Correctness of Sensor Network Applications by Software Bounded Model Checking

Frank Werner and David Faragó

Institute for Theoretical Computer Science
Karlsruhe Institute of Technology (KIT)
`werner@kit.edu`, `farago@kit.edu`

Abstract. We investigate the application of the software bounded model checking tool CBMC to the domain of wireless sensor networks (WSNs). We automatically generate a software behavior model from a network protocol (ESAWN) implementation in a WSN development and deployment platform (TinyOS), which is used to rigorously verify the protocol. Our work is a proof of concept that automatic verification of programs of practical size ($\approx 21\,000$ LoC) and complexity is possible with CBMC and can be integrated into TinyOS. The developer can automatically check for pointer dereference and array index out of bound errors. She can also check additional, e.g., functional, properties that she provides by `assume`- and `assert`-statements. This experience paper shows that our approach is in general feasible since we managed to verify about half of the properties. We made the verification process scalable in the size of the code by abstraction (eg, from hardware) and by simplification heuristics. The latter also achieved scalability in data type complexity for the properties that were verifiable. The others require technical advancements for complex data types within CBMC's core.

Keywords: Software Bounded Model Checking, CBMC, automatic protocol verification, embedded software, Wireless Sensor Networks, TinyOS, abstraction, simplification heuristics.

1 Introduction

We strongly rely on embedded systems, which are widely used in highly distributed as well as safety-critical systems, such as structural monitoring of bridges [22], intrusion detection [12], and many industrial use cases, e.g., using SureCross from Banner Engineering Corp. [19] or Smart Wireless from Emerson Electric Co [10]. Hence they must become more dependable and secure.

The application of formal methods to embedded systems could be the key to solve this. Although embedded devices carry only some hundred kilobytes of memory, their verification is neither simple nor easily automated (cf. related work below) because they run complex algorithms for the underlying protocols, distributed data management, and wireless communication. In the special domain of wireless sensor networks (WSNs), these techniques are all combined in a single product, making it a challenging candidate for the application of formal methods. In this domain, powerful and extensible development and deployment frameworks are used, e.g., TinyOS [20]. By integrating formal methods seamlessly, i.e., fully automated, into such a framework, their usage by developers is most likely. This implies automatic generation of the model (cf. Section 4.2) for our verification process, which solves further problems:

- The development of manual artifacts is costly since the model is only required for verification.
- Since the verification model and the implementation must stay in conformance, the model must rapidly change, especially during the design phase. Hence additional work is required.
- There is a high danger to abstract from fault-prone details, e.g., due to missing constructs in the modeling language.

In this paper, we investigate a concast protocol [5] implementation called *ESAWN* [2] (*Extended Secure Aggregation for Wireless sensor Networks*). It is from the domain of WSNs and uses the development and deployment platform TinyOS. From this implementation, we automatically generate a software behavior model that fully comprises the protocol behavior of the sensor node. Then the model is used to rigorously verify the protocol using the software bounded model checking tool *CBMC* (*C Bounded Model Checking*) [6]. We used version 2.9, the most recent when we started, and implemented several heuristic simplifications and slicing rules (on a side branch of CBMC's repository) to make verification possible.

Related Work. Most approaches for protocol verification in WSNs use either a heavy abstraction from the actual implementation or only consider parts of the model behavior. The presented work is the first, as far as we know, to use software bounded model checking (SBMC) for verification.

The authors of [3] and [4] considers the application of verification techniques to software written in TinyOS, or more precisely, in the TosThreads C API. Instead on analyzing an integrative model with an operating system part and a protocol implementation, low level services are modeled and statistically verified against safety specifications. The verification tool employed was SATABS, which performs predicate abstraction using SAT and can handle ANSI-C and C++ programs. In this work the overall model size checked is at most 440 LoC. In our approach, we apply our abstraction to a more complex security protocol consisting of 21 000 LoC and obtain a model with about 4 400 LoC, which we subsequently check.

The T-Check tool [14] builds on TOSSIM and provides state space exploration and early detection of software bugs. The authors use a combination of model checking, random walks, and heuristics, to combat the complexity of nondeterministic branching. Also the results show the applicability of the tool and the fact that actual violated properties are found, this random search is not exhaustive and purely depends on the implemented heuristics [13] for finding liveness bugs, and the user's experience.

The Anquiro tool [15] is used for the verification of WSN software written for the Contiki OS using different levels of abstraction, which the user can select from. In comparison, our abstraction only eliminates direct function calls to the hardware and assembler constructs. Thereby, our abstraction is even able to detect erroneous packet fragmentations and reassembling errors. This is closer to the actual implementation - at the cost of complexity. In addition, since we use CBMC and its transformation mechanisms, we are able to directly point to the violating line of code.

The work in [3] considers the application of verification techniques to software written in TinyOS, or more precisely, in the TosThreads C API. Instead of analyzing an integrative model consisting of the operating system part and the protocol implementation, services are modeled and verified individually. The verification tool employed was SATABS, which performs predicate abstraction using SAT and can handle ANSI-C and C++ programs. In this work the overall model size checked is at most 440 LoC. In our approach, we apply our abstraction to a more complex protocol consisting of 21 000 LoC and obtain a model with about 4 400 LoC, which we subsequently check.

Insense [18] is a composition-based modeling language which translated models in a concurrent high-level language to Promela, to enable verification of WSN software by Spin. A complete model of the protocol under investigation has to be created, though, even if an implementation, e.g., in TinyOS, already exists. This is very time intensive and error prone. Though Spin is very well capable of analyzing concurrent and distributed settings, we experienced problems with state space explosion when checking a high-level behavior model in small topologies [21].

Structure of this paper. In Section 2, we introduce SBMC: in general, the SBMC tool CBMC, its capability to use nondeterminism, and the complexity of SBMC. Then we describe the heuristic improvements we contributed to make CBMC cope with our protocol. Section 3 explains the ESAWN protocol. Section 4 introduces the TinyOS platform in general and then the abstract behavior model we generate from its NULL platform. In Section 5, we specify the additional properties that we checked on the ESAWN protocol. The verification results are given in Section 6. Section 7 concludes this paper.

2 Bounded Program Verification

2.1 Software Bounded Model Checking

CBMC [6] is one of the most popular SBMC implementations for C programs. Before CBMC is described in detail, we will first review the technique SBMC. SBMC computes a solution to the following problem: For a program P, a bound k and a property f, does a path p within the bound k exist that violates the property f?

A *program state* can be characterized by the content of the heap, stack, all registers and a program counter. A *path* is a sequence of program states where a transition between states is triggered by the C statement at the program counter.

One interpretation of the *length* of a path in a program is the number of statements in a program. *Properties* declare some error states, or invalid sequences of program states, that shall never occur in any execution, e.g., certain values assigned to a variable.

A SBMC problem has three possible results [7]:

1. The property f holds for all paths.
2. The property f does not hold for at least one path p'.
3. The bound k is too small for at least one path p'.

In the two latter cases, a counter-example path p' is computed – a path having the form of a concrete program execution. For a program that contains a finite set of finite paths, k can always be set large enough such that a sound and complete verification of the property f can be achieved. If the bound is chosen too small, it can iteratively be increased.

Embedded systems (e.g., those following the MISRA C standard [1]) commonly use reactive systems that consist of an infinite outer loop, but within that loop, all possible paths are bounded.

We can usually show ultimate correctness for typical properties even when only considering these inner, finite functions (without the infinite loop around them), such that case 3 from above does not occur. This finitization might require underspecification (see Section 2.3 and 4.2).

2.2 CBMC

CBMC implements SBMC for C programs. Properties have to be specified by `assert(f)` statements. The semantics of such a statement is that whenever a program execution reaches the statement, the condition `f` must evaluate to `true`. CBMC also offers `assume(f)` statements, which we do not need. In CBMC, the positive integer *bound* denotes the maximum number of allowed loop body executions on a path and the maximum recursive depth. The *recursive depth* of a path is the number of stack frames it contains. For a given program, the bound limits the number of statements on any path. In CBMC the bound can be set individually for each loop occurring in the program.

The software behavior is encoded into a satisfiability (SAT) instance that is checked using a SAT solver (Minisat2 [9] in the case of CBMC). If the SAT problem is satisfiable, CBMC generates a concrete counter-example from the satisfying assignment produced by the solver. If the SAT problem is not satisfiable, the property holds for all program executions and the program always terminates.

2.3 Nondeterminism

In CBMC, we can set variables and return values of functions nondeterministically. Thus the model can subsume all possible behaviors of the implementation in a simple way. It usually contains even more behaviors than the implementation, i.e., the model is *over-approximated*, also known as *underspecified*. Then the verification can have false negatives, i.e., false error reports. But a successful verification implies a correct implementation, i.e., we do not have false positives.

2.4 Complexities

Covering all program states within the bound means that many values of the heap, stack, registers and program counter need to be considered (especially when nondeterminism is used). This leads to a combinatorial explosion of exponential size, called the *state space explosion.* The SAT problem encoding the SBMC problem has at least as many variables as the number of bits potentially addressed in the C program. Though the SAT problem is NP-complete, real world instances of SAT problems can be solved surprisingly fast. We found that the generated SAT instances of the investigated protocol posed a problem to Minisat2 simply because of their size. The problems were solved rather efficiently when we used preprocessing before calling Minisat2. For this, we engineered the following heuristic improvements into CBMC.

2.5 Extending CBMC Optimization Heuristics

Even for simple execution scenarios of our large scale program, e.g., one message shall be correctly processed, the size of the propositional formula that CBMC generates surpasses 4 GB. Therefore we extended CBMC with optimization heuristics, which respect non-simple types as arrays, pointers and structures and use slicing rules (enabled by the option `-slice`) and with simplifications that stem from the domain of compiler optimizations (enabled by the option `-use-sd`). They strongly reduce the problem size and complexity and are applied after the code is transformed into a more rudimentary, intermediate language (see below), but before it gets encoded into a SAT problem. The simplifications use the following steps, detailed in the following paragraphs:

– Constant propagation for arrays, pointers and structures, which can be computed efficiently in an unwound program.
– Expression simplification that uses the additional information generated by the constant propagation.
– Simplifying guards for statements by early satisfiability detection, using the above expression simplification.

SSA Encodings in CBMC. In order to solve the bounded software model checking problem, CBMC facilitates inlining (resp. unwinding of function calls and loops) up to an upper bound. CBMC also introduces single static assignments (SSA, see for example [17]) in the transformed program: Every assignment is replaced by a *versioned assignment* such that each identifier is assigned at most once. In order to transform a sequence of n assignments to a symbol, n new identifiers are introduced by appending a version number to the original identifier. Read accesses are replaced by read accesses to the currently active version. At program points where two control-flows join, e.g., the end of an `IF` block, a new

```
int x = 0;                                      x0 = 0;
if (x==0)                                       if (x0==0)
   x = x + 2;            becomes                   x1 = x0 + 2;
assert(x==0);                                   // phi:
                                                x2 = (x0==0) ? x1 : x0;
                                                assert(x2==0);
```

Fig. 1. Exemplary SSA translation

phi assignment is introduced that determines which version should be used for the consecutive read accesses (cf. Figure 1).

In SSA form, use-definition chains are easily computed, which will in the following be used for constant propagation. In CBMC, the SSA statements have *guards*, i.e., necessary and sufficient conditions for the statements to be executed.

Field- and Array-Sensitive Constant Propagation. Many implementations (like the ESAWN protocol) rely on heavy use of arrays, pointers and structures. CBMC already contains many optimizations, but does not yet facilitate constant propagation for non-simple data types before the generation of the SAT problem: Hence sequences as `a[0] = 0; if (a[0] == 0)` are not simplified. In contrast to the built-in approach, we have implemented the propagation on the level of the SSA representation of the program by flattening these complex data types.

Expression Simplifier. Using the additional information generated by the constant propagation, we have added an expression simplifier. Any expression that can be simplified by one of the three following rules is replaced by its simplified expression. It has to be noted that all expressions are side-effect free at this level of encoding:

– Boolean expressions with Boolean operands: If an expression has Boolean type and any Boolean operand must evaluate to a constant `true` or `false`, the expression is simplified, e.g., `expr && false` becomes `false` and `false? expr1:expr2` becomes `expr2`. Additional cases where more than one operand evaluates to a constant are also simplified.
– Boolean comparisons: Cases where Boolean operands have non-Boolean type operands are also simplified, e.g., `c <= c` becomes `true`, with `c` being a constant or versioned identifier.
– Integer expressions with constant integer or Boolean operands: Arithmetic expressions `+`,`-`,`*`,`/`,`<<`,`>>` where all operands are constants are simplified according to their C semantics.

The last rule can be extended to `float` and `double` type variables. As the ESAWN implementation does not use such types, they are not yet implemented. As we will show later, the above rules provide necessary simplifications for the verification of the ESAWN implementation.

Early Satisfiability Detection. The above simplifier can be effectively used to simplify guards for statements. If a guard always evaluates to `false`, a statement can be removed from the encoding as it cannot be executed anyway. If a reachable guard evaluates to `true` and the statement expresses an `assert` statement that evaluates to `false`, the program cannot be verified. The heuristic can often detect the reachability and stops further encoding with an according message. If the `assert` statement is always true it can be removed.

The effectiveness of early satisfiability detection lies in the fact that the unwinding bounds for loops are unknown and can only be determined by many runs of CBMC. For loops that are executed a fixed number of times, i.e., most of the ESAWN loops, the heuristic detects that loop bounds are chosen too small. Hence the overall process of finding the correct loop bounds is greatly improved.

We have 32 loops in total. For one loop, we were able to infer the required unwindings: It belongs to a `memset` function, which has to be iterated very often when duplicating memory locations. Consequently, we set the required unwindings to a sufficiently high and safe value of 20. The unwindings for the other loops were determined iteratively by automatically running the unwinding check provided by CBMC, and incrementing the unwinding setting if the unwinding-assertion failed.

3 The ESAWN Protocol

The protocol under investigation is called ESAWN [2]. It offers means to handle the transportation and aggregation of messages in sensor networks from many senders to one receiver, so called *concasts*. By using an end-to-end authenticity, the transport of sensible data is possible even in the presence of multiple malicious nodes under the control of an adverse acting entity. The protocol runs in two phases: First an initialization is necessary, before the actual probabilistic concast can be performed in the second phase.

We first consider the second phase, in which the actual sensor data is passed around and aggregated all along the way to the sink, the root node. In this phase, packets of type ESAWN are used. Since the packets are relayed down the aggregation tree via intermediate nodes, their entries are encrypted such that only the destination can decode its contents (see Section 5.2).

For the concast with probability, each node checks the authenticity of each received aggregate only with a fixed probability p. Otherwise it just assumes that the aggregate is authentic. Since authentication is costly, this is a trade-off between low energy consumption (low p) and high probability of authenticity (high p).

To be able to check for authenticity, a node sends its information to a fixed number w of additional child nodes, called *witnesses*. The employed concast saves additional energy by buffering packets and sending them all together later on using an aggregation function f_{agg}.

An example setup is given in Figure 2 where 5 nodes are used. The leaf node n_0 triggers the probabilistic concast by sending a packet (with its data value D_0)

to its successor on the aggregation path n_1. Since this node could be cheating, additional packets are sent to node n_2 and n_3, which act as witnesses to assure the proper behavior of node n_1. The nodes n_i ($i \in \{1, ..3\}$) are collecting all incoming packets, then check authenticity with probability p and finally, if all incoming packets were authentic, send out packet $agg_i = f_{agg}(agg_{i-1}, D_i)$ (with D_i being the new data value from n_i and $agg_0 := D_0$). The root node n_4 finally collects all data. It is located at the base station and accessible by the user.

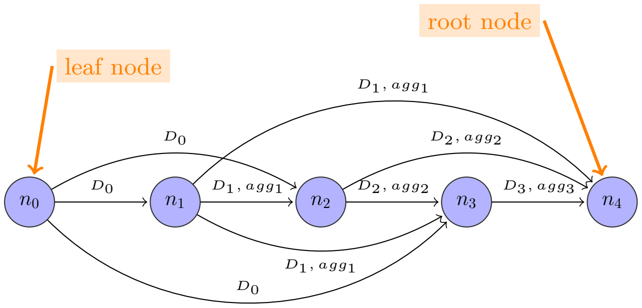

Fig. 2. ESAWN scenario of an aggregation tree with $w = 2$ witnesses

In the initialization phase, the parameter settings and the aggregation tree are made known to all nodes in the network. For this task, the ESAWN protocol uses $STATUS$ packets, which are also encrypted (see Section 5.1). So the number of nodes (num_nodes), probability p and the number of witnesses w are sent around in the network using SET packets. In addition, the aggregation tree is spread using packets of type $SETAGG$ which contain the $parent\_ids$ for each node. Finally, a packet of type GO triggers the second phase of the protocol. The GO packet contains a value specifying the frequency at which nodes send their data (0 means only one concast). Further packet types exist, which we do not consider since they play only a minor role for the verification of relevant global properties.

4 TinyOS Platform and Model Abstraction

4.1 The TinyOS Platform

TinyOS is an open source operating software for embedded devices and widely used for programming embedded devices. Its component based architecture and event driven execution model make it very suitable for resource constrained hardware systems with respect to memory, computation power and energy shortness.

TinyOS is an operating system and a software development platform that offers means to deploy the implementation on various hardware platforms through a modular design. So once a protocol, e.g., ESAWN, is implemented, it can be deployed automatically to the desired sensor type. With many possible combinations of interacting components, automatic verification within TinyOS is the solution for checking that the resulting composition behaves as expected.

In more detail, software in TinyOS is initially written in nesC, a C dialect having special constructs for embedded devices. Before the software can be deployed on sensor nodes, it is firstly translated from the modular description in nesC into an intermediate ANSI C representation, which includes specific constructs for interaction with the hardware. The C code could theoretically be used as model for the verification process already. But an abstraction is required when considering the size and complexity of the C code: Essentially, the hardware part, which includes register assignments and interrupt handling, would inhibit a successful verification process because of state space explosion. Due to this reasons another approach – described in the following section – is required. It abstracts appropriately from the hardware part by generating an *abstract behavior model.*

4.2 A Behavior Model Abstraction

The NULL Platform. The *NULL* platform is a hardware model included in the TinyOS environment. It can be used to generate a hardware independent software behavior model. In particular the platform can be understood as a skeletal structure containing only the functionality of the protocol plus some overhead in form of the *scheduling* functions for jobs and the *job queue.* But all hardware specific functions (e.g., for the UART and LEDs) are removed, i.e., empty function bodies are generated. Since this abstraction exactly comprises the pure protocol behavior under investigation, it is safe. Besides strongly reducing complexity, this abstraction has the major advantage that we do not have to take hardware platforms into account when specifying properties.

Abstract behavior model. We made four modifications to the NULL platform for our verification: instrumentation with `assert()` statements, a so-called autostart function, rudimentary packet transportation functionality and a task loop finitization:

The *autostart* function imitates some of the omitted hardware functionality, most of all the input from the environment. The function makes parameters known to a node by inserting them in its receive queue. Tasks can be enqueued to the *task queue* to let the node perform certain actions like sleep, start the processing of packets, etc. Essentially the autostart function brings us in the favorable position to bring the nodes in any state.

Since the NULL platform is hardware independent, also functionality that transports packets to the transceiver chip is lost. Since all protocols for WSNs depend excessively on packet sending and receiving, our generation rudimentarily inserts this into the function bodies of the sending and receiving functions that are empty in the NULL platform.

Task loop finitization changes the scheduler which periodically executes the task loop: The protocol usually does not terminate because of its controller-like nature: As long as the sensor nodes are active, new packets are generated and processed. The original task loop is hence infinite. By limiting the execution number of the task loop, a bounded model is obtained that is well suited for verification since its complexity is limited. In consequence, the model will run either until all tasks from the task queue are processed or an upper bound is reached, which we compute with injected code and check via assertions. If finished, it simply stops the node. In the course of this finitization, we are also able to further reduce the scheduler's complexity by replacing the complex functions for initializing the scheduler queue and the assignment of the empty task element by the necessary core functionality in the autostart function. With the help of nondeterminism for our correctness proofs, it is sufficient to show that individual packets are transported and processed in accordance with the protocol. Hence regarding the finite task loop is sufficient. Having only finitely many terminating paths, CBMC's verification is sound and complete. An overview of the generated model is depicted in Figure 3.

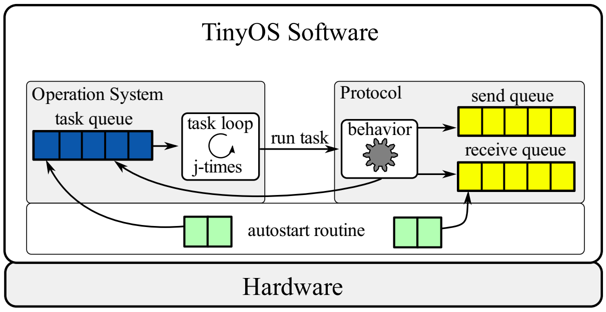

Fig. 3. Abstraction from TinyOS

Sensors are also not present in the NULL platform. But the implementation of the ESAWN protocol was using the node's IDs as sensor data to be transmitted, anyways, for clarity reasons. Since the IDs are unique, this approach also reduces the verification complexity.

We modified the NULL platform with manual intervention, but the modifications for the task loop finitization and packet transportation can be automated straightforwardly, e.g., by introducing a verification platform into TinyOS. The autostart function cannot be completely automated, since the initialization depends on the protocol and contains the configuration we want to consider.

With these modifications to the NULL platform, we get our *abstract behavior model*. From originally 21 000 lines of C code, as in the example of a real hardware platform (MicaZ nodes), the abstract behavior model only contains 4 400 lines of C code and CBMC statements.

Simulation. Besides verification, the abstract behavior model can also be used for simulation: We enriched the abstract behavior model with debugging statements and executed it. A few internal variables of TinyOS that were set nondeterministically in the model were now set to various specific values. This simulation can be strengthened by setting these variables automatically (e.g., randomly) until a desired coverage is reached, to even better complement the verification process.

5 Specification

After acquiring the abstraction in Section 4.2, we start specifying properties. These properties need to be local, since we only have a limited scope of a single node since the original code is intended for deployment. This means we cannot specify properties that include two or more nodes, only the behavior of one node at a time can be verified. Correctness is shown with local properties by non-deterministically setting the network into all relevant states using our powerful autostart function and then checking the according desired behavior at a single node (cf. the exemplary REQ4 below). This is achieved using `assert()` statements incorporated into the sources, to be able to check additional properties, i.e., monitor the behavior and stop the execution in case of an error. In Section 5.3 we will show a solution for global properties. We formulate the desired functional behavior as requirements (REQ), which are all translated into properties that are verifiable by CBMC using CBMC's `assert` functions. These assertions check whether the corresponding variables (e.g., a node's locally stored parameter w or outgoing packet queue) are set correctly. The assertions are either located after a node's computation or within the alarm function that is built into the protocol. This instrumented function is then able to indicate wrong behavior of the protocol, potential attacks and also erroneous packets.

5.1 STATUS Packets

The entries of the $STATUS$ packets are encrypted with an RC5 cipher. To avoid state space explosion, the encryption procedure was automatically removed for the abstract behavior model without changing the underlying protocol, i.e., nodes send their data as plain text.

For the autostart function, we chose an initialization of the network as described in Section 3 (cf. Figure 2), so $w = 2$ and $num\_nodes = 5$. p was set to 1, so we check with the strictest possible authentication and can fully avoid the complexity caused by probability, i.e., the random variables and the function computing the seed. Whether the values are set correctly by the autostart function is checked via the assertions for the following three requirements, which are categorized by packet type.

The first requirement covers packets of type SET, which are sent initially by the base station to make protocol parameters known to the network. The following property states that a node processes this type of packet correctly:

$$SET(num\_nodes, w, p) \text{ sets variables correctly} \qquad \text{(REQ1)}$$

The second requirement considers packets that make the aggregation tree public using $SETAGG$ packets. For this reason each node is informed about its successor nodes that it will send packets to. The $SETAGG$ packets contain the fields $node\_id$ and $parent\_id$ and must be sent to every node in the network.

$$SETAGG(node\_id, parent\_id) \text{ sets variables correctly} \qquad \text{(REQ2)}$$

The following requirement is about protocol conform behavior after receiving a *GO* packet: Only leaf nodes initiate concasts and the frequency value f in the *GO* packet (cf. page 122) must be respected.

$$\text{correct action upon reception of } GO(f) \tag{REQ3}$$

We omit the trivial requirements for the packet type *RESET*, which causes a hard reset of the node, and *ALARM*, which is simply forwarded.

5.2 ESAWN Packets

Entries of ESAWN packets are encrypted using symmetric keys (cf. SKEY [23]). Again, we consider unencrypted packets instead. Similarly to *STATUS* packets, we also split the correct handling of ESAWN packets into several requirements.

Firstly, we require that ESAWN packets are correctly transported. This also implies that packets have been correctly aggregated and are correctly forwarded (e.g., correct computation of the relay count). As aggregation function f_{agg} the sum is used ($f_{agg}(a, b) = a +_{\text{int}} b$). We check this requirement exemplary for the packet P that contains D_1, agg_1 sent to n_2:

$$\text{correct reception of packet } P \tag{REQ4}$$

Secondly, we require that ESAWN packets are correctly authenticated (which also implies correct aggregation). For this, a node n_i has to alarm if any of the last w aggregates is incorrect (n_0 to n_w can only check fewer aggregates):

$$(\exists j \in \{1, .., w\} : agg_{i-j} - D_{i-j} \neq agg_{i-j-1}) \iff alarm_i \tag{REQ5}$$

Finally, we must also check that this $alarm_i$, a certain alarm function built into n_i, behaves correctly, i.e., issues an *ALARM* packet to be sent. We do this by checking whether *ALARM* packets are put in the outgoing packet queue out_{n_i} of n_i:

$$alarm_i \implies ALARM \text{ packets in } out_{n_i} \tag{REQ6}$$

5.3 Global Properties

Global properties can achieve stronger and more comfortable formulations, for instance: `if some node alarms, then eventually the sink will receive an ALARM packet`. Since we are verifying the derived code that can be deployed on a sensor node, the verification process cannot handle multiple nodes so far, i.e., it does not consider distributed settings where messages are interchanged. To imitate this, we implemented simple multitasking between nodes: When the current node sends out a packet, a switch between nodes takes place. For this, we modified TinyOS' send routine: The local variables of the current node are saved and the local variables of the destination node are loaded. The packet being sent is enqueued into the receive queue. With this, a distributed network behavior can be imitated to some degree, with packets being sent to their destination without delay.

The trade-off of using global properties is an increased complexity. Therefore, we successfully verified only very simple ones and will use more powerful global properties only in future work after local properties no longer cause problems (cf. next section).

6 Verification Results

For the verifications, we used CBMC version 2.9 with our additional heuristics (cf. Section 2.5), some bug fixes related to complex data types and compiled for 64bit processors because some verifications required a lot of memory (see below).

Table 1. Verification results for *STATUS* packets for a valid loop unwinding of 4

SET packets			*SETAGG* packets			*GO* packets		
check	passes	|claims|	check	passes	|claims|	check	passes #	|claims|
REQ1	yes	6	REQ2	no	4	REQ3	yes	4
unwinding	yes	37	unwinding	yes	37	unwinding	yes	37
bounds	yes	60	bounds	yes	60	bounds	yes	59
pointer	no	181	pointer	no	177	pointer	no	175

As described in the previous section, the generated code is manually instrumented with the assertions that specify REQ1 to REQ6. All other assertions are inserted automatically by CBMC. The first verification step is finding the required number of loop unwindings using the according assertions to be sure the verification of the other properties is complete.

Table 1 displays the performed verifications for the *STATUS* messages, their results and number of required claims, which are CBMC's internal assertions. For the verification, we fixed node n_2, which exhibits all the behavior relevant to our verification. An unwinding depth of 4 is sufficient. The properties for code safety detect array index out of bounds and bad pointer dereferences. All checks had to be performed for each REQ since the autostart function was adjusted to each REQ. The pointer checks failed for every packet type. Debugging the source code of CBMC showed that this is not a failure of the protocol, but CBMC does not find correct symbols during its pointer-analysis.

For REQ1 and REQ3, all other checks are successful. The assertions for REQ2 are violated: The cause seems to be that CBMC is unable to handle arrays of structures, which are heavily used for the queues. This is one example where CBMC does not scale related to data type complexity.

Besides verifying these properties, we raised our confidence in the correctness of ESAWN by successful simulation (cf. 4.2) and fault injections in the code and in the assertions, all of which CBMC found.

Unfortunately, we were not able to verify REQ4 to REQ6 because the unwinding checks were problematic: At first we had difficulties setting the loop unwindings just as high as necessary, which is crucial. For instance, when we set the unwindings to 11 for all loops, CBMC requires 30GB of RAM (and over 3 hours) to detect that not enough unwindings were made. For 12 unwindings, CBMC gives segmentation faults because 32GB are exceeded. We solved the difficulty of finding the smallest possible unwinding value for each loop by searching automatically. But as the search is very time consuming, it is important to start with sensible values. When we used `--unwind 6 --unwindset 1:20`, i.e., unwindings 20 for the first loop (`memset`, which needs to be able to copy values sufficiently often) and unwindings 6 for all others, verification came much further with much less memory: With 2.5GB, CBMC reached the stage `passing to decision procedure`. Unfortunately, CBMC then halts with the error message `unexpected array expression: typecast`. Because CBMC aborted with a typecast exception, we tested whether the unwindings might be sufficient by injecting a fault into one of our assertions for REQ4 to REQ6. But these verifications also caused typecast exceptions. This shows again that CBMC does not scale with data type complexity.

CBMC offers two possibilities when enough unwindings cannot be reached efficiently: Firstly, paths with more unwindings can simply be ignored. But this leads to a bad testing coverage: In our case, a lot of packets in the queue need to be processed for initialization. Thus the processing of the ESAWN packets – and therefore their bugs – would not be reached. Secondly, we could have used nondeterminism at points where the maximum unwindings are reached, and possibly over-approximate (cf. Section 2.3). In our case, we would need to generate packets nondeterministically. Because of CBMCs difficulties with complex data types, it cannot create them nondeterministically. Hence the only solution would be the cumbersome manual implementation of nondeterministically generating a protocol-conform sequence of packets whenever maximum unwindings are reached. But that would counteract our intent of a fully automatic verification process. We also tried the current CBMC version 3.6. Since it does not include our heuristics (cf. Section 2.5), we encountered segmentation faults, e.g., when passing the problem to propositional reduction, already with 4 unwindings. We alternatively tried VCC [8], a SBMC tool similar to CBMC and currently developed at Microsoft Research. We experienced similar problems as in the first steps with CBMC: Pointer constructs present in the generated model could not be handled correctly and resulted in a syntax error while parsing. This shows that handling complex data types in SBMC tools is currently problematic, but a necessary improvement for verifying realistically complex programs.

7 Conclusion

7.1 Summary

We have described a proof of concept for an automatic verification process for realistically large and complex sensor network applications that can be integrated into the software design process. To be able to handle such large scale programs, the process must be automatic and requires the abstractions and heuristics we provided. It generates an abstract behavior model that is then verified by CBMC. We were able to prove correctness for the *SET* and *GO* packets, but not for the *SETAGG* and ESAWN packets, due to technical difficulties in CBMC, e.g., unsupported arrays of structures, pointer bugs and typecast exceptions. It shows that, in our case, CBMC does not scale well with the complexity of data types. Since we learned from our case study that this is very important for the successful verification of programs of practical size, CBMC (and VCC) can improve by not only supporting flat C data types, such as a single struct or array, but also their closure, i.e., nested types. A different solution is using a simpler intermediate language, e.g., LLVM (see Section 7.2).

Many of the technical difficulties in CBMC were caused by large function parameters ($\approx$ 500 byte) in the source code of ESAWN. In some cases, this can be considered a design flaw in ESAWN since frequent, unnecessary copying (because of C's call-by-value evaluation) is inefficient. We have informed the developer of ESAWN about this.

Our heuristics (cf. Section 2.5) improved the scalability in data type complexity, and even more the scalability in the size of the code: Without them, state space explosion prevents verification of even the simplest instances for the ESAWN protocol. A general lesson learned is that recent advances in compiler optimizations for the generation of runtime code can also improve static analysis mechanisms in real world settings, which is another argument for LLVM.

Our abstractions (cf. Section 4.2) also improve scalability in the size of the code and additionally allow hardware independence. Using our heuristics and abstractions, we have seen that, in general, CBMC is powerful enough to be employed in the verification process for large scale programs.

7.2 Future Work

As further SBMC tools emerge and improve, we can use our case study as benchmark for them, e.g., for NEC's VeriSol via F-Soft [11]. We can also consider unbounded model checking tools, e.g., use our generation of the abstract behavior model and apply SATABS afterwards, in the line of the recently published paper [3]. If this approach is infeasible, a combination of SBMC and predicate abstractions (cf. [16]) might be able to cope with our large protocol. At our institute, we are currently developing a new SBMC tool which will be based on the LLVM compiler toolkit. We expect that with this new tool, many of the technical difficulties can be avoided, and also better scalability can be achieved.

Promising enhancements in our abstract behavior model are: Firstly, improving multitasking between nodes for verifying global properties. We can reduce the large memory requirement by not storing the local variables (e.g., w, p and the whole aggregation tree) of all simulated nodes independently, but compactly or even only once. We can also implement a more general multitasking that allows several leaf nodes and delayed transmission of packets. This is achieved by using one extended scheduling function that comprises all jobs of all simulated nodes. With these improvements, all distributed properties we have verified with the tool Spin in [21] using hand-written models will be verifiable automatically.

Secondly, we can use an alternative to setting $p = 1$: By settling for a quantitative instead of a qualitative inspections, i.e., by using CBMC's nondeterminism instead of probabilistic choices, we are able to avoid the complexity of using probability and still investigate all possibilities of the probabilistic concast. The trade-off in this approach is the loss of quantitative results and the additional complexity that nondeterminism might cause.

Thoroughly investigating the protocol's robustness is another important research direction, made possible by the powerful autostart function. Since it can set all state variables to arbitrary values, also hazardous situations can be constructed.

Incorporating our generation scheme (simulation features inclusive) as a verification platform into TinyOS will enable many developers of WSN protocols to easily check correctness of their implementations.

References

1. Motor Industry Research Association. MISRA-C 2004: Guidelines for the Use of the C Language in Critical Systems. Motor Industry Research Association (September 2004)
2. Blaß, E.-O., Wilke, J., Zitterbart, M.: Relaxed Authenticity for Data Aggregation in Wireless Sensor Networks. In: 4th International Conference on Security and Privacy in Communication Networks (SecureComm 2008), Istanbul, Turkey (September 2008)
3. Bucur, D., Kwiatkowska, M.: Ambient Intelligence. In: Tscheligi, M., de Ruyter, B., Markopoulus, P., Wichert, R., Mirlacher, T., Meschterjakov, A., Reitberger, W. (eds.) AmI 2009. LNCS, vol. 5859, pp. 101–105. Springer, Heidelberg (2009)
4. Bucur, D., Kwiatkowska, M.: Towards Software Verification for TinyOS Applications. In: Proc. 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2010), Computing Laboratory, Oxford University, UK, pp. 400–401. ACM, New York (April 2010)
5. Calvert, K.L., Griffioen, J., Sehgal, A., Wen, S.: Concast: Design and implementation of a new network service. In: Proceedings of 1999 International Conference on Network Protocols (1999)
6. CBMC: Bounded Model Checking for ANSI-C, `http://www.cprover.org/cbmc/` (March 2010)
7. Clarke, E.M., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
8. Cohen, E., Moskal, M., Schulte, W., Tobies, S.: A practical verification methodology for concurrent programs. Technical Report MSR-TR-2009-15, Microsoft Research (February 2009)
9. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
10. Emerson Smart Wireless, `http://www2.emersonprocess.com/en-us/plantweb/wireless/pages/pages/wirelesshomepage.aspx`
11. Gupta, A.: From hardware verification to software verification: Re-use and re-learn. In: Yorav, K. (ed.) HVC 2007. LNCS, vol. 4899, pp. 14–15. Springer, Heidelberg (2008)

12. Ioannis, K., Dimitriou, T., Freiling, F.C.: Towards intrusion detection in wireless sensor networks. In: Proceedings of the 13th European Wireless Conference (2007)
13. Killian, C., Anderson, J.W., Jhala, R., Vahdat, A.: Life, death, and the critical transition: Detecting liveness bugs in systems code. In: Proc. of the 4th Symposium on Networked Systems Design and Implementation (NSDI), Cambridge, MA, USA (2007)
14. Li, P., Regehr, J.: T-Check: Bug Finding for Sensor Networks. In: Proc. 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2010), School of Computing, University of Utah, USA, pp. 174–185. ACM, New York (April 2010)
15. Mottola, L., Voigt, T., Österlind, F., Eriksson, J., Baresi, L., Ghezzi, C.: Anquiro: Enabling Efficient Static Verification of Sensor Network Software. In: Proc. 1st International Workshop on Software Engineering for Sensor Networks (SESENA - Colocated with 32nd ACM/IEEE International Conference on Software Engineering ICSE), ACM, New York (2010)
16. Post, H., Sinz, C., Kaiser, A., Gorges, T.: Reducing false positives by combining abstract interpretation and bounded model checking. In: ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, Washington, DC, USA, pp. 188–197. IEEE Computer Society, Los Alamitos (2008)
17. Rosen, B., Wegman, M., Zadeck, F.: Global value numbers and redundant computations. In: 15th ACM Symposium on principles of Programming Languages, pp. 12–27 (1988)
18. Sharma, O., Lewis, J., Miller, A., Dearle, A., Balasubramaniam, D., Morrison, R., Sventek, J.: Model Checking Software. In: Păsăreanu, C.S. (ed.) SPIN Workshop. LNCS, vol. 5578, pp. 223–240. Springer, Heidelberg (2009)
19. SureCross Wireless Industrial I/O Sensor Network Applications, `http://www.bannerengineering.com/en-us/wireless/surecross_web_appnotes`
20. TinyOS: An open-source OS for the networked sensor regime (March 2010), `http://www.tinyos.net`
21. Werner, F., Steffen, R.: Modeling Security Aspects of Network Aggregation Protocols. In: 8. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze, Hamburg, pp. 83–86 (August 2009)
22. Xu, N., Rangwala, S., Chintalapudi, K.K., Ganesan, D., Broad, A., Govindan, R., Estrin, D.: A wireless sensor network for structural monitoring. In: SENSYS, pp. 13–24. ACM, New York (2004)
23. Zitterbart, M., Blaß, E.-O.: An Efficient Key Establishment Scheme for Secure Aggregating Sensor Networks. In: ACM Symposium on Information, Computer and Communications Security, Taipei, Taiwan, pp. 303–310 (March 2006) ISBN 1-59593-272-0