IOWA STATE UNIVERSITY
Digital Repository

6-22-2007

Verifying Fault-Tolerance of Sensor Network Applications Using Auto-generated Fault Injection Mechanisms

Youssef Wasfy Hanna
Iowa State University, youssef.hanna@gmail.com

Hridesh Rajan
Iowa State University

Recommended Citation

Verifying Fault-Tolerance of Sensor Network Applications Using Auto-generated Fault Injection Mechanisms

Abstract

The deployment scenarios for sensor networks are often hostile. These networks also have to operate unattended for long periods. Therefore, fault-tolerance mechanisms are needed to protect these networks from various faults such as node failure due to loss of power, compromise, etc and link failure due to network intrusion, etc. A number of fault-tolerance techniques have been developed specifically for wireless sensor networks. Verifying these fault-tolerant techniques is necessary for reliability and dependability checks. Formal methods such as model checking have been used for verification of such fault-tolerance mechanisms; however, building the models is a tedious job which makes model checking a hard task to accomplish. Techniques that allow model checking source code ease this task. These approaches automate the process of verification model construction. There are two aspects of automated verification model construction. First, a model of the application needs to be built. Second, a model of faults has to be created to expose problems with the application. In a previous work, we developed a framework, which we called Slede, to automatically extract PROMELA models from sensor network applications written in the nesC language. The contribution of this work is the design and implementation of a mechanism for automatically generating fault models from a partial specification of the application. By automatically generating fault models, our approach eases the verification of fault-tolerance for sensor network applications.

Verifying Fault-Tolerance of Sensor Network Applications Using Auto-generated Fault Injection Mechanisms

Youssef Hanna and Hridesh Rajan

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

Verifying Fault-Tolerance of Sensor Network Applications Using Auto-generated Fault Injection Mechanisms

Youssef Hanna
Dept. of Computer Science
Iowa State University
226 Atanasoff Hall
Ames, IA, 50011
ywhanna@iastate.edu

Hridesh Rajan
Dept. of Computer Science
Iowa State University
226 Atanasoff Hall
Ames, IA, 50011
hridesh@iastate.edu

ABSTRACT

The deployment scenarios for sensor networks are often hostile. These networks also have to operate unattended for long periods. Therefore, fault-tolerance mechanisms are needed to protect these networks from various faults such as node failure due to loss of power, compromise, etc and link failure due to network intrusion, etc. A number of fault-tolerance techniques have been developed specifically for wireless sensor networks. Verifying these fault-tolerant techniques is necessary for reliability and dependability checks. Formal methods such as model checking have been used for verification of such fault-tolerance mechanisms; however, building the models is a tedious job which makes model checking a hard task to accomplish. Techniques that allow model checking source code ease this task. These approaches automate the process of verification model construction. There are two aspects of automated verification model construction. First, a model of the application needs to be built. Second, a model of faults has to be created to expose problems with the application. In a previous work, we developed a framework, which we called *Slede*, to automatically extract PROMELA models from sensor network applications written in the nesC language. The contribution of this work is the design and implementation of a mechanism for automatically generating fault models from a partial specification of the application. By automatically generating fault models, our approach eases the verification of fault-tolerance for sensor network applications.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Formal Methods; D.2.4 [**Software/Program Verification**]: Model Checking; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Mechanical verification, Specification technique

General Terms

Reliability, Verification

Keywords

sensor networks, fault tolerance, model checking, slede, fault injection model generation, fault-tolerant systems

1. INTRODUCTION

Many informal and formal verification techniques are applied to verify the dependendability of fault-tolerance mechanisms. In one specific domain namely sensor networks, common verification techniques applied include simulating the system behavior using a commonly used simulator called TOSSIM [17], manual Fagan-style code inspection [8], and test runs of the protocol implementation on sensor network test beds. For example, Koushanfar *et al* demonstrated effectiveness of their fault-tolerance mechanism using canonical examples [16].

Partial but systematic informal verification can often reveal faults and improve confidence in program behavior; however, the problem with testing and simulation is that it does not necessarily provide a complete validation of the required goals due to the difficulty of ensuring that the test cases will cover all possible fault occurence scenarios [23]. More rigourous alternative is the use of formal methods such as model checking (e.g. [24, 3]) to verify whether a system satisfies its fault-tolerance requirements. Model checking as a verification technique has shown significant potential in recent years [1, 10].

There are two issues with applying model checking techniques to check fault-tolerance properties of sensor network applications. First, building the models is just too hard, often requiring significant model checking expertise from sensor network experts. Second, there may be discrepancies between a hand-written model and the actual implementation of the sensor network application, therefore there is no guarantee that the faults in the actual implementation are revealed by verifying the model [20]. Tools that verify source code such as Java path finder [13], Bandera [4], etc, make model checking more accessible to a broader audience.

In a previous work, we presented *Slede*, a domain-specific verification framework for sensor network security protocols implementations [12]. Slede allows PROMELA models [14] to be extracted from nesC [9] implementations of sensor network security protocols. Slede also featured an automatic intrusion model generator. The contribution of this work is an extension of Slede for more rigorous verification of fault-tolerance techniques utilized in sensor network applications. The key idea is to automatically generate fault models from partial specification of the application.

We provide an annotation language for describing the partial specification of the application. This description includes message structures, message sequencing, topology and objectives. Based on this description and a template for the fault injection model, our

tool generates a customized fault injection model for the sensor network application. The extracted application model, the auto-generated fault injection model and the verification objectives are then fed to the SPIN model checker [14] that verifies the model and gives a counter example in nesC (if any). We demonstrate the various aspects of our approach through a motivating example.

In the next section, we provide some necessary background on sensor networks and fault tolerance techniques in sensor networks. We also briefly describe the model extraction methodology and the verification framework *Slede* proposed by our previous work [12]. Section 3 illustrates the benefit of verifying sensor network application directly from the code through an example. Our approach for generating the fault injection models is described in Section 4. Section 5 discusses related work and Section 6 concludes.

2. BACKGROUND

In this section, we briefly describe key ideas in sensor networks, fault-tolerance techniques used in sensor networks and the nesC language, the dominant implementation language for sensor network paradigm [9].

2.1 Sensor Networks

A sensor network is a collection of low cost, small form factor, embedded devices called sensor nodes. A sensor node is often battery operated and therefore power constrained. A typical sensor node such as Berkeley Mote runs on two AA batteries and is often expected to operate for up to 3-6 months without maintenance. These nodes typically also have limited computational, communication and storage capacity.

In the last few years, wireless sensor networks have been deployed in both civil and military applications such as volcanic eruption monitoring, target monitoring, security and remote surveillance [7]. These networks are often deployed unattended for long periods of time. Therefore, it is important that fault-tolerance mechanisms are put in place to guard against physical failures such as battery running out, etc and malicious outside behaviors such as network intrusion.

2.2 Fault-tolerance Techniques for Sensor Network Applications

Fault-tolerance mechanisms such as double and triple fault-tolerance techniques have proven to be effective for traditional environments, where for all practical purposes it is assumed that operating nodes have no limits on storage space, bandwidth and computational power.

These assumptions about storage space, bandwidth, and computational power no longer hold in resource constrained environments such as sensor networks. Therefore, traditional fault-tolerance techniques are not very effective in these environements [16]. Many fault-tolerance techniques have been developed for the wireless sensor networks. For example, Gupta and Younis [11] proposed a runtime recovery mechanism for failed sensors in failed clusters that avoids full-scale re-clustering based on consensus of healthy gateway which allows detection and handling of faults in one faulty gateway. Koushanfar *et al* [16] proposed a heterogenous back-up scheme where one type of sensors is substituted with another.

2.3 The nesC Language

nesC [9] is an extension of the C language designed to develop sensor network applications. nesC applications consist of modules, interfaces and configurations. nesC modules are similar to early Ada and ML modules in that they cannot be instantiated, but

```
module CompM {
  provides interface StdControl;
  uses interface Timer;
}
implementation {
  command result_t StdControl.init() {...}
  event result_t Timer.fired() {...}
}

configuration Comp {
}
implementation {
  components Main, CompM, SingleTimer;
  Main.StdControl -> CompM.StdControl;
  CompM.Timer -> SingleTimer.Timer;
  ...
}
```

Figure 1: A NesC Example

they serve as containers. A module can contain state declarations (shared by other elements of the modules), command declarations (methods) and event handlers. An event handler is similar to a method; yet, it is executed only when the event is triggered. An interface is a collection of related commands/events. A module that provides an interface has to implement its commands, while a module that uses an interface has to implement its events.

An example module in nesC is shown in Figure 1. Module CompM provides interface StdControl, so it has to implement the interface commands (e.g. StdControl.init()). CompM also uses the interface Timer, so it has to implement its events (e.g. Timer.fired). A configuration component is responsible for connecting the components that are using interfaces to the components that provide their implementation. For example, component Main uses interface StdControl and is wired to component CompM. Every application has a single top-level configuration.

2.4 Slede: A Domain Specific Verification Framework for Sensor Networks

In our previous work, we proposed Slede, a domain specific verification framework for sensor networks [12]. An overview of Slede is presented in Figure 2. Slede features new mechanisms for extracting PROMELA models from nesC implementations and for generating intrusion models from protocol specifications.

Slede accepts the complete nesC language. The front-end generates an abstract syntax tree of the protocol implementation, which is then passed to the protocol model generator that is responsible for automatically extracting verifiable PROMELA models [14].

The protocol implementation is translated as a whole into one PROMELA process that is a global object describing the behavior of the protocol. Every node is represented as an instance of the generated process. To generate a finite-state model, we bound the number of principals involved in the protocol, and provide an annotation language to specify a verification topology (see [12] for details on our annotation language).

During the protocol model generation, all system calls (calls to the libraries of TinyOS) are replaced by calls to environment models provided as Slede's library. Slede's library provides PROMELA models for sending and receiving messages, LED manipulation, etc. The pointer arithmetic is translated to embedded C code inside the PROMELA model and the event constructs in nesC are modeled by inserting a checking statement between every two statements in the model, where an event handler may be executed [15].

The generated model containing the model of the protocol implementation, the intrusion model and the environment models are given as inputs to the SPIN model checker [14], which verifies
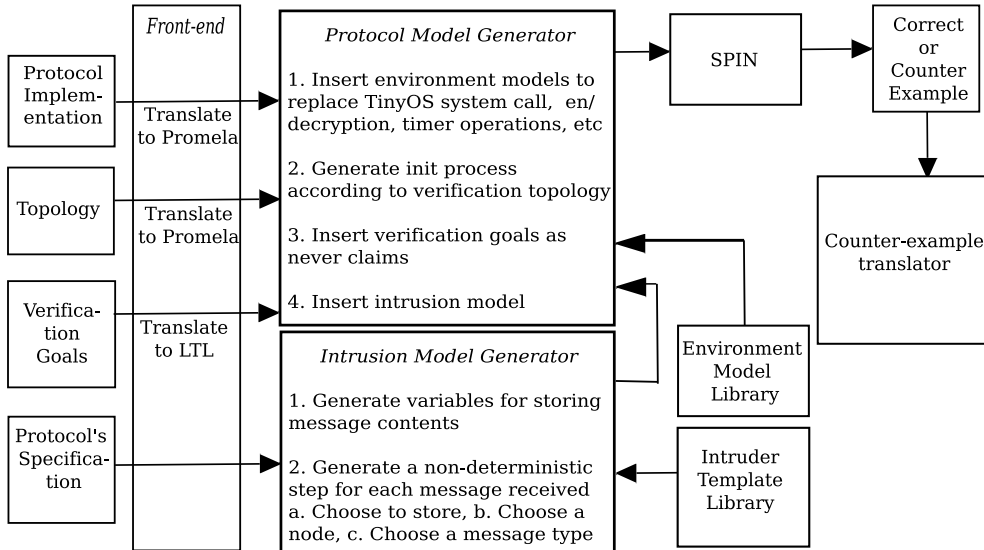
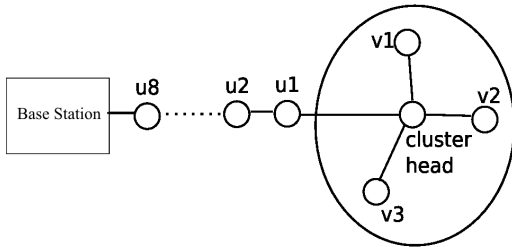**Figure 2: Overview of the Slede Verification Framework [12]**

Figure 3: An Example Network with a 4 nodes cluster (*t = 3*) [25]

whether the model violates the objectives which have been translated into LTL formulas [21]. If the objectives are satisfied, the protocol is verified as secure. Otherwise, SPIN produces a counter example that violates the objectives of the fault-tolerance mechanism. This counter example is then translated to a sequence of nesC statements. The verification may not terminate if the PROMELA model is too large.

3. MOTIVATING EXAMPLE

In this section, we give an example of a fault-tolerance mechanism for sensor networks and show the benefit of verifying it from source code as opposed to verification from abstract specification.

Zhu *et al* [25] propose a mechanism that protects the base stations of sensor networks from malicious faults. The protocol allows the base station of a sensor network to detect false injected data that may be injected from nodes compromised by the adversary, when deploying the sensor network in a military application for example. Such an attack is usually known as *false data injection attack*. The base station can verify the authentication of information it receives as long as the number of compromised nodes is at most t, where $t + 1$ is the size of the smallest nodes cluster in the network. Moreover, their scheme tries to filter out the false injected data before reaching the base station, thus saving the energy of transmitting false data from one node to another.

Figure 3 illustrates an example of a network where $t = 3$. In this example, there is only one cluster of size ($t + 1$) with the nodes *v1, v2, v3* and *cluster head*. If 3 of the 4 nodes of the cluster injected false data, then the base station will be able to detect the false information. On the other hand, if the 4 nodes of the cluster emit false data, then the number of compromised nodes has exceeded the threshold t, rendering the base station unable to detect such an attack.

Verifying the dependability of fault-tolerance scheme such as this one using process algebra or model checking has proven to be an efficient approach. In order to verify this scheme using traditional techniques, the user has to specify the system, the fault injection model as well as the failing behaviour for every fault in the process algebra or model checking specification language. For example, a state representing a crash extends the behavior of the system by allowing the system to move to this state.

These fault injection models are dependent on the specification of the system under verification. For example, Bernardeschi *et al* [2] model the system as a set of CCS processes, and faults are modelled directly by actions of the processes themselves. One consequence of such dependency is that a change in the specification of the system will require manually changing the fault injection model accordingly, which is tedious and error-prone, specially if the fault injection model developer is not the system model implementer.

We believe that automatic generation of fault injection models from the implementation and specification provided as comments in the code solves this problem. The fault models follow a fixed pattern independent of the application, thus alleviating the user from worrying about changing them when changing the implementation. We describe our approach for fault injection model generation in the next section.

4. AUTOMATED FAULT INJECTION MODEL GENERATION

In our approach, fault models are generated as intruders to the network. Slede automates this process. Intruder models follow the Dolev-Yao style intrusion model [6]. An intruder in this model can intercept all messages and modify on their contents.

The intruder can provide verification of dependability of the fault-tolerance mechanisms. For example, in the authentication scheme described in the previous section, the goal is to guarantee that the base station will detect any injected false data packets when no more than a certain number of *t* nodes are compromised. Fault generation in the verification of such protocol will be presented by different intruders that can intercept messages from the legitimate nodes. Not forwarding the messages will be a representation of the node compromise. Injecting false data will be emulated by the intruder that can create a message with random values or modify on the content of the intercepted message. The protocol should satisfy its goal (in this example that it detects false injected data) with the presence of such intruders who provide fault actions.

```
1  bit knowsNonceA;
2  active proctype Intruder() {
3   Crypt data, saved;
4   mtype msg;

6   do
7   :: network? msg,_,data -> /* Msg Intercepted */
8      if /* perhaps store the message */
9      :: saved = data;
10     :: skip;
11     fi;

13  :: /* Replay or send message */
14     if /* choosing message type */
15     :: msg = msg1;
16     ...
17     fi;
18     if /* choosing recipient */
19     :: recipient = agentA;
20     ...
21     fi;
22     if /* replay saved msg or assemble it */
23     :: data = saved;
24     :: if
25        :: knowsNonceA -> data.info = nonceA;
26        :: data.info = 0;
27        fi;
28     fi;
29     network ! (msg,rcpt,data);
30  od;
31 }
```

Figure 4: Intruder Model Pattern Based on [19]

The pattern used for the fault model generation is based on the intruder model described in [19]. The structure of the intruder in PROMELA is shown in Figure 4. There is one loop (lines 6-30) where the intruder either intercept the message (line 9), or send either the intercepted message (line 23) or a new one (lines 24-27). The intruder is allowed to save only one message, and knows about the structure of the messages sent.

In order to inject faults in the protocol, the intruder requires some knowledge about the protocol such as message construct and message sequencing. Both the construct and sequencing of the messages are almost always available. This knowledge will help reduce the size of the model, as opposed to an intruder that generates purely random values which might lead to state explosion.

This information is presented as comments in the protocol implementation. An example message declaration for the fault-tolerance mechanism described in the previous section is given in Figure 5 (lines 2-5), where a message type `Hello` is defined. This message type in specification is mapped to the structure `helloMsg` in the implementation, which is sent by the base station at the beginning

```
1  /*@
2  @ message Hello mapsto helloMsg{
3  @   int sender mapsto src;
4  @   ...
5  @ }
6  @ message Data mapsto dataMsg{
7  @   int data mapsto ev;
8  @   ...
9  @ }
10 ... // other message structures
11 @ node SensorM CH{ }
12 @ node SensorM V1{ CH; }
13 @ node SensorM V2{ CH; }
14 ... // other nodes connections
15 @ node SensorM BS{ U8; }
16 @ protocol p {
17 @   (SensorM, SensorM, Hello)
18 @   (SensorM, SensorM, dataMsg)
19 ... // rest of message sequencing
20 @ }
21 @ Leds.greenOn()
22 @*/
```

Figure 5: An Example Verification Configuration

of the protocol. The example defines that a message of this type will contain an integer field representing the sender of the message. This field is mapped to the field `src` of the structure `helloMsg`.

The topology of the protocol show which nodes are connected to each other (lines 11-15). Lines 16-18 illustrate the message sequencing of the protocol. Finally, the objective of the protocol is described as reaching a state where the green LED is turned on (line 21), an action done by the base station when it ensures that the data received from the cluster is valid.

5. RELATED WORK

Verification of fault-tolerant systems is not a new topic. Bernardeshi *et al* [3] proposed to verify fault-tolerant systems based on CSS/Meije process algebra [5]. They model the system as a set of processes communicating with each other and interacting with environment using actions, and model fault actions directly by actions of the processes themselves. This requires that the fault actions are dependent on the implementation of the model, which requires manual resolving. On the other hand, in our approach no manual efforts are needed.

Schneider *et al* [24] proposed verifying fault-tolerant systems using model checking. They define fault behavior of a faulty process at its interface with the system and automate the generation of all possible failure s cenarios. This is done at the design level. Our approach is similar in spirit, but works at the implementation level. The advantage of Schneider's approach is that they can detect errors early, in the design itself. The advantage of our approach is that, even in cases where the implementation does not mirror the design, we will be able to find faults.

Liu and Joseph [18] have used a single notation and model to specify fault-tolerance, schedulability as well as timing. This is also done at the design level, similar to Schneider's approach. Our approach does not yet provide schedulability; however, it has advantages similar to that over Schneider's approach.

Rushby [22] verifies time-triggered systems from algorithm specified as functional programs using the PVS verification system. Their approach requires knowledge of the PVS verification system. Our approach does not require the user to be aware of any verification system. Instead, it hides the details of the verification

system behind an annotation language, which is very similar to the domain-specific language nesC.

6. CONCLUSION

In this work, we presented our approach for automatic generation of fault injection models. Our work builds on our previous work, where we presented *Slede* a verification framework for sensor network security protocol implementations. The key advantage of Slede is that it automatically extracts verifiable models from nesC implementations. The approach described in this work auto-generates fault injection models from partial specification of the system, provided an annotations in the implementation itself. Our approach thus makes it easier to maintain correspondence between the fault models and system models, which is hard to achieve in manual verification methods. It also brings the advantages of explicit-state model checking to the verification of fault tolerance of sensor network applications.

Acknowledgments

7. REFERENCES

[1] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, New York, NY, USA, 2002. ACM Press.

[2] C. Bernardeschi, A. Fantechi, and S. Gnesi. Model checking fault tolerant systems. *Software Testing, Verification, and Reliability*, 12:251–275, December 2002.

[3] C. Bernardeschi, A. Fantechi, and L. Simoncini. Formally verifying fault tolerant system designs. 43(3):191–205, 2000.

[4] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Preanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 439–448, New York, NY, USA, 2000. ACM Press.

[5] G. B. Didier Austry. Algébre de processus et synchronisation. Technical Report Rapports de Recherche No.187, Institut National de Recherche en Informatique et en Automatique, 1983.

[6] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12):198–208, mar 1983.

[7] D. Estrin, W. Michener, G. Bonito, and T. W. Participants. A report from a national science foundation sponsored workshop: Environmental cyberinfrastructure needs for distributed networks. Technical report, Scripps Institute of Oceanography, Aug 2003.

[8] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Syst. J.*, 38(2-3):258–287, 1999.

[9] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the 2003 conference on Programming language design and implementation*, pages 1–11, 2003.

[10] P. Godefroid. Model checking for programming languages using verisoft. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–186, New York, NY, USA, 1997. ACM Press.

[11] G. Gupta and M. Younis. Fault-tolerant clustering of wireless sensor networks. 3:1579–1584, March 2003.

[12] Y. Hanna and H. Rajan. Slede: A domain specific verification framework for sensor network security protocol implementations. Technical Report 07-09, Computer Science, Iowa State University, 2007.

[13] K. Havelund and G. Rosu. Monitoring Java programs with Java PathExplorer. In K. Havelund and G. Roşu, editors, *Runtime Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 23July 2001.

[14] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[15] G. J. Holzmann and M. H. Smith. A practical method for verifying event-driven software. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 597–607, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[16] F. Koushanfar, M. Potkonjak, and A. Sangiovanni-Vincentelli. Fault tolerance techniques in wireless ad-hoc sensor networks. In *Sensors 2002*, pages 1491–1496, 2002.

[17] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137, New York, NY, USA, 2003. ACM Press.

[18] Z. Liu and M. Joseph. Specification and verification of fault-tolerance, timing, and scheduling. *ACM Trans. Program. Lang. Syst.*, 21(1):46–89, 1999.

[19] S. Merz. Model checking: A tutorial overview. In F. C. et al., editor, *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 3–38. Springer-Verlag, Berlin, 2001.

[20] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. Cmc: a pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev.*, 36(SI):75–88, 2002.

[21] A. Pnueli. The temporal logic of programs. In *The 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57, New York, 1977. IEEE.

[22] J. Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. In M. D. Cin, C. Meadows, and W. H. Sanders, editors, *Dependable Computing for Critical Applications—6*, volume 11, pages 203–222, Garmisch-Partenkirchen, Germany, 1997. IEEE Computer Society.

[23] F. Schneider, S. Easterbrook, J. R. Callahan, and G. J. Holzmann. Validating requirements for fault tolerant systems using model checking. In *ICRE '98: Proceedings of the 3rd International Conference on Requirements Engineering*, pages 4–13, Washington, DC, USA, 1998. IEEE Computer Society.

[24] W. Steiner, J. Rushby, M. Sorea, and H. Pfeifer. Model checking a fault-tolerant startup algorithm: From design exploration to exhaustive fault simulation. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, page 189, Washington, DC, USA, 2004. IEEE Computer Society.

[25] S. Zhu, S. Setia, S. Jajodia, and P. Ning. An interleaved hop-by-hop authentication scheme for filtering of injected false data in sensor networks. *sp*, 00:259, 2004.