Anquiro: Enabling Efficient Static Verification of Sensor Network Software

Luca Mottola, Thiemo Voigt,
Fredrik Österlind, Joakim Eriksson
Swedish Institute of Computer Science
{luca, thiemo, fros, joakime}@sics.se

Luciano Baresi, Carlo Ghezzi
Politecnico di Milano, Italy
{baresi, ghezzi}@elet.polimi.it

ABSTRACT

We present ANQUIRO, a domain-specific model checker for statically verifying the correctness of sensor network software. In this context, static verification has hitherto received little attention, as state space explosion problems may prevent applying these techniques. ANQUIRO overcomes this limitation by providing different abstraction levels depending on the functionality to verify, and by implementing domain-specific state abstractions within the checking engine. We demonstrate the use of ANQUIRO in verifying the correctness of a widely used data dissemination protocol. This study allows us to identify issues that the protocol may overlook. Moreover, our evaluation of ANQUIRO's performance shows that it drastically reduces the number of states generated during the verification, preventing state space explosion problems.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*

General Terms

Design, Verification

Keywords

Wireless sensor networks, Static verification

1. INTRODUCTION

Many deployments of Wireless Sensor Networks (WSNs) have failed due to software and hardware issues [10]. Failures may happen even after careful testing in simulation and testbeds, since the real-world conditions encountered at deployment time are in general different from those in the lab. These trigger untested execution paths, revealing previously unknown issues. As a result, assessing the correctness of WSN software is a major challenge.

The current practice, described in Section 2, includes using simulators/emulators to investigate the system behavior prior to deployment, or relying on WSN-specific debugging tools to deal with issues arising in the field. Both types of solutions only identify issues *when* and *if* they arise. In addition, the former approaches expose only a partial view on the possible system executions, dictated by arbitrary simulation parameters such as random seeds. The latter approaches, instead, may identify issues without however showing the corresponding *causes*. Some of these tools must also defend against so-called "heisenbugs" [27], where the debugging infrastructure affects the system behavior to the point of causing issues that would not be present otherwise.

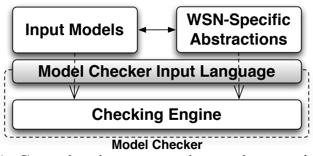
Static verification appears to be an ideal complement to the above approaches, providing *complete* and *sound* verification of WSN software against user-specified properties. Completeness entails that a property violation is always found if present, whereas soundness guarantees that no false property violations are detected. The properties may describe desired behaviors or invariants over the system state, e.g., a requirement that a tree-based routing protocol never creates loops. Upon detecting a property violation, the tools return *counter-examples* showing the complete system execution leading to the property violation. This represents an asset during development, as it makes easier to identify the causes of the issue.

Existing approaches for static verification of WSN software, however, focus on very specific problems, e.g., concurrency issues in multi-threading libraries [3]. Generalizing these solutions is difficult because of the diversity in the functionality to verify. These range from application-level processing to hardware-specific functionality, and thus operate at different abstraction levels. In most cases, details that are relevant to an abstraction level are immaterial for others. Even within a single abstraction level, simple fragments of code may generate large state spaces. Many of these states are generated only because of the generality of algorithms used in general-purpose tools [9], which do not exploit domain-specific knowledge to save on generated states. However, these states may not be necessary to examine for the property at hand.

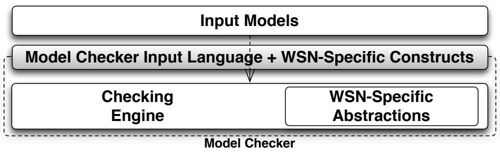
Leveraging this observation, we build upon our prior work [2] on domain-specific verification of Publish/Subscribe architectures, and present ANQUIRO, a WSN-specific model checker based on the Bogor model checker [19]. The tool works in a two-step fashion. The source code is first translated into ANQUIRO-specific models. Next, it is verified against user-provided properties. ANQUIRO overcomes state space explosion problems during verification by:

- providing different abstraction levels depending on the functionality to verify, as described in Section 3. For instance, if users want to verify the implementation of application-level functionality, the tool operates by abstracting away the low-level communication aspects that are irrelevant from the application perspective. These may include, for instance, different causes of packet losses, as the application is only interested in whether packets are lost, independently of the cause.
- augmenting Bogor's modeling language with domain-specific constructs depending on the abstraction level, and implement-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE '10, May 2-8 2010, Cape Town, South Africa
Copyright 2010 ACM 978-1-60558-969-5/10/05 ...$10.00.



(a) Standard approach to the verification of WSN functionality.


Input Models
Model Checker Input Language + WSN-Specific Constructs
Checking Engine
WSN-Specific Abstractions
Model Checker

(b) ANQUIRO approach to the verification of WSN functionality: domain-specific abstractions are embedded within the tool.

Figure 1: The ANQUIRO model checker embeds domain-specific modeling abstractions within the checking engine.

ing their semantics *inside* the tool, as described in Section 4. This technique, intuitively described in Figure 1, gives ANQUIRO full control on how the state space evolves during verification, thus enabling domain-specific state abstractions. These allow us to transparently merge different system states that are irrelevant to a given abstraction level, drastically improving the speed and scalability of the verification.

To demonstrate the use of ANQUIRO, Section 5 reports on the verification of a widely used WSN dissemination protocol [12]. As property to check, we study the core guarantee the protocol is to provide: the ability to eventually deliver data to all nodes. ANQUIRO shows us that the protocol may overlook an issue preventing correct operation. We complete the discussion with an evaluation of ANQUIRO's run-time performance. Our results show that our domain-specific state abstractions provide orders of magnitude improvements over standard approaches. This allows ANQUIRO to verify even large instances of our problem in reasonable time. Section 6 concludes the paper with a discussion of our plans for further development of ANQUIRO and brief concluding remarks.

2. STATE OF THE ART

In the following, we survey current approaches for checking the correctness of WSN software.

2.1 Simulation/Emulation

Simulators [11, 17] and emulators [23] are routinely used for performance evaluation. The behavior of WSN software is driven by random events—such as radio communication or clock drifts—which simulators mimic using pseudo-random models. For instance, the radio model determines whether a message is lost during a particular execution. Such pseudo-random behaviors are determined by simulation parameters, e.g., the choice of the initial random seed. This determines the random behavior of the entire simulated execution.

The current state of the art in WSN simulation/emulation includes well-established tools with mature user support. Using these tools for verifying WSN software, however, exposes only a partial subset of the possible system executions. To improve coverage, multiple random seeds are used to trigger executions, albeit it is not possible to ensure full coverage. Therefore, no absolute guarantees are provided on the correctness of the WSN software at hand.

2.2 Debugging

Solutions for in-field debugging of WSN software exist in the current state of the art. Some of them [13,26] are based on passively

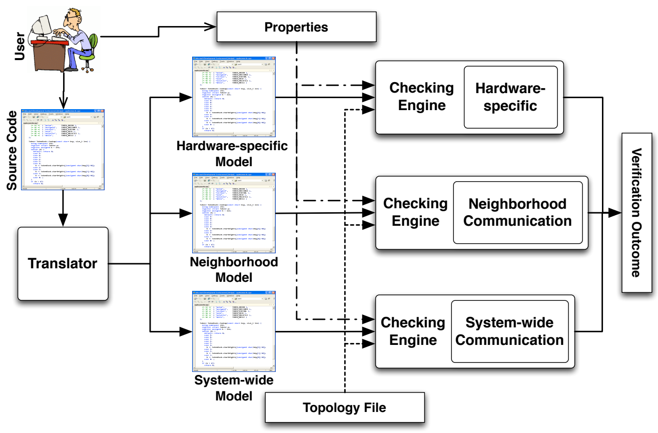


Figure 2: Verification flow using ANQUIRO.

monitoring the distributed execution of WSN software, e.g., using sniffers co-located with the WSN. These solutions, however, may not have access to all relevant information. Therefore, even if an issue is found, developers may not have sufficient visibility into the system operation to understand the corresponding causes.

Alternative solutions [4, 14, 18, 20, 21, 25, 27] gather run-time information by instrumenting the application code to check properties of interest, or provide on-line interfaces to inspect the state of WSN nodes. As already mentioned, these tools are useful to investigate issues that already emerged at run-time, but might not anticipate problems prior to deployment. In addition, the possibility of heisenbugs limits the applicability of these solutions to given application domains, e.g., scenarios that are not time-sensitive.

2.3 Static Verification

Static verification of embedded software has widely been used in safety-critical scenarios [5]. The applicability of such solutions in the WSN domain, however, is limited to few examples focused on the verification of specific functionality [3, 8, 16, 24]. The limiting factor for a generalization of such approaches, as we pointed out, lies in the diversity of WSN functionality and the associated state space explosion problems. Using standard tools for software verification, these issues may ultimately result in the impossibility to carry out the verification effort.

We overcome this limitations with a domain-specific verification tool, described next.

3. THE ANQUIRO MODEL CHECKER

The current design of ANQUIRO includes three abstraction levels to use depending on the functionality to verify. Figure 2 shows the typical verification flow using ANQUIRO. The user feeds the source code to check as input to a dedicated translator, and selects the abstraction level to consider. Based on this information, the translator automatically outputs ANQUIRO-specific models to be used for the verification. Once this completes, ANQUIRO returns a confirmation message if all properties are satisfied. Otherwise, it shows the first counter-example found where at least one property is violated. This includes the complete sequence of actions to reach the state where the property is violated.

The specifics of the available abstraction levels and an example property are shown next, along with the network model we adopt.

3.1 Abstraction Levels

The abstraction levels we consider cover a large fraction of the WSN functionality that users may need to verify. They are:

Hardware-specific: the models describe operations specific to a hardware architecture. These may include read/write operations in memory and interactions with external devices, e.g., radio chips. This abstraction level is therefore amenable for verifying low-level functionality, e.g., device drivers.

Neighborhood Communication: we model local processing in atomic steps independently of the hardware platform, and communication only between devices within direct radio range. This eases the description network-level functionality such as routing protocols.

System-wide Communication: we model communication between any devices in the network. This abstracts away the functionality of the underlying network. Therefore, this abstraction level is amenable to verify application-level processing independently of network-level functionality.

3.2 Properties

The properties to verify may describe a desired behavior (lifeness) or invariants to check against all possible system executions (safety). In ANQUIRO, we specify properties using Linear Temporal Logic (LTL), leveraging an existing Bogor plug-in that enables LTL verification.

The individual formulae may refer to variable names in the original source code, as these symbols are preserved during the translation process, and are quantified over network nodes. Variables in different source files are distinguished based on the file name. For instance, a property to check that in a tree-based routing protocol no nodes ever select themselves as parent is specified as:

$$\forall i \in Nodes \quad \Box(collection.parent_i \neq i) \tag{1}$$

where *collection* is the source file where variable *parent* is found, and i is any node in the network.

3.3 Network Model

The user must also provide a topology file that describes the connectivity between nodes. The file lists all node pairs the tool should consider as possibly able to exchange data. Notably, the use of such information in ANQUIRO is different from that in simulators. Wireless communication is inherently unreliable. To achieve complete verification, ANQUIRO must check all possible system executions corresponding to every packet being delivered or lost. Therefore, every time the input models describe a send operation over some network link, ANQUIRO splits the system execution in two branches: one corresponding to the packet being delivered, the other representing the case where the packet is lost.

ANQUIRO must do the above for *every* transmitted message, or completeness of the verification may compromised. Nevertheless, there may be system executions where simultaneous link failures render a node completely disconnected from the network. This would represent a case where most (if not all) properties of interest would trivially fail. The tool may thus exclude system executions where the network is partitioned. This procedure occurs inside our dedicated checking engine, described next.

4. ANQUIRO INTERNALS

We design and implement ANQUIRO as an extension to the Bogor model checker [19]. This allows us to augment the modeling language with domain-specific constructs, while providing the internal hooks necessary to the implementation of the corresponding semantics. To do so, we define the additional constructs in a Bogor preamble that points to the implementation of their semantics. This implementation has full access to the state space as it evolves during the verification. Thus, it can drive the generation and exploration of the system states according to the semantics of the domain-specific constructs.

```
 1 extension NeighborhoodComm
 2             for sics.anquiro.NeighborhoodComm {
 3   typedef type<'a>;
 4   // Opening a channel
 5   expdef NeighborhoodComm.type<'a> openChannel(int);
 6   // Communication API
 7   actiondef sendUnicast
 8           (NeighborhoodComm.type<'a>, 'a, int);
 9   actiondef sendBroadcast
10           (NeighborhoodComm.type<'a>, 'a);
11   expdef boolean waitingMessage
12           (NeighborhoodComm.type<'a>);
13   actiondef getNextMessage
14           (NeighborhoodComm.type<'a>, lazy 'a); }
```

Figure 3: WSN-specific language constructs to model neighborhood communication.

```
1 extension Timer for sics.anquiro.Timers {
2   typedef type;
3   // Creating a new timer
4   expdef Timer.type createTimer();
5   // Timer operations
6   actiondef startOneShotTimer(Timer.type, int);
7   actiondef startPeriodicTimer(Timer.type, int);
8   actiondef stopTimer(Timer.type);
9   expdef boolean timerFired(Timer.type); }
```

Figure 4: WSN-specific language constructs to model timers.

We describe next the internals of ANQUIRO. Formally verifying the correctness of both the translation process and our domain-specific abstractions is in our immediate research agenda. To this end, we plan to leverage our previous experience [1] in similar scenarios. Both the translator and checking engine are written in Java for easier integration with Bogor.

4.1 Translator

The current implementation of the ANQUIRO translator provides partial support for WSN software written in the C language for the Contiki OS [7]. The translation process occurs using simplified versions of known techniques to convert C programs into finite state machines [5]. Depending on the chosen abstraction level, some calls to the underlying operating system or device drivers are replaced with domain-specific constructs. We describe next an example when the user selects neighborhood communication as abstraction level. In Section 4.2, we illustrate the implementation of the semantics of the domain-specific constructs in this example.

Modeling constructs. Figure 3 shows the Bogor preamble modeling neighborhood communication. The preamble describes an abstract data type and its associated operations. Mirroring the network support in Contiki, communication is channel based. Every connection to a different channel is represented as a different instance of the abstract data type. This is parametric in the message type transmitted through the channel.

Line 1-2 bind the constructs listed in the preamble to the underlying implementation inside the checking engine, included in a Java class named `NeighborhoodComm`. We use the expression in line 5 to open a new channel, obtaining a new instance of the abstract data type that represents the open connection. Line 7-10 describe operations to send unicast or broadcast messages. Line 11-12 define a guard that yields true when a message is received. The message is retrieved using the operation defined in line 13-14, passing a reference to an empty message filled with received data when the operation returns[1].

The translator may also need to describe time-triggered behaviors, which are common in WSN implementations. To this end, we leverage a timer abstraction, whose operations are defined in

[1]Bogor's `lazy` modifier acts as a pass-by-reference.

```
static struct abc_conn abc;
PROCESS_THREAD(example_abc_process, ev, data) {
  static struct etimer et;
  PROCESS_BEGIN();
  abc_open(&abc, 128, NULL);
  while(1) {
    /* Delay 2 seconds */
    etimer_set(CLOCK_SECOND * 2);
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
    /* Load packet and send */
    packetbuf_copyfrom("Hello", 6);
    abc_send(&abc);  }
  PROCESS_END(); }
```

Figure 5: Contiki C code sending periodic broadcast messages.

```
// Message definition
record MyMessage { string msg; }
// Sending of periodic broadcast messages
active thread exampleAbcProcess() {
  NeighborhoodComm.type<MyMessage> abc_ch_128;
  Timer.type et;
  loc loc0:  // Channel open and timer setup
    do { abc_ch_128 := NeighborhoodComm.
                             openChannel<MyMessage>(128);
         et := createTimer();
         Timer.startPeriodicTimer(et, 2 * SECOND);
    } goto loc1;
  loc loc1:  // Timer fires: sending message
    when Timer.timerFired(et) do {
      MyMessage msg1 := new MyMessage;
      msg1.msg := ``Hello'';
      NeighborhoodComm.
             sendBroadcast<MyMessage>(abc_ch_128,msg1);
    } goto loc1; }
```

Figure 6: ANQUIRO model obtained from the code in Figure 5.

Figure 4. Every timer is represented as a different instance of a corresponding abstract data type. Line 1 binds the semantics of the operations in the `Timer` abstract data type to an underlying Java class. The expression in line 4 creates a new timer instance, which serves as parameter for all other functionality. The operations in line 6-8 serve to start/stop periodic or one-shot timers. The guard in line 9 yields true when a timer fires.

Translation example. Figure 5 shows a fragment of Contiki C code implementing periodic broadcasting of messages. It uses Contiki's `timer` library to interrupt the `while` loop every two seconds, and the `abc` module in Contiki's network stack to perform the transmission on channel `128`.

Figure 6 shows the ANQUIRO model output by the translator from the code in Figure 5. The translator automatically maps every Contiki process to a Bogor thread. The use of Contiki's `timer` library is translated into the use of the timer extension in Figure 4. The use of the `abc` module is mapped to the operations modeling neighborhood communication, listed in Figure 3. Specifically, in state `loc0` we perform all setup operations and start a periodic timer. In `loc1`, a guard suspends the execution of the model until the timer fires. When so, we create and fill the message, which is sent in broadcast before returning to checking the guard.

4.2 Checking Engine

We employ a custom reduction technique that leverages localized executions at given subsets of nodes. Especially in large scale systems, there may be executions that replicate identically in different parts of the system. If so, it is not necessary to re-examine these executions if they only differ in the subsets of nodes involved. To recognize these situations, we employ a dedicated hashing technique to tag the local node states. Our technique separates the state of a node as determined by the code it is running from node-specific information, e.g., its identifier. The former information is used to recognize the same state at different nodes, and thus to identify executions that only differ in the subsets of nodes involved.

In addition, inside the checking engine we implement the semantics of the modeling constructs at the different abstraction levels.

Hardware-specific. We are currently investigating how to hook existing hardware emulators to the checking engine. Instead of using the emulator for a complete execution of the code, the checking engine asks it to perform single steps in given computations, corresponding to given transitions in the state space. The checking engine continuously exchanges state information with the emulator: the execution step needs to start from a given initial state and the results of the computation must be returned to the checking engine as next state information.

Neighborhood communication. The implementation revolves around modeling broadcast communications and packet losses. The former is a form of multi-point communication that is difficult to model using standard approaches [15]. Embedding the implementation of such semantics inside the model checker allows us to abstract away states that are irrelevant for the functionality to verify, e.g., those generated to demultiplex the content of messages addressed to different receivers.

According to our network model, we generate two different system executions at every 1-hop packet transmission. These correspond to whether a packet is delivered or not, on a per-link basis. The loss occurs independently of what may cause it in reality, e.g., collisions or external interference, which is irrelevant from the network-level perspective. Nevertheless, not distinguishing these aspects allows us to abstract away different executions that are perceived as equivalent by network functionality. The checking engine may also exclude executions where the network is partitioned, which may cause most properties of interest to fail trivially.

System-wide communication. The implementation of this semantics focuses on abstracting away the effects of multi-hop communication. At this abstraction level, the user is interested in the end-to-end behavior of the communication network. Therefore, unlike the case of neighborhood communication, different system executions are generated for every packet transmission independently of the relative position of sender and receiver(s). In case of system-wide broadcast transmissions, we use the same techniques as in the neighborhood case to abstract away details of multi-point communication that are irrelevant for the verification.

5. EVALUATION

We demonstrate the use of ANQUIRO to verify a widely used data dissemination protocol against the specification of the core guarantee it is to provide. Throughout the study, we leverage neighborhood communication as abstraction level.

5.1 Data Dissemination with Trickle

We may expect that the software running on the nodes needs to be updated after deployment, e.g., to correct software bugs. To accomplish this task, data dissemination protocols are used. Their objective is to distribute new data to all nodes. The dissemination process starts at a base station that first injects the data to distribute. A fundamental guarantee such protocols are to provide is that *all* nodes eventually receive the data. For instance, it may be fatal if some nodes do not receive a software update and continue to run a possibly incompatible version of the code.

A widely used dissemination protocol is Trickle [12]. It achieves eventual delivery of data by making nodes periodically broadcast in the 1-hop neighborhood meta-data representing the current state, e.g., the latest version number of the software. Based on this information, a node may recognize that another device has more recent information. If so, the node proactively pulls the more recent data from the more up-to-date device. This guarantees that, if the network is connected, all nodes eventually receive the most recent data and the version numbers are eventually consistent.

To alleviate the overhead due to periodic broadcasting of meta-data, Trickle employs a form of "polite gossiping" to suppress redundant transmissions. Whenever a node hears that at least k of its neighbors already transmitted its same version number, it does not broadcast during the same period. To preserve the eventual consistency guarantee, nodes with more recent state always broadcast if they hear one of their neighbors with older meta-data.

5.2 Verifying Trickle with Anquiro

We consider Trickle's implementation in the default Contiki network stack and the corresponding default values for all protocol parameters. First, we isolate Trickle-specific code by eliminating functionality that is not directly tied to its operation, e.g., packet fragmenting and reassembling. Next, we develop a simple application to trigger the dissemination process. This distributes a single integer value after a given boot-up time. Finally, we assemble the minimal set of C source files to run the dissemination. These include the aforementioned application and a subset of Contiki's network stack down to the `abc` module. Using neighborhood communication, this is the lowest layer to considered, as it maps directly to domain-specific modeling constructs, as described in Section 4.1.

As property to verify, we specify Trickle's eventual consistency guarantee as follows:

$$\forall i \in Nodes \quad \Diamond(dissemination.version_i = 1) \qquad (2)$$

When nodes boot, their version number is initialized to 0. After the first dissemination process, their version number should eventually advance to 1 once they receive the disseminated data.

We configure ANQUIRO not to model network partitions, as they would make property (2) fail trivially. We investigate several topologies, in particular, *i)* grid topologies where every node talks to four immediate neighbors, *ii)* random topologies obtained with TOSSIM's random topology generator [11], and *iii)* some peculiar topologies that may be critical for dissemination protocols [22].

Outcome. ANQUIRO verifies property (2) for grid and random topologies. However, we find a set of peculiar topologies where eventual consistency is *not* satisfied. Figure 7 depicts one such topology. ANQUIRO shows the grey node never updating its version number. Based on the counter-example, we understand that this is due to the black node constantly suppressing its broadcasting of meta-data. In turn, this is because the white nodes always broadcast first, which causes the black node to constantly surpass the threshold of k neighbors already broadcasting the same meta-data. In this situation, the grey node never hears meta-data reporting a version number greater than its, and never requests the data.

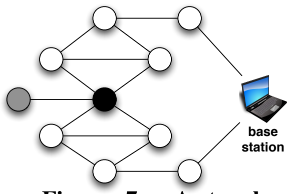


Figure 7: A topology where Trickle fails in delivering data to all nodes.

The issue above is due to an aspect that the Trickle's original description [12] overlooks, which reflects in most existing implementations: nodes must broadcast meta-data also when they have not received any data yet. By doing so, the grey node would eventually broadcast version number 0. According to the protocol operation, the black node would then broadcast version number 1 even if all its neighbors already broadcasted the same information. This allows the grey node to be informed of the new version number and to request the disseminated data. This feature is not present in the implementation considered, which caused the issue to emerge during the verification. Nevertheless, understanding the causes of the issue enables devising a fix quite easily.

Nodes	States	CPU time (min)	Memory (Mb)
10	10986	68.23	456.76
20	50871	234.01	765.76
30	109841	562.23	1282.76
40	NC	NC	Out of memory

Table 1: Performance in the absence of domain-specific abstractions. (NC indicates incomplete verification).

5.3 Run-Time Performance

We leverage our case study to investigate ANQUIRO's run-time performance. We use grid and random topologies with a varying the number of nodes. In the latter case, we test 100 randomly generated topologies with the same average density as the grid configuration, and average the results. We do not report results obtained from the peculiar topologies, as they represent specific cases that are not representative of the average use of our tool.

As performance metrics, we measure the *number of states* and peak *memory consumption* during the verification, as well as the *CPU time* to complete the verification. The former figure is indicative of the effectiveness of our domain-specific state abstractions, which aim to reduce the size of the state space. The peak memory consumption measures the maximum amount of computing resources spent during the verification. This is usually the bottleneck, as the verification may not complete because of memory overflows. The CPU time to complete the verification is an indication of how practical the approach may be when the verification does complete.

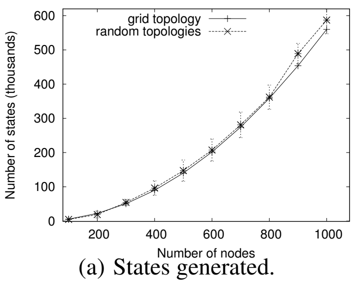
We run the experiments using a Linux desktop PC with a P4 3.2Ghz CPU and 2 Gb RAM, a standard Sun JVM version 1.5, the DJProf tool [6] to measure memory consumption, and Linux `time` command to measure the total CPU time.

Results. To provide a baseline for comparison, we run an initial set of experiments by exposing as generated states information that ANQUIRO would normally hide to the checking engine. This makes our tool operate in a way very similar to standard approaches, shown in Figure 1(a). Table 1 shows ANQUIRO's performance in this configuration, using grid topologies. The results for random topologies are essentially the same. If all states were exposed to the checking engine, the tool would be unable to complete the verification even with only 40 nodes, due to memory overflows.

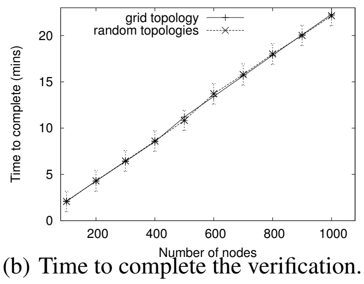
Our domain-specific state abstractions provide orders of magnitude improvements: Figure 8(a) depicts the number of states inspected under this setting. The chart shows a quadratic increase in this metric. This is expected, because the system state is obtained from the combination of the local states at the different nodes. As the number of them grows, the number of possible combinations increases. Random topologies show about the same behavior as grid ones. Thus, the performance is mainly dictated by the number of nodes, rather than by topological characteristics.

The quadratic trend in Figure 8(a) does not reflect in the CPU time to complete the verification, shown in Figure 8(b). ANQUIRO completes the verification within minutes even in large scenarios, demonstrating its general applicability. The trend here is essentially linear. This is due to the local state hashing technique, illustrated in Section 4. Even if several nodes are considered, the state of many of them may be the same. ANQUIRO recognizes this situation and avoids inspecting system executions where the behavior of the system is the same but it manifests at different nodes.

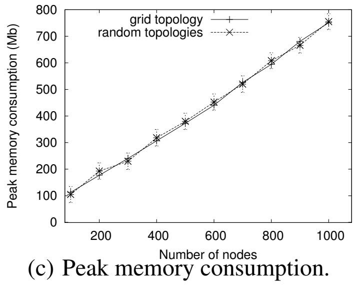
The above reasoning applies to Figure 8(c) as well, showing the *peak* memory consumption. The absolute values are well within the limits of today's PCs. The trend is linear as well. Indeed, memory is consumed mainly to store information about system executions that are actually inspected during the verification.



(a) States generated.



(b) Time to complete the verification.



(c) Peak memory consumption.

Figure 8: ANQUIRO **performance.**

6. CONCLUSION AND FUTURE WORK

We presented ANQUIRO, a domain-specific model checker for static verification of WSN software. ANQUIRO overcomes state space explosion problems by providing different abstraction levels based on the property to verify, and by leveraging domain-specific state abstractions to reduce the number of states inspected. We illustrated the use of ANQUIRO in verifying Trickle, discovering an issue that is commonly overlooked. ANQUIRO's run-time performance demonstrated that our techniques provide orders of magnitude improvements over standard solutions. Our research agenda includes completing the implementation of the ANQUIRO translator and the hardware-specific abstraction level, as well as applying ANQUIRO to the verification of entire WSN applications.

Acknowledgements. The authors thank Adam Dunkels for the feedback received on early versions of this paper.

7. REFERENCES

[1] L. Baresi, G. Gerosa, C. Ghezzi, and L. Mottola. Playing with time in publish-subscribe using a domain-specific model checker. In *Proc. of the SAVCBS Workshop*, 2007.

[2] L. Baresi, C. Ghezzi, and L. Mottola. On accurate automatic verification of publish-subscribe architectures. In *Proc. of the* 29th *Int. Conf. on Software Engineering (ICSE)*, 2007.

[3] D. Bucur and M. Kwiatkowska. Bug-free sensors: The automatic verification of context-aware TinyOS applications. In *Proc. of the European Conference on Ambient Intelligence (AmI)*, 2009.

[4] W. Cao, T. Abdelzaher, J. Stankovic, K. Whitehouse, and L. Luo. Declarative tracepoints: a programmable and application independent debugging system for wireless sensor networks. In *Proc. of the Int. Conf. on Embedded Network Sensor Systems (SENSYS)*, 2008.

[5] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, 2004.

[6] DJProf. Java Memory Profiler. `www.mcs.vuw.ac.nz/djp/djprof/`.

[7] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proc. of the Workshop on Embedded Networked Sensors (EmNetS)*, 2004.

[8] Y. Hanna, H. Rajan, and W. Zhang. Slede: A domain-specific verification framework for sensor network security protocol implementations. In *Proceedings of the ACM Conf. on Wireless network security*, 2008.

[9] G. J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5), 1997.

[10] K. Langendoen, A. Baggio, and O. Visser. Murphy loves potatoes: experiences from a pilot sensor network deployment in precision agriculture. In *Proceedings of the 20th Int. Parallel and Distributed Processing Symposium (IPDPS 2006)*, 2006.

[11] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *Proc. of the Conf. on Embedded Networked Sensor Systems (SENSYS)*, 2003.

[12] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proc. of NSDI*, Mar. 2004.

[13] K. Liu, M. Li, Y. Liu, M. Li, Z. Guo, and F. Hong. Passive diagnosis for wireless sensor networks. In *Proc. of the Conf. on Embedded Network Ssensor Systems (SENSYS)*, 2008.

[14] L. Luo, T. He, G. Zhou, L. Gu, T. F. Abdelzaher, and J. A. Stankovic. Achieving repeatability of asynchronous events in wireless sensor networks with EnviroLog. In *Proc. of INFOCOM*, 2006.

[15] P. Merino and J. M. Troya. Modelling and verification of the ITU-T multipoint communication service with SPIN. In *Proc. of the* 2nd *Int. Wrkshp. on SPIN Verification*, 1996.

[16] NESL. Lighthouse Project. `projects.nesl.ucla.edu/public/lighthouse/`.

[17] F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-level sensor network simulation with COOJA. In *Proc. of the Workshop on Practical Issues in Building Sensor Network Applications (SenseApp)*, 2006.

[18] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the sensor network debugger. In *Proc. of the Conf. on Embedded Networked Sensor Systems (SENSYS)*, 2005.

[19] Robby, M.-B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proc. of the* 9th *European Software Engineering Conf.*, 2003.

[20] K. Römer and J. Ma. PDA: passive distributed assertions for sensor networks. In *Proc. of the Conf. on Information Processing in Sensor Networks (IPSN)*, 2009.

[21] T. Sookoor, T. Hnat, P. Hooimeijer, W. Weimer, and K. Whitehouse. Macrodebugging: global views of distributed program execution. In *Proc. of the Conf. on Embedded Networked Sensor Systems (SENSYS)*, 2009.

[22] F. Stann, J. Heidemann, R. Shroff, and M. Z. Murtaza. RBP: robust broadcast propagation in wireless networks. In *Proc. of the Conf. on Embedded Networked Sensor Systems (SENSYS)*, 2006.

[23] B. Titzer, D. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *Proc. of Conf. on Information Processing in Sensor Networks (IPSN)*, 2005.

[24] S. Tschirner, L. Xuedong, and W. Yi. Model-based validation of QoS properties of biomedical sensor networks. In *Proc. of the* 8th *ACM Int.Conf. on Embedded Software (EMSOFT)*, 2008.

[25] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: using RPC for interactive development and debugging of wireless embedded networks. In *Proc. of Conf. on Information Processing in Sensor Networks (IPSN)*, 2006.

[26] M. Woehrle, C. Plessl, J. Beutel, and L. Thiele. Increasing the reliability of wireless sensor networks with a distributed testing framework. In *Proc. of the Workshop on Embedded Networked Sensors*, 2007.

[27] J. Yang, M. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. *Proc. of the Conf. on Embedded Networked Sensor Systems (SENSYS)*, 2007.