Formal Verification of Synchronisation, Gossip and Environmental Effects for Critical IoT Systems

Matt Webster[1], Michael Breza[2],
Clare Dixon[1], Michael Fisher[1], and Julie McCann[2]

[1] Department of Computer Science, University of Liverpool, Liverpool, L69 3BX, UK
{matt,cldixon,mfisher}@liverpool.ac.uk
https://www.liverpool.ac.uk/computer-science/
[2] Department of Computing, Imperial College London, London, SW7 2AZ, UK
{mjb04,j.mccann}@doc.ic.ac.uk
https://www.imperial.ac.uk/computing

Abstract. The Internet of Things (IoT) promises a revolution in the monitoring and control of a wide range of applications, from urban water supply networks and precision agriculture food production, to vehicle connectivity and healthcare monitoring. For applications in such critical areas, control software and protocols for IoT systems must be verified to be both robust and reliable. Two of the largest obstacles to robustness and reliability in IoT systems are effects on the hardware caused by environmental conditions, and the choice of parameters used by the protocol. In this paper we use probabilistic model checking to verify that a synchronisation and dissemination protocol for Wireless Sensor Networks (WSNs) is correct with respect to its requirements, and is not adversely affected by the environment. We show how the protocol can be converted into a logical model and then analysed using the probabilistic model-checker, Prism. Using this approach we prove under which circumstances the protocol is guaranteed to synchronise all nodes and disseminate new information to all nodes. We also examine the bounds on synchronisation as the environment changes the performance of the hardware clock, and investigate the scalability constraints of this approach.

Keywords: Internet of Things · Critical Systems · Formal Verification.

1 Introduction

In this paper we use formal verification, via the logical method of *probabilistic model-checking* [17], to analyse and verify critical communication protocols used for the Internet of Things (IoT) [46]. IoT systems often involve networks of small, resource-constrained, computer devices embedded in an environment. These devices have low-power sensors, radios for communication, and can potentially control motors and other devices to perform actuation to change their

environment. A common class of IoT systems, called Wireless Sensor Networks (WSN), enable the monitoring and control of critical infrastructures made up of large, complex systems such as precision agriculture or smart water networks. Such systems require control software that can synchronise the events of the nodes in the system, and disseminate parameters and code updates. WSN and IoT deployments are increasingly mobile, allowing for wider applications and new challenges in their design and deployment [37,36].

A key problem with the development of critical IoT systems is *ensuring* that they will function correctly, or at least, fail in a way that is non-destructive to the systems that they monitor and control. The use of probabilistic models is crucial because it allows us to quantitatively analyse the system with the dynamical effects caused by the environment — one of the most significant causes of failure for WSNs [29]. WSNs deployed on critical infrastructure suffer from the effects of cyber-physical interactions in a way not seen with office or domestic computing. Environmental conditions such as precipitation or changes in temperature will affect the performance of the sensor nodes, and can degrade the WSN potentially causing node failure. The control software that provides event synchronisation and controls message dissemination needs to be programmed to be reliable in the light of these potential problems. Errors here can make the infrastructure itself inefficient at best, or even unstable and failing in the worst case.

1.1 Formal Verification and Probabilistic Model Checking in PRISM

Formal methods are a family of techniques used to verify software and hardware, typically using mathematical, proof-based approaches [17]. These include techniques such as automated theorem proving [18], in which full mathematical proof is carried out, and model checking [3], in which every state of a model (also known as the model's *state space*) can be examined exhaustively. Formal methods allow for *formal verification*, where models of software and hardware systems can be proved to satisfy certain requirements. These requirements are typically provided using a precise formal logical language such as temporal logic [17]. In this paper we use *probabilistic* model-checking [28], a variant of traditional model-checking that allows for probabilities to be incorporated into a model, and for quantitative analyses to be carried out on such models.

The probabilistic model checker, PRISM [28,39], consists of two parts: a modeling language, and a model checker. The PRISM modeling language can be used to specify the behaviour of a *probabilistic* finite state automaton (P-FSA), which can then be formally verified via the model checker. For example, we can model a simple sensor node in the PRISM version 4.4 modelling language as follows:

```
module sensorNode
  state: [0..1] init 0;
  [] state=0 -> 0.99: (state'=0) + 0.01: (state'=1);
  [] state=1 -> 0.99: (state'=1) + 0.01: (state'=0);
endmodule
```

This sensor node is modelled as a module in PRISM. We have one variable, 'state', which can be set to 0 or 1 (which we define as representing 'transmit' and 'idle'

respectively). Note that we define an initial state of 0 for this variable. There are two lines denoting commands. The first command says that if the state is 0, then remain in state 0 with probability 0.99 or transition to state 1 with probability 0.01. The second command is similar, but with 0 and 1 reversed. In general, commands take the form

$$[\mathsf{s}]\ \mathsf{guard}\ \texttt{->}\ \mathsf{p}_1 : \mathsf{u}_1 + \ldots + \mathsf{p}_n : \mathsf{u}_n;$$

where p_i are probabilities and u_i are lists of variable updates. In the case where only one list of updates is made with probability 1.0, a simpler form is used (e.g., [s] guard -> u;). The letter s denotes an optional synchronisation. Synchronised commands execute simultaneously with synchronisation commands from other modules that share the same label, and can be used for inter-module communication. Another way for modules to communicate is via the use of *local variables*, which can be read by all modules, as well as *global variables* which can be read by, and written to, all modules.

Multiple modules can be specified within a PRISM model. Models are executed by selecting non-deterministically a command (from any module) whose guard evaluates to true. If there are no commands whose guards are true, then the model has reached a fixed point and will stop executing.

Once a formal model has been developed in the PRISM language, it can be formally verified, with respect to some requirement, using the PRISM model checker. PRISM requirements can be formalised as *properties* using logical languages such as probabilistic computation tree logic (PCTL*) and probabilistic linear temporal logic (P–LTL) [3]. Different logics allow different kinds of properties to be specified. In this paper we will use PCTL* to specify properties.

PCTL* is based on a discrete formulation of time as a tree-like structure, starting from a particular point and extending into the future. The following are well-formed PCTL* formulae: 'p', meaning that p is true; '$\neg p$', meaning that p is false; '$p \Longrightarrow q$', meaning if p is true then q is true; $p \wedge q$', meaning that both p and q are true; '$\mathsf{F}\ p$', meaning p is true now or at some point in the future; and '$\mathsf{G}\ p$', meaning p is true now and at every point in the future. PRISM also allows the use of standard numerical operators such as $=$, $\geq$ and $\leq$.

Formal verification works by analysing the entire state space of a model in order to determine whether a particular property holds. For example, for the sensor node model above, we can use PCTL* to specify the probability that sensor node is eventually in the 'idle' state:

$$P^{=?}[\mathsf{F}\ (\mathsf{state} = 1)]$$

We can then use PRISM model checker to determine that this probability is 1.0:

$$P^{=?}[\mathsf{F}\ (\mathsf{state} = 1)] = 1.0$$

More complex properties can be formed, e.g., the following property says that the probability that the model will always be in the 'idle' state eventually is 1.0:

$$P^{=?}[\mathsf{G}\ \mathsf{F}\ (\mathsf{state} = 1)] = 1.0$$

This kind of property is said to specify the probability that the model is in the 'idle' state *infintely often*.

2 Related Work

Formal methods have been used previously for design and analysis of WSN and IoT. For example, Chen et al. [12] provide a survey of a number of approaches to formal verification of routing protocols for WSNs. Kim et al. [26] conduct a formal security analysis of an authorization toolkit for the Internet of Things using the Alloy verification tool. Mouradian & Augé–Blum [35] describe the formal verification of real-time WSN protocols using the UPPAAL model checker. Tobarra et al. [42] use the Avispa model checking tool to formally verify a security protocol for WSNs. Usman et al. [44] demonstrate formal verification of mobile agent-based anomaly detection for WSNs using the Symbolic Analysis Laboratory model checking tool. Dong et al. [15] use a formal specification language for sensor networks and perform formal verification using SAT-solvers. However, none of these approaches uses a probabilistic model checker, as is the case in this paper, to determine the probability of success or failure for a particular requirement. Fruth [19] used PRISM to analyse contention resolution and slot allocation protocols for WSNs, but not synchronisation or dissemination protocols. Synchronization [21,20,40] and gossip protocols [4,16,27,22,25,14] have been formally verified but not together, and not accounting for environmental effects.

Mohsin et al. [34] used PRISM to formally assess security risks in IoT systems, but not risks due to the environment. Modelling of embedded systems and the environment have been explored by Baresi et al. [5], who used a UML-based MADES approach to model a system, and by Basile et al. [6], who performed statistical model checking of an energy-saving cyber-physical system using UPPAAL SMC. These approaches can find when constraints are not met, but do not perform an exhaustive search of the entire state space, as is the case here. Boano et al. explored the effects of temperature on CPU processing time and transceiver performance though TempLab, a WSN test-bed which allows for the manipulation of the temperature of each individual sensor node [7]. Lenzen et al. [31] studied the effect of temperature on the hardware clocks chips used as timers on many common WSN sensor node platforms.

3 Modelling a WSN Protocol in PRISM

It is possible to model various WSN protocols in PRISM. In order to illustrate the approach, we create a model of a non-trivial decentralised WSN management protocol known as FiGo (an abbreviation of Firefly–Gossip) [11]. FiGo enables synchronisation of different sensors' clocks in order to unify the measurement of time across the network based on firefly-like synchronisation. FiGo also enables consensus on key information between nodes via *gossiping*, in which nodes pass on new information to their neighbors. FiGo was chosen because it is a simple

protocol that contains elements, such as epidemic propagation [23], found in more complex protocols like Trickle [32] and RPL [8].

3.1 The Firefly-Gossip (FiGo) Protocol

Current techniques for large-scale computer management are not suitable for WSNs due to the unreliable nature of the nodes and their networks. A potential solution is to use management protocols, such as FiGo, that scale well and are robust to the failure of individual nodes. In applications such as precision agriculture [43,38], wireless nodes need to be synchronised to be able to deliver time-correlated samples of data such as moisture levels and temperature, and to analyse the data. If the analysis shows a problem, control messages need to be sent to nodes with actuators, e.g., to increase irrigation in a drought, or decrease it if a particular disease is discovered.

Synchronisation of WSNs is essential in many applications, for example in adaptive sensing for smart water networks [24]. WSNs allow urban water providers to monitor the water flow to match customer demand. Synchronisation enables the sensor nodes to measure, communicate and aggregate the flow rates and water pressure data. A control algorithm on the actuator nodes can open or close valves to stabilise water flow for the network, or re-route water in the case of a major leak. Importantly, the control software can also disseminate new control algorithms or critical security updates to all the sensing and actuation nodes via gossiping.

FiGo is typical of a class of algorithms that combine firefly synchronisation [45] and gossip protocols [23] into a single epidemic process [11]. This mixture of synchronisation and dissemination processes is used to bring the internal states of WSN nodes to a stable, global equilibrium where all nodes are synchronised with respect to both time and metadata. Experiments have shown such protocols to be both scalable and resilient to individual node failure [11,9,10]. A typical FiGo algorithm is shown in Figure 1.

FiGo algorithms have been deployed for the synchronisation and management of several WSN deployments run by the Adaptive Emergent Systems Engineering group at Imperial College[3]. For example, they were used to organise pollution sensors for an experiment with mobile data mules as part of an Imperial College Grand Challenge project, and to synchronise and control the sampling rate for a rainfall monitoring sensor network as part of a floodplain monitoring project done in collaboration with the Imperial College Department of Civil Engineering. They are currently undergoing evaluation for deployment across the Liverpool Sensor City IoT-X/LoRaWAN network[4].

3.2 A Prism Model of FiGo

A Prism model of FiGo was developed precisely capturing the control flow of the algorithm in Figure 1. The algorithm begins with a number of variable

[3] http://wp.doc.ic.ac.uk/aese/
[4] http://www.sensorcity.co.uk/

```
1  clock := 0
2  cycleLength := 100
3   refractoryPeriod  := cycleLength/2
4  dutyCycle := cycleLength
5  nextBroadcast := random(0, cycle  length)
6  metadata := 0
7  same count := 0
8  while(true)
9      if  ((clock  = nextBroadcast) or                                  |
10           (clock  = (nextBroadcast + refractoryPeriod)               |
11           mod cycleLength)) and                                      | transmit
12           sameCount < sameThreshold then                             |
13         transmit()                                                   |
14         sameCount := 0                                               |
15     else  if  clock  <= cycleLength then                             |  clockCheck
16         if  message.overheard()  then                                |  listen
17             if  clock  >= refractoryPeriod  then                     |
18                 clock  := CAvg(s1LocalClock,s2LocalClock)            |  sync1
19             end if                                                   |
20             if  message.metadata() > metadata then                   |
21                 metadata := message.metadata()                       |  sync2
22             else  if  message.metadata() < metadata and              |
23                 sameCount < 1 then                                   |  sync3
24                 transmit()                                           |
25             else  if  metadata = message.metadata() and              |
26                 message.time() = clock then                          |  sync4
27                 sameCount := sameCount + 1                           |
28             end if                                                   |
29         end if                                                       |
30     end if                                                           |
31     if  clock  = cycleLength                                         |
32         clock  := 0                                                  |
33         sameCount := 0                                               |  updateClock
34     else                                                             |
35         clock  := clock  + 1                                         |
36     end if                                                           |
37 end while
```

Fig. 1. Phases of the FiGo Gossip–Synchronisation Algorithm.

assignments which are directly translated into variable assignments in PRISM. Some of the variables are not updated at all in the model, so these are set as global constants in PRISM, e.g.:

```
const int cycleLength = 100;
const int refractoryPeriod = floor(cycleLength/2);
```

The main loop of the algorithm is then divided into a number of phases. For example, the transmit phase corresponds to the if-statement in lines 9 to 14. The next if-statement consists of a number of nested if-statements called clockCheck, listen, sync1, sync2, and so on. The final phase corresponds to the final if-statement in the main loop and is called updateClock. These phases are defined as global constants and are used as the values of a local variable s1Phase which contains the currently-executing phase:

```
s1Phase : [0..7] init transmit;
```

Note that s1Phase refers to the phase of the first sensor node module, which is called s1. The phases of other sensors (sensors s2, s3, etc.) are called s2Phase, s3Phase, etc.

When one phase has finished executing the next phase is chosen according to the control flow of the algorithm in Figure 1. For example, during the sync1 phase in lines 17 to 19 the algorithm checks whether the clock is outside a "refractory period" set to half of the cycle length. If it is, then the sensor updates its clock to the average of its own clock and the clock of the other sensor. The "circular average" is used, in which the average of 90 and 10 are 0 (with respect to the clock cycle of length 100), rather than 50. The circular average ensures that the update to the clock variable moves it closer to the clock of the other sensor.

In the PRISM model, this behaviour is shown in the following three commands:

```
[] s1Phase=sync1 & s1LocalClock>=refractoryPeriod & diff<=floor(cycleLength/2)
  -> (s1LocalClock'=s1avg1) & (s1Phase'=sync2);
[] s1Phase=sync1 & s1LocalClock>=refractoryPeriod & diff>floor(cycleLength/2)
  -> (s1LocalClock'=s1avg2) & (s1Phase'=sync2);
[] s1Phase=sync1 & !( s1LocalClock>=refractoryPeriod ) -> (s1Phase'=sync2);
```

The first two commands say that if the sensor is in the sync1 phase and the clock is greater than or equal to refractoryPeriod, then set s1's clock to the circular average of s1's clock and s2's clock. The third command says that if these conditions are not set, then proceed to the next phase of the algorithm, sync2.

The sensor which we have modelled here is called s1. To model communication between sensor nodes we need at least one more sensor in the model, s2. The sensor s2 is exactly the same as s1, except all references to "s1" in the code are modified to "s2." Communication in the model is achieved asynchronously through the use of inboxes: when a sensor sends a message to another sensor it does so by leaving the message in an inbox, which can then be read by the receiving sensor when it is ready to do so.

The resulting combined model is around 140 lines of code long including variable declarations, and can be found in the online repository[5]. This PRISM model is an almost direct translation from the pseudocode to PRISM and has not been optimised for formal verification.

4 Formal Verification of FiGo Using PRISM

We build a formal model in PRISM, in a manner analogous to compiling a program: the source code, in this case the PRISM model, is automatically converted into a mathematical model, essentially a finite state structure. During this construction, PRISM calculates the set of states reachable from the initial state and the transition matrix which represents a probabilistic finite state automaton. Building revealed that the full model consisted of 4,680,914 reachable states,

[5] http://livrepository.liverpool.ac.uk/3021710/

with 9,361,828 transitions between those states, and took 21 minutes on an Intel Core i7-3720QM CPU @ 2.60GHz laptop, with 16 GB of memory, running Ubuntu Linux 16.04. As we shall see in Section 4.1, it was possible to reduce the size of this model significantly.

One of the key features of PRISM is that it can find the probability of a particular property holding through some path through a computation tree. For example, we can create a property to determine the probability that eventually the two sensors are synchronised:

$$P^{=?}[\mathsf{F}\ (\mathsf{s1Clock} = \mathsf{s2Clock})] = 1.0 \qquad [23.8\text{s}] \tag{1}$$

In this case the probability is 1.0, meaning that on *all* paths through the model the clocks will eventually synchronise. (The time taken for model checking was 23.8 seconds.) That is not to say that they remain synchronised, or that they become synchronised again once they are no longer synchronised. If we wish to verify the latter, that synchronisation happens repeatedly, then we can create a probability with a different formula:

$$P^{=?}[\mathsf{G}\ \mathsf{F}\ \mathsf{s1Clock} = \mathsf{s2Clock}] = 1.0 \qquad [100\text{s}] \tag{2}$$

This probability, in which synchronisation occurs infinitely often, is 1.0. We can strengthen the property further: we can determine the probability that, once the clocks are synchronised, they remain synchronised:

$$P^{=?}[\mathsf{F}\ \mathsf{G}\ \mathsf{s1Clock} = \mathsf{s2Clock}] = 0.0 \qquad [75.6\text{s}] \tag{3}$$

In this case the probability of this property being true is 0.0, meaning that it is *never* the case that the two clocks synchronise and then remain synchronised forever. The reason this is so can be seen by examining a simulation, or trace, of the model. (A simulation is a sample path or execution of the model [39].) Below is a simulation of the model showing how de-synchronisation occurs after synchronisation:

action	s1Phase	s1Clock	s2Phase	s2Clock
s1	updateClock	4	updateClock	4
s2	updateClock	4	transmit	5
s2	transmit	5	transmit	5

The table shows the values of certain state variables during an execution of the model. The leftmost column, 'action", shows which module, s1 or s2, is currently executing. In the first state, both clocks have the value "4" and are synchronised. However, a transition occurs in which one of the sensors, in this case, s2, increments its clock value resulting in de-synchronisation. However, in the next state we can see that the sensor s1 updates its clock as well, resulting in synchronisation.

We might postulate that once synchronisation occurs, then de-synchronisation will occur at some point. This can be encoded as the following property:

$$P^{=?}\left[\mathsf{G}\ \left(\begin{array}{l}\mathsf{s1Clock} = \mathsf{s2Clock} \implies \\ \mathsf{F}\ \neg(\mathsf{s1Clock} = \mathsf{s2Clock})\end{array}\right)\right] = 1.0 \qquad [123\text{s}] \tag{4}$$

We can also verify whether once de-synchronisation has happened, that synchronisation will eventually happen:

$$P^{=?}\left[\mathsf{G}\left(\begin{array}{c}\neg(\mathsf{s1Clock}=\mathsf{s2Clock})\implies\\ \mathsf{F}\ \mathsf{s1Clock}=\mathsf{s2Clock}\end{array}\right)\right]=1.0\qquad[175\mathrm{s}]\tag{5}$$

Property 1 tells us that synchronisation will occur at some point during the execution of the model and Property 2 tells us that synchronisation will occur infinitely often. Properties 4 and 5 tell us even more: that periods of synchronisation are separated by periods of de-synchronisation, and vice versa.

4.1 Increasing the Model's Accuracy

Examining simulations using PRISM reveals that clocks will rapidly de-synchronise after synchronisation, as we saw in the previous section. This is a result of the way clocks were handled in this model: we allowed for clocks to tick at any rate. Therefore it is possible for clocks to tick unevenly, as in this case. In fact, it is possible for one clock to tick indefinitely without the other clock ticking. This assumption of the model can be seen to correlate with a real-world sensor system in which clocks are unreliable and may vary widely in comparative speeds.

The FiGo sensor network we are modelling is based on the 'MICAz' sensor mote developed by Memsic Inc. [33] The network is homogeneous across nodes, meaning that the same hardware and software is present on each node. This includes the microcontroller, in this case the 'ATmega128L' developed by Atmel Corporation [2]. This microcontroller has a clock speed of 16 MHz and operates at up to 16 million instructions per second. As the network is homogeneous we can model the clock speed as constant across different nodes. In practice, and as we shall see in Section 5, clock speeds are never exactly the same. However, treating the clock speeds as constant is much closer to reality than one clock being able to tick indefinitely without the other ticking.

Clock speeds were made constant by introducing synchronisations in the updateClock phase:

```
[tick] s1Phase=updateClock & s1Clock<cycleLength
  -> (s1Clock'=s1Clock+1) & (s1Phase'=transmit);
[tick] s1Phase=updateClock & s1Clock=cycleLength
  -> (s1Clock'=0) & (s1SameCount'=0) & (s1Phase'=transmit);
```

The first command says that if the clock is less than the cycle length (equal to 99 in this model), then increment the clock, but if the clock is equal to 99, then reset the clock to zero.

These commands both use a synchronisation label, tick, and correspond to a similar set of commands in the s2 sensor module, which use the same label. The label means that one of these commands must execute at the same time as one of the corresponding commands in the s2 module. Since these commands handle clock updates, this ensures that the clocks will update synchronously, and therefore it is impossible for one clock to tick faster than the other. This models more closely the homogeneous network on which FiGo is used.

One advantage of constant clock speeds is that it reduces the total number of states of the probabilistic model. In this case the model reduced in size from 4,680,914 states with 9,361,828 transitions to 8,870 states and 13,855 transitions. The time taken for model building also decreased, from 21 minutes to 17 minutes.

Property 1 was formally verified for this revised model:

$$P^{=?}[\mathsf{F\ s1Clock} = \mathsf{s2Clock}] = 1.0 \qquad [5.4\text{s}] \tag{6}$$

In the definition of the FiGo algorithm, the variable nextBroadcast is assigned a random value between 0 and 99. During model translation, however, this random value was modified to a constant integer value. We used PRISM variables to automatically check every possible value of nextBroadcast. This is done by removing the values of the global constants that represent the next broadcast value. Then, PRISM can be used to perform automatic, and exhaustive, model-checking of a property across a range of values for these constants, by automatically building and verifying a model for each value. However, the PRISM model has a large verification time of 17 minutes. PRISM needs to build a model for each value of the two variables above, meaning that 10,000 models would need to be constructed, each taking 17 minutes. To reduce the size of the model the duty cycle length was reduced from 100 to 20. This reduces the size of the model to 1,947 states and 3,040 transitions, and takes 16.8 seconds to build. The duty cycle length can be reduced from 100 to 20 without significantly affecting the accuracy of the model, as there is still a large enough range of possible values to allow for an accurate depiction of clock synchronisation via circular averaging.

Property 2 was verified with a range of $[0, 20]$ for both variables, modelling every possible combination of the two nextBroadcast values. The results showed that the probability that synchronisation will occur infinitely often is always 1.0.

4.2 Gossip and Synchronisation

The properties examined thus far have concerned clock synchronisation. The other main function of the FiGo algorithm is to spread information across a network using a gossip protocol in which sensors tell their neighbours about new information. In the case of the FiGo algorithm, this is represented by an integer variable whose initial value is zero, but which may increase when a node is updated with a new piece of information. This captures a common function of WSNs that must share new information, roll-out software updates, etc.

In order to analyse metadata synchronisation the model was modified to allow new metadata values. This was done by creating a branching point during the updateClock phase of the algorithm:

```
[tick] s1Phase=updateClock & s1Clock=cycleLength & s1Metadata<3
  -> (1-pUpdateMetadata): (s1Clock'=0) & (s1SameCount'=0) & (s1Phase'=transmit)
   + pUpdateMetadata: (s1Clock'=0) & (s1SameCount'=0) &
     (s1Metadata'=s1Metadata+1) & (s1Phase'=transmit);
```

The metadata can take any value from 0 to 3, representing a sequence of three possible updates. This updated command allows the metadata to be incremented

at the point the duty cycle ends. This happens with probability pUpdateMetadata which is equal to 0.5, a value chosen to represent that new metadata will happen, on average, every other duty cycle. Therefore the probability that the metadata will not be updated at the end of the duty cycle is also 0.5. This functionality is included in s1, but not in s2, to model a sensor node that receives updates first. For example, this could be the sensor node located closest to an engineer who is updating node software, which will therefore receive an update first. Adding this branch point to the model introduces new states for the various values of the local metadata variables. This increased the size of the model from 1,947 states and 3,040 transitions to 4,776 states and 7,467 transitions for a model with a duty cycle of 20. It is now possible to form properties that verify the gossip part of the FiGo algorithm. For example:

$$P^{=?}[\mathsf{F}\ \mathsf{s1Metadata} = \mathsf{s2Metadata}] = 1.0 \qquad [0.041\text{s}] \tag{7}$$

This formula says that the probability that the metadata is eventually synchronised across nodes is 1.0. As is the case with software version numbers, the metadata increases but never decreases, i.e., once it reaches 3 it stays at 3. Therefore it can also be verified that at some point the metadata is synchronised (i.e., when it is equal to 3) and remains so:

$$P^{=?}[\mathsf{F}\ \mathsf{G}\ \mathsf{s1Metadata} = \mathsf{s2Metadata}] = 1.0 \qquad [2.0\text{s}] \tag{8}$$

Furthermore, we can verify that the Firefly and Gossip parts of the algorithm both work, and that eventually the two sensors will by synchronised on both time and metadata, and will remain so:

$$P^{=?}\left[\mathsf{F}\ \mathsf{G}\ \begin{pmatrix}\mathsf{s1Metadata} = \mathsf{s2Metadata}\\ \wedge\ \ \mathsf{s1Clock} = \mathsf{s2Clock}\end{pmatrix}\right] = 1.0 \qquad [1.6\text{s}] \tag{9}$$

To examine the scalability of the model, the two-sensor network was extended to three and four sensors. A complete graph topology was used, so that every node can communicate with every other node. A range of clock duty cycle lengths was examined for 2–, 3– and 4–sensor networks. The aim was to see how the total time to verify Property 2 (including build and verification time) was affected. The results are summarised in Figure 2.

The 2– and 3–sensor networks could be verified formally with a clock cycle length of up to 100 for 2-sensor networks, and 28 for 3–sensor networks. However, the 4–sensor network could not be analysed at all. The amount of time taken to verify this property increases with cycle length, and increases significantly with the number of sensors (see Figure 2). This is due to a state space explosion [13] occurring as a result of a larger number of large variables occurring in the model (e.g., the duty cycle has a range of up to 100 for each sensor). The state space also increases with cycle length due to increased non-determinism in the model: the larger the duty cycles for the clocks of each sensor, the more combinations of these clock values there are in the model.

All of the probabilities for Property 2 for the different network and duty cycle sizes were found to be 1.0, showing that synchronisation happens infinitely often

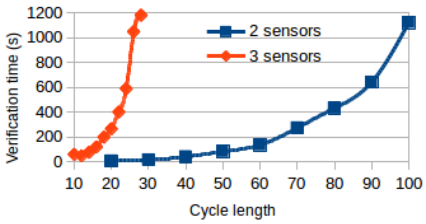

Fig. 2. Total time for formal verification of Property 2 for 2– and 3–sensor networks.

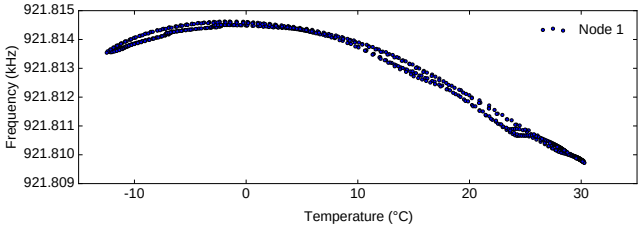

Fig. 3. Mica2 hardware clock frequency for different ambient temperatures [31].

in all the cases examined. It should be noted that these results only pertain to the model examined in this paper, and other models and protocols may permit larger sensor networks to be analysed. While state space explosion is a recurrent theme in model checking, it can be mitigated through abstraction and re-modelling to reduce the size of the state space.

5 Environmental Effects on Hardware

Microcontrollers such as the ATmega128L [2] are often set to process instructions at a particular speed, known as the clock speed. (Here, the clock speed refers to the clock internal to the microcontroller, not the clock used in the FiGo algorithm). These clock speeds can vary slightly due to environmental conditions (principally temperature). Laboratory tests with synchronised MICAz [33] sensor nodes, which use the ATmega128L controller, have revealed that the drift in clock speed can be pronounced over a period of hours. Lenzen et al. [31] studied the effect of varying ambient temperature on the clock speed of a 'Mica2' node, which uses the same processor as the MICAz node used in this paper. It was found that drift was up to one microsecond per second for a difference of five degrees Celsius (see Figure 3).

Using the raw data from [31] it was determined that at 0.0 degrees Celsius the operating frequency was 921,814 Hz, and at 30.0 degrees Celsius the frequency was 921,810 Hz. Therefore, for each tick of the clock, the amount of time taken per tick for a processor at 30.0 degrees Celsius will be 1.000004339 times longer

than for a clock at 0.0 Celsius. Eventually the warmer clock will lag the colder clock by one whole tick, i.e., the colder clock will have ticked twice and the warmer clock will have ticked once.

Suppose that clock c_1 has ticked n_1 times, with each tick having length l_1. Then, after a period of time, the total time elapsed is $n_1 l_1$. Similarly for clock $c2$, after n_2 ticks the total time elapsed is $n_2 l_2$. After a period of time, the clocks will tick in unison, so that $n_1 l_1 = n_2 l_2$. Suppose that clock c_2 has ticked exactly once more than c_1, so that $n_1 = n_2 + 1$. Therefore we know that $(n_2 + 1)l_1 = n_2 l_2$. If we let c_1 be the colder clock, and c_2 be the warmer clock, then we know that c_2's tick is 1.000004339 times longer that the tick of c_1, so that $l_2 = 1.000004339 l_1$. Therefore $(n_2 + 1)l_1 = 1.000004339 l_1 n_2$. Therefore $n_2 = 230,467$, and we know that after $230,468$ ticks of c_2's clock it will be exactly one tick behind c_1's clock.

Therefore, on average, every $230,468$ ticks, the warmer clock will lag behind the colder one by one whole tick. We can convert this to a probability, 1 in $230,468$, or 0.000004339, which can be incorporated into the PRISM model:

```
[tick] s1Phase=updateClock & s1Clock=1
  -> (1-pClockDrift): (s1Clock'=s1Clock+1) & (s1Phase'=start)
   + pClockDrift: (s1Clock'=s1Clock+2) & (s1Phase'=start);
```

This command says that if it is time to update the clock, then increase the clock value by 1 with probability $1 - \mathsf{pClockDrift}$, or by 2 with probability $\mathsf{pClockDrift}$, where $\mathsf{pClockDrift} = 0.0004339$. Note that $\mathsf{pClockDrift}$ is 100×0.000004339. This is because clock drift is modelled as happening once per duty cycle (specifically, when $\mathsf{s1Clock} = 1$), which is every hundred clock ticks. This helps reduce the state space because this branching point can only happen once per duty cycle, rather than on every tick. Note that the clock is increased by 2 when clock drift occurs. This is to ensure that the clock drifts only once per duty cycle — if the clock was increased by 0 (representing a slower clock rather than a faster one) then the precondition of this command would be true on the next iteration of the algorithm meaning that the clock could drift more than once in the duty cycle. As clock drift can be modelled either by one clock slowing by one tick, or the other clock speeding up by one tick, the accuracy of the model is not affected.

It is possible to calculate the effect of clock drift on the stability of clock synchronisation. One way to do this is use a *steady-state* probability in PRISM, denoted $S^{=?}[s]$, which is the probability that a model is in a particular state s at any given time. For example it was found that:

$$S^{=?}[\mathsf{s1Clock} = \mathsf{s2Clock}] = 0.996709321 \qquad [0.5\text{s}] \tag{10}$$

i.e., the probability that the model is in a synchronised state is equal to 0.996709-321. That is to say, 99.67% of the time the model is in a synchronised state.

It should be noted that the numerical methods normally used to determine the steady state probabilities in PRISM were not suitable in this case, as they either did not converge or returned a value of 1.0 after a very short execution time, indicating a possible problem with the use of the numerical method. One possible reason for this is the closeness of the probability of clock drift to zero.

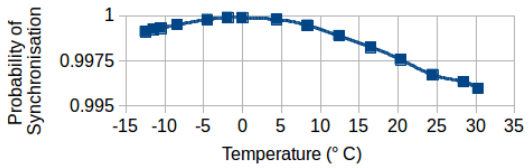

Fig. 4. Probability of synchronisation for varying temperatures of a second node.

Instead, 'exact model checking' was used, a technique in which the model checker builds the state space explicitly, and returns a probability based on the number of states matching the specified formula divided by the total number of states. Exact model checking is not enabled by default as it requires a lot of time and memory [39], but in this case the model was sufficiently small to allow its use.

Experiments with different values for pClockDrift showed that the steady state probability of synchronisation is dependent on the clock drift rate. If the clock drifts more often, then the model will spend less time in a synchronised state. The varying clock drift rates due to ambient temperature were examined to determine the effect on synchronisation of operating at varying temperatures. Various clock speeds were taken from the data in Lenzen et al.[31] corresponding to different temperatures. These were compared against a base clock speed of 921814.624 Hz. This value was chosen as it was the highest frequency observed, and it occurred at approximately zero degrees Celsius. Therefore the drift rates in our experiment were relative to a reference node operating at that temperature.

Figure 4 shows the effect on synchronisation between two nodes when one node is at zero degrees Celsius, and a second node is at a varying ambient temperature between -12.48 degrees Celsius and 30.48 degrees Celsius. It can be seen that the steady-state probability never drops below 0.9959677566, and decreases with increased difference in temperature between the two nodes. The shape of the curve closely matches that in Figure 3, as expected.

6 Conclusion and Future Work

We have shown how formal methods, in particular probabilistic model checking using PRISM, can be used to model and verify protocols used in critical IoT systems. Models were developed based on a straightforward translation from a pseudocode-style language into the PRISM modelling language. Key requirements of a gossip–synchronisation algorithm were encoded using probabilistic computation tree logic (PCTL*) and then verified formally using PRISM. These requirements included clock synchronisation, metadata synchronisation and steady-state probability of synchronisation.

Environmental effects, such as temperature, can affect a WSN node's hardware and cause clock drift. We have explored the use of formal verification to quantify the extent to which clock drift affects the synchronisation of WSN nodes. Results such as these can be useful for system designers who may wish to

adjust the parameters of FiGo, or even develop new algorithms, to better cope with sources of unreliability such as clock drift. These new algorithms can then be verified formally in a similar way to that described in this paper.

We have also demonstrated that state space explosion is a key challenge in the formal verification of WSNs. State space explosion issues are common when using model checkers like PRISM [13], and the results in Figure 2 are typical. However, it is often possible to compensate for state space issues through the use of abstraction and re-modelling. For example, rather than modelling the algorithm completely for each sensor, we could model it in detail for a single sensor, and model the rest of the network of n nodes with a second module in PRISM. In doing so the module size would be kept to a minimum, but would still allow for verification of the behaviour of the node in response to a network. A possible application of this approach would be to verify how long a particular sensor node takes to synchronise with an already-synchronised network. Another possibility is to use a population model (e.g., [21,20]), in which sensors are not modelled in detail, but rather the whole network, or several sub-networks, are modelled in order to verify properties concerning overall sensor network behaviour. These approaches, which could also be applied to investigate different sensor network topologies, are intended for future work.

Another way to compensate for state space explosion is to complement model checking with other verification methods, e.g., simulation. For example, sensor networks consisting of thousands of nodes can be analysed by simulation software [9]. Of course, the disadvantage of simulation is that it does not allow exhaustive examination of the state space, and is therefore prone to missing highly improbable events that can be detected using model checking: so-called 'black swans' [41]. However, this can be mitigated through analysis of sufficiently large numbers of simulations, as is the case with statistical model checking [30,1]. Naturally, we advocate the use of a range of different methods of verification for critical IoT systems, as their different characteristics are often complementary.

Our intention is to extend this approach beyond specific synchronisation and distribution algorithms, through the generation of a more general approach to critical IoT systems design. Such systems have commonly-used programming archetypes, such as sense–compute–send cycles for sensor nodes, or clock duty cycles. Formal modelling of these elements is often straightforward and could potentially be automated. In addition, simulation, algorithm animation, testing and a range of formal verification elements could all be included in a single tool to provide a strong and useful apparatus for the exploration and analysis of a range of design decisions. While there is much work still to be done to facilitate this, the research reported in this paper shows how certain design choices can be explored in a more precise, formal way.

Acknowledgment. The authors would like to thank Philipp Sommer for the experimental data from [31]. This work was supported by the EPSRC-funded programme grant S4 (EP/N007565/1) and the FAIR-SPACE (EP/R026092/1), RAIN (EP/R026084/1). and ORCA (EP/R026173/1) RAI Hubs.

References

1. Agha, G., Palmskog, K.: A survey of statistical model checking. ACM Transactions on Modelling and Computer Simulation **28**(1), 6:1–6:39 (2018). https://doi.org/10.1145/3158668
2. Atmel Corporation: ATmega128L: 8-bit Atmel microcontroller with 128 kBytes in-system programmable flash. http://www.atmel.com/images/doc2467.pdf (2018), last accessed 6/4/18.
3. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
4. Bakhshi, R., Bonnet, F., Fokkink, W., Haverkort, B.: Formal analysis techniques for gossiping protocols. ACM SIGOPS Operating Systems Review **41**(5), 28–36 (Oct 2007). https://doi.org/10.1145/1317379.1317385
5. Baresi, L., Blohm, G., Kolovos, D., Matragkas, N., Motta, A., Paige, R., Radjenovic, A., Rossi, M.: Formal verification and validation of embedded systems: the UML-based MADES approach. Software & Systems Modeling **14**(1), 343–363 (Feb 2015). https://doi.org/10.1007/s10270-013-0330-z
6. Basile, D., Di Giandomenico, F., Gnesi, S.: Statistical model checking of an energy-saving cyber-physical system in the railway domain. In: Proceedings of the Symposium on Applied Computing. pp. 1356–1363. SAC '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3019612.3019824
7. Boano, C., Zúñiga, M., Brown, J., Roedig, U., Keppitiyagama, C., Römer, K.: TempLab: A testbed infrastructure to study the impact of temperature on wireless sensor networks. In: Proceedings of the 13th International Symposium on Information Processing in Sensor Networks. pp. 95–106. IEEE Press (2014)
8. Brandt, A., et al.: RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. RFC 6550 (2012). https://doi.org/10.17487/rfc6550
9. Breza, M.: Bio-Inspired Tools for a Distributed Wireless Sensor Network Operating System. Ph.D. thesis, Imperial College, London (2013)
10. Breza, M., McCann, J.: Polite broadcast gossip for IoT configuration management. In: 3rd International Workshop on Sensors and Smart Cities. IEEE (2017)
11. Breza, M., McCann, J.A.: Lessons in implementing bio-inspired algorithms on wireless sensor networks. In: 2008 NASA/ESA Conference on Adaptive Hardware and Systems. pp. 271–276 (June 2008). https://doi.org/10.1109/AHS.2008.72
12. Chen, Z., Zhang, D., Zhu, R., Ma, Y., Yin, P., Xie, F.: A review of automated formal verification of ad hoc routing protocols for wireless sensor networks. Sensor Letters **11**(5), 752–764 (2013)
13. Clarke, E.M., Klieber, W., Nováček, M., Zuliani, P.: Model checking and the state explosion problem. In: Tools for Practical Software Verification, vol. 7682, pp. 1–30. Springer LNCS (2012)
14. Crouzen, P., van de Pol, J., Rensink, A.: Applying formal methods to gossiping networks with mCRL and Groove. SIGMETRICS Perform. Eval. Rev. **36**(3), 7–16 (Nov 2008). https://doi.org/10.1145/1481506.1481510
15. Dong, J.S., Sun, J., Sun, J., Taguchi, K., Zhang, X.: Specifying and verifying sensor networks: An experiment of formal methods. In: Liu, S., Maibaum, T., Araki, K. (eds.) 10th International Conference on Formal Engineering Methods, ICFEM 2008, Kitakyushu-City, Japan, October 27-31, 2008. pp. 318–337. Springer (2008), http://dx.doi.org/10.1007/978-3-540-88194-0_20
16. Fehnker, A., Gao, P.: Formal Verification and Simulation for Performance Analysis for Probabilistic Broadcast Protocols, pp. 128–141. Springer (2006), http://dx.doi.org/10.1007/11814764_12

17. Fisher, M.: An Introduction to Practical Formal Methods Using Temporal Logic. Wiley (2011)
18. Fitting, M.: First-Order Logic and Automated Theorem Proving. Springer (1996)
19. Fruth, M.: Formal Methods for the Analysis of Wireless Network Protocols. Ph.D. thesis, University of Oxford (2011)
20. Gainer, P., Linker, S., Dixon, C., Hustadt, U., Fisher, M.: Investigating parametric influence on discrete synchronisation protocols using quantitative model checking. In: Proceedings of QEST 2017. vol. 10503, pp. 224–239. Springer LNCS (2017)
21. Gainer, P., Linker, S., Dixon, C., Hustadt, U., Fisher, M.: The Power of Synchronisation: Formal Analysis of Power Consumption in Networks of Pulse-Coupled Oscillators. ArXiv e-prints (2017), https://arxiv.org/abs/1709.04385. Last accessed 23/5/18.
22. Haverkort, B.R., Siegle, M., van Steen, M.: Quantitative analysis of gossiping protocols. SIGMETRICS Perform. Eval. Rev. **36**(3), 2 (2008). https://doi.org/10.1145/1481506.1481508
23. Jelasity, M., Montresor, A., Babaoglu, O.: Gossip-based aggregation in large dynamic networks. ACM Transactions on Computer Systems **23**(3), 219–252 (2005)
24. Kartakis, S., Yu, W., Akhavan, R., McCann, J.A.: Adaptive edge analytics for distributed networked control of water systems. In: 2016 IEEE First International Conference on Internet-of-Things Design and Implementation (IoTDI). pp. 72–82. IEEE (2016)
25. Katoen, J.P.: How to model and analyze gossiping protocols? SIGMETRICS Perform. Eval. Rev. **36**(3), 3–6 (Nov 2008). https://doi.org/10.1145/1481506.1481509
26. Kim, H., Kang, E., Lee, E.A., Broman, D.: A toolkit for construction of authorization service infrastructure for the Internet of Things. In: Proceedings of the Second International Conference on Internet-of-Things Design and Implementation, IoTDI 2017, Pittsburgh, PA, USA, April 18-21, 2017. pp. 147–158. ACM (2017). https://doi.org/10.1145/3054977.3054980
27. Kwiatkowska, M., Norman, G., Parker, D.: Analysis of a gossip protocol in PRISM. SIGMETRICS Performance Evaluation Review **36**(3), 17–22 (Nov 2008). https://doi.org/10.1145/1481506.1481511
28. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Proc. 23rd International Conference on Computer Aided Verification (CAV'11). LNCS, vol. 6806, pp. 585–591. Springer (2011)
29. Langendoen, K., Baggio, A., Visser, O.: Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture. In: 20th International Parallel and Distributed Processing Symposium. IEEE (2006)
30. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: An overview. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) Runtime Verification. pp. 122–135. Springer (2010)
31. Lenzen, C., Sommer, P., Wattenhofer, R.: Optimal clock synchronization in networks. In: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems. pp. 225–238. SenSys '09, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1644038.1644061
32. Levis, P., Lee, N., Welsh, M., Culler, D.: TOSSIM: accurate and scalable simulation of entire TinyOS applications. In: Proceedings of the 1st international conference on Embedded networked sensor systems. pp. 126–137. ACM New York, NY, USA (2003)

33. MEMSIC, Inc.: MICAz wireless measurement system. http://www.memsic.com/userfiles/files/Datasheets/WSN/micaz_datasheet-t.pdf (2018), last accessed 6/4/18.
34. Mohsin, M., Sardar, M., Hasan, O., Anwar, Z.: IoTRiskAnalyzer: A probabilistic model checking based framework for formal risk analytics of the Internet of Things. IEEE Access **5**, 5494–5505 (2017)
35. Mouradian, A., Augé-Blum, I.: Formal verification of real-time wireless sensor networks protocols with realistic radio links. In: Proceedings of the 21st International conference on Real-Time Networks and Systems. pp. 213–222 (2013)
36. Munir, S.A., Ren, B., Jiao, W., Wang, B., Xie, D., Ma, J.: Mobile wireless sensor network: Architecture and enabling technologies for ubiquitous computing. In: Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops. AINAW '07, vol. 2, pp. 113–120. IEEE, Washington, DC, USA (2007). https://doi.org/10.1109/AINAW.2007.257
37. Nahrstedt, K., Li, H., Nguyen, P., Chang, S., Vu, L.H.: Internet of mobile things: Mobility-driven challenges, designs and implementations. In: First IEEE International Conference on Internet-of-Things Design and Implementation, IoTDI 2016, Berlin, Germany, April 4-8, 2016. pp. 25–36 (2016). https://doi.org/10.1109/IoTDI.2015.41
38. Ojha, T., Misra, S., Raghuwanshi, N.S.: Wireless sensor networks for agriculture: The state-of-the-art in practice and future challenges. Computers and Electronics in Agriculture **118**, 66–84 (2015)
39. Parker, D.: PRISM 4.4 Manual. Department of Computer Science, University of Oxford (April 2018), http://www.prismmodelchecker.org/manual/Main/Welcome. Last accessed 18/4/18.
40. Pfeifer, H., Schwier, D., von Henke, F.W.: Formal verification for time-triggered clock synchronization. In: 7th IFIP International Working Conference on Dependable Computing for Critical Applications (DCCA-7). pp. 207–226. IEEE (1999)
41. Taleb, N.N.: The Black Swan: The Impact of the Highly Improbable. Penguin Books (2007)
42. Tobarra, L., Cazorla, D., Cuartero, F.: Security in wireless sensor networks: A formal approach. In: From Problem Toward Solution: Wireless Sensor Networks Security, chap. 8, pp. 145–164. Nova (2009)
43. Ur-Rehman, A., Abbasi, A.Z., Islam, N., Shaikh, Z.A.: A review of wireless sensors and networks' applications in agriculture. Computer Standards & Interfaces **36**(2), 263–270 (2014)
44. Usman, M., Muthukkumarasamy, V., Wu, X.W.: Formal verification of mobile agent based anomaly detection in wireless sensor networks. In: 8th IEEE Workshop on Network Security. pp. 1001–1009. IEEE (2013). https://doi.org/10.1109/LCNW.2013.6758544
45. Werner-Allen, G., Tewari, G., Patel, A., Welsh, M., Nagpal, R.: Firefly-inspired sensor network synchronicity with realistic radio effects. In: Proceedings of the 3rd international conference on Embedded networked sensor systems. pp. 142–153. ACM New York, NY, USA (2005)
46. Yinbiao, S. et al.: Internet of Things: Wireless sensor networks. International Electrotechnical Commission White Paper (July 2014), http://www.iec.ch/whitepaper/pdf/iecWP-internetofthings-LR-en.pdf. Last accessed 23/5/18.