Exploring the Effects of Environmental Conditions and Design Choices on IoT Systems Using Formal Methods

Matt Webster[a,*], Michael Breza[b], Clare Dixon[a], Michael Fisher[a], Julie McCann[b]

[a] *Department of Computer Science, University of Liverpool, Liverpool, L69 3BX, UK*
[b] *Department of Computing, Imperial College London, London, SW7 2AZ, UK*

Abstract

Wireless communication protocols are often used in critical applications, e.g., urban water supply networks or healthcare monitoring within the Internet of Things. It is essential that control software and protocols for such systems are verified to be both robust and reliable. The effects on the hardware caused by environmental conditions and the choice of parameters used by the protocol are among the largest obstacles to robustness and reliability in wireless systems. In this paper we use formal verification to verify that a wireless sensor network synchronization and dissemination protocol is not adversely affected by these factors.

Keywords: Formal Verification, Protocol Verification, Internet of Things, Network Performance Evaluation, Model Checking, Wireless Sensor Network

1. Introduction

The Internet of Things (IoT) promises a revolution in the monitoring and control of a wide range of applications [1, 2], from water supply networks [3] and precision agriculture food production [4, 5, 6], to vehicle connectivity [7] and healthcare monitoring [8]. For applications in such critical areas, control software and protocols for IoT systems must be verified to be both robust and

*Corresponding author.
Email addresses: matt@liverpool.ac.uk (Matt Webster), mjb04@doc.ic.ac.uk (Michael Breza), cldixon@liverpool.ac.uk (Clare Dixon), mfisher@liverpool.ac.uk (Michael Fisher), jamm@imperial.ac.uk (Julie McCann)

reliable. Two of the largest obstacles to robustness and reliability in IoT systems are the effects on the hardware caused by environmental conditions, and the choice of parameters used by the protocol. In this paper we use a formal method of logic-based exhaustive analysis to verify that a Wireless Sensor Network (WSN) synchronization and dissemination protocol is not adversely affected by the environment or the choice of parameters used in the protocol. We show how the protocol can be converted into a logical model and then analyzed using the probabilistic model-checker, PRISM. Using this approach we prove under which circumstances the protocol is guaranteed to synchronize all nodes and disseminate new information to all nodes. We examine the bounds on synchronization as the environment changes the performance of the hardware clock, as well as the effects of varying important protocol parameters such as refractory period and energy consumption. We also show the scalability of this approach.

Systems for the Internet of Things (IoT) often involve networks of small, resource-constrained, computer devices embedded in an environment. These *sensor nodes* 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 synchronize 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 [9, 10].

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. In this paper we use formal verification, via the logical algorithmic method of *probabilistic model-checking* [11], to analyze and verify critical communication protocols used for the Internet of Things (IoT) [12]. The use of probabilistic models is crucial because it allows us to quantitatively analyze the system with the dynamical effects caused by the environment — one of the largest causes of failure for WSN [13]. WSN 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 rain or changes in temperature will affect the performance of the sensor nodes. These effects will influence

the correctness of protocols and algorithms running on the sensor nodes and can potentially cause protocol or node failure. The system software that provides event synchronization and controls message dissemination needs to be correct and 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

Formal methods are a family of techniques used to verify software and hardware, typically using mathematical, proof-based approaches [11]. These include techniques such as automated theorem proving [14], in which full mathematical proof is carried out, and model checking [15], 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 [11]. In this paper we use *probabilistic* model-checking [16], 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. In traditional model checking the result of formal verification is a Boolean value indicating whether or not a model satisfies the prescribed property; by contrast, probabilistic model checking allow us to find the probability that a particular model satisfies that property. Probabilistic model checking is essential for capturing and analyzing the stochastic nature of WSN, e.g., communication errors, or changes in ambient temperature resulting in clock drift. Additionally, the *quantitative* nature of probabilistic model checking allows it to be used for performance evaluation based on the time elapsed or energy consumed, for example [17, 18, 19, 20].

1.2. Contribution and Organization

This paper's contributions are as follows:

- A method of analyzing the reliability and correctness of communication protocols for WSN and IoT using formal verification and model checking, exemplified through the formal modelling and verification of a high-level communication protocol called FiGo. The translation from pseudocode to the formal model and the resulting formal verification are described in detail so that similar communication protocols may be analyzed in a similar way.

- Probabilistic formal models of a high-level communication protocol based on a design-level pseudocode-based description.
- Formal models of environmental conditions across a wireless sensor network based on the effects of temperature differences between nodes and the resulting hardware clock drift.
- Formal verification of key requirements of the communication protocol related to reliability, time and energy usage, encoded as logical properties that are used during model checking.

This work extends [21] via Section 7 on performance evaluation, an extended comparison with the literature in Sections 2 and 5.4, and extended explanations throughout.

The paper is organised as follows. We start with an overview of related work in Section 2. Next we cover some background material on formal methods and a decentralized wireless network management protocol called FiGo in Section 3. In Section 4 we show how a formal model of FiGo was developed using PRISM, and in Section 5 we describe how FiGo was formally verified through exhaustive analysis using the PRISM model checker. In Section 6 we introduce environmental effects into the PRISM model to capture the effects of temperature variation on hardware clock drift and examine the FiGo protocol's resilience to such effects. In Section 7 we evaluate the performance of the FiGo protocol with respect to synchronization time, stability and energy usage. In Section 8 we discuss our findings and make some more detailed comparisons with results from the literature. Conclusions are provided in Section 9.

2. Related Work

Formal methods have been used previously for design and analysis of WSN and the IoT. For example, Chen et al. [22] provide a survey of a number of approaches to formal verification of routing protocols for WSN. Kim et al. [23] conduct a formal security analysis of an authorization toolkit for the Internet of Things using the Alloy verification tool. Mouradian & Augé–Blum [24] describe the formal verification of real-time WSN protocols using the UPPAAL model checker. Tobarra et al. [25] use the Avispa model checking tool to formally verify a security protocol for WSN. Usman et al. [26] demonstrate formal verification of mobile agent-based anomaly detection for

WSN using the Symbolic Analysis Laboratory model checking tool. Dong et al. [27] 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 [28] used PRISM to analyze contention resolution and slot allocation protocols for WSN, but not synchronization or dissemination protocols. Synchronization [29, 30, 31] and gossip protocols [32, 33, 34, 35, 36, 37] have been formally verified but not together, and not accounting for environmental effects. Performance evaluation of protocols based on time elapsed and energy consumed has been examined before [29, 17, 20], but not with the inclusion of environmental effects.

Mohsin et al. [38] 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. [39], who used a UML-based MADES approach to model a system. The approach can find when constraints are not met, but does 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 [40]. Lenzen et al. [41] studied the effect of temperature on the hardware clocks chips used as timers on many common WSN sensor node platforms.

3. Background

In this paper we use the probabilistic model checker, PRISM [16, 42] to enable formal verification of a typical communications protocol for wireless sensor networks and the Internet of Things. PRISM 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, a sensor node that can either transmit a message or remain idle can be modelled simply using a P-FSA as shown in Figure 1.

The two states, 'transmit' and 'idle', are linked with arrows, or transitions. These transitions specify how the state of the P-FSA can change. The annotations on the transitions show the probability that the transition will

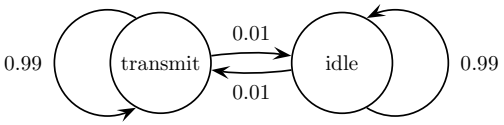


Figure 1: A simple probabilistic finite state automaton with two states.

be taken. In this case there is a probability of 0.99 that the sensor node remains in its current state, and a probability of 0.01 that the state will change. Note that these values can either be chosen arbitrarily, or with reference to a particular practical sensor node implementation. For example, a specific sensor node may spend more time transmitting than idling, in which case the probabilities associated with these transitions can be modified.

We can model the sensor node above in the PRISM 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

$$[s]\ \text{guard} \text{ -> } p_1 : u_1 + \ldots + p_n : u_n;$$

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 synchronization. Synchronized commands execute simultaneously with synchronization 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 formalized as *properties* using probabilistic computation tree logic (PCTL*) [42]. 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 for propositions p and q:

- 'p', meaning that p is true;
- '$\neg p$', read as 'not p,' meaning that p is false;
- '$p \Longrightarrow q$', read as 'p implies q,' meaning if p is true then q is true;
- '$p \wedge q$', read as 'p and q,' meaning that both p and q are true;
- '$p \vee q$', read as 'p or q,' meaning that p or q (or both) are true;
- '$\mathsf{F}\ p$', read as 'finally p' or 'eventually p', meaning p is true now or at some point in the future; and
- '$\mathsf{G}\ p$', read as 'globally p' or 'always p,' meaning p is true now and at every point in the future.
- '$\mathsf{G}\ \mathsf{F}\ p$', read as 'always eventually p' or 'infinitely often p,' meaning that it is always the case that p is eventually true.
- '$\mathsf{F}\ \mathsf{G}\ p$', read as 'eventually always p,' meaning that at some point in the future p becomes true, and remains true.
- '$P^{=?}[M]$', referring to the probability that M is true for some well-formed formula M.
- '$S^{=?}[M]$', referring to the steady-state probability that M is true, i.e., the probability that at some point the formula M is true.
- '$R^{=?}[M]$', referring to the expected *reward* in states where M is true. Rewards let us quantify things within a model, e.g., time taken or energy used. Rewards are described in more detail in Section 7.

PRISM also allows the use of standard numerical operators such as $=$, $\geq$ and $\leq$ in PCTL* formulae. A full syntax and semantics for PCTL* can be found elsewhere [15].

Model checking works by analyzing 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 that the sensor node is eventually in the 'idle' state:

$$\mathsf{F}\ (\mathsf{state} = 1)$$

This can then be turned into a property in PRISM by adding the $P^{=?}[\ldots]$ operator around the PCTL* formula. This operator indicates that we are querying PRISM for the probability of the formula being true on some path through the model:

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

In this case, PRISM has determined that the probability is 1.0. More complex properties can be formed. For example, the following property gives the probability that the model will always be in the 'idle' state at some point:

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

This is also equal to 1.0 for the sensor node PRISM model.

3.1. The Firefly-Gossip (FiGo) Protocol

In the next section we describe the construction of a PRISM model of a decentralized wireless network management protocol called FiGo. FiGo was chosen because it is a relatively-simple protocol that contains characteristics found in more commonly used protocols like Trickle [43] and RPL [44] such as the use of local-only information and dissemination via epidemic propagation [45]. Unlike these protocols, FiGo serves two functions, it both synchronizes clocks and forces the agreement of new information for all of the nodes in the network. In FiGo, these two functions are coupled, and so it was important to the authors of FiGo that its correctness for both functions be verified.

FiGo synchronizes sensor node's clocks in order to unify the measurement of time across the network using firefly-like synchronization. In nature, fireflies are bioluminescent beetles that create bursts of light to attract mates or prey. Fireflies have been observed to synchronize these bursts with nearby

fireflies. The FiGo protocol replicates this behaviour algorithmically to allow WSN nodes to synchronize with one another. A key parameter in firefly synchronization algorithms is the *refractory period*, a period after transmitting for which no further transmissions are made. Within a wireless sensor network, a refractory period enables nodes to conserve energy usage in a similar way to real fireflies. In addition to synchronization, FiGo enables agreement on key information between nodes via *gossiping* new information to their neighbours.

Current techniques for large-scale computer management are not suitable for WSN 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 [46]. In applications such as precision agriculture [4, 5], wireless nodes need to be synchronized to be able to deliver time-correlated samples of data such as moisture levels and temperature, and to analyze 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.

Synchronization of a WSN is essential in many applications, for example in adaptive sensing for smart water networks [47]. WSN allow urban water providers to monitor the water flow to match customer demand. Synchronization 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 stabilize 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 synchronization [48] and gossip protocols [45] into a single epidemic process [49]. This mixture of synchronization and dissemination processes is used to bring the internal states of WSN nodes to a stable, global equilibrium where all nodes are synchronized with respect to both time and metadata. Experiments have shown such protocols to be both scalable and resilient to individual node failure [49, 46, 50]. A typical FiGo algorithm is shown in Figure 2.

FiGo algorithms have been deployed for the synchronization and management of several WSN deployments run by the Adaptive Emergent Systems

Engineering group at Imperial College[1]. For example, they were used to organize pollution sensors for an experiment with mobile data mules as part of an Imperial College Grand Challenge project, and to synchronize 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 [51].

In practice, FiGo algorithms like the one in Figure 2 are implemented on a network of microcontroller-based computers equipped with sensors and radio communications such as the MicaZ sensor nodes [52] shown in Figure 3. One instance of the FiGo algorithm runs on each node. The nodes are switched on randomly and the FiGo algorithm enables synchronization and gossiping across the network. As sensor nodes are switched on at different times it is not possible to assume that the algorithms execute from the same starting point. For example, one node may be executing line 1 of the algorithm, while another is executing line 5, and other is executing line 12 within the main control loop. Furthermore, we cannot assume that nodes execute synchronously as sequences of instructions corresponding to different lines within the algorithm may execute at different rates, and as we shall see in Section 6, environmental factors such as temperature variation may affect the clock speed at which the microcontrollers themselves operate.

4. A PRISM Model of FiGo

It is possible to model various WSN protocols in PRISM. In order to illustrate the approach, we create a model of the decentralized WSN management protocol known as FiGo [49]. A PRISM model of FiGo was developed precisely capturing the control flow of the algorithm in Figure 2. An excerpt from the PRISM model can be seen in Appendix A. The algorithm begins with a number of variable 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

[1] http://wp.doc.ic.ac.uk/aese/

to 14. The next if-statement consists of a number of nested if-statements called 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 were used as the values of a local variable s1Phase which contains the currently-executing phase:

```
s1Phase : [0..9] init transmit;
```

Note that s1Phase refers to the phase of the first sensor node module, which is called s1. Other sensor node modules are called s2, s3, etc.

When one phase has finished executing the next phase is chosen according to the control flow of the algorithm in Figure 2. For example, during the sync1 phase in lines 16 to 19 the algorithm checks whether a message has been overheard (i.e., received), and if the clock is outside the refractory period. If they are, then the sensor node updates its clock to the average of its own clock and the clock of the other sensor node. The "circular average" is used, in which the average of 90 and 10 is 0 (modulo 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 node. The following procedure calculates the circular average of two numbers a and b modulo 100:

```
if ( | a−b | > 50 )
    return ceiling ((a+b+100)/2) mod 100
else
    return ceiling (( a +b)/2)
```

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

```
[] s1Phase=sync1 & s1Clock>=refractoryPeriod & diff ≤ floor(cycleLength/2) →
  (s1Clock'=s1avg1) & (s1Phase'=sync2);
[] s1Phase=sync1 & s1Clock>=refractoryPeriod & diff >floor(cycleLength/2) →
  (s1Clock'=s1avg2) & (s1Phase'=sync2);
[] s1Phase=sync1 & !( s1Clock>=refractoryPeriod ) → (s1Phase'=sync2);
```

The PRISM formulae diff, s1avg1 and s1avg2 are defined as follows:

```
formula diff = max(s1Clock,s2Clock)−min(s1Clock,s2Clock);
formula s1avg1 = ceil((s1Clock+s1InboxClock)/2);
formula s1avg2 = mod(ceil((s1Clock+s1InboxClock+cycleLength)/2),cycleLength);
```

The first two commands say that if the sensor node s1 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 the s1's clock and s2's clock. The third command

says that if we are in the sync1 phase and we are within the refractory period, then proceed to the next phase of the algorithm, sync2.

PRISM models are based on modules which execute concurrently. Each sensor node in the PRISM model is represented by a single module. For example, in a two-node model there are two modules. The PRISM model is based on a state transition system, and therefore time is modelled implicitly in the transitions between states. Modules are able to communicate using: (i) synchronizations, (ii) global variables that may be updated by any module, or (iii) local variables which may be updated by just one module but which are readable by all. PRISM allows modules to be executed concurrently in different ways based on the use of ideas from communicating sequential processes and process algebra [42]. The PRISM model of FiGo is based the PRISM default of an alphabetised parallel composition of the transitions across each of the modules. For example, for a n-node model with modules $\mathsf{s1}, \mathsf{s2}, \ldots, \mathsf{s}n$ the composition $\mathsf{s1} \parallel \mathsf{s2} \parallel \ldots \parallel \mathsf{s}n$ is used. The modules are synchronized only on labelled transitions which appear in all modules. For example, the tick synchronizations used in Section 5.1 cause the modules to synchronize when executing the transitions with that label.

The sensor node 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 online[2]. This PRISM model is an almost direct translation from the pseudocode to PRISM and was not initially optimized for formal verification.

5. Formal Verification Using PRISM

We build a formal model in PRISM, in our case using version 4.3.1, in a manner analogous to compiling a program: an input file containing the PRISM model is automatically converted into a probabilistic finite state au-

[2] http://dx.doi.org/10.17638/datacat.liverpool.ac.uk/1118

tomaton. During this construction, PRISM calculates the set of states reachable from the initial state and the transition matrix which defines transitions between states. Building revealed that the full model consisted of 4,680,914 reachable states, 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 5.1, 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 synchronized:

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

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

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

Therefore the probability that synchronization occurs infinitely often is 1.0. We can strengthen the property further: we can verify that, once the clocks are synchronized, that they remain synchronized:

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

In this case the probability of this property being true is 0.0, meaning that it is *never* the case that the two clocks synchronize and then remain synchronized 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 [42].) Below is a simulation of the model showing how desynchronization occurs after synchronization:

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 synchronized. However, a transition occurs in which one of the sensors, in this case, s2, increments its clock value resulting in de-synchronization. However, in the next state we can see that the sensor s1 updates its clock as well, resulting in synchronization.

We might postulate that once synchronization occurs, then de-synchronization will occur at some point. This can be encoded as the following property, which evaluates to 1.0:

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

We can also verify whether once de-synchronization has happened, that synchronization will eventually happen with a probability of 1.0:

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

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

5.1. Increasing the Model's Accuracy

Examining simulations using PRISM reveals that clocks will rapidly de-synchronize after synchronization, 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 node developed by Memsic Inc. [52] (see Figure 3). 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 [53]. 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 assume that the clock speed is constant across different nodes. The current model assumes that clock speeds may vary, however, so this model should be updated to use a constant clock speed.

Clock speeds were made constant by introducing synchronizations 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 synchronization 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 the FiGo algorithm is implemented.

Another advantage of constant clock speeds is that it reduces the total number of states of the probabilistic model. Constant clock speeds were introduced to increase the accuracy of the model, but also have a side-effect of reducing model size and complexity. 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.

Properties 1–5 were formally verified for this revised model and were found to have the same probabilities as before, but with significantly reduced times for model checking, e.g.:

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

5.2. Improving Efficiency Further

As described above, the PRISM model has a long build time of 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 synchronization via circular averaging. Therefore, from this point on the reduced duty cycle length was used.

Naturally, reducing verification time makes verification more convenient, but it also enables batch processing of verification jobs known as *experiments* within the PRISM model checker. To demonstrate this, we used the experiment feature within the PRISM model checker to automatically check every combination of nextBroadcast values. In the original definition of the FiGo algorithm, the variable nextBroadcast is assigned a random value between 0 and 99 for each sensor. During construction of the PRISM model, however, these random values were modified to a constant integer value. In order to verify that the FiGo algorithm would work with every possible value of the nextBroadcast constants we used the experiment feature automatically check every combination of nextBroadcast values. This is done by removing the values of the global constants that represent the next broadcast value. This effectively makes them variables:

```
const int s1NextBroadcast;
const int s2NextBroadcast;
```

Then, PRISM can be used to perform automatic, and exhaustive, model-checking of properties across a range of values for these constants. Property 2 was verified with a range of $[0, 20]$ for both variables. The results can be found in the online repository[3] and showed that the probability that synchronization will happen infinitely often (i.e., Property 2) is 1.0 for every combination of values, meaning that the FiGo algorithm reaches synchronization regardless of the particular values of the nextBroadcast variables.

5.3. Gossip and Synchronization

The properties examined thus far have concerned clock synchronization. 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 a new piece of information. In the case of the FiGo algorithm, this is represented by an integer variable whose initial value is zero, but which

[3] `http://dx.doi.org/10.17638/datacat.liverpool.ac.uk/1118`

may increase when a node is updated with a new piece of information. This captures a common function of WSN that must share new information, roll-out software updates, etc.

The FiGo algorithm currently modelled in PRISM sends metadata to the other sensors, and when it receives metadata it compares it against its own local metadata value. In the models examined previously, the metadata was fixed at zero across all sensors to allow us to examine clock synchronization only. In order to analyze metadata synchronization 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 from the initial value. 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 6,018 states and 9,408 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:

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

This property says that the probability that the metadata is eventually synchronized across nodes is 1.0. It can also be verified that metadata is synchronized infinitely often:

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

Furthermore, we can verify that the Firefly and Gossip parts of the algorithm both work, and that infinitely often the two sensors will by synchronized on

both time and metadata:

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

This allows us to verify the original requirements of the FiGo authors [46] that both the Firefly synchronization and Gossip parts of the algorithm can correctly co-exist.

To examine the scalability of the FiGo algorithm as it is currently modelled, 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 total time to verify Property 2 (including build and verification time) was affected. The results are summarized in Figure 4. All of the probabilities for Property 2 for the different network and duty cycle sizes were found to be 1.0, showing that synchronization happens infinitely often in all the cases examined.

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 analyzed 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 4). This is due to a state space explosion [54] 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). These variables include the inboxes which have to be updated with the clock and metadata values for synchronization and gossip respectively. 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. It is possible to reduce the size of the state space by reducing the number of variables in the model. For example, the use of inboxes results in values being copied from the clock value of a sensor node to the inbox of each of the other sensor nodes. We can, however, avoid the use of inboxes by reading values directly from other modules (as all variables are readable by other modules) or by using global variables. This would enable us to eliminate the need for inbox variables, thereby reducing the size of the state space. Strategies like this based on re-modelling and abstraction are often used to manage the state space explosion problem and reduce the time and memory

required during model checking. More information on these approaches can be found in Section 5.4.

5.4. Overcoming State Space Explosion

It should be noted that the results presented earlier only pertain to the FiGo model examined in this paper, and other models and protocols may permit larger sensor networks to be analyzed. 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. For example, it may be possible to reduce the size of the state space significantly by using a more abstract model in which less relevant details are left out. In this paper, we chose a straightforward translation from pseudocode to PRISM for the following reasons. Firstly, the time taken to write the model is greatly reduced by using a straightforward translation. Secondly, as it is more straightforward, the relationship between lines of pseudocode and their implementation in the PRISM model should be clearer, meaning that it is less likely that errors have been introduced by more significant abstractions. Thirdly, if the authors of the pseudocode algorithm decide that modifications are necessary, then these should be more straightforward to implement by modifying the existing PRISM model. Of course, as we saw in the previous section, the use of a straightforward translation closely resembling the pseudocode of the algorithm, and using little abstraction, is likely to result in a model with significantly more states than would be found by using a more abstract model. As is often the case in engineering, there are trade-offs: in this case, between ease of translation, clarity, extensibility, time and memory required for model checking, and so on.

The complexity of model checking reflects the high level of confidence gained through the exhaustive examination of the state space. Larger sensor networks can be modelled using simulation, but simulation does not allow for the complete traversal of the state space. However, model checking often cannot model at the same level of detail as simulation. Therefore, we advocate the use of model checking in concert with simulation, experimentation (with real sensor networks) and other verification tools, as is the case with corroborative verification and validation [55], in order to develop the highest levels of confidence in protocols for critical IoT systems.

In this section the size of the PRISM model was reduced using a number of simplifications, or abstractions. The aim of any abstraction is to remove unnecessary detail, so that only pertinent information remains. For example,

in order to reduce the build time of the PRISM model from 17 minutes to 16.8 seconds the duty cycle time was reduced from 100 to 20. Another example is in the handling of metadata, in which there are only 4 possible values of the metadata used. In each case these abstractions allow us to reduce the time and memory needed for probabilistic model checking, but without significantly affecting the accuracy of the model. Of course, as is the case in many engineering situations, there is a trade-off. At one end of the scale, there are highly tractable models which are quick to compute and analyze, but lack sufficient detail to allow us to draw any meaningful conclusions for verification. At the other end of the scale there are highly detailed models which cannot be analyzed in a reasonable amount of time. The job of the verification engineer is to find a 'happy medium' between these two extremes, so that a model is sufficiently detailed to be interesting, but is sufficiently simple to allow tractable analysis.

Simulation is often used as a verification tool in a similar way to model checking. Models of the system are built and then executed in order to analyze the behaviour of the system across varying parameters. However, the key difference between model checking and simulation is that model checking is exhaustive, and examines every possible state of a model. In contrast, simulation examines only the states which were encountered during a finite number of simulations. These states are often a very small subset of the set of possible states. However, simulations can be more detailed than models for model checkers, as not every state has to be examined.

In this paper we focus our efforts on formal verification of protocols for WSN and critical IoT systems in order to show the applicability of the approach to more complex protocols such as FiGo. However, when verifying systems for use in critical IoT systems, or any other kind of critical system, the most preferable option is to use a variety of verification and validation techniques so that the best features of each can be leveraged [55].

6. Environmental Effects on Hardware

IoT systems are often deployed out-of-doors in environments that are hostile to electronics. Appropriate packaging can mitigate many of the affects of the damp and animal intrusion, but can not help prevent changes in temperature. It has been shown that the functioning of microcontrollers commonly used in IoT sensor nodes, such as the ATmega128L, are particularly affected by temperature. Specifically, the speed of the oscillator used the internal

microcontroller clock changes depending on the temperature. (In this case the clock speed refers to the clock internal to the microcontroller, not the clock used in the FiGo algorithm described earlier.) Laboratory tests with synchronized MICAz sensor nodes have revealed that the drift in clock speed can be pronounced over a period of hours. Due to the pronounced effects of temperature on the function of the sensor nodes, and the difficulty of mitigating these with packaging, we focus on solely on temperature in this work.

Lenzen et al. [41] 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 5). Using the raw data from [41] 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 clocks c_1 and c_2 starting ticking at the same point. When clock c_1 has ticked n_1 times, with each tick having length l_1, 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 again (see Figure 6), 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 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 probabilistic 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 synchronization. One way to do this is use a *steady-state* probability in PRISM, which is the probability that a model is in a particular state at any given time. For example it was found that:

$$S^{=?}[\mathsf{s1Clock} = \mathsf{s2Clock}] = 0.999307518 \ \ [0.5\text{s}] \tag{10}$$

i.e., the probability that the model is in a synchronized state is equal to 0.999307518. That is to say, 99.93% of the time the model is in a synchronized 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. 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 [42], but in this case the model was sufficiently small to allow its use.

Experiments with different values for $\mathsf{pClockDrift}$ showed that the steady state probability of synchronization is dependent on the clock drift rate. If the clock drifts more often, then the model will spend less time in a synchronized state. The varying clock drift rates due to ambient temperature were

examined in order to determine the effect on synchronization of operating at varying temperatures. Various clock speeds were taken from the data in Lenzen et al.[41] corresponding to different temperatures. The clock speeds 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 7 shows the effect on synchronization 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 0.999151067, and decreases with increased difference in temperature between the two nodes. The shape of the curve closely matches that in Figure 5, as expected.

7. Performance Evaluation and Design Choices

In the previous sections we showed how probabilistic model checking can be used to establish that a communications protocol for a wireless sensor network satisfies key requirements concerning reliability. In this section we show how probabilistic model checking can also be used to investigate performance trade-offs. For example, a key parameter in the implementation of FiGo is the refractory period; a period during which the node does not modify its clock in response to messages from other nodes. The refractory period is based on observations of firefly synchronization, in which fireflies do not receive messages from other nodes within a period of time after 'firing', i.e., sending a message. The FiGo algorithm in Figure 2 uses a refractory period of 50 within a duty cycle of 101 clock ticks, meaning that the algorithm will not receive messages for half of its duty cycle.

Our first experiment investigated the steady-state probability of synchronization using the property shown in the previous section.

$$S^{=?}[\mathsf{s1Clock} = \mathsf{s2Clock}] \tag{11}$$

This property was examined for a PRISM model with a duty cycle of 20 steps, clock drift of 0.000004339, refractory period between 0 and 15, s1NextBroadcast between 0 and 5, and s2NextBroadcast between 6 and 11. Verifying with respect to these parameters took over 19 hours. The results, shown in Figure 8, show how the median probability of synchronization decreases as the refractory period increases. This was expected, as a longer refractory period causes

the sensor node to be non-responsive to other messages for a longer period of time, causing the node to spend more time unsynchronized.

The relationship between the amount of time required to synchronize for different refractory periods can be investigated further using reward-based properties in PRISM. Reward structures in PRISM allow values to be associated with particular states or transitions of interest. For example, in the case of time, a reward can be associated with every time step in the model as follows:

```
rewards
  [] true : 1;
endrewards
```

This reward structure tells the model checker to associate the value 1 with every state in the model (as the proposition true is true in every state). A reward-based property is then defined as follows:

$$R^{=?}[p]$$

This property returns the average, or expected, total reward for a path leading up to states where p is true. For example, we could count the number of times that the PRISM modules synchronize on the tick label by using the following rewards structure:

```
rewards
  [tick] true : 1;
endrewards
```

We can also find the time taken for a sensor node to synchronize with another node. The first step is to set p to be the synchronization condition:

$$R^{=?}[\mathsf{s1Clock} = \mathsf{s2Clock}] \tag{12}$$

Using this reward-based property with the reward structure above, we can find the average amount of time taken to reach the synchronization state. However, the structure will assign a reward of 1 to every state in the model. This will include states in which other nodes are active. We would like to only pay attention to states in which a particular node is active. Therefore the reward structure above had to be modified in order to measure the time taken for a single node. To do this we add the label x to each transition in a single node's module, e.g.:

```
[x] s1Phase=clockCycleCheck & s1Clock<cycleLength → (s1Phase'=synchronize);
```

Three transitions already contain a label, tick, which is used for synchronization of the clock updates (as described in Section 5.1). A few transitions that update global variables could not have a reward label, as PRISM assumes that all labels denote synchronization, and synchronized transitions are not allowed to modify global variables. In this case, 'dummy' transitions were inserted into the model to follow directly after the transitions which update global variables. These dummy transitions were then labeled to allow a reward to be assigned to them. (As this technique involves adding transitions to the model, the amount of time used to build and verify the model increases slightly.)

The following reward structure was then used:

```
rewards
  [x] true: 1;
  [tick] true : 1;
endrewards
```

This assigns a reward of 1 to every transition for a single node. The reward was assigned to a transition, rather than a state, as it was simpler to pick out states of a single model using labels on transitions, rather than using conditions on states.

The average time required for synchronization was investigated for different refractory periods. The results can be seen in Figure 9. The graph shows that the mean synchronization time increases with the refractory period. (Standard deviation is shown via error bars.) This was expected as increasing the refractory period prevents the node from responding to messages for a greater period of time, meaning that the time taken for synchronization to occur will increase.

Rewards can also be used to investigate the approximate energy consumption of a node [29, 42]. Manufacturer specifications state that the MicaZ platform draws 19.7 mA of current in receive mode, and 11 mA when sending at -10 dBm [52]. These values can be added to a reward structure as follows:

```
rewards
  [rx] true : 19.7;
  [tx] true : 11;
endrewards
```

The action labels `rx` and `tx` refer to receiving and transmitting respectively. Transitions in the model are then labelled accordingly. (The MicaZ node uses half-duplex communication, so sending and receiving are mutually-exclusive.)

The implementation of the algorithm requires that the MicaZ node is always in receive-mode, except when it is transmitting. For simplicity, we assume that receiving a message takes the same amount of time as transmitting, but it is equally possible to model situations where receiving and transmitting require different amounts of time. We can also add more detail to the model, e.g., to take into account idle periods when the microprocessor does not draw as much current.

Verifying the model with respect to Property 12 then reveals the average energy consumption for synchronization. The refractory period can then be varied (in the same manner as before) to determine the effect on energy consumption. The results are shown in Figure 10. It can be seen that the average energy consumed to reach synchrony increases with the refractory period. Again, this is expected as increasing the refractory period increases the amount of time to synchronize (Figure 9) and this, in turn, means that more energy is consumed.

The results in Figures 8–10 indicate that a refractory period of 3 leads to better results for this version of the algorithm: the least synchronization time, the least energy usage and the greatest stability of synchronization. The significance of this precise value is not clear; however examination of related work on firefly synchronization protocols may give us some clues. Please see Section 8 for more information.

It is possible that the refractory period could be better used, however. For example, if the algorithm was set to enter a low-power mode during its refractory period, then energy could be saved. In fact, the ATmega128L microcontroller used on the MicaZ sensor node has such a low-power mode, known as, 'extended standby' [53], which uses an average current of 0.25 mA, compared to 8.65 mA [56].

This can be represented within the PRISM model by an updated rewards structure:

```
rewards
  [rx] s1Clock>refractoryPeriod : 19.7;
  [tx] true: 11;
endrewards
```

For simplicity, the minimal current of 0.25 mA is treated as zero. The reward for receiving (rx) is now only applied when the clock is outside its refractory period. The effect on the energy consumed for different refractory periods is shown in Figure 11, which shows that the average energy consumed decreases as the refractory period increases. This indicates a trade-off between (i)

energy consumption, and (ii) stability and synchronization time when the low-power mode is used. (The results in Figures 8 and 9 are still valid as the model of the algorithm has not changed.) While energy consumption can be minimized by increasing the refractory period, stability and synchronization time are best when the refractory period is equal to 3.

8. Discussion

We have shown how formal methods, in particular probabilistic model checking using PRISM, can be used to model and verify wireless sensor networks that use synchronization/distribution algorithms such as the Firefly–Gossip algorithm. Models were developed based on a straightforward translation from a pseudocode-style language into the PRISM modelling language. Key requirements of the FiGo algorithm were then encoded using probabilistic computation tree logic which were then verified formally using PRISM. These requirements included clock synchronization, metadata synchronization and steady-state probability of synchronization, verifying that the FiGo algorithm can be used reliably in WSN to synchronize clock and metadata values. Due to the nature of model checking, this verification is based an exhaustive analysis showing that the models satisfy properties based on those requirements. In turn, this gives a high level of assurance that the system has been designed correctly.

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 synchronization of WSN nodes. Results such as these can be extremely 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 synchronization algorithms can then be verified formally in a similar way to that described in this paper.

We have also demonstrated that the state space explosion is a key challenge in the formal verification of WSN. State space explosion issues are common when using model checkers like PRISM [54], and the results in Figure 4 are typical. However, it is often possible to alleviate 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. One possible application of this approach would be to verify how long a particular sensor node takes to synchronize with an already-synchronized network. Another possibility is to use a population model (e.g., [29, 30]), in which no one sensor is 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 outside the scope of this paper and are intended for future work.

Another way to compensate for the state space explosion is to complement formal verification with other verification methods, e.g., simulation. Simulation can provide a greater level of detail. For example, sensor networks consisting of thousands of nodes can be analyzed by simulation software [46]. 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' [57]. (A more detailed comparison of simulation and model checking can be found in Section 5.4.) Naturally, we advocate the use of a range of available methods of verification for critical IoT systems, as their different characteristics are often complementary. An holistic approach including, but not limited to, algorithmic analysis, simulation, experimentation, field reports, as well as formal verification, can be used to provide a high level of confidence in critical systems [55].

In Section 5.1 we described how the PRISM model was improved by introducing synchronization between the modules. This had the dual effect of increasing the accuracy of the model (as the actual hardware systems on which FiGo is implemented is homogeneous) and increasing the efficiency of model checking by reducing the number of the states in the model by a significant margin. A similar approach was taken by Dixon et al. [58] when model checking swarm robotic systems. Initially, they investigated a few different approaches to synchronization, from full synchrony, through strict and non-strict turn-taking between modules/robots and ending at fair asynchrony in which robots were allowed to operate at different speeds up to a certain point. The latter case would be analogous to modelling a wireless sensor network where clock speeds are allowed to vary within a particular margin. After some analysis the authors concluded that full sychrony allowed the most accurate modelling of their homogeneous robot swarm. Additionally, it was found that this had a positive effect on the efficiency of model checking. Of

course, this abstraction is more difficult to use when heterogeneous networks or swarms are used. However it may still be possible to use fair asynchrony as a compromise between model accuracy and efficiency.

In Section 7 we showed how the FiGo algorithm works best with a refractory period of 3 across a range of possible refractory periods from 0 to 20. Specifically, we found that the probability of synchronization, synchronization time, and energy consumption were optimal at that point. (This result is distinct from the case where the low-power mode was used. In that case the energy consumption decreased as the refractory period increased.) As mentioned in that section, the reasons for this are unclear. However, similar results were presented by Gainer et al. [30] in their analysis of firefly-like synchronization in networks of pulse-coupled oscillators. The authors used a population-model within PRISM and observed that the optimal refractory period with respect to synchronization time and stability is observed at around half the cycle length (analogous to the duty cycle length of the FiGo algorithm). A similar result for networks of pulse-coupled oscillators was reported by Degesys et al. [59]. The corresponding value for FiGo with a duty cycle length of 20 steps would be at approximately 10 steps. Clearly this is different from the optimal refractory period of 3 seen with FiGo. However the placement of the optimal refractory period at a point midway between the two extremes of the duty cycle appears to be a common theme.

9. Conclusion

We have shown how formal methods, in particular probabilistic model checking using PRISM, can be used to model and verify the performance of WSN protocols under the influence of environmental conditions, like temperature, that affect their hardware.

We have also shown that formal verification can provide an important basis for investigation and analysis of WSN protocol design choices. Formal verification can therefore be productively used as a design tool in the development of WSN algorithms and protocols. By formally modelling the algorithms a designer can assess various parameters and options before turning to either detailed simulation or even practical implementation. Furthermore. we showed how the performance of algorithms for WSN can be analyzed quantitatively using probabilistic model checking, providing insights into trade-offs between key performance indicators, such as stability, time efficiency and energy consumption.

Our intention is to extend applicability beyond just specific synchronization and distribution algorithms, through the generation of a more general approach to WSN design. This will incorporate simulation, algorithm animation, testing and a range of formal verification elements, to provide a strong and useful tool 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.

CRediT author statement

Matt Webster: Methodology, Investigation, Software, Validation, Formal analysis, Writing - Original Draft. **Michael Breza**: Investigation, Software, Resources, Writing - Original Draft. **Clare Dixon**: Conceptualization, Writing - Review & Editing, Supervision, Funding acquisition. **Michael Fisher**: Conceptualization, Writing - Review & Editing, Supervision, Funding acquisition. **Julie McCann**: Conceptualization, Writing - Review & Editing, Supervision, Funding acquisition.

Acknowledgments

The authors would like to thank Philipp Sommer for the experimental data from [41]. The work is supported by EPSRC Programme Grant EP/N007565/1, "Science of Sensor System Software (S4)", the EPSRC FAIR-SPACE (EP/R026092/1) and ORCA (EP/R026173/1) RAI Hubs, and the Royal Academy of Engineering.

Appendix A. PRISM Model of FiGo

An excerpt of the code for the PRISM model of the example described in Section 4 is reproduced here. This excerpt contains the code for a module called sensor1. All models also contained a similar module called sensor2 in which references to "sensor1" and "s1" are replaced by "sensor2" and "s2". In the three- and four-sensor models described in Section 5 there are additional modules for sensor3 and sensor4. The full modules, together with files containing properties verified, are provided online [4].

[4] http://dx.doi.org/10.17638/datacat.liverpool.ac.uk/1118

```
1  module sensor1
2     s1Phase : [0..9] init transmit;
3     s1Clock : [0.. cycleLength] init 5;
4     s1LocalMetadata : [0..1] init 1;
5     s1SameCount : [0..2] init 0;
6
7     // if local clock = next broadcast and same count < same threshold then
8     // transmit local clock and local metadata
9     [] s1Phase=transmit & (s1Clock=s1NextBroadcast | s1Clock=(mod(
          s1NextBroadcast+refractoryPeriod,cycleLength))) & s1SameCount<sameThreshold
          → (s2InboxClock'=s1Clock) &(s2InboxMeta'=s1LocalMetadata) &(s2InboxFull'=
          true) & (s1SameCount'=0) & (s1Phase'=updateClock);
10    [] s1Phase=transmit & !((s1Clock=s1NextBroadcast | s1Clock=(mod(
          s1NextBroadcast+refractoryPeriod,cycleLength))) & s1SameCount<sameThreshold)
           → (s1Phase'=clockCycleCheck);
11
12    // if clock ≤ cycle length then listen
13    [] s1Phase=clockCycleCheck & s1Clock<cycleLength → (s1Phase'=listen);
14    [] s1Phase=clockCycleCheck & !(s1Clock<cycleLength) → (s1Phase'=updateClock);
15
16    // if a message is overheard, then synchronize
17    [] s1Phase=listen & s1InboxFull → (s1InboxFull'=false) & (s1Phase'=
          synchronise1);
18    [] s1Phase=listen & !s1InboxFull → (s1Phase'=updateClock);
19
20    // if local clock > refractory period then adjust local clock to average
21    // of local clock and time in message
22    [] s1Phase=synchronise1 & s1Clock>=refractoryPeriod & diff ≤ floor(cycleLength
          /2) → (s1Clock'=s1avg1) & (s1Phase'=synchronise2a);
23    [] s1Phase=synchronise1 & s1Clock>=refractoryPeriod & diff >floor(cycleLength/2)
           → (s1Clock'=s1avg2) & (s1Phase'=synchronise2a);
24    [] s1Phase=synchronise1 & !( s1Clock>=refractoryPeriod ) → (s1Phase'=
          synchronise2a);
25
26    // if metadata > local metadata then local metadata := metadata
27    [] s1Phase=synchronise2a & s1InboxMeta>s1LocalMetadata → (s1LocalMetadata'=
          s1InboxMeta) &(s1Phase'=updateClock);
28    [] s1Phase=synchronise2a & !(s1InboxMeta>s1LocalMetadata) → (s1Phase'=
          synchronise2b);
29
30    // else if metadata < local metadata AND same count < 1 then transmit data now
31    [] s1Phase=synchronise2b & s1InboxMeta<s1LocalMetadata &s1SameCount<1 → (
          s2InboxClock'=s1Clock) &(s2InboxMeta'=s1LocalMetadata) & (s2InboxFull'=true)
            & (s1Phase'=updateClock);
```

```
32      [] s1Phase=synchronise2b & !(s1InboxMeta<s1LocalMetadata &s1SameCount<1) →
            (s1Phase'=synchronise2c);
33
34      // else if the metadata and the time are unchanged then increment sameCount
35      [] s1Phase=synchronise2c & s1InboxMeta=s1LocalMetadata &s1InboxClock=s1Clock
            & (s1SameCount<1) → (s1SameCount'=s1SameCount+1) &(s1Phase'=
            updateClock);
36      [] s1Phase=synchronise2c & s1InboxMeta=s1LocalMetadata &s1InboxClock=s1Clock
            & (s1SameCount>=1) → (s1Phase'=updateClock);
37      [] s1Phase=synchronise2c & !(s1InboxMeta=s1LocalMetadata &s1InboxClock=
            s1Clock) → (s1Phase'=updateClock);
38
39      // local clock = local clock + 1
40      [] s1Phase=updateClock & s1Clock=1 → (1−pClockDrift): (s1Clock'=s1Clock+1) & (
            s1Phase'=transmit)
41          + pClockDrift: (s1Clock'=s1Clock+2) & (s1Phase'=transmit);
42      [] s1Phase=updateClock & s1Clock<cycleLength & s1Clock!=1 → (s1Clock'=s1Clock
            +1) & (s1Phase'=transmit);
43      [] s1Phase=updateClock & s1Clock=cycleLength → (s1Clock'=0) & (s1SameCount
            '=0) & (s1Phase'=transmit);
44  endmodule
```

References

[1] L. D. Xu, W. He, S. Li, Internet of Things in industries: A survey, IEEE Transactions on Industrial Informatics 10 (4) (2014) 2233–2243. doi:10.1109/TII.2014.2300753.

[2] A. Whitmore, A. Agarwal, L. Da Xu, The Internet of Things—a survey of topics and trends., Information Systems Frontiers: A Journal of Research and Innovation 17 (2) (2015) 261.

[3] D. Koo, K. Piratla, C. J. Matthews, Towards sustainable water supply: Schematic development of big data collection using internet of things (iot), Procedia Engineering 118 (2015) 489 – 497, defining the future of sustainability and resilience in design, engineering and construction. doi:https://doi.org/10.1016/j.proeng.2015.08.465.

[4] A. ur Rehman, A. Z. Abbasi, N. Islam, Z. A. Shaikh, A review of wireless sensors and networks' applications in agriculture, Computer Standards & Interfaces 36 (2) (2014) 263–270.

[5] T. Ojha, S. Misra, N. S. Raghuwanshi, Wireless sensor networks for agriculture: The state-of-the-art in practice and future challenges, Computers and Electronics in Agriculture 118 (2015) 66–84.

[6] M. C. Vuran, A. Salam, R. Wong, S. Irmak, Internet of underground things in precision agriculture: Architecture and technology aspects, Ad Hoc Networks 81 (2018) 160 – 173. doi:https://doi.org/10.1016/j.adhoc.2018.07.017.

[7] M. Gerla, E. Lee, G. Pau, U. Lee, Internet of vehicles: From intelligent grid to autonomous cars and vehicular clouds, in: 2014 IEEE World Forum on Internet of Things (WF-IoT), 2014, pp. 241–246. doi:10.1109/WF-IoT.2014.6803166.

[8] S. M. R. Islam, D. Kwak, M. H. Kabir, M. Hossain, K. Kwak, The Internet of Things for health care: A comprehensive survey, IEEE Access 3 (2015) 678–708. doi:10.1109/ACCESS.2015.2437951.

[9] K. Nahrstedt, H. Li, P. Nguyen, S. Chang, L. H. Vu, 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, 2016, pp. 25–36. doi:10.1109/IoTDI.2015.41.

[10] S. A. Munir, B. Ren, W. Jiao, B. Wang, D. Xie, J. Ma, 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 - Volume 02, AINAW '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 113–120. doi:10.1109/AINAW.2007.257.

[11] M. Fisher, An Introduction to Practical Formal Methods Using Temporal Logic, Wiley, 2011.

[12] S. Yinbiao et al., Internet of Things: Wireless sensor networks, International Electrotechnical Commission White Paper (July 2014).

[13] K. Langendoen, A. Baggio, O. Visser, Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture, in: Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International, IEEE, 2006, pp. 8–pp.

[14] M. Fitting, First-Order Logic and Automated Theorem Proving, Springer, 1996.

[15] C. Baier, J.-P. Katoen, Principles of Model Checking, MIT Press, 2008.

[16] M. Kwiatkowska, G. Norman, D. Parker, PRISM 4.0: Verification of probabilistic real-time systems, in: Proc. 23rd International Conference on Computer Aided Verification (CAV'11), Vol. 6806 of LNCS, Springer, 2011.

[17] M. Kwiatkowska, G. Norman, D. Parker, PRISM: Probabilistic model checking for performance and reliability analysis, ACM SIGMETRICS Performance Evaluation Review 36 (4) (2009) 40–45.

[18] B. R. Haverkort, J.-P. Katoen, Performance and verification, SIGMETRICS Perform. Eval. Rev. 32 (4) (2005) 3–3. doi:10.1145/1059816.1059817.

[19] C. Baier, B. Haverkort, H. Hermanns, J.-P. Katoen, Model checking meets performance evaluation, SIGMETRICS Perform. Eval. Rev. 32 (4) (2005) 10–15. doi:10.1145/1059816.1059819.

[20] M. Kwiatkowska, G. Norman, D. Parker, Probabilistic model checking in practice: Case studies with PRISM, SIGMETRICS Perform. Eval. Rev. 32 (4) (2005) 16–21. doi:10.1145/1059816.1059820.

[21] M. Webster, M. Breza, C. Dixon, M. Fisher, J. McCann, Formal verification of synchronisation, gossip and environmental effects for wireless sensor networks, Electronic Communications of the EASST 76, Automated Verification of Critical Systems 2018 (AVoCS 2018) (2019). doi:10.14279/tuj.eceasst.76.1078.

[22] Z. Chen, D. Zhang, R. Zhu, Y. Ma, P. Yin, F. Xie, A review of automated formal verification of ad hoc routing protocols for wireless sensor networks, Sensor Letters (5) (2013).

[23] H. Kim, E. Kang, E. A. Lee, D. Broman, 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, 2017, pp. 147–158. doi:10.1145/3054977.3054980.

[24] A. Mouradian, I. Augé-Blum, Formal verification of real-time wireless sensor networks protocols with realistic radio links, in: Proceedings of RTNS 2013, 2013, pp. 213–222.

[25] L. Tobarra, D. Cazorla, F. Cuartero, Security in wireless sensor networks: A formal approach, in: From Problem Toward Solution: Wireless Sensor Networks Security, Nova, 2009, Ch. 8.

[26] M. Usman, V. Muthukkumarasamy, X.-W. Wu, Formal verification of mobile agent based anomaly detection in wireless sensor networks, in: 8th IEEE Workshop on Network Security, 2013.

[27] J. S. Dong, J. Sun, J. Sun, K. Taguchi, X. Zhang, Specifying and Verifying Sensor Networks: An Experiment of Formal Methods, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 318–337. doi:10.1007/978-3-540-88194-0_20.

[28] M. Fruth, Formal methods for the analysis of wireless network protocols, Ph.D. thesis, University of Oxford (2011).

[29] P. Gainer, S. Linker, C. Dixon, U. Hustadt, M. Fisher, The power of synchronisation: Formal analysis of power consumption in networks of pulse-coupled oscillators, in: J. Sun, M. Sun (Eds.), Formal Methods and Software Engineering, Springer International Publishing, Cham, 2018, pp. 160–176.

[30] P. Gainer, S. Linker, C. Dixon, U. Hustadt, M. Fisher, Investigating parametric influence on discrete synchronisation protocols using quantitative model checking, in: N. Bertrand, L. Bortolussi (Eds.), Quantitative Evaluation of Systems, Springer International Publishing, Cham, 2017, pp. 224–239.

[31] H. Pfeifer, D. Schwier, F. W. von Henke, Formal verification for time-triggered clock synchronization, in: 7th IFIP International Working Conference on Dependable Computing for Critical Applications (DCCA-7), 1999.

[32] R. Bakhshi, F. Bonnet, W. Fokkink, B. Haverkort, Formal analysis techniques for gossiping protocols, SIGOPS Oper. Syst. Rev. 41 (5) (2007) 28–36. doi:10.1145/1317379.1317385.

[33] A. Fehnker, P. Gao, Formal Verification and Simulation for Performance Analysis for Probabilistic Broadcast Protocols, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 128–141. doi:10.1007/11814764_12.

[34] M. Kwiatkowska, G. Norman, D. Parker, Analysis of a gossip protocol in PRISM, SIGMETRICS Perform. Eval. Rev. 36 (3) (2008) 17–22. doi:10.1145/1481506.1481511.

[35] B. R. Haverkort, M. Siegle, M. van Steen, Quantitative analysis of gossiping protocols, SIGMETRICS Perform. Eval. Rev. 36 (3) (2008) 2–2. doi:10.1145/1481506.1481508.

[36] J.-P. Katoen, How to model and analyze gossiping protocols?, SIGMETRICS Perform. Eval. Rev. 36 (3) (2008) 3–6. doi:10.1145/1481506.1481509.

[37] P. Crouzen, J. van de Pol, A. Rensink, Applying formal methods to gossiping networks with mCRL and Groove, SIGMETRICS Perform. Eval. Rev. 36 (3) (2008) 7–16. doi:10.1145/1481506.1481510.

[38] M. Mohsin, M. Sardar, O. Hasan, Z. Anwar, IoTRiskAnalyzer: A probabilistic model checking based framework for formal risk analytics of the Internet of Things, IEEE Access 5 (2017) 5494–5505.

[39] L. Baresi, G. Blohm, D. Kolovos, N. Matragkas, A. Motta, R. Paige, A. Radjenovic, M. Rossi, Formal verification and validation of embedded systems: the UML-based MADES approach, Software & Systems Modeling 14 (1) (2015) 343–363. doi:10.1007/s10270-013-0330-z.

[40] C. Boano, M. Zúñiga, J. Brown, U. Roedig, C. Keppitiyagama, K. Römer, 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, IEEE Press, 2014, pp. 95–106.

[41] C. Lenzen, P. Sommer, R. Wattenhofer, Optimal clock synchronization in networks, in: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems, SenSys '09, ACM, New York, NY, USA, 2009, pp. 225–238. doi:10.1145/1644038.1644061.

[42] D. Parker, PRISM Manual, Department of Computer Science, University of Oxford, http://www.prismmodelchecker.org/manual/Main/Welcome. Last accessed 5 Nov 2019. (November 2019).

[43] P. Levis, N. Lee, M. Welsh, D. Culler, TOSSIM: accurate and scalable simulation of entire TinyOS applications, in: Proceedings of the 1st International Conference on Embedded Networked Sensor Systems, ACM New York, NY, USA, 2003, pp. 126–137.

[44] A. e. a. Brandt, RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks, RFC 6550, last accessed 30 November 2019. (2012). doi:10.17487/rfc6550.
URL https://rfc-editor.org/rfc/rfc6550.txt

[45] M. Jelasity, A. Montresor, O. Babaoglu, Gossip-based aggregation in large dynamic networks, ACM Transactions on Computer Systems (TOCS) 23 (3) (2005) 219–252.

[46] M. Breza, Bio-inspired tools for a distributed wireless sensor network operating system, Ph.D. thesis, Imperial College, London (2013).

[47] S. Kartakis, W. Yu, R. Akhavan, J. A. McCann, Adaptive edge analytics for distributed networked control of water systems, in: Internet-of-Things Design and Implementation (IoTDI), 2016 IEEE First International Conference on, IEEE, 2016, pp. 72–82.

[48] G. Werner-Allen, G. Tewari, A. Patel, M. Welsh, R. Nagpal, Firefly-inspired sensor network synchronicity with realistic radio effects, in: Proceedings of the 3rd international conference on Embedded networked sensor systems, ACM New York, NY, USA, 2005, pp. 142–153.

[49] M. Breza, J. McCann, Lessons in implementing bio-inspired algorithms on wireless sensor networks, IEEE COMPUTER SOC, 2008, pp. 271–276.

[50] M. Breza, J. McCann, Polite broadcast gossip for IOT configuration management, in: 3rd International Workshop on Sensors and Smart Cities, IEEE, 2017.

[51] A. Verhoef, B. Choudhary, P. J. Morris, J. McCann, A high-density wireless underground sensor network (WUSN) to quantify hydro-ecological

interactions for a UK floodplain; project background and initial results, in: EGU General Assembly Conference Abstracts, EGU General Assembly Conference Abstracts, 2012, p. 6346.

[52] MEMSIC, Inc., MICAz wireless measurement system, http://www.memsic.com/userfiles/files/Datasheets/WSN/micaz_datasheet-t.pdf, last accessed 30 November 2019. (2019).

[53] Atmel Corporation, 8-bit Atmel microcontroller with 128KBytes in-system programmable flash, ATmega128/L Datasheet, www.atmel.com/images/doc2467.pdf., rev. 2467X-AVR-06/11. Last accessed 30 November 2019. (2019).

[54] E. M. Clarke, W. Klieber, M. Nováček, P. Zuliani, Model checking and the state explosion problem, in: Tools for Practical Software Verification, Springer, 2012, pp. 1–30.

[55] M. Webster, D. Western, D. Araiza-Illan, C. Dixon, K. Eder, M. Fisher, A. G. Pipe, A Corroborative Approach to Verification and Validation of Human–Robot Teams, International Journal of Robotics Research (2019). doi:10.1177/0278364919883338.

[56] M. Krämer, A. Geraldy, Energy measurements for MicaZ node, GI/ITG KuVS Fachgesprch Drahtlose Sensornetze (2006).

[57] N. N. Taleb, The Black Swan: The Impact of the Highly Improbable, Penguin Books, 2007.

[58] C. Dixon, A. F. Winfield, M. Fisher, C. Zeng, Towards temporal verification of swarm robotic systems, Robotics and Autonomous Systems 60 (11) (2012) 1429 – 1441, Towards Autonomous Robotic Systems 2011. doi:http://dx.doi.org/10.1016/j.robot.2012.03.003.

[59] J. Degesys, P. Basu, J. Redi, Synchronization of strongly pulse-coupled oscillators with refractory periods and random medium access, in: Proceedings of the 2008 ACM Symposium on Applied Computing, SAC '08, ACM, New York, NY, USA, 2008, pp. 1976–1980. doi:10.1145/1363686.1364164.

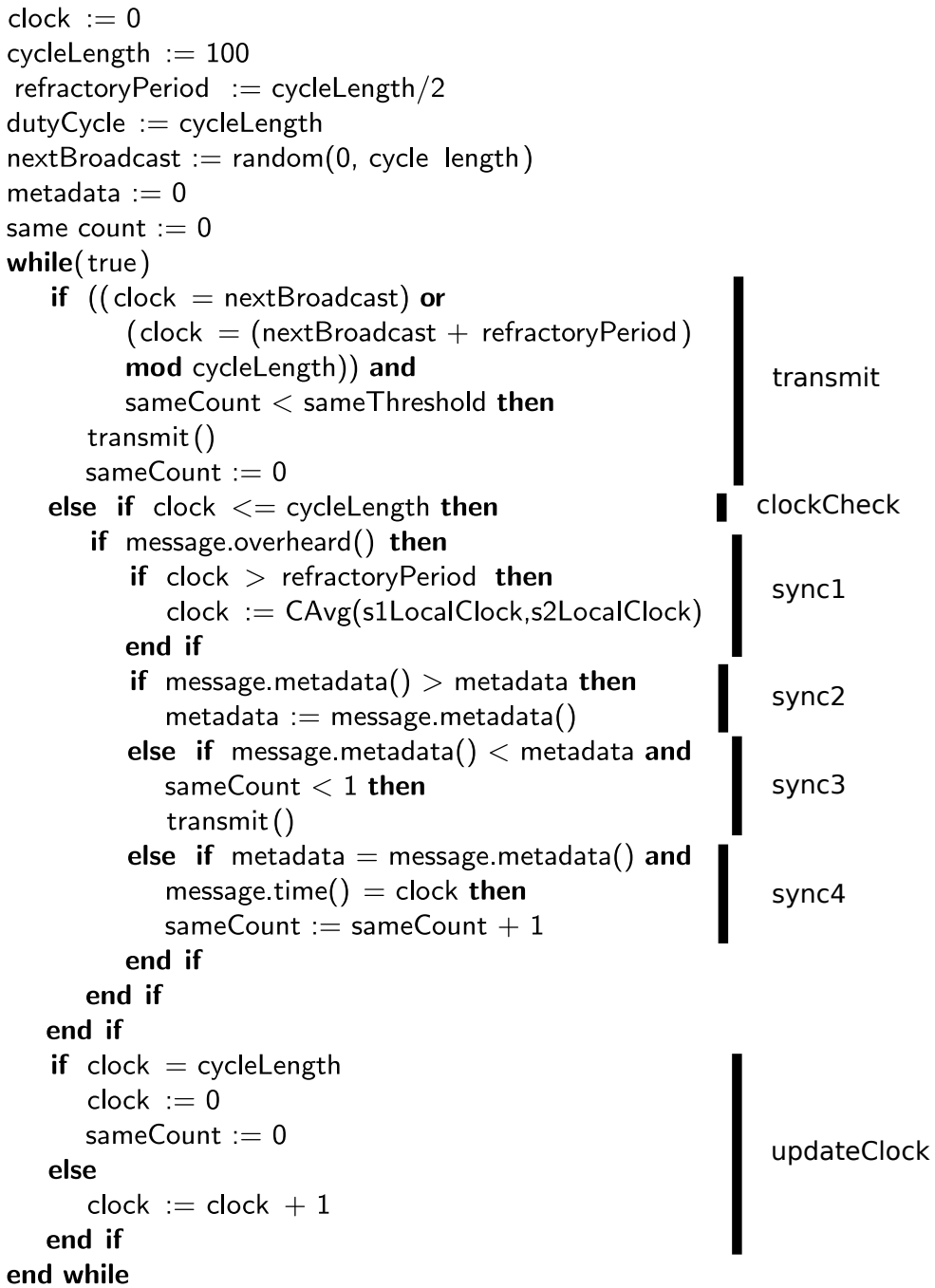


Figure 2: Phases of the FiGo Gossip–Synchronization Algorithm.

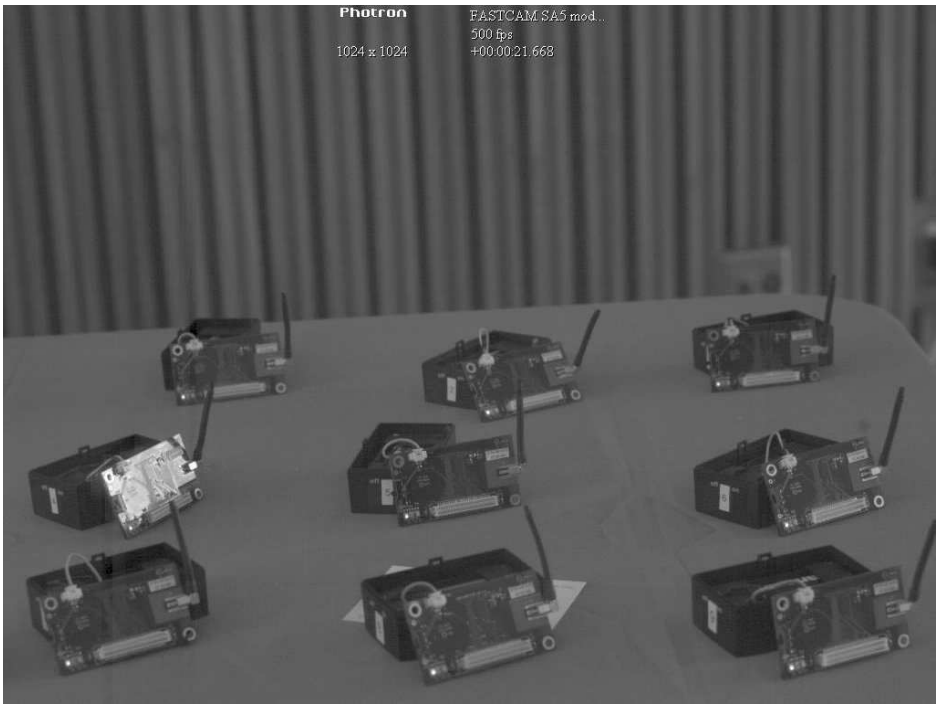


Figure 3: Nine MicaZ sensor nodes running the FiGo algorithm.

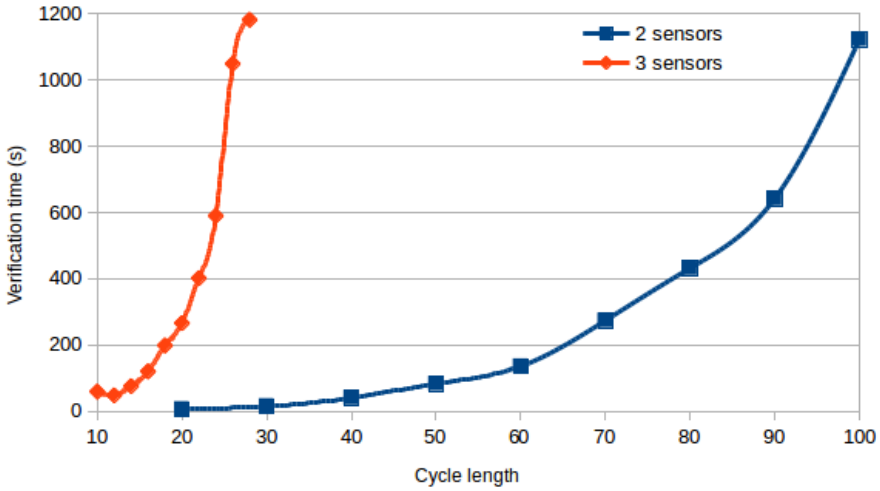


Figure 4: Total time for formal verification of Property 2 for 2– and 3–sensor networks.

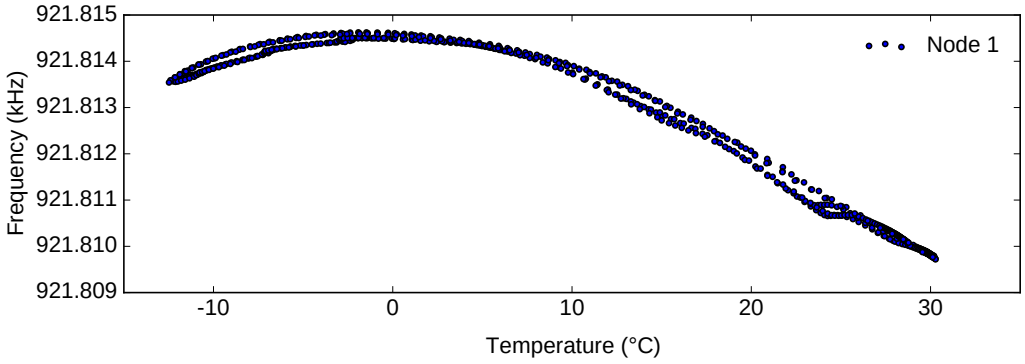


Figure 5: Hardware clock frequency for a Mica2 node for different ambient temperatures [41].

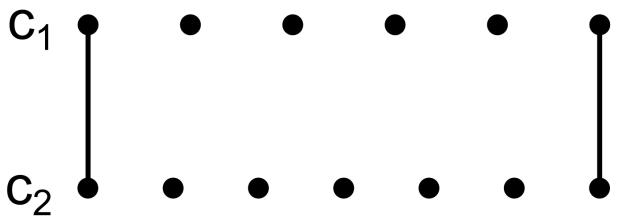


Figure 6: Clock re-synchronization. Ticks are shown as black circles.

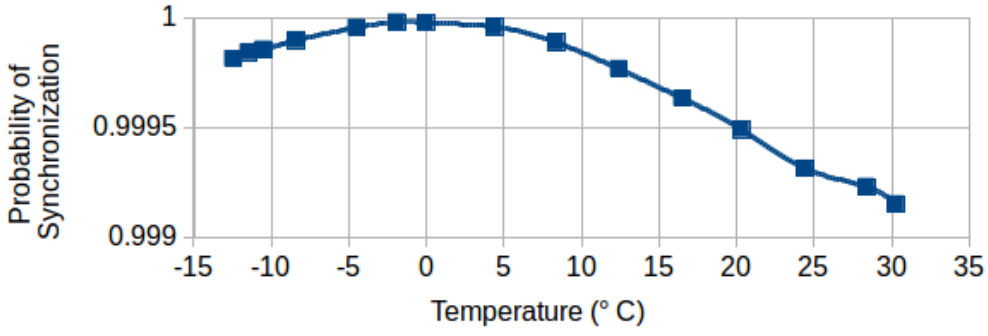


Figure 7: Steady-state probability of FiGo synchronization for varying temperatures of a second node.

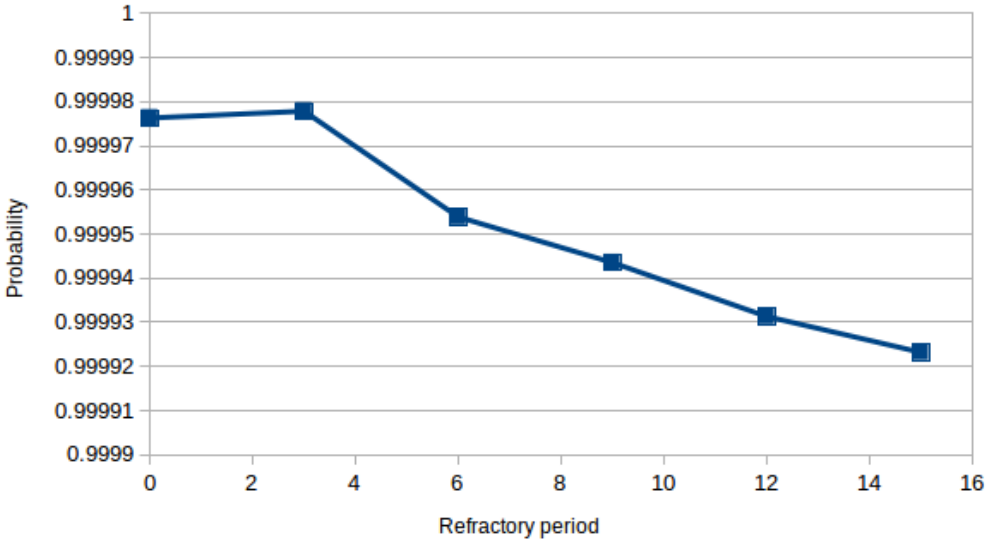


Figure 8: Median probability of synchronization for different refractory periods.

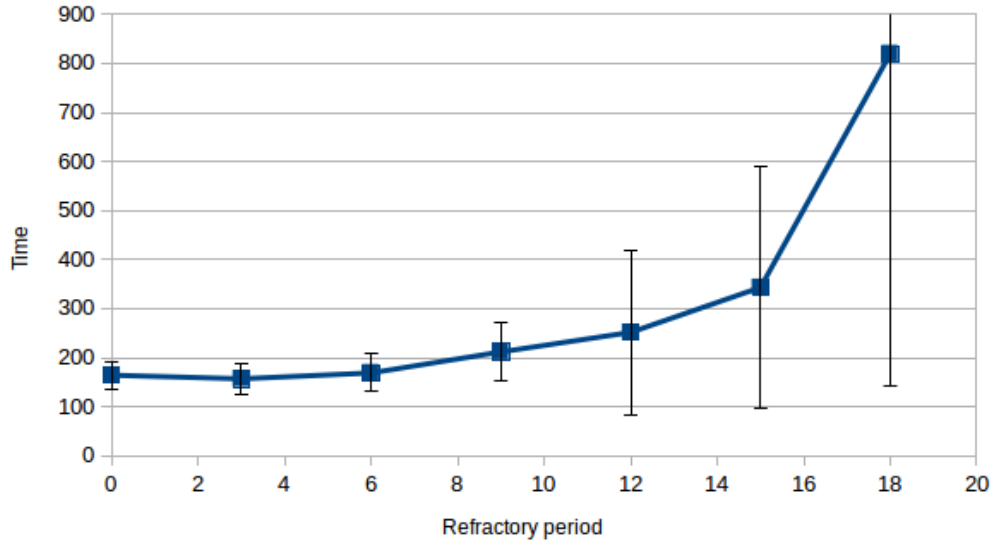


Figure 9: Mean synchronization time for different refractory periods.

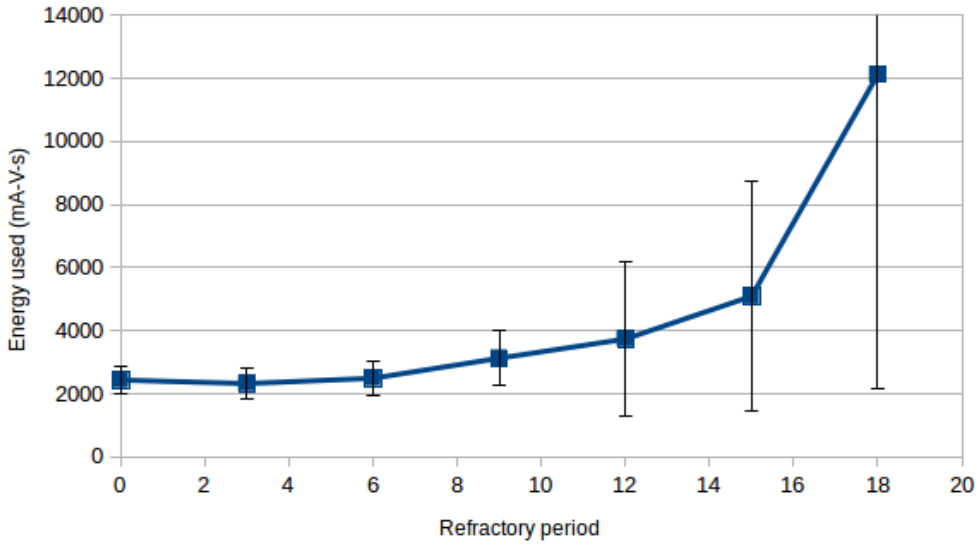


Figure 10: Mean energy consumption for different refractory periods.

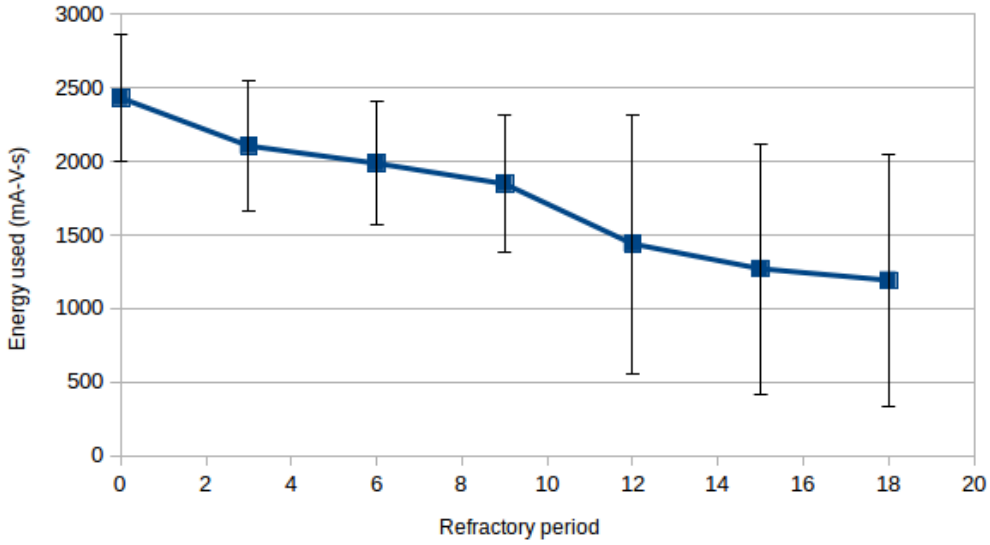
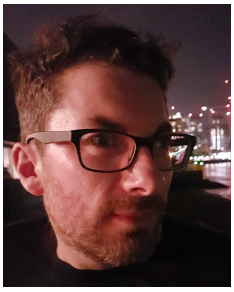


Figure 11: Reduced mean energy consumption with low-power mode.



Matt Webster is a senior postdoctoral researcher at the University of Liverpool. With over 15 years of academic and industrial research experience, his research aims to make computer systems safer and more reliable through the development and application of techniques from formal methods. His research interests include formal verification of the Internet of Things, verification of AI in space robotics, certification of autonomous unmanned aircraft, human-robot interaction, model-checking agent programming languages, computer security and artificial life.

Michael Breza completed his PhD at Imperial College London in 2013. His work focuses on ensuring the operational robustness and reliability of distributed sensor systems deployed in adverse environments such as space. He does this by studying the causes and rates of failures on distributed sensor systems. His current interest is in failure models of sensors and their communication networks, and the way that these failures influence the operation of systems that rely on their data. Example systems include control systems for water distribution and recycling networks, and industrial processes.

Clare Dixon is a Professor in the Department of Computer Science, University of Liverpool, Liverpool, U.K. Her research interests include formal verification and temporal and modal theorem-proving techniques, in particular applied to robotics and autonomous systems. She is a member of the Autonomy and Verification Laboratory and the Centre for Autonomous Systems Technology at the University of Liverpool. She is also a member of the British Standards Institution AMT/10 Committee on Robotics.

Michael Fisher holds a Royal Academy of Engineering Chair in Emerging Technologies in the Department of Computer Science of the University of Liverpool. He is the Director of the University's Centre for Autonomous Systems Technology and co-chairs the IEEE Technical Committee on the Verification of Autonomous Systems. His a Fellow of both BCS and IET, and is a member of both BSI and IEEE standards committees concerning Robotics and Autonomous Systems.

Julie McCann is Professor of Computer Systems at Imperial College London where leads the Resilient and Robust Infrastructure challenge part of the Data Centric Engineering theme in the Alan Turing Institute. She is PI for the NRF funded Singapore Smart Sensing project, the Logistics 4.0 project with PeTraS and the Tate Modern, Imperial PI for the EPSRC Science of Sensing Systems Software (S4) programme grant, and Co-Director of the Digital Oceans project with the China Shipbuilding Industry Corp. She is currently Director of the cross-Imperial Smart Connected Futures Network.