligible in what regards RAM and CPU load, but is up to 2.67KB of ROM for a basic LTL pattern, which amounts to 5.55% of the ROM integrated on a Telos revision B platform.

In what follows, Section 2 briefly overviews background matters in what regards both the theory of runtime checking and the TinyOS system; Sections 3 and 4 describe and evaluate, respectively, our monitoring framework, and Section 5 covers the related work and concludes.

2 Background

2.1 Theoretical background

In the future fragment of propositional linear-time temporal logic (LTL), formulas are composed out of atomic propositions from a finite set AP, boolean logic operators, and the temporal operators **X** ("next time") and **U** ("until"); other temporal operators are defined in terms of these, e.g., **G** (the invariant "globally") and **F** ("eventually"). Whether a word over AP satisfies a LTL formula ϕ is defined inductively over the formula structure.

Given the set AP over which a temporal property is written, $\Sigma := 2^{AP}$ is a finite language over AP. A (nondeterministic) transition-based generalized Büchi automaton (TGBA) is a tuple $A := (\Sigma, Q, T, q_0, F)$, where Σ is a finite language as above, Q is a finite set of states, $q_0 \in Q$ is an initial state, F is a finite set of acceptance conditions, and T is the transition relation $T \subseteq Q \times \Sigma \times F \times 2^Q$, i.e., each transition is labelled and has attached acceptance conditions. General verification of LTL properties against a given system model is traditionally automata-theoretic: the LTL property is translated into a Büchi automaton A, such that a word σ over AP correctly described by ϕ allows a sequence of transitions such that each letter in σ matches a transition label, and the sequences satisfy each acceptance condition in A (infinitely many times, in the case of an infinite σ); i.e., a TGBA can be constructed for a given LTL property ϕ such that it accepts exactly the temporal words described in ϕ. Building a monitor for LTL only requires the particular case of *finite* words σ. The model-checking library SPOT [7, 6][1] implements a number of LTL-to-TGBA translations, of which we used Couvreur/FM based on [4] and further formula and automaton simplifications; the resulting automaton compares well in terms of size with other translation algorithms, including that of LTL2BA [10].

Any nondeterministic automaton over finite words may be translated into an equivalent deterministic one; this *deterministic monitor* then has constant computational complexity, instead of linear in the size of the automaton. To generate a deterministic monitor from a nondeterministic Büchi automaton, [5] prunes all states which cannot start an accepting run, restricts the transition

[1] An online engine is at `http://spot.lip6.fr/ltl2tgba.html` (last access: August 2012).

relation to the new set of states, and modifies all states to be accepting. SPOT implements a similar pruning method, with resulting acceptance conditions on transitions.

2.2 Practical background

A long line of low-power, integrated hardware platforms have been developed as WSN nodes in the past 15 years. The Telos [19] general design integrates computation, communication, storage, and sensing: an 8MHz, 16-bit MSP430-model microcontroller runs with 10KB of RAM, 48KB of ROM code storage, and, when active and with the radio on, draws 19mA of current from the battery pack. The equally mainstream alternative platform from the Mica family, MicaZ, runs on an 8-bit RISC-based ATmega128L microcontroller with 4KB of RAM. In our evaluation, we focus our experimentation on the popular Telos revision B platform (i.e., TelosB), with occasional comparisons to MicaZ.

Software for these platforms is constructed on the low-duty-cycle principle: the processor and radio transceiver are asleep for most of a duty interval, and periodically awaken for sensing, computation and communication duties. TinyOS [14] is a relatively young open-source operating system for WSN nodes, intended to allow such low-power duty cycling. TinyOS itself is written in a novel language, network embedded systems C (nesC), which extends C with code *components* (which may be either *configurations* or *modules*) wired through *interfaces*. Hardware Presentation Layer (HPL) components form the lowest level and interact directly with the hardware; higher-level system logic (e.g., device drivers or networking protocols) consists simply of one or more newly programmed components wired together with the relevant lower-level system components. A programmer's TinyOS application (i.e., the highest level of software abstraction) is programmed in the same way.

This component-based system design comes with advantages when implementing a monitor: (i) variables global to the entire OS are few, and (ii) a given, e.g., system peripheral or data structure pertaining to a network protocol is (each) de facto controlled from a single nesC system component. These facts simplify the task of code instrumentation for logging system events.

As usual for low-level networked systems, the associated programming languages and compilers support *asynchrony* natively: nesC interfaces are a bundle of asynchronous *events* and/or synchronous *commands*. For example, a hardware interrupt itself is an event to a low-level nesC component; this component's event handler may *signal* further events to higher-level components, triggering in effect an asynchronous event chain. All event handlers are non-interruptible and must be kept brief by the programmer, while any deferred computation in the form of synchronous TinyOS *tasks* should instead be *post*ed to the system's task queue.

We also capitalize on these OS features related to timeliness and asynchrony when designing our runtime logging and trace checking to be *real-time*: all system events of interest are notified through asynchronous nesC events, and every checking step is an `atomic` deferred task.

Fig. 1 depicts a schematic architecture of TinyOS, with our added runtime checker.

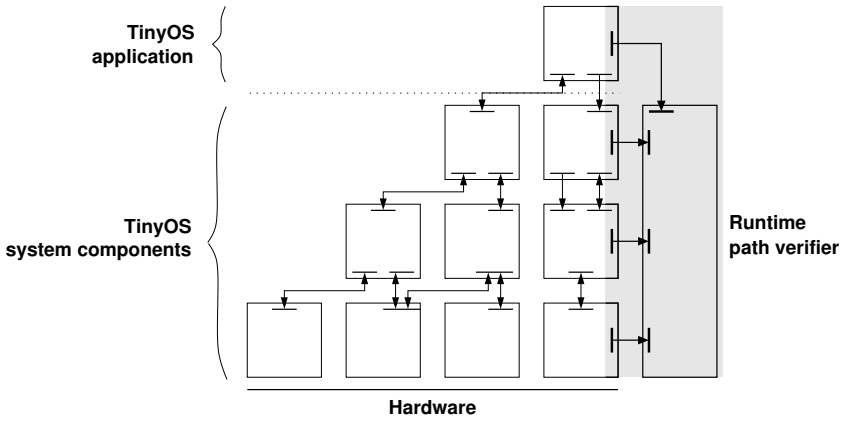

Fig. 1. An abstract schematic of TinyOS showing the system being implemented as nesC components wired through interfaces. A TinyOS application is effectively a set of nesC components, wired together in a graph-like structure. Our components and interfaces implementing runtime verification are emphasized (on the right). The runtime checker wires to any existing nesC system components of interest, and is thus notified of relevant system events.

3 TinyOS system events, state, and monitor synthesis

System events and representing atomic propositions

As introduced in Section 1, system events may include, syntactically, boolean conditions over nesC variables and program checkpoints. Logging these system events is a relatively simple task, due to TinyOS's component-based design and the ensuing separation of concerns (described in Section 2.2). Furthermore, each nesC component of interest (such as a device driver or a protocol implementation) generally has a well-defined set of "important" variables and program checkpoints, such as the routing table (in the case of a network protocol) or the entry point of the event signalling the acquisition of new sensor data (in the case of a sensor's device driver). Thus, we found that exhaustively instrumenting a component once is sufficient for checking a large set of realistic temporal properties over the entire OS.

As testbed, we consider a representative sample TinyOS application from the existing codebase[2], Oscilloscope. This application is effectively a wiring together

[2] The source repository for TinyOS is at `http://code.google.com/p/tinyos-main/` (last access: August 2012).

of a set of most-used TinyOS system components, including the drivers for the on-board sensors and the basic wireless networking stack. The top-level application logic then simply adds a duty cycle of 250ms, for which interval a timer is programmed to signal a periodic alarm event. In each of these cycles, a sensor is sampled with a call/signal command/event pair; when the number of successful readings has filled a small buffer, the buffer is transmitted wirelessly to a fixed address.

We instrument for logging some of these crucial TinyOS system components, i.e., part of the implementation of the Hardware Presentation Layer (which effectively means that the state of any of the microcontroller's I/O peripherals is logged), and the high-level application logic itself. In what regards HPL, we consider as relevant system events the change in state of each bit in the microcontroller's peripheral registers; given the memory model of this platform, the peripherals on a TelosB platform form a set of 56 event types. For the high-level logic, we mark 10 conditions over variables and checkpoints. AP then equals this union set of system events.

Representing system state, and matching transitions

Only a subset of these atomic propositions in AP need monitoring for a given (set of) LTL properties ϕ. It is thus somewhat memory-inefficient to statically index each of the 66 event types in AP by a non-negative integer, and then statically encode the system state as a bit vector of 66, where each bit i is assigned the truth value of the corresponding $p_i \in AP$. We improve on this by instead dynamically generating the encoding for the system state per (set of) LTL properties, together with the monitor generation: for each atomic proposition p in ϕ, the lowest available index greater than zero is assigned. The trivial propositions *true* and *false* are treated in the same way. The resulting system state is thus a minimal bit vector of the size of ϕ (instead of the size of AP). Matching a transition in the automaton is then simply checking the required bits in the state bit vector.

Notifying system events

Fig. 1 shows the components and interfaces added to the original TinyOS codebase for runtime checking. The *checker* itself is implemented as a new nesC component, automatically generated from a given LTL formula ϕ. This component `provides` a nesC event for any other logged component to `signal`; we list the header of this nesC checker in Fig. 2.

The *logging* of system events is then implemented as follows: each nesC component instrumented with logging simply *wires* to the checker (introduced above) through a nesC `notify` event, which is then `signal`ed by the instrumented component at particular program checkpoints or variable writes.

We list the wiring for an HPL component in Fig. 3. We note that we did this additional nesC wiring and instrumentation manually. While a suitable tool implementing program analysis would automatize this process, we found that the instrumentation overhead is acceptably low: we only needed three new

```
configuration PaxLTLC {
   provides async event void notify(uint16_t ap, bool val);
}
module PaxLTLP {
   // implements runtime checking for a given LTL property
}
```

Fig. 2. The header of our nesC runtime checker.

`signal` calls to log any change in the state of 56 microcontroller pins in the `HplMsp430GeneralIOP` modules in Fig. 3, due to the fact that all I/O ports are generated from a single "generic" module.

Monitor synthesis

Take the LTL formula $\mathbf{F}\, r \rightarrow (\, !p\, \mathbf{U}\, (\, s \,|\, r\,))$ (the *Precedence* pattern with a *Before* scope from the KSU LTL pattern repository [8], meaning that atomic proposition s being true precedes p being true, and all before r is true). SPOT generates the equivalent deterministic monitor, as in Fig. 4.

As monitor encoding, we adopt a nesC version of the C++ `front_det_ifelse` encoding in [23], proven fairly efficient over experimentation for System C. The encoding (of the Precedence property above, in Fig. 5) keeps track of the execution of the automaton with the integer variables `current` and `next`. At most one `if` branch on each of the two levels of conditionals is taken, for each notification event. The verification ends, and the property is proven violated, when at the arrival of an event notification, no transition is enabled. We generate these monitor implementations *automatically* from LTL properties, based on a nesC template.

4 Evaluation

Properties

Our test suite includes the basic KSU collection of future-time LTL property *patterns* and *scopes* [8]. The six property patterns are those of *Universality* (p is true), *Absence* (p is false), *Existence* (p eventually becomes true) and the related *Bounded Existence* (p becomes true at most twice), *Precedence* (s precedes p) and *Response* (after p, s eventually follows). To form more complex properties, any pattern is composed with any of five scopes: *Globally*, *Before* r, *After* q, *Between* q *and* r and *After* q *before* r. Thus, for the Oscilloscope application, any invariant over the values of sensed data is written as a Universality pattern in a Global scope (which we abbreviate by U-G); a specification requiring at least one successful sensing operation before a packet is sent may be written as an Existence-Before, abbreviated E-B.

Five of the ensuing thirty combined property types have trivial monitors, as they cannot be violated in finite time; these are the Existence-Globally (E-G),

```
configuration HplMsp430GeneralIOC { [..]
}
implementation {
   [..]
   components PaxLTLC;
   [..]

   PaxLTLC.notify <- P10.pax_notify;
   PaxLTLC.notify <- P11.pax_notify;
   PaxLTLC.notify <- P12.pax_notify;
   // where P10, P11, etc are instantiations of HplMsp430GeneralIOP
   [..]
}

generic module HplMsp430GeneralIOP([..]) {
   [..]
   uses async event void notify(uint16_t ap, bool val);
}
implementation
{
   [..]
   async command void IO.set() {
      [..]
      signal pax_notify((PORTx*10+pin), TRUE);}
   async command void IO.clr() {
      [..]
      signal pax_notify((PORTx*10+pin), FALSE);}
   [..]
}
```

Fig. 3. Wiring and instrumentation added to the HPL components which control the microcontroller pins on the TelosB; the logging is done by signalling the `notify` event.

Existence-After (E-A), etc, and we omit them from the evaluation results. To these, we add two composite properties which are practically useful:

$$\bigvee_{i=1}^{k} \mathbf{G} p_i \quad \text{and a generic event-sequence chain} \quad p_1 \mathbf{U} (p_2 \mathbf{U} (\dots \mathbf{U} p_k))$$

and also multiple basic monitors checking the same application.

Into these property types, we *randomly* input combinations of atomic propositions from our list of relevant system events. The resulting specifications are either violated or satisfied by the system software; our monitors will report whether the checking has finished (thus a violation was encountered) in real time, at the end of each monitoring step—variable `finished_checking` in the monitor implementation from Fig. 5 records the verification status, and is reported to the system users.

Metrics, experimental setup, and results

As our runtime checker shares all on-board resources with the original application, we evaluate the monitor's performance in terms of computational (CPU

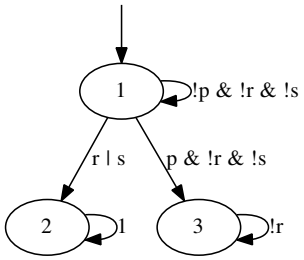

Fig. 4. Deterministic monitor generated for the specification $\mathbf{F}\, r \rightarrow (\, !p\, \mathbf{U}\, (\, s \,|\, r\,))$. State 1 is the initial state, and each transition is accepting.

```
implementation {
   async event void notify(uint16_t ap,
                           bool val) {
      // store (ap, val)
      if (!finished_checking)
         post step();
   }

   task void step() {
      atomic {
      // calculate new state with (ap, val)
      current_checking_steps++;
      current = next; next = -1;

      if (current == 1) {
         if ((call stateBV.get(r)) ||
             (call stateBV.get(s)))
            next = 2;
         else if ((call stateBV.get(p)) &&
                  !(call stateBV.get(r)) &&
                  !(call stateBV.get(s)))
            next = 3;
         else if (!(call stateBV.get(p)) &&
                   !(call stateBV.get(r)) &&
                   !(call stateBV.get(s)))
            next = 1;
      }
      else if (current == 2) {
         next = 2;
      }
      else if (current == 3) {
         if (!(call stateBV.get(r)))
            next = 3;
      }
      finished_checking = (next == -1);
}}}
```

Fig. 5. NesC monitor encoding for the monitor in Fig. 4. This forms most of the implementation for the `PaxLTLC` component introduced in Fig. 2.

overhead) and memory (RAM and ROM overhead) when running one or more checkers in the `PaxLTLC` component.

We measure the difference in the size of the binary application code between the monitored and the original, uninstrumented and unchecked, version used as a baseline (i.e., the ROM overhead). In Fig. 6, we evaluate the ROM overhead for example formulas of the basic LTL patterns composed with scopes, minus the cases with trivial monitors; patterns and scopes are shown as pairs X-Y of their abbreviations. In general, the difference we observed between the code size of a monitor for a property pattern, and that of the same monitor over different atomic propositions, is due only to the difference in the code instrumentation needed to notify of the occurrence of corresponding system events; the checker

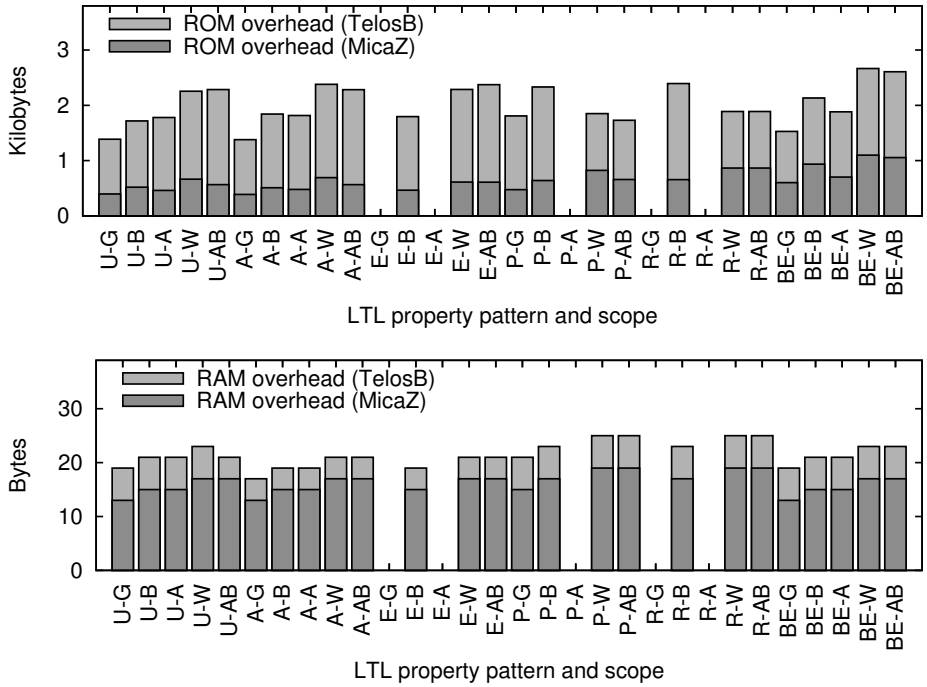

Fig. 6. Maximum code (ROM) and RAM overhead (in the latter case, we include data, uninitialized data, and maximum stack use) encountered by varying the subset of AP in each formula. Results as compiled by the platform compilers of the TelosB and MicaZ platforms out of the C software generated from nesC by the TinyOS compiler. The *original* application takes 19KB (ROM, TelosB), 13.7KB (ROM, MicaZ), 483B (RAM, TelosB), and 927B (RAM, MicaZ).

component is otherwise identical. In Fig. 6, we plot only the maximum values we encountered per property type.

A similar method is used to evaluate the RAM overhead, with the exception of the fact that only the `data` and `bss` (i.e., uninitialized data) binary segments are immediately readable from the compiled binary. TinyOS implements no dynamic memory allocation, which means that we only need to calculate the *maximum stack use* (an intrinsic part of the RAM metric). For this, we used actual execution runs of the monitored application in an experimental setup for TelosB; for MicaZ, we used `tos-ramsize`, a platform-specific static analysis tool for soundly assessing this metric, integrated in TinyOS. Fig. 6 also gives the final evaluation of the RAM overhead.

It is to note that for both ROM and RAM overheads, the microcontroller features (e.g., 16- versus 8-bit RISC) are crucial to the outcome; also, the RAM overhead is low in general, while ROM overhead averages around 2KB per LTL property. To add some perspective, the maximum automaton size among these

properties is 133 (where we take the size of an automaton to be the number of states times the number of transitions).

As for monitoring composite properties on a TelosB, the upper limit on a feasible k in $\bigvee_{i=1}^{k} \mathbf{G}p_i$ was $k = 5$; for $k = 6$, the code size (now including the original 19KB of code in ROM) exceeded the 48KB available on-board the platform; this automaton has 63 states and 665 transitions. For the event-sequence chain $p_1\mathbf{U}(p_2\mathbf{U}(\ldots\mathbf{U}p_k))$, we reached up to and including $k = 10$. For multiple basic monitors, the overhead is expectedly upper-bounded by the sum of the overheads in the single-monitor case.

To assess computational overhead, we run our monitored TinyOS application in a cycle-accurate emulator, MSPSim [9], tailored to the MSP430 microcontroller on the TelosB platform. Such an emulator executes all CPU operations with correct timing up to individual clock ticks, as these operations would also be executed on the given microcontroller architecture; an emulator is generally used for testing hardware or hardware-and-software designs. Cycle-accurate emulator executions of the same sensor software in the same context will always be consistent.

When running the application over the MSP430 emulator, we sample the CPU load and stack use every 20ms. We show the results of CPU overhead (i.e. the *difference* between the load of monitored runs and that of the original unmonitored application run) for a single monitor in Fig. 7. In order to capture the worst-case overhead, for this calculation we considered only those combinations of system events in the LTL formula, and only those intervals of the corresponding emulated executions with a still *active monitor*, i.e., before the monitor detected that the property at hand had been violated, and thus finished its checking procedure.

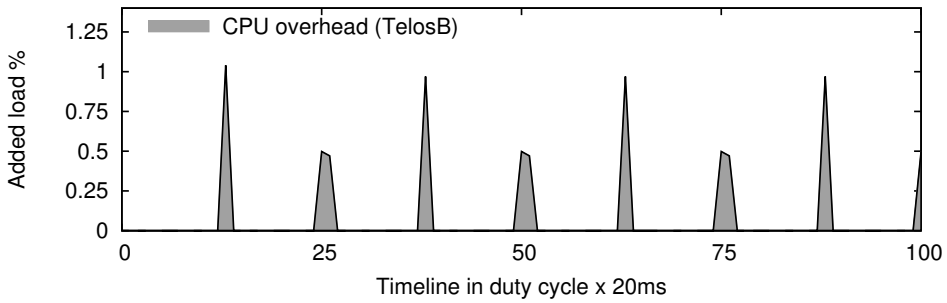

Fig. 7. CPU overhead (the average of 30 emulation runs) with a single running monitor.

To note is that the CPU overhead follows the system's duty cycle, as expected, and that it only rises up to about 1% load increase for the 40ms after a new system event. This metric can be easily translated into actual mW of power consumed by the computation overhead, through integration. Also, since

at each checking `step()` the monitor will also report whether a violation was just encountered, this gives that a *user notification* can be triggered after these 40ms of processing the new event.

Finally, we gather all emulation runs and assess overhead by automaton size; we show results in Fig. 8.

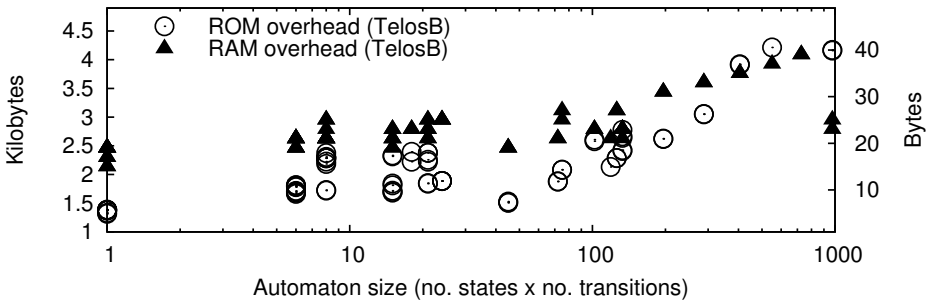

Fig. 8. Collected overhead results by automaton size, for a single monitor. ROM scale on left, RAM scale on right.

5 Related work and conclusions

Other than SafeTinyOS [3] and the Interface Contracts for TinyOS [1], NodeMD [16] also contributes a checker for deadlock, livelock (using checkpoints and expiration times), stack overflow, and assertions, based purely on program analysis and code instrumentation. None of these tools applies formal methods for runtime verification, and only support temporal properties expressed as small automata by the programmer, or quantitative temporal properties with heavy code instrumentation to check timing conditions. However, we share with these tools the online, embedded manner of running a checker. The same application area is covered in [20], for probabilistic properties and with the added feature of networking (and thus the added positive of supporting global network properties) and the negative in that this checker, while based on formal methods, runs externally to the WSN on a desktop, with 300k lines of code code added to the simulator. Temporal checking for C using aspect-based code instrumentation and state-machine monitors is covered in [13]. We found valuable insights in [23], an experimental study into the optimization of state-machine monitors for SystemC.

A few closing remarks are in order. Given the resulted feasibility of monitoring TinyOS execution traces on an embedded platform, as shown in this prototype, we may sustain the argument that, for such practical applications with (1) possibility for offline construction of the monitor, and (2) tightly bound online computational resources, alternative solutions could replace our regener-

ation of the TGBA per each new property with maintained databases of minimal automaton translations for standard temporal properties, such as the Büchi store [24]. Also, further study is needed in what regards alternative solutions such as nondeterministic automata as checkers, and into quantitative temporal properties.

References

1. Archer, W., Levis, P., Regehr, J.: Interface contracts for TinyOS. In: Proceedings of the International Conference on Information Processing in Sensor Networks (IPSN). pp. 158–165. ACM (2007)
2. Bucur, D., Kwiatkowska, M.: On software verification for sensor nodes. Journal of Systems and Software 84(10), 1693 – 1707 (2011)
3. 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). pp. 205–218. ACM (2007)
4. Couvreur, J.M.: On-the-fly verification of linear temporal logic. In: Wing, J., Woodcock, J., Davies, J. (eds.) Formal Methods, Lecture Notes in Computer Science, vol. 1708, pp. 711–711. Springer Berlin / Heidelberg (1999)
5. d'Amorim, M., Rosu, G.: Efficient monitoring of omega-languages. In: Etessami, K., Rajamani, S. (eds.) Computer Aided Verification, Lecture Notes in Computer Science, vol. 3576, pp. 311–318. Springer Berlin / Heidelberg (2005)
6. Duret-Lutz, A.: LTL translation improvements in SPOT. In: Proceedings of the Fifth International Conference on Verification and Evaluation of Computer and Communication Systems. pp. 72–83. VECoS, British Computer Society (2011)
7. Duret-Lutz, A., Poitrenaud, D.: SPOT: An extensible model checking library using transition-based generalized Büchi automata. In: Proceedings of the IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems. pp. 76–83. MASCOTS, IEEE Computer Society, Washington, DC, USA (2004)
8. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st International Conference on Software Engineering. pp. 411–420. ICSE, ACM, New York, NY, USA (1999)
9. Eriksson, J., Dunkels, A., Finne, N., Österlind, F., Voigt, T.: MSPsim – an Extensible Simulator for MSP430-equipped Sensor Boards. In: European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session. Delft, The Netherlands (2007)
10. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: Berry, G., Comon, H., Finkel, A. (eds.) Proceedings of the 13th International Conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 2102, pp. 53–65. Springer, Paris, France (Jul 2001)
11. 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). pp. 40–49. ACM (2005)
12. 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). pp. 1–11. ACM (2003)

13. Havelund, K.: Runtime verification of C programs. In: Proceedings of the 20th IFIP TC 6/WG 6.1 International Conference on Testing of Software and Communicating Systems: 8th International Workshop. pp. 7–22. TestCom/FATES, Springer-Verlag, Berlin, Heidelberg (2008)
14. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for networked sensors. SIGPLAN Not. 35(11), 93–104 (2000)
15. Jurdak, R., Wang, X.R., Obst, O., Valencia, P.: Wireless Sensor Network Anomalies: Diagnosis and Detection Strategies, Intelligent Systems Reference Library, vol. 10, chap. 12, pp. 309–325. Springer, Berlin, Heidelberg (2011)
16. 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). pp. 43–56. ACM (2007)
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). pp. 174–185. ACM (2010)
18. 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)
19. Polastre, J., Szewczyk, R., Culler, D.: Telos: Enabling Ultra-Low Power Wireless Research. Fourth International Symposium on Information Processing in Sensor Networks (IPSN) pp. 364–369 (April 2005)
20. Sammapun, U., Lee, I., Sokolsky, O., Regehr, J.: Statistical runtime checking of probabilistic properties. In: Proceedings of the 7th International Conference on Runtime Verification. pp. 164–175. RV, Springer-Verlag, Berlin, Heidelberg (2007)
21. 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). pp. 186–196 (2010)
22. Sharma, O., Lewis, J., Miller, A., Dearle, A., Balasubramaniam, D., Morrison, R., Sventek, J.: Towards verifying correctness of wireless sensor network applications using Insense and SPIN. In: Proceedings of the 16th International SPIN Workshop on Model Checking Software. pp. 223–240. Springer-Verlag, Berlin, Heidelberg (2009)
23. Tabakov, D., Vardi, M.Y.: Optimized temporal monitors for SystemC. In: Proceedings of the First International Conference on Runtime Verification. pp. 436–451. RV, Springer-Verlag, Berlin, Heidelberg (2010)
24. Tsay, Y.K., Tsai, M.H., Chang, J.S., Chang, Y.W.: Büchi store: An open repository of Büchi automata. In: Abdulla, P., Leino, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 6605, pp. 262–266. Springer Berlin / Heidelberg (2011)
25. Zheng, M., Sun, J., Liu, Y., Dong, J.S., Gu, Y.: Towards a model checker for nesC and wireless sensor networks. In: Proceedings of the 13th International Conference on Formal Methods and Software Engineering. pp. 372–387. ICFEM, Springer-Verlag, Berlin, Heidelberg (2011)