Mechanized extraction of topology anti-patterns in wireless networks

Matthias Woehrle[1], Rena Bakhshi[2], and Mohammad Reza Mousavi[1,3]

[1] Embedded Software Group, Delft University of Technology, The Netherlands
m.woehrle@tudelft.nl
[2] Vrije Universiteit Amsterdam, Department of Computer Science, The Netherlands
rbakhshi@few.vu.nl
[3] Eindhoven University of Technology, Eindhoven, The Netherlands
m.r.mousavi@tue.nl

Abstract. Exhaustive and mechanized formal verification of wireless networks is hampered by the huge number of possible topologies and the large size of the actual networks. However, the generic communication structure in such networks allows for reducing the root causes of faults to faulty (sub-)topologies, called anti-patterns, of small size. We propose techniques to find such anti-patterns using a combination of model-checking and automated debugging. We apply the proposed technique on two well-known protocols for wireless sensor networks and show that the techniques indeed find the root causes in terms of canonical topologies featuring the fault.

1 Introduction

Wireless (sensor) networks are increasingly used in critical areas such as geoscience [4] and medicine [22], where correct and seamless operation is imperative. Automated formal methods, in general, and model-checking, in particular, have been used to ensure the correctness of computer systems and communication protocols, but their application to wireless (sensor) networks is barred by the well-known state-space explosion problem. This problem is severely intensified in this domain due to the huge number of possible topologies (initial states) for model-checking as well as the huge number of possible actions (next-steps) to be taken by the numerous sensor nodes present in the protocol.

In the field of model checking, some reduction techniques such as symmetry and partial-order reduction have been proposed in order to combat the state space explosion problem. Nevertheless, our experiments show that even a combination of the traditional techniques falls short of providing a solution for the state spaces resulting from sizable wireless networks.

However, the generic structure of communication primitives in wireless network protocols may come to the rescue: nodes of such networks work independently, run similar or identical protocols, which are designed regardless of the size of the network, and the protocols comprise generic and simple communication primitives. Due this generic structure, a potential problem in a large network should be traceable to a generic root cause, which also shows itself in an "anti-pattern" of small size, i.e., minimal faulty sub-topologies that demonstrate the causes of possible failures. Subsequently, the problem of model-checking large networks (with a huge number of possible topologies) is reduced to checking a small set of anti-patterns in large networks. For this approach to be successful, the anti-patterns should canonically capture the essence of possible flaws in the design. Moreover, these anti-patterns can then also serve as guidelines for the network designers to make sure that the network topology never forms such anti-patterns in its future evolutions. In this paper, we propose an approach based on a combination of model-checking and automated debugging to find anti-patterns. Searching for anti-patterns is a formal test-based approach; it is sound but not complete, because the topology space is too large to be explored exhaustively. We examine the above-mentioned approach on a number of simple, yet typical, protocols for wireless sensor networks: a probabilistic code dissemination protocol, called Trickle [21] and a medium access control protocol, called LMAC [28].

The contributions of this paper can be summarized as follows:

– We provide two complementary algorithms for determining and understanding anti-patterns.
– We show, by means of case studies, that the detected anti-patterns can canonically describe faults / implicit assumptions in protocol descriptions.

The structure of the paper is as follows. In Section 2 we provide an overview of the related work. Section 3 describes our approach and its two main algorithms. In Section 4 we describe the case studies and use them to demonstrate our approaches. We conclude with Section 5.

2 Related work

This paper focuses on understanding the root causes that let a protocol fail on particular topologies. As such, our work is closely related to isolating faults in software executions, of which the goal is to facilitate debugging by finding a minimal failing execution. Most prominently, delta debugging [29] uses a set of passing and failing conditions in order to efficiently uncover a small failing execution. Complementary to our approach is the approach to explain counterexamples of model-checking runs [6, 12]. Note that we cannot generally rely on having a complete set of counterexamples; some model-checking tools, such as PRISM [13], do not provide any counter-example. For some expressive modal and temporal logics, it is arguably questionable what the counter-example should be [8]. Additionally, as illustrated by our case studies, some topology-dependent bugs cannot be easily understood and generalized using counter-examples, yet are easily comprehensible and generalizable by identifying the topology anti-pattern. Similarly, there has been previous work on leveraging logs of sensor networks executions in order to find root-causes of errors [18, 23]. Different from these works, we work on high-level specifications of protocols and are particularly interested in faults due to specific topology patterns. Hence, in contrast to finding the exact location in one node's software, we analyze topological anti-patterns to understand the root cause of a protocol failure.

Another related approach for debugging is diagnosis in general, and spectrum-based diagnosis [1] in particular. In spectrum-based diagnosis passed or failed executions are scrutinized and annotated with information about the execution of each line (block or module) of code. Note that for diagnosis we do not differentiate whether in a particular run a block caused the failure or not; we just consider whether it is part of the whole execution. Obviously we need a different formulation (than line of code) for debugging topologies. In particular, we investigated using subgraph inclusions as "basic blocks" and count the specific subgraphs included in a topology and count their occurrence in faulty and working examples. Thereby we can use the same ideas as in spectrum-based diagnosis of software using different similarity coefficients such as the Jaccard coefficient. Diagnosis has the advantage of only requiring a fixed set of executions, while our methods necessitate additional model checking runs and are thus more time intensive. However, as demonstrated by our experimental results, the results of our algorithms are much clearer compared to the results obtained by using spectrum-based diagnosis.

We use model-checking as a vehicle for our bug-hunting approach. Model checking is an exhaustive and fully automatic state space exploration technique that has been successfully applied to many academic and industrial systems [7]. However, a naive attempt for model checking wireless sensor networks is bound to fail, due to the well-known state-space explosion problem. To overcome this problem several techniques have been proposed to reduce the state space of such networks, in particular: symmetry [9, 14] and partial order reduction [11, 25, 27], abstraction [2, 17, 19], and approximation [5, 20], as well as domain-specific reduction techniques [16, 24]. Our earlier experiments showed that, even after applying reduction techniques, model-checking networks of actual size still remains practically infeasible. Also reducing the number of topologies using measures of symmetry did not lead to a workable subset for networks of considerable size. Hence, we decided to change our strategy and first find anti-patterns of small size, which characterize possible causes for failure, and then efficiently search for these anti-patterns in networks of larger size. Our experimental results show that this does lead to an effective and efficient debugging procedure. In view of the probabilistic features of our case studies, in this paper, we focus on model checking of probabilistic models and to this end, we use PRISM [13] as our probabilistic model-checking tool.

An alternative approach that has been used in proving correctness of wireless (sensor) network protocols is computer-assisted theorem proving [15]. The advantage of this approach is that it can provide a general proof of correctness under given assumptions. The disadvantage is that the assumptions under which the protocol works correctly is not usually precisely specified and sometimes even not known to the designers. Moreover, theorem proving requires some affinity with the proof tools and the underlying mathematical theories. The two approaches can, however, be combined by finding anti-patterns using our approach, generalizing them and using them as assumptions (i.e., absence of generalized anti-patterns) as proof obligations.

We use two case studies of topology-related faults in wireless sensor networks that have been previously discussed. The Trickle protocol [21] has been shown to be flawed in the presentation of Anquiro [24] with respect to its threshold value for overhearing broadcast transmission. We had to make our own model of Trickle in PRISM for our experiment, since the tool presented in [24] is not available; however, our technique is applicable to any model-checking and automated verification, including that of [24]. The LMAC protocol [28] has been first modeled and verified by Fehnker et al. [10] based on timed automata models using UPPAAL [3]. The authors considered a range of different topologies, of size up to five nodes, and manually determined the cause of failure. As we demonstrate in our case studies, our approach automatically arrives at the root cause without necessitating manual generation of test cases nor analysis of 61 topologies.[4]

3 Identifying topology anti-patterns

The goal of this work is to detect *anti-patterns*: small faulty topologies that characterize faults or implicit assumptions inherent to a particular protocol. Once these anti-patterns are exhaustively enumerated, the problem of checking correctness for larger designs is reduced to finding anti-patterns in them, which is much more efficient than model checking the state space. Our approach is inspired by the seminal work of Zeller et al. [29] on delta debugging. Similar to delta debugging, we investigate two complementary approaches for identifying topology anti-patterns: (*i*) *minimization* of topologies to find a set of minimal topologies that fail and (*ii*) *isolation* of a single edge that changes a passing topology to a failing topology.

The premise of our approaches is that we check, for a given topology, whether the protocol model P violates the required properties ϕ given a certain topology g, i. e., $(\mathcal{P}||g) \not\models \phi$; in the present paper, we achieve this by means of model checking. Our starting point is always a failing topology (or a set thereof) and we search for the root cause of failures in these topologies. As we only decrease topology sizes (and therefore state space) in our algorithms, the runs of the model checker should always return a pass or fail answer. In case the run does not terminate with a definite answer, we assume a passing run, since we cannot prove the presence of a fault. Note that we assume that wireless network protocols are designed for any type of network, i. e., protocol properties should hold invariant of the topology. Hence, it follows that if: $(\mathcal{P}||g) \not\models \phi \implies \forall g' \subset g : (\mathcal{P}||g') \not\models \phi$.

3.1 Minimization

For minimization, we start with a set of topologies G, where a topology $g \in G$ is a graph, i. e., $g = (V, E)$. We focus on the set of failed runs F, i. e., $F = \{g \in G \mid (\mathcal{P}||g) \not\models \phi\}$. Given F, we try to find a set of smallest topologies S in order to determine anti-patterns.

[4] Note that Fehnker et al. included duplicate topologies in their work. The actual number of unique topologies of size 5 is 58.

Minimization algorithm Algorithm 1 summarizes our approach. Based on the set of failing topologies F and using the procedure REDUCE, we minimize each failing topology w. r. t. its number of (i) nodes and (ii) edges by calling the procedure MINIMIZE. Note that this order is implied by the fact that reduction in the number of nodes (removing all of its connected edges) is more granular than removing a single edge.

Procedure MINIMIZE reduces the number of nodes or edges respectively. The procedure MINIMIZE searches for subgraphs of smaller size and adapts the bound on the topology size until decreasing the bound results in no failing topologies. We use the same procedure both for nodes and edges (parameterized by [NODES/EDGES]), as they work identically, except for the generation of subgraphs using the function SUB. SUB removes nodes or edges, respectively, depending on its parameter. SUB(g,n) returns all (connected) complete edge-induced subgraphs of graph g of order n (NODES), or size n respectively (EDGES). A complete edge-induced subgraph of graph $g = (V, E)$ is a graph $g' = (V', E')$, $V' \subset V$ and $E' = \{(v_1, v_2)|(v_1, v_2) \in E \wedge v_1, v_2 \in V'\}$. Note that we can trivially speed up the algorithm by memoizing calls to MINIMIZE with previously checked topologies.

The output of REDUCE is the set of smallest topologies S.[5] As we see in the case studies, this results in a small set of minimal topologies that may represent the essence of the fault.

3.2 Isolation

Minimization results in a small set of graphs that (may) explain the underlying fault of the protocol. Additional to minimization, we also perform fault isolation, i.e., to identify the discriminating edge that lets a protocol fail. We start on the one hand with a failing topology and on the other hand with a passing topology and close in on the fault. Algorithm 2 presents the details of the approach: The user provides one failing topology f_{in}. We use as an initial passing topology a graph with a single node that trivially satisfies requirements.[6] The algorithm relies on building the relative complement δ of the failing and the passing topology, i. e., $\delta = E_{f_-} \setminus E_{f_+}$, where f_-, f_+ are the currently smallest failing or largest passing topology respectively. We sample from δ to shrink the failing and extend the passing topologies, respectively, such that the new topology f_{new} is also a connected graph. Please note that in this way the passing topology is always a subgraph of the failing topology. If the newly created topology passes we assign it to the currently largest passing topology f_+, else it is the currently smallest failing topology f_-. Thereby, we iteratively increase/decrease the topologies until they differ by a single edge. This single edge is instructive on why the protocol fails.

[5] When building set S, we check that each element $s \in S$ is unique modulo graph isomorphism.

[6] Depending on the protocol requirements, a larger passing topology with more nodes may be used.

Algorithm 1 Network minimization based on binary search

```
procedure $\textsc{reduce}(\mathcal{P}, \phi, F)$
    /* **input**
        $\mathcal{P}$, Protocol model
        $\phi$, Protocol properties
        $F = \{f_1, \ldots, f_m\}$, Set of faulty topologies with $f_i = (V_{f_i}, E_{f_i})$
    **output**
        $S = \{s_1, \ldots, s_n\}$, Set of smallest topologies */
    $H = S = \emptyset$
    // Minimize faulty topologies
    for $f \in F$ do
        $H = H \cup \textsc{minimize[nodes]}(\mathcal{P}, \phi, F, 1, |V_f|)$
    end for
    for $h \in H$ do
        $S = S \cup \textsc{minimize[edges]}(\mathcal{P}, \phi, H, 1, |E_h| - |V_h| + 1)$
    end for
    return S
end procedure
procedure $\textsc{minimize[nodes/edges]}(\mathcal{P}, \phi, T, \mathit{low}, \mathit{high})$
    /* **input**
        $\mathcal{P}$, $\phi$ as before
        $T$, Set of faulty topologies
        $\mathit{low}, \mathit{high} \in \mathbb{N}$, Upper and lower bound
    Topology minimization using binary search */
    if $\mathit{high} > \mathit{low}$ then
        $\mathit{middle} = \lfloor(\mathit{low} + \mathit{high})/2\rfloor$
        $U = \emptyset$
        for all $t \in T$ do
            for all $r \in \textsc{sub[nodes/edges]}(t, \mathit{middle})$ do
                if $(\mathcal{P}||r) \not\models \phi$ then
                    $U = U \cup \{r\}$
                end if
            end for
        end for
        if $U \neq \emptyset$ then
            return $\textsc{minimize[nodes/edges]}(\mathcal{P}, \phi, U, \mathit{low}, \mathit{middle} - \mathit{1})$
        else
            return $\textsc{minimize[nodes/edges]}(\mathcal{P}, \phi, T, \mathit{middle} + \mathit{1}, \mathit{high})$
        end if
    else
        return $T$
    end if
end procedure
```

Algorithm 2 Fault isolation using a delta debugging strategy

```
1: procedure ISOLATE($\mathcal{P}, \phi, f_{in}$)
2:     /* input
3:         $\mathcal{P}$, Protocol model
4:         $\phi$, Protocol properties
5:         $f_{in}$, Faulty topology
6:     output
7:         $f_-$, Smallest failing topology
8:         $f_+$, Largest passing topology */
9:     $f_- = f_{in}, f_+ = (\{0\}, \emptyset)$ // Note that $f_+ \subseteq f_-$
10:    // Loop until one-edge difference between $f_-, f_+$
11:    while SIZE($f_- - f_+$) $> 1$ do
12:        $\delta = E_{f_-} \setminus E_{f_+}$
13:        $f_{new} = f_- - \delta', \delta' \subseteq \delta,$ s.t. $f_{new}$ is connected
14:        if $(\mathcal{P} || f_{new}) \not\models \phi$ then
15:            $f_- = f_{new}$
16:        else
17:            $f_+ = f_{new}$
18:        end if
19:    end while
20:    return $f_-, f_+$
21: end procedure
```

3.3 Discussion

The minimization and isolation algorithm are different than the original delta debugging formulation as graphs as relational data have a different structure than execution traces: The difference for the minimization algorithm is that instead of partitioning as described in delta debugging, we check all subgraphs of a given size (w. r. t. nodes and edges). Further research is needed to investigate different partitioning/bisection strategies, in particular how to handle the cut set of the partitioning, and compare them with the subgraph-based approach proposed in this work. Similarly, since we need to build a complement graph for the isolation algorithm, it does not matter whether we grow from the passing graph or decrease the failing graph. As such in our formulation we only remove from the failing graph yet still approach the isolating edge from both sides.

Please note that minimized topologies also include isolation information. In a minimal topology removing *any* edge will remove the fault. Since the minimization algorithm necessitates more model checking runs evaluations than isolation, there is a tradeoff between execution time and quality of results. Finally, we need to consider that both algorithms are heuristics. That means if we have multiple faults in a protocol our algorithms potentially misses some of them. Since faults are typically gradually fixed, this is not an issue. Additionally, we show in the case studies in Sec. 4.5 that the algorithm can find the causes of multiple faults.

4 Case studies

To demonstrate our methodology, we considered two protocols for wireless sensor networks, namely, Trickle [21] and LMAC [28]. In this section, we briefly describe each of the case studies and present the anti-patterns detected using our approach.

4.1 Experimental setup

We base our experiments on a set of randomly generated undirected graphs with a dedicated sink node. We generate these graphs using the algorithm described in Rodionov et al. [26]. Our protocols features a notion of sink (a node from which the updates originate, see below). We run PRISM 4.0.1. All algorithms and graph operations are performed using Python 2.7 and NetworkX 1.6[7]. We automatically generate PRISM models for a fixed topology of the network. Our script takes a topology description as input, and generates the concrete PRISM model.

4.2 PRISM

We modeled both protocols using the probabilistic model checker PRISM [13]. The model checker automatically computes precise quantitative results based on an exhaustive analysis of a formal model. We specified the protocols in PRISM's state-based input language as discrete-time Markov chains (DTMCs), since they exhibit probabilistic behavior. In PRISM, a system consists of a set of communicating modules, each with its local variables of the integer type. The evolution of each module is described by a set of guarded commands of the form: $\texttt{[a] c} \rightarrow \texttt{p}_1\texttt{:e}_1 \texttt{ + ... + } \texttt{p}_n\texttt{:e}_n\texttt{;}$. Such a transition consists of the predicate `c` on the state variable, also called a *guard*, the action label `a`, and a probabilistic update relation $\texttt{p}_i\texttt{:e}_i$. If `c` evaluates to true, then update $\texttt{e}_i$ is applied with the probability $\texttt{p}_i$. Modules can synchronize either on global shared variables or on common actions labels. Note that PRISM implements CSP-style synchronization over an action label `a`: it requires the participation of all modules with the common action label `a` simultaneously.

Once the system is specified in terms of its modules, PRISM constructs a stochastic transition system for the composition of specified modules. Analysis is performed through model checking such systems against properties specified in the probabilistic temporal logic PCTL (for the DTMC model).

4.3 Verifying Trickle

Description: Trickle is a probabilistic code dissemination (maintenance) protocol. The goal of the protocol is to update all nodes with new versions of a deployed software. The software update is first published at a sink (also called base or root) node and is propagated among the involved nodes using a "polite gossiping" approach.

In a nutshell, the protocol works as follows:

[7] http://networkx.lanl.gov/

– Each node that hears about a new update, pulls the update from the source and schedules an announcement to inform the new update to its neighbors.
– If prior to the announcement, the node hears at least w neighbors announcing the update, it cancels its own announcement. (We call w the *broadcast parameter.*)
– If a node hears a neighbor announcing an older update (than its local version), it schedules an announcement of its own (newer) update as above.

If the network is connected, all nodes executing the Trickle protocol should eventually receive the published update.

Implementation: We are interested in Trickle's control flow and thus modeled a spread of 'the most recent update' throughout the network, executing Trickle. Thus, it is sufficient to use a single bit to indicate whether a node received the update (as 1), or an older version (as 0).

Each node i is modeled as a PRISM module, maintaining local variables $\mathtt{rcv}_{ij}$ for all its neighbors j. These variables indicate whether the recent update has been received by node i from its neighbor j. Only the sink, i. e., node 0 has a constant `rcv`0 with the value 1, thereby initially publishing the update. The nodes communicate via message channels, represented by the action labels $\mathtt{msg}_i$, with node i 'broadcasting over this channel' to all its neighbors. The broadcast medium is implemented as an additional module `broadcaster`, simulating nodes that initiate a broadcast. The module chooses the broadcasting node uniformly at random. The node modules only wait for announcements and receive the update. Our model is parameterized on the broadcast parameter w, assumed to have value $w := 3$ in the following. This parameter defines an upper threshold: if a node has heard broadcasts from w neighbors, it stops broadcasting itself. The broadcaster has two types of transitions, a labelled command and a non-labelled one; the node modules have only labelled commands. Each synchronization (labelled) command in the model is guarded by the constant w. As soon as any node exceeds w, only non-synchronizing (local) transitions are enabled for this node at the broadcaster module.

In our experiments, we verify whether all nodes eventually receive the recent datum with probability 1. This can be formulated for PRISM as:

$$\mathit{filter}(\mathit{forall}, \mathcal{P} \geq 1[\mathcal{F}(\text{"}\mathit{all}\text{"})]) \tag{1}$$

where the state `all` is specified as the conjunction for n nodes: $\bigwedge_{i=0}^{n-1} \bigvee_{j} (\mathtt{rcv}_{ij} != 0)$. Simply put, `all` is the state where every node i has received the recent update $\mathtt{rcv}_{ij} != 0$ from a neighbor j (at least once).

Minimization results: For Trickle, we generate a set of 50 random topologies of order 8; in this set of topologies there are four faulty topologies. We minimize these four topologies to a single anti-pattern that is shown in Fig. 1. In this figure, and all other figures to come, we denote the sink node with a black circle and the failing node with a gray circle. We can clearly see how the broadcast parameter (set to 3), prevents the parent of the gray node to send an update. This means that while the gray node is failing, the cause is actually

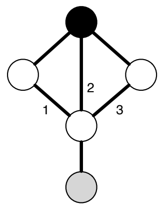

Fig. 1. Result of minimizing Trickle.

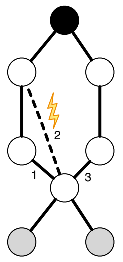

Fig. 2. Result of isolation for Trickle.

that in this topology, it has a single parent that is blocked by the broadcast parameter.

Isolation results: We select one faulty topology and perform isolation. Fig. 2 presents the results where the dashed edge indicates the edge difference between the passing and the failing topology. In this case the isolation algorithm merely removes a single edge from the initial failing topology, yet grows the passing topology to seven nodes. Although the resulting graphs feature two additional nodes compared to the minimization result in Fig. 1, the underlying fault is clearly visible in the differentiating edge 2. That is, adding a third predecessor node to the (single) parent of the gray node results in a failure.

4.4 Verifying LMAC

Description: LMAC [28] is a medium access control protocol for wireless sensor networks. LMAC is from the class of time-division multiple-access protocols: time is segmented into (time) frames and frames are split into fixed-length time slots. In each time slot a single node (in a given range) should have exclusive channel access for transmission in order to avoid collisions.

The goal of the protocol is to assign time slots to nodes in a distributed fashion. A node can then transmit its messages in the time slot it owns. For the nodes that are within range of each other, only one node owns a given time slot, so that only one node can transmit at a time. To limit the overall number of time slots, LMAC allows for reuse of time slots by the nodes at a non-interfering distance.

Nodes maintain a table of the time slot occupancy for its neighborhood. This table is synchronized with other nodes by transmitting a short control message in their time slots. In its time slot, a node broadcasts a bit array of slots chosen by its (one-hop) neighbors and itself. When a node receives such a control message from a neighbor, it stores the respective time slots of the two-hop neighborhood in its table.

LMAC is initiated at a gateway node, which is the first to select a time slot to control, and, thereafter, initiates the protocol by sending its slot occupancy table. In the bootstrap state, i. e., after a node receives its first message, it listens for an entire time frame to any messages from its neighbors. Based on the control messages, nodes will determine the time slots that are currently occupied. Since a node cannot control a time slot occupied by its one-hop and two-hop neighbors, it randomly chooses one of the remaining time slots.

A node, that already owns a time slot, executes the protocol in three steps:

– It listens for messages during the time slots other than its own one. If the node detects a collision, it stores the corresponding time slot in order to notify its neighbors. This is necessary since a node cannot detect collisions it may have caused by itself, since the radio of a sensor node cannot transmit and listen at the same time.
– During its own time slot, the node transmits a control message, which includes the time slots it knows are occupied, and the time slots where it detected collisions.
– If the node is notified about a collision that occurred in its time slot, it chooses a number of time frames to wait, and proceeds to choose another available time slot as described above.

Eventually all nodes should be able to transmit messages in their time slots, without interfering with each others transmissions.

Implementation: We verify the time slot distribution procedure of LMAC. Each node i is modeled as a PRISM module, which maintains several local variables: the selected time slot $\texttt{own\_ts}_i$, $\texttt{state}_i$ indicating the state node i is in, and a time slot with detected collision $\texttt{col\_ts}_i$. In addition, every node i maintains an array slot $\texttt{ts}_{ij}$ to record the occupancy of all time slots j.

According to the LMAC protocol and the model in [28], nodes are assumed to be globally time synchronized. Thus, we model a global clock as a separate module `timer` with the current time slot number `timeslot` and the current time frame `timeframe` as variables. Three types of transitions of `timer` enable time progress in the model: *(i)* the non-synchronizing transition is enabled if a current time slot `timeslot` is not controlled by any node in the network, *(ii)* nodes transmit and receive control messages using the labelled transition `ts`, *(iii)* synchronization on the labelled transition `decide` allows nodes to select a time slot to control. A node decides uniformly at random on a new time slot to control (if more than one are unoccupied), and on the number of time frames to back-off.

Our PRISM model is parameterized by `t`, the number of time slots in each time frame. Note that there need to be sufficient time slots in a time frame for a slot allocation to be feasible, e. g., at least n time slots for cliques of size n.

Similar to Fehnker et al. [28], we introduced two rules in our model that were underspecified in the informal description of LMAC:

– a node may not select a new time slot, if it did not received a control message from at least one of its neighbors;
– upon sending a control message to the neighbors, a node resets all array entries $\texttt{ts}_{ij}$, except for its own controlled time slot. Thereby it propagates only time slot information received in the recent time frame.

In our experiments with LMAC, we verify whether any two nodes i and j that are one or two-hop neighbors will eventually proceed to choose a new time slot if they experienced a collision. This property is expressed as follows:

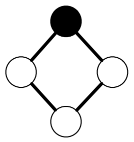
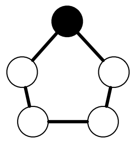

Fig. 3. Result of minimizing LMAC.

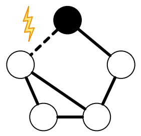

Fig. 4. Result of isolation for LMAC.

$$\begin{aligned}&\mathit{filter}(\mathit{forall},((\mathtt{own\_ts}_i = \mathtt{own\_ts}_j)\&(\mathtt{state}_i = 2)\&(\mathtt{state}_j = 2))\\ &\quad => \mathcal{P} \geq 1[\mathcal{F}\,((\mathtt{state}_i = 0)|(\mathtt{state}_j = 0))]) \end{aligned} \tag{2}$$

where $\mathtt{state}_i = 2$ denotes that the node i is broadcasting in the current time slot. $\mathtt{state}_i = 0$ means that the node i is recording occupancy of the time slots from control messages of its neighbors in order to select a new time slot.

Minimization results: We run minimization on a (sub-)set of six node networks; in particular we chose three topologies that can be scheduled using merely four slots ($\mathtt{t}$:= 4). The minimization algorithm returns two topologies – a four node ring and a five node ring as shown in Fig. 3. This is the optimal (minimal) result: Rings of more than four nodes cause the LMAC property to fail. Note that for LMAC this is not because of some parameterization of the protocol or a specific issue with the neighborhood but an emergent detrimental property of the protocol. If two neighboring nodes that have no common neighbors end up in a collision, this collision cannot be detected by other nodes and hence cannot be resolved.

Isolation results: We perform isolation on one of the topologies that we used for the minimization algorithm. [8] The result is depicted in Fig. 4. We can see here that the closing of this ring of four causes the fault. Note that we see here one additional detail – the ring of three in the lower left is not a problem; yet a ring of four in the upper right is.

4.5 Handling multiple faults

As a final test in our case study we look at the Trickle protocol and additionally inject an error for topologies that have a path from the sink in the graph of at least 5 hops in order to represent a second fault that is depth or forwarding related. Our minimization algorithm returns two topologies. One is related to the trickle fault; it is exactly the same as the fault described in Sec. 4.3. The other topology is a graph consisting of a chain of 6 nodes. As we can see the results clearly indicate the two different types of faults that we inject. In contrast, when we select a single topology and perform isolation we only investigate one underlying cause. In this case, we consciously select a topology that is due to both faults. As we can see in Fig. 5 isolation returns one of the two faults. In this case we find (the simpler) depth-related inject that gets triggered by the edge that increases the distance of the gray node to 5 hops.

[8] The faulty topology corresponds to number 29 in [28].

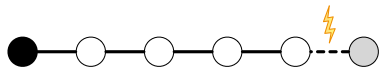

Fig. 5. Isolation of multiple faults.

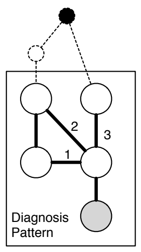

Fig. 6. Anti-patterns detected using diagnosis.

4.6 Comparison to diagnosis

In order to compare our results with diagnosis, we ran spectrum-based diagnosis as described in Sec. 2 for the Trickle testcase using subgraphs of order 4 and 5. We assume here a certain size of error which is reasonable, as we would start debugging from small-size topologies. The highest ranked subgraph, which is shown in the box in Fig. 6, is of order 5. Note that in this figure, we arrange the graph to make the pattern contained in the box more visible. In particular, we add a potential embedding into a larger graph outside the box just for clarification; however, this embedding is not part of the diagnosis result. In the pattern, the nodes on the top are not connected to a sink node. As we can see, such a pattern does not necessarily demonstrate the core cause. Hence, these results are not as helpful for debugging purposes as the patterns generated using the anti-pattern approach, which clearly identifies the source of the problem.

5 Conclusions

In this paper, we presented an approach, inspired by delta debugging, to find the root causes of failure in wireless network protocols. The causes are represented in terms of minimal topologies, called anti-patterns. We also developed another approach inspired by fault diagnosis and showed that the approach based on delta debugging is more effective in demonstrating the root causes of failure.

Although anti-patterns explain the faults or implicit assumptions of a protocol, their presence in a larger network does not necessarily lead to a failure. We plan to extend our notion of anti-pattern to capture the boundary conditions on the nodes, which also capture when these faults do lead to a failure in larger topologies. We think that the isolation technique does provide additional information that can be used in characterizing the boundary conditions under which the fault will necessarily into failure.

Acknowledgements: Matthias Woehrle is supported by the Dutch Technology Foundation STW and the Technology Programme of the Ministry of Economic Affairs, Agriculture and Innovation. The authors thank Koen Langendoen, Tim Willemse, Wan Fokkink and the anonymous reviewers for their valuable feedback.

References

1. R. Abreu, P. Zoeteweij, R. Golsteijn, and A. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.
2. R. Bakhshi, J. Endrullis, S. Endrullis, W. Fokkink, and B. Haverkort. Automating the mean-field method for large dynamic gossip networks. In *Proc. Conf. on Quantitative Evaluation of SysTems (QEST)*, pages 241–250. IEEE Computer Society, 2010.
3. G. Behrmann, A. David, and K. G. Larsen. A tutorial on uppaal. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.
4. J. Beutel, S. Gruber, A. Hasler, R. Lim, A. Meier, C. Plessl, I. Talzi, L. Thiele, C. Tschudin, M. Woehrle, and M. Yuecel. PermaDAQ: A scientific instrument for precision sensing and data recovery in environmental extremes. In *Proc. 8th ACM/IEEE Int'l Conf. on Information Processing in Sensor Networks (IPSN 2009)*, pages 265–276, San Francisco, CA, USA, April 2009. ACM/IEEE.
5. M. Cadilhac, T. Hérault, R. Lassaigne, S. Peyronnet, and S. Tixeuil. Evaluating complex MAC protocols for sensor networks with APMC. In *Proc. Workshop on Automated Verification of Critical Syst (AVoCS'06)*, volume 185 of *ENTCS*, pages 33–46. Elsevier, 2007.
6. S. Chaki, A. Groce, and O. Strichman. Explaining abstract counterexamples. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, SIGSOFT '04/FSE-12, pages 73–82, New York, NY, USA, 2004. ACM.
7. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.
8. E. M. Clarke and H. Veith. Counterexamples revisited: Principles, algorithms, applications. In N. Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 41–43. Springer, 2004.
9. E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1/2):105–131, 1996.
10. A. Fehnker, L. van Hoesel, and A. Mader. Modelling and verification of the LMAC protocol for wireless sensor networks. In *Proc. Conf. on Integrated Formal Methods (IFM)*, volume 4591 of *LNCS*, pages 253–272. Springer, 2007.
11. P. Godefroid. Using partial orders to improve automatic verification methods. In E. Clarke and R. Kurshan, editors, *Proceedings of 2nd International Workshop Computer Aided Verification (CAV'90)*, volume 531 of *LNCS*, pages 176–185. Springer, 1990.
12. A. Groce and W. Visser. What went wrong: explaining counterexamples. In *Proceedings of the 10th international conference on Model checking software*, SPIN'03, pages 121–136, Berlin, Heidelberg, 2003. Springer-Verlag.
13. A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In *Proc. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3920 of *LNCS*, pages 441–444. Springer, 2006.
14. C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, 1996.

15. M. Kamali, L. Laibinis, L. Petre, and K. Sere. Self-recovering sensor-actor networks. In M. R. Mousavi and G. Salaün, editors, *FOCLASA*, volume 30 of *EPTCS*, pages 47–61, 2010.
16. J.-P. Katoen, T. Kemna, I. Zapreev, and D. Jansen. Bisimulation minimisation mostly speeds up probabilistic model checking. In *Proc. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *LNCS*, pages 76—92. Springer, 2007.
17. J.-P. Katoen, D. Klink, M. Leucker, and V. Wolf. Three-valued abstraction for continuous-time Markov chains. In *Proc. Conf. on Computer Aided Verification (CAV)*, volume 4590 of *LNCS*, pages 311–324. Springer, 2007.
18. M. M. H. Khan, H. K. Le, H. Ahmadi, T. F. Abdelzaher, and J. Han. Dustminer: troubleshooting interactive complexity bugs in sensor networks. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 99–112, New York, NY, USA, 2008. ACM.
19. M. Kwiatkowska, G. Norman, and D. Parker. Game-based abstraction for markov decision processes. In *Proc. Conf. on Quantitative Evaluation of SysTems (QEST)*, pages 157–166. IEEE Computer Society, 2006.
20. S. Laplante, R. Lassaigne, F. Magniez, S. Peyronnet, and M. de Rougemont. Probabilistic abstraction for model checking: An approach based on property testing. In *Proc. IEEE Symp. on Logic in Comput. Sci. (LICS 2002)*, pages 30–39. IEEE Computer Society, 2002.
21. P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, page 2. USENIX Association, 2004.
22. K. Lorincz, B.-r. Chen, G. W. Challen, A. R. Chowdhury, S. Patel, P. Bonato, and M. Welsh. Mercury: a wearable sensor network platform for high-fidelity motion analysis. In *Proc. ACM Conf. on Embedded Networked Sensor Systems*, SenSys '09, pages 183–196, New York, NY, USA, 2009. ACM.
23. M. Maifi, H. Khan, T. Abdelzaher, and K. K. Gupta. Towards diagnostic simulation in sensor networks. In *Distributed Computing in Sensor Systems*, pages 252–265. Springer, 2008.
24. L. Mottola, T. Voigt, F. Österlind, J. Eriksson, L. Baresi, and C. Ghezzi. Anquiro: enabling efficient static verification of sensor network software. In *Proc. ICSE Workshop on Software Engineering for Sensor Network Applications (SESENA)*, pages 32–37, New York, NY, USA, 2010. ACM.
25. D. Peled. All from one, one for all: on model checking using representatives. In C. Courcoubetis, editor, *Proceedings of 5th International Conference Computer Aided Verification (CAV'93)*, volume 697 of *LNCS*, pages 409–423. Springer, 1993.
26. A. S. Rodionov and H. Choo. On generating random network structures: trees. In *Proceedings of the 2003 international conference on Computational science: PartII*, ICCS'03, pages 879–887, Berlin, Heidelberg, 2003. Springer-Verlag.
27. A. Valmari. A stubborn attack on state explosion. In E. Clarke and R. Kurshan, editors, *Proceedings of 2nd International Workshop Computer Aided Verification (CAV'90)*, volume 531 of *LNCS*, pages 156–165. Springer, 1990.
28. L. van Hoesel and P. Havinga. A lightweight medium access protocol (LMAC) for wireless sensor networks: Reducing preamble transmissions and transceiver state switches. In *Proc. Workshop on Networked Sensing Systems (INSS)*, pages 205–208. Society of Instrument and Control Engineers (SICE), 2004.
29. A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28:183–200, February 2002.