Run Time Assurance of Application-Level Requirements in Wireless Sensor Networks

Yafeng Wu, Krasimira Kapitanova, Jingyuan Li,
John A. Stankovic, Sang H. Son, and Kamin Whitehouse
Department of Computer Science, University of Virginia
{yw5s, krasi, jl3sz, stankovic, son, whitehouse}@virginia.edu

ABSTRACT

Continuous and reliable operation of WSNs is notoriously difficult to guarantee due to hardware degradation and environmental changes. In this paper, we propose and demonstrate a methodology for *run-time assurance* (RTA), in which we validate at run time that a WSN will function correctly, despite any changes to the operating conditions since it was originally designed and deployed. We use program analysis and compiler techniques to facilitate automated testing of a WSN at run time. As a proof of concept, we implemented a framework for designing and automatically testing WSN applications. We evaluate our implementation on a network of 21 TelosB nodes, and compare performance with an existing network health monitoring solution. Our results indicate that in addition to providing the application-level verification function, RTA misses 75% fewer system failures, produces 70% fewer maintenance dispatches, and incurs 33% less messaging overhead than network health monitoring.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems

General Terms

Design, Experimentation, Performance

Keywords

Wireless sensor networks, Petri Nets, code generation, automatic test generation, run time application validation

1. INTRODUCTION

Emerging wireless sensor network (WSN) technologies are applicable to a wide range of mission-critical applications, including fire fighting and emergency response, infrastructure monitoring, military surveillance, and medical applications. These applications must operate reliably and continuously due to the high cost of system failure. However, continuous and reliable operation of WSNs is notoriously difficult to guarantee due to hardware degradation and environmental changes, which can cause operating conditions that were impossible for the original system designers to foresee. This is particularly true for applications that operate over long time durations, such as a building monitoring system that must operate for the lifetime of the building. Wireless noise and interference may change dramatically as new wireless technologies are developed and deployed in or near a building, and sensor readings and network topology may change as the occupancy, activities, and equipment in a building evolve over time.

In this paper, we propose and demonstrate a methodology for *run-time assurance* (RTA), in which we validate at run time that a WSN will function correctly in terms of meeting its high-level application requirements, irrespective of any changes to operating conditions since it was originally designed and deployed. The basic approach is to use program analysis and compiler techniques to facilitate automated testing of a WSN at run time. The developer specifies the application using a high-level specification, which is compiled into both (i) the code that will execute the application on the WSN, and (ii) a set of input/output tests that can be used to verify correct operation of the application. The test inputs are then supplied to the WSN at run time, either periodically or by request. The WSN performs all computations, message passing, and other distributed operations required to produce output values and actions, which are compared to the expected outputs. This testing process produces an end-to-end validation of the essential application logic.

RTA differs from network health monitoring, which detects and reports low-level hardware faults, such as node or route failures [21, 26]. The end-to-end application-level tests used for RTA have two key advantages over the tests of individual hardware components used for health monitoring: 1) fewer false positives - RTA does not test nodes, logic, or wireless links that are not necessary for correct system operation, and therefore produces fewer maintenance dispatches than health monitoring systems; 2) fewer false negatives - a network health monitoring system will only validate that all nodes are alive and have a route to a base station, but does not test more subtle causes of failure such as topological changes or clock drift. In contrast, the RTA approach tests the ways that an application may fail to meet its high-level requirements because it uses end-to-end tests. Network health monitoring improves system reliability by detecting

IPSN'10, April 12–16, 2010, Stockholm, Sweden.

some types of failures, but stops short of actually validating correct system operation. The goal of RTA, instead, is to provide a positive affirmation of correct application-level operation.

We have implemented a framework for designing and automatically testing WSN applications using the RTA methodology. The developer specifies the application using a high-level Sensor Network Event Description Language (SNEDL) [10], which is an extended Petri net model. Our system compiles the SNEDL model down to TinyOS [15] code that runs on the Telos nodes [3], as well as tests the defined mappings between sensor input values and system outputs. We use program analysis techniques to identify the minimal set of tests that will cover the essential application logic. This analysis uses new techniques that exploit information about network topology and the redundancy of nodes based on sensing range, and builds on existing techniques to cover all execution paths in the program [13]. Once the minimal set of tests has been identified, our system deploys the TinyOS code onto the network and periodically executes the tests. This implementation serves as a proof of concept of our RTA methodology, which can also be applied more generally to other programming models besides SNEDL.

We evaluate our implementation by designing a fire detection system and executing it on a network of 21 TelosB nodes. We artificially introduce failures into the system, including node failures and location errors, and compare the performance of RTA to that of an existing health monitoring solution [26]. Our results indicate that RTA misses 75% fewer system failures and also produces 70% fewer maintenance dispatches than health monitoring. Furthermore, our program analysis techniques reduce the number of tests required such that RTA incurs 33% less messaging overhead than health monitoring. The main contributions of this paper are: 1) A novel RTA methodology that positively affirms correct system operation at run time; 2) A prototype implementation based on the SNEDL description language; 3) New analysis techniques exploiting network topology and sensing redundancy to reduce the number of necessary tests; 4) A quantitative evaluation of our RTA methodology and implementation, and comparison with an existing network health monitoring system.

2. RELATED WORK

Although testing has always been a major part of software development, a very limited amount of work has been done in the area of testing WSN applications. This is partially due to a few characteristics of WSN applications such as operating in concurrent, event-based, and interrupt-driven manner, which considerably complicates the development of code representation. Nguyen et al. [18] proposed application posting graphs to represent behaviors of WSN applications. Regehr [22] designed a restricted interrupt discipline to enable random testing of nesC programs. Lai et al. [14] studied inter-context control-flow and data-flow adequacy criteria in nesC programs. However, all previous work is intended for testing applications prior to deployment when the size of the test suite is not as critical. Therefore, WSN-specific approaches for decreasing the size of the number of necessary tests have not been considered until now.

There is a great array of fault tolerance and reliability techniques developed over the last 50 years many of which have been applied to WSNs [2, 19, 28, 34]. We expect that any WSN that must operate with high confidence will utilize many of these schemes. However, most existing approaches, such as eScan [36] and CODA [31], aim to improve the robustness of individual system components. Therefore, it is difficult to use such methodologies to validate the high-level functionality of the application. Similarly, self-healing applications [8, 6, 35], although attempting to provide continuous system operation, are not capable of demonstrating adherence of the system to key high-level functional requirements.

Debugging WSN applications is a complicated process and many different approaches exist. Marionette [32] and Clairvoyant [33] are source-level debuggers allowing access to source-level symbols. MDB [29] supports the debugging of microprograms. SNTS [23], Dustminer [12], and LiveNet [25] use overhearing to gain visibility into the network operations. Some debugging approaches are based on invariants [7], others attempt to use data mining to discover hard to find bugs [11]. EnviroLog [17] uses a *record and replay service* where it stores and replays the I/O on the sensor nodes. However, all of these debugging mechanisms are either used prior deployment or in a *post mortem* manner, where data about the application is collected and then analyzed offline. Therefore, these techniques do not provide a way to monitor and analyze the application behavior at run time.

Many applications have been developed to achieve sensor network hardware verification. Sympathy [21], for example, is concerned with detecting routing problems. Health-monitoring systems such as MANNA [27], LiveNet [25], and Memento [26] employ sniffers or specific embedded code to monitor the status of a system. However, such applications monitor low-level components of the system instead of high-level application requirements. Therefore, they cannot be used as a substitute for our RTA framework. Instead, if available, such applications could be used as a monitoring component. Information from these systems could be used for RTA checks or to activate further system state checks.

There are very few overall system-management systems for WSNs [1, 30]. Most of them manage only a few system properties such as energy [9], topology, or bandwidth. However, we have not found any that address RTA for high confidence systems in terms of application-level requirements.

3. RTA METHODOLOGY

The RTA methodology is built around the following three principles:

Run time verification: The RTA principle requires that a system demonstrates at run time that it is able to perform its key services. Currently, testing and debugging techniques are used to fully test the system at design time, and deployment-time validation [16] is used to verify the system during deployment. However, due to the changing environment and the dynamic nature of failures, we argue that even when these techniques have been employed, RTA is still necessary.

Application-level guarantees: Many sensor networks are complicated systems, consisting of numerous components and protocols. Each component may use various fault tolerance, self-healing, or other reliability mechanisms to operate robustly. However, even if one can guarantee that each separate component works correctly, the system may still fail to perform some high-level operations. And a user (such as a fire inspector) is only concerned with whether the system performs the way they want, rather than whether each component works correctly. The goal of the RTA principle is to address this and focus on verifying the application-level services.

Correctness demonstration: There are different ways to demonstrate services at run time. One option is to monitor system health information and infer system correctness from this information. Such health monitoring techniques are shown to be effective for certain applications with regular traffic [4, 24, 25]. However, many critical events, such as fire or volcano eruptions, are rare. Also, in complex WSN systems, it is hard to determine which states should be monitored and how to infer the correctness of services. Monitoring too many states is inefficient and may cause many false positives, but monitoring too few states could fail to reveal failures. Memento [26] uses a periodic heart beat method to determine if a node is still alive. This approach is not suitable for RTA for two main reasons: 1) node responsiveness does not imply proper functioning of the system and its application semantics; 2) node failure does not imply the failure of the overall system or the application. If we have some level of node redundancy, a small number of node failures may not affect the system application at all. Thus, in general, health monitoring techniques are not sufficient to provide confidence in application-level requirements for WSNs at run time. To address this, we employ testing to verify the proper operation of the application. Using tests allows us to check if the application provides the results we expect regardless of what the state of its components might be.

4. IMPLEMENTATION FRAMEWORK

Figure 1 shows how the components of the RTA framework interact with each other. The framework needs three inputs: the SNEDL model of the application logic, the test specification, and the topology of the network. Both the automated code generation and the automated test generation mechanisms need the SNEDL application model. Note that the RTA framework is flexible enough to use other modeling languages, as long as they are able to clearly and unambiguously define the application-level behavior of the system. The test generation mechanism also needs information about what tests the user wants to run (the test specification), and how many and what nodes there are in the regions of the network that will be tested (the network topology). After the code for the nodes has been generated and deployed, and the proper sets of tests have been created, the RTA execution mechanism can start monitoring the application's functionality by running RTA tests on the sensor network.

4.1 The SNEDL Programming Model

SNEDL [10] is the first event specification language to support key features of WSNs. SNEDL can capture the structural, spatial, and temporal properties of a complex event detection system, which can be used to assist system designers to identify inconsistencies and potential problems. As a description language, it is an extension of Petri nets that combines features of Timed, Color, and Stochastic Petri nets. These additional features make SNEDL a Turing complete language [20].

Figure 2 shows an example SNEDL model of a WSN application. The application monitors the temperature and humidity levels and signals if they go above some predefined thresholds. The SNEDL model consists of places (circles), transitions (rectangles or bars), directed arcs, and tokens (dots inside places).

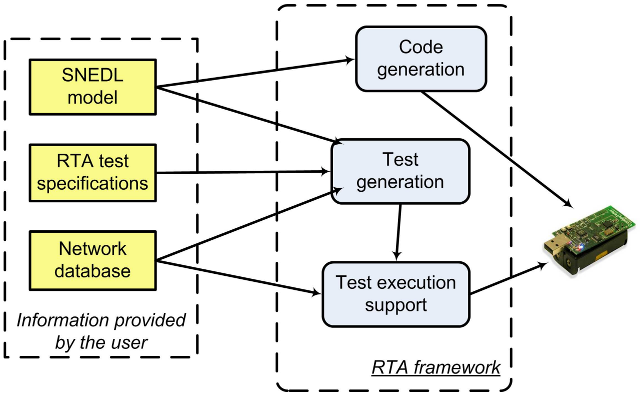


Figure 1: The main components of the RTA framework are the automatic code generator, the automatic test generator, and the test execution support.

- **Places** represent the states in which the application can be. In SNEDL, dashed places, also called sensor events, are used to abstract sensors (places 1 and 3). The physical sensor readings are represented as tokens generated by the sensor events. The path of these tokens through the Petri net corresponds to the application behavior when the corresponding sensor readings stored in the tokens are detected. Higher level events are constructed using sensor events and/or other higher level events.
- **Transitions** model various kinds of actions. They represent the decision part of the application.
- In a traditional Petri net **arcs** represent changes between states and the way in which tokens are created or destroyed [5]. There are three types of arcs in SNEDL: logic, radio, and hybrid. Logic arcs connect places and transitions that are part of the application logic of the same node. A radio arc, shown as a dashed line (arc a3), denotes communication among nodes. In Figure 2, in order for transition T3 to fire, the application logic on a node needs to reach place 4 and also receive a message from another node that has reached place 2. A hybrid arcs combines the functionality of logic and radio arcs.
- **Tokens** hold sensor readings. The tokens that arrive at each sensor event are associated with temporal and spatial attributes, and therefore information about when and where the data has been sensed can be retrieved. For example, if a token with a time stamp *t*, capacity *c*, and location attribute (x, y, r) reaches a temperature sensor event (place 3 in Figure 2), we can say that a temperature sensor at location (x,y) with sensing range *r* has detected a value *c* at time *t*. Tokens can also contain information about the application execution. For example, when a transition is fired, the tokens injected into that transition's output places could store information specific to the transition's implementation.

An advantage of SNEDL is that it provides a different perspective of a WSN system. By representing a WSN application as a SNEDL model we can view it as a flow of tokens from places in the Petri net to other places. There are several reasons why this is useful for the RTA specification.

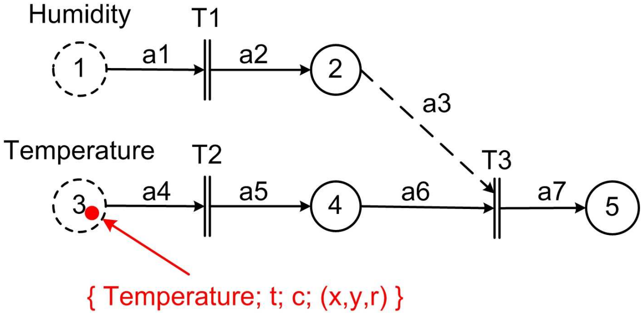


Figure 2: SNEDL model of an application that detects when the temperature and humidity values exceed some predefined thresholds.

First, this is very suitable for testing purposes and makes running RTA tests extremely straightforward. Second, a token-flow model allows us to easily differentiate between a real event and a test event. Since the test events are specified by test-tokens, all we have to do is mark the test-tokens as such. Then, whenever a transition is triggered or a place is reached, we can always check the type of the token that caused this to happen and react accordingly. Third, such a model helps the collection of event-logs and makes it easy to define flow-traces in the system.

The SNEDL model of the WSN application is one of the inputs to our RTA framework. The user is asked to write a script to specify the transitions and places of their model as well as the connections among them. We provide a set of predefined transition types which, we believe, covers the majority of logical operations used in a WSN application, such as, greater, less, equal, minimum, maximum, difference between two consecutive values, average, and moving average. If, however, the user wants to specify transitions with more complex functionality, they can do that by writing the necessary code themselves.

4.2 Automated Test Generation

For more than a decade automatic test generation has been widely used in software testing. It has improved the quality of testing and reduced the effort and time spent on testing. Despite its many advantages, this approach has not been applied to WSN application testing. One of the main benefits of the RTA framework is that it also provides automatic test generation. The RTA test generator takes three inputs:

1. the SNEDL model of the application. Since the application code is automatically generated based on the SNEDL model, we analyze the model itself;
2. the network topology;
3. the test specification - the user provides information about the events they want tests generated for, the areas of the network they want to test, and the times the test should be run. Consider the following scenario: a user wants to test the network for the occurrence of fire (event *EFire*). They want the tests to be run on the nodes in two rooms (Room1 and Room2) at 7am on the 1st day of each month. The nodes in each room are considered equivalent, i.e. they run the same application and are equipped with the same set of sensors. The specification describing this example scenario is:

```
//equivalence - 'no equivalence' or 'region equivalence'
region equivalence

//Declare the basic elements of the language
Time T1;
Region R1, R2;
Event EFire;

//Define the elements
T1=07:00:00, */1/2010;

R1={Room1};
R2={Room2};
EFire = Fire @ T1;
```

A challenge in testing is to maintain a high degree of coverage while reducing the size of the test suite and thus shortening the testing stage. There are additional reasons, specific to WSN applications, for keeping the test suites small. Extensively testing the network would not only increase the cost of the project but will also significantly reduce the lifetime of the system. In addition, memory constraints prevent us from storing too many test inputs on the nodes. Therefore, it is essential to develop methods that would help decrease the number of tests run by the RTA mechanism.

The number of tests needed to fully test the SNEDL model of an application is m^n, where n is the number of inputs and m is the size of the sensing value range of the sensors. Since there are multiple sensor nodes in the network, the total number of tests would be m^{s*n} where s is the number of sensors. In an example network with 100 sensor nodes, each of which has a temperature sensor with value range 0-100 C and a humidity sensor with value range 0-100 RH, the number of tests for all possible input combinations would be 100^{2*100}, or 10^{400}. Running that many tests on a sensor network is infeasible. To address this problem we utilize three test reduction techniques. The first technique has been widely used for reducing the number of tests for software applications. The second and third ones, however, are novel and also unique to the nature of WSN applications.

4.2.1 Static model analysis

The first test reduction technique performs a static analysis of the SNEDL model. The transitions that fire based on the value of the tokens shape the input-to-output behavior of the model. Therefore, we consider these transitions to be the *important* transitions in this analysis. To test such a transition, it is enough to identify the "passing" and "non-passing" sets of inputs and then provide values to represent these two sets. For example, to test a transition with a firing rule "the value should be greater than 27", we need two values - one less or equal to 27, and another one greater than 27. Our static analysis identifies the important transitions and their passing and non-passing sets. Then it chooses a random value to represent each of these sets. This reduction decreases the number of input values to 2^t, where t is the number of important transitions. Therefore, the number of tests for the whole network is decreased to 2^{s*t}, which is significantly smaller than m^{s*n} since the value range of a sensor is much larger than 2 and t is comparable with n.

A limitation of this step is that it is tightly coupled with SNEDL. The reduction step could work with another modeling language only if this language is able to precisely identify the application logic and the important parts of the application that define the input-to-output behavior of the application.

4.2.2 Network topology

A unique characteristic of WSNs application is that, unlike other software applications, they are strongly dependent on the topology of the network. For example, two sensors are more likely to influence each other's decisions if they are neighbors. We take advantage of this feature to further decrease the size of the test suite. We divide the network into *regions*. The size and shape of these regions is very flexible. The default size we use is a room in a building, but a region could also be a group of nodes located close to each other. The RTA test specification contains information about which regions should be tested. When creating the set of tests for a region we only include the nodes within that region. An example application to justify this choice would be detecting an event E in a building with many rooms. A node determines if E has occurred based on its own readings and the readings of its neighbors, where a neighbor is a node in the same room. In this case it is sufficient to test the rooms separately since nodes do not consider the readings of sensors outside of their own room. This approach reduces the number of tests to $R * (2^{sR*t})$, where R is the number of regions in the network and sR is the number of sensors in a region. Since sR is much smaller than the number of nodes in the network, this reduction substantially decreases the number of tests.

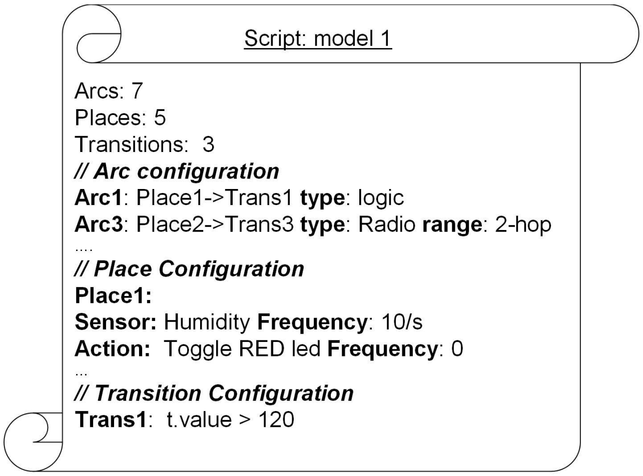
4.2.3 Redundancy

Another unique characteristic of WSNs is node redundancy. The assumption that all nodes in a region are redundant helps us reduce the number of tests that are sufficient to test the network even further. If we assume that all nodes in a region are equivalent to each other, we can use the same test inputs to check the behavior of each one of them. Using this assumption we can decrease the number of tests from $R * (2^{sR*t})$ to $R * (2^t)$. For example, if a network is deployed in a building with 8 rooms ($R = 8$) and there are 5 nodes in each room ($sR = 5$) with 5 important transitions in their application logic ($t = 5$), applying this reduction step decreases the number of tests from 2^{28} to 2^8. Often, if the application logic is more complex, the number of tests could remain large even after applying all three reduction techniques. In these cases, designers could choose a test subset that has an acceptable level of coverage. Alternatively, a test schedule could be devised such that tests are chosen on a rotation principle and only a few tests are run each time.

4.3 Automated Code Generation

The input to the automated code generator (ACG) is a script file that designers write to specify the SNEDL model. An example script for the model in Figure 2 is shown in Figure 3. The script specifies the transitions and their types, the arcs, and the places in the model. Places have two attributes: sensors and actions. At sensor event places, places 1 and 3 in Figure 2, nodes need to access sensors to get data. The designers have to specify the sensor type and the sampling frequency for these places. Actions are the operations that a node has to perform after reaching a particular place. A place can be associated with multiple actions but with only one sensor.

Nodes in heterogeneous WSNs have different functionalities and run different code. In these cases, the ACG needs to partition the SNEDL model accordingly before generating the TinyOS code. Consider a fire detection system that consists of smoke sensor nodes, temperature nodes, and cluster-head nodes. Smoke and temperature nodes sense and notify the cluster-head nodes if they detect abnormal readings. Cluster-head nodes collect information from the sensors and run algorithms to determine if there are fires. Figure 4 shows how the SNEDL model of this example application would be partitioned. The partitions are determined by generating a directed graph with places and transitions as graph nodes, and logic and hybrid arcs as edges. We use breath-first search to find the connected components in the graph which partitions the SNEDL model.



Script: model 1

Arcs: 7
Places: 5
Transitions: 3
// Arc configuration
Arc1: Place1->Trans1 **type**: logic
Arc3: Place2->Trans3 **type**: Radio **range**: 2-hop
....
// Place Configuration
Place1:
Sensor: Humidity **Frequency**: 10/s
Action: Toggle RED led **Frequency**: 0
...
// Transition Configuration
Trans1: t.value > 120

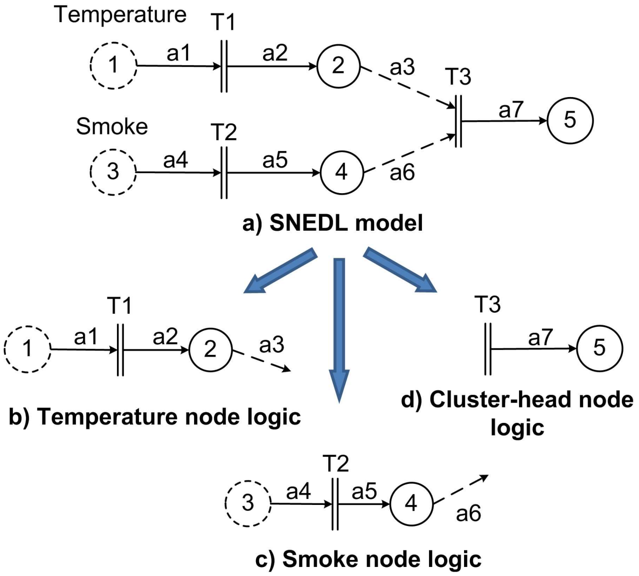
Figure 3: A script specifying the SNEDL model of an application describes all arcs, places, and transitions with their types and attributes. The description of each SNEDL element (arc, transition, or place) requires not more than a couple lines of code.

4.3.1 Code Structure

TinyOS programs are built out of software components. In order to translate the SNEDL model into TinyOS code, we add a SNEDL structure above the TinyOS component model. This structure consists of components that represent places and transitions. It also allows code executions to be driven, processed, represented, and recorded as tokens passing through places and transitions.

The core of the code structure is the concept of *token*. A token is defined as an encapsulation of a value, temporal and spatial information, and an RTA ID. ID 0 is used to indicate the real application execution and other ID numbers are used for tests. The tokens from a particular test use the test's ID as their RTA ID. Tokens pass through components to transfer information and drive the execution. By recording the traversing history of tokens, we can continuously monitor and collect traces of system execution.

Figure 5 shows the component structure of the generated code. We generate components for places and transitions. Another major component, *SNEDL_ResolverM*, works as a system execution engine. This component accepts tokens from one logic object (arc, place, or transition) and automatically transfers them to the next object. Using this component, each logic object can be configured to block the incoming tokens or log the tokens that pass through



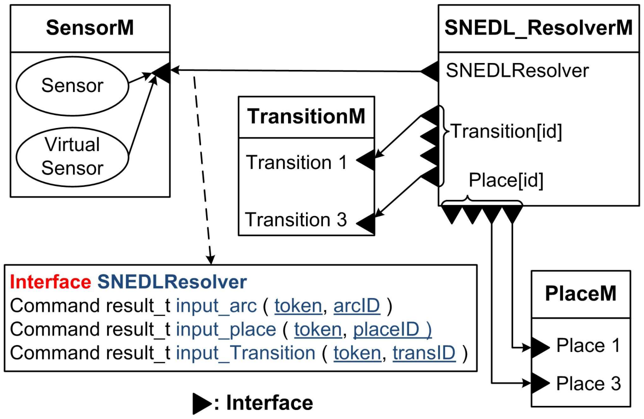
a) SNEDL model

b) Temperature node logic

d) Cluster-head node logic

c) Smoke node logic

Figure 4: The code generator automatically partitions a SNEDL model based on the logic and radio arcs in the model. Each piece is then translated into TinyOS code for the different types of nodes.



```
Interface SNEDLResolver
Command result_t input_arc ( token, arcID )
Command result_t input_place ( token, placeID )
Command result_t input_Transition ( token, transID )
```

Figure 5: The TinyOS code is generated following the SENDL structure. The code structure includes components representing places and transitions. The SNEDLResolverM component drives the execution by transferring tokens through places and transitions. The SensorM component provides both the real and the testing data.

it. *SNEDL_ResolverM* also provides an interface to input a token into any logic object and then trigger the execution.

The *SensorM* component contains all sensors from the SNEDL model. The ACG generates the code to sample sensors, supplement the readings with location and time and encapsulate them into tokens that are forwarded to *SNEDL_ResolverM*. To accommodate testing, the ACG generates a virtual sensor for each physical sensor. These virtual sensors have a buffer to store virtual test readings and can also be sampled. The SensorM component can switch between real sensors and virtual sensors at run time.

The advantages of this code structure are that: 1) it efficiently supports self-testing at run time. We can either run end-to-end tests by using virtual sensors, or partial tests by injecting tokens directly into any logic object; 2) it is easy to monitor execution states and collect running traces by using the log mode; 3) when real events and tests occurs at the same time, we can use the tokens' RTA IDs to distinguish between them. This gives us the flexibility to cancel tests by blocking the test tokens.

Although the ACG currently generates only TinyOS code, it can be adapted to generate code in other languages. For example, instead of components, we can build structures or classes for the different SNDEL objects, and change the code templates in the ACG to generate code in the new languages.

4.4 Test Execution Support

The test execution support is responsible for automatically running the RTA tests on the networked sensing devices. Supporting the execution of RTA tests involves the following steps:

1. After system initialization, every node in the sensor network uses a "sync" interface provided by the service layer to request time synchronization with the WSN gateway and request its own location. The WSN gateway will get synchronized with the time and location server and then process the "sync" requests from each node. The location server gets node locations by querying the database (Figure 1). After this synchronization process, each node is assigned its current time and location. To improve the reliability, the WSN gateway periodically sends "sync" messages to guarantee the time correctness of the nodes.

2. RTA tests are scheduled or requested via test specifications at the user terminal. RTA test data is generated by the automated test generation mechanism, delivered to the WSN gateway, and then shipped to each node using the test deployment protocol. Each test is scheduled to automatically start and stop at specified times using the test control protocol. Each RTA test packet is assigned a unique sequence number. By checking the continuity of the sequence numbers, nodes can detect delivery failures and request the retransmission of lost packets from the gateway.

3. When the scheduled test start time is reached, each sensor node involved in testing enters testing mode and starts reading from its virtual sensors which store the RTA test data. The virtual sensor readings then create virtual events in the network. These virtual events are detected and processed by the SNEDL logic at the application layer. When test mode is entered, each sensor node stops sampling from the corresponding real sensors to avoid communication conflicts. However, to avoid the interruption of real event monitoring, users can specify that sensor nodes should continue sampling even in test mode.

4. When virtual events are detected by the application, the results are reported to the gateway and then used for test result resolution.

The test execution support completely automates the RTA test process. After the users provide the test specification, the specification is processed by the test generation mechanism which generated the test data. This test data is then delivered to the sensor nodes in the network via a WSN gateway. When it is time to run the scheduled tests, the sensor nodes start reading from their virtual sensors to simulate virtual events. These events are processed by the application logic, and the application layer outcomes are reported

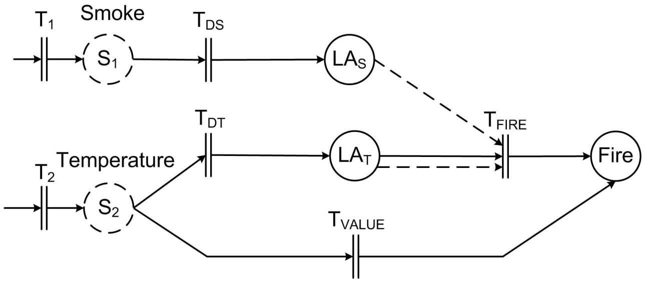


Figure 6: A fire detection application uses smoke and temperature readings to determine the presence of fire. The SNEDL model of this application logic concisely and unambiguously depicts the application logic.

back via the WSN gateway.

With the integration of the ACG, the test generation mechanism, and the test execution support, the RTA methodology can be applied to different user applications with minimal effort. Users specify their applications using SNEDL, and then the ACG generates code for each sensor node, reflecting changes in the application logic. The test generation mechanism uses the same SNEDL model to generate the RTA tests according to the test specifications. These tests are automatically deployed and run on the sensor networks via the test execution support and users can receive automatic updates about the RTA test results.

5. CASE STUDY

We present a case study to demonstrate the usability of our RTA methodology. In this scenario, we are required to design and build a fire detection system for a building. The building has seven floors and there are ten rooms per floor. There is at least one node with a smoke sensor and one node with a temperature sensor in each room. The fire detection application uses both smoke and temperature readings to determine the presence of fire. The fire detection algorithm we use is composed of two separate algorithms. The first monitors the temperature and smoke increase rates and if they both exceed some predefined thresholds, the algorithm reports the presence of fire. The second algorithm reports fire if the temperature value exceeds a predefined threshold. Figure 6 shows the SNEDL model describing the logic of the application. The elements of this model are:

- Places S_1 and S_2 represent the two types of sensor events, while LA_s, LA_t, and Fire stand for, Local smoke alarm, Local temperature alarm, and Fire, respectively.

- Transitions: Transition T_{DS} is fired if the smoke increase rate goes above a specific predefined value. Firing transition T_{DS} sends the firing value over the radio and raises a Local smoke alarm. Transition T_{DT} fires if the temperature increase ratio goes above another predefined threshold. This also causes the application to send a message over the radio and raise a Local temperature alarm. Transition T_{FIRE} is fired if fire has been detected. It is activated only if there are more than two tokens ready to enter it and if these tokens have been generated from sensor readings from the same room.

Next, we implement the system. First, we write a new script file according to this SNEDL model, and input it into the ACG. The ACG partitions the whole logic and generates the code for the smoke and temperature sensors.

To automatically generate the test suite, we need the user to provide us with an RTA test specification. The testing specification in Section 4.2 could be used for this scenario. Knowing the test specification, the SNEDL model, and the topology, we can generate the necessary set of tests to be run by the execution support. These tests need to provide such inputs to places S_1 and S_2 that the presence of fire is simulated. When these tests are run, the application is expected to report the presence of fire based on the virtual readings from the tests.

This case study shows how the RTA methodology is used to guide and help system designers build a fire detection system with RTA capabilities. As demonstrated by this example, the formal SNEDL model and the RTA specification language make the application logic and the RTA requirements clear, unambiguous, and easy to understand. The ACG, combined with the SNEDL model, helps generate an accurate logical implementation of the WSN system high-level behavior. More importantly, our methodology fully addresses the RTA requirements.

At first this case study might seem simplistic. However, several key points must be emphasized. It is important to recognize that this simple model can represent a fire detection system that exists across many floors and uses many sensors. Although we had initially specified that we have seven floors with ten rooms each, these numbers do not confine the application model. The same model could successfully be used to detect fires in a skyscraper with hundreds of rooms. In other words, even though the model is simple, it can represent a large scale system. This case study also uses relatively simple logic for temperature and smoke sensors. However, the power of the underlying Petri net allows us to describe arbitrarily complex logic and control flow. An additional advantage is that the SNEDL model of a complex application is not necessarily too big or complicated since much of the complexity for such systems is encompassed in the logic associated with the transitions.

6. EVALUATION

To investigate the performance of our RTA methodology, we have implemented a prototype Fire Detection (FD) system that is based on the SNEDL model presented in the case study. The system is built on an indoor testbed. The testbed is composed of 21 TelosB nodes, placed in a 7 × 3 grid on a board. One node is chosen as the base station and the rest are divided into different rooms. The TelosB nodes are equipped only with a light sensor. We have used the light sensors to simulate temperature sensors. We use the fire detection algorithm described in Section 5 and the nodes only contain the logic for the temperature sensors. Once a node detects a fire, it sends a report to the base station. For the communication topology we constrain the nodes to only directly communicate with nodes in the same room. For multi-hop communication we use a simple geographic forwarding routing protocol.

6.1 Test reduction

We have analytically estimated the number of tests necessary to fully test the SNEDL model for both the prototype FD application and the FD application presented as a case study. Our calculations are shown in Table 1 and Table

rooms	2	4	5	10
nodes in room	10	5	4	2
baseline (no reduction)	$11 * 10^{35}$	$11 * 10^{35}$	$11 * 10^{35}$	$11 * 10^{35}$
static analysis reduction	$11 * 10^{11}$	$11 * 10^{11}$	$11 * 10^{11}$	$11 * 10^{11}$
topology reduction	$2 * 10^{7}$	4096	1280	160
redundancy reduction	8	16	20	40

Table 1: Combining static analysis techniques and knowledge about the network topology and node redundancy reduces the size of the test suite for the FD application by 35 orders of magnitude.

rooms	2	4	5	10
nodes in room	10	5	4	2
baseline (no reduction)	$11 * 10^{91}$	$11 * 10^{91}$	$11 * 10^{91}$	$11 * 10^{91}$
static analysis reduction	$11 * 10^{17}$	$11 * 10^{17}$	$11 * 10^{17}$	$11 * 10^{17}$
topology reduction	$2 * 10^{9}$	131072	20480	640
redundancy reduction	16	32	40	80

Table 2: When all three reduction steps are applied, adding an extra sensor only doubles the number of necessary tests instead of causing an exponential increase.

2, respectively. For both tables, the result of applying a particular reduction step is calculated as the number of automatically generated tests when the current reduction and all previous reductions steps have been applied together.

The results in both tables show that the first reduction step, statically analyzing the SNEDL model, provides the highest decrease in the number of tests. However, even after using this reduction, running a hundred billion tests on a sensor network is still unreasonable, if not impossible, considering the limited resources of the sensor nodes. In addition, the situation is further exacerbated by the fact that the tests must be run periodically.

Applying our second reduction step which takes advantage of the topology additionally decreases the number of tests. The impact of this step is much more noticeable in cases where there are just a few nodes per region. As shown in Table 1, in the case where we have 10 sensor nodes per region, the drop in the number of tests is just 10^5 times, while the same step leads to a 10^{10} times reduction in the scenario with 2 nodes per region. Similar results can be seen in Table 2.

The decrease in the number of tests after applying the last reduction step is significant as well. Compared to the previous reduction step, here we see the opposite effect: the more sensors there are per region, the higher improvement we get. This is due to the fact that since the nodes in a region are considered to be equivalent, all of them can use the same input test values.

A number of conclusions can be drawn based on the values in Table 1 and Table 2. First, if no reduction is applied, the number of tests we have to run increases exponentially with the number of sensors needed by the application. However, if all three reduction steps are applied, adding an extra sensor merely doubles the number of necessary tests. Second, it is better to create fewer regions with more nodes than more regions each containing a small number of nodes. Third, all three reduction steps need to be applied in order to generate a small set of tests. We realize that there might be cases where the sensor nodes are not equivalent and the last reduction step cannot be performed. In such situations, to avoid exhausting the network's resources, it might be necessary to either only run a carefully chosen subset of the generated tests or schedule the tests in such a way that all generated tests are run but over a more extended period of time.

The automated test generation mechanism is not designed to handle node mobility. Therefore, changes in the topology might cause the RTA tests to fail even if the system is still able to function correctly. This problem can be alleviated in networks with low node mobility levels where can rely on the test execution support to automatically detect topology changes and regenerate and redistribute tests when such changes occur. However, this approach is not feasible when the level of node mobility is high.

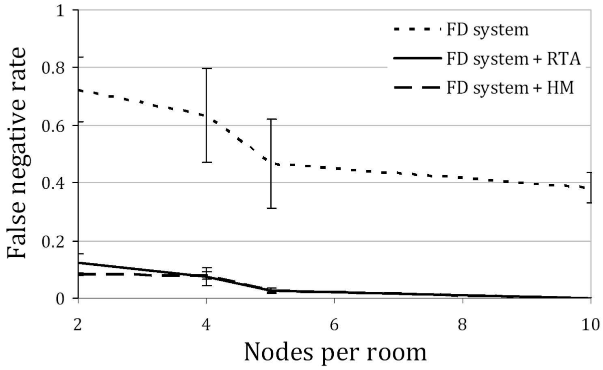


Figure 7: RTA and HM achieve similar false negative rates.

6.2 Robustness to Failure

The goal of RTA is to maintain the accurate operation of a WSN application despite the unreliable and failure-prone underlying infrastructure. To evaluate the robustness of our RTA approach, we have compared it to a health monitoring (HM) system and a pure system with no RTA or health monitoring support. All three fire detection systems have the same application logic. The RTA FD system is generated with the help of our RTA framework. The rooms are tested on a rotation basis and a different room is picked each time a test should be run. The FD system with HM monitoring is implemented following the HM mechanism introduced by Memento [26]. Memento uses heartbeats from neighbor nodes to determine if a node is functional. We assume that when an RTA test fails or the HM system detects a node failure the failed nodes are immediately repaired.

6.2.1 Experimental Results

To demonstrate RTA's efficiency in maintaining application robustness while significantly reducing maintenance

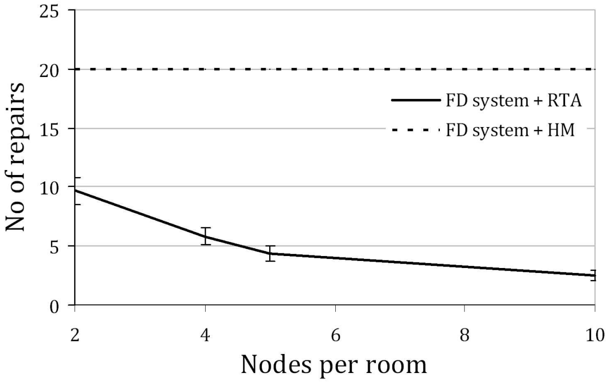


Figure 8: RTA requires 50%-70% fewer repairs than HM.

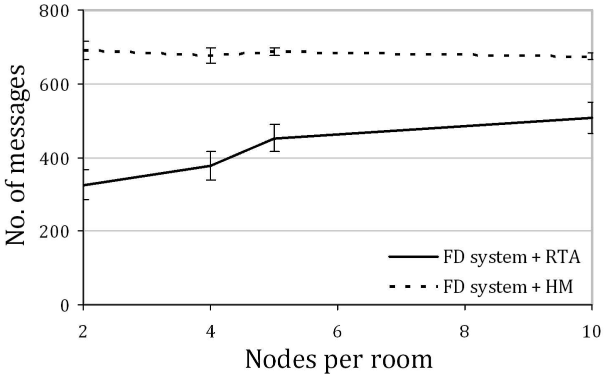


Figure 9: On average, RTA uses 33% less messages than HM.

cost, we have run multiple experiments under different settings. We use a flashlight to generate the fire events in our testbed. To measure the system's performance under realistic scenarios, the fire event sequences are generated using a Poisson process generator. The room in which a fire occurs is chosen randomly. A random Poisson process is also used to compute the failure sequence and determine which nodes stop executing the application and at what time. During the experiments, the WSN gateway injects node-failures by sending messages to the selected nodes and stopping their execution strictly according to the failure sequence. For each experiment, we run the three systems sequentially with the same fire and failure sequences. A node is periodically chosen to fail until no operational nodes are left and the experiment stops. The robustness of the system is represented by the false negative rate which we define as the ratio between the number of unreported fires and the total number of generated fires. Maintenance cost is measured with the number of repairs, the Mean Time to Repairs (MTTR), i.e. the average time between two consecutive repairs, and the number of messages sent during testing.

In the first set of experiments we measure the robustness and maintenance costs of the three systems. To study the impact of redundancy, we vary the number of nodes per room from 2 to 10. All experiments last 20 minutes and both the RTA and the HM system run a test every minute. The fire event rate is 0.5/s, and the node failure rate is 1/s. Figure 7 shows the results of these experiments. Each data point is the average of 5 trials with a 90% confidence interval. The results reveal that, compared to a pure system, both RTA and HM significantly reduce the false negative rates of the FD application. This is expected as both mechanisms can detect node failures and repair nodes immediately. Comparing the RTA system to the HM system shows that both systems demonstrate similar robustness levels. The only visible difference is when there are only 2 nodes per room. Since according to the application logic we need at least two nodes per room, any node failure will lead to a false negative. In this case, the false negative rate of the RTA system is slightly greater than that of the HM system because the RTA system only tests one room at a time while the HM system tests all nodes every time. Increasing the level of redundancy eliminates this difference in the false negative rates. With 5 nodes per room, both systems have a false negative rate close to zero.

We also compare the maintenance cost of the RTA and HM systems. Figure 8 shows the number of repairs for both systems. We can see that the RTA system requires fewer repairs than the HM system. The number of repairs required by the HM system is constant since it triggers a repair every time a node fails. On the other hand, the RTA system repairs nodes only when the operational nodes in a room cannot detect the fire. Our results show that with a certain level of redundancy, the RTA mechanism can significantly reduce the number of repairs while still providing high confidence. For example, with 10 nodes per room, the RTA system guarantees no false negatives and requires only a total of 2.5 repairs. On average, the RTA system produces 70% fewer maintenance dispatches. We also measure the MTTR of both systems. Results show that the RTA system has much longer MTTR than the HM system. For example, with 5 nodes per room, the MTTR of the RTA system is 5.6 seconds, with 1.5 seconds for the HM system.

Another type of maintenance cost is the extra messages incurred by running tests. The base station in the RTA system needs to send messages to trigger tests and nodes are required to report virtual fire events during tests. For the HM system, nodes need to send heartbeats, and the base station should be notified if no heartbeat is received from some node. These extra messages use bandwidth, consume additional energy, and reduce the system lifetime. We have measured the number of the extra messages for both systems and the results are shown in Figure 9. We see that the RTA system introduces much less message overhead. The HM system always uses over 650 messages, because nodes keep sending heartbeats. On average, the RTA system uses 33% less messages than the HM system.

Node level failures are lower level failures and the HM mechanism has been specifically designed to detect them. In the next experiment we compare the performance of the two systems when application level failures are introduced. To do this we insert a new type of error, which we call *location error*. A location error can be defined as any unexpected changes in the location of a node from one region/room to another. Such a location change should lead to a failure if it is against the application's requirements. In our scenario, we need to have at least two nodes per room. Therefore, a location change that decreases the number of nodes in

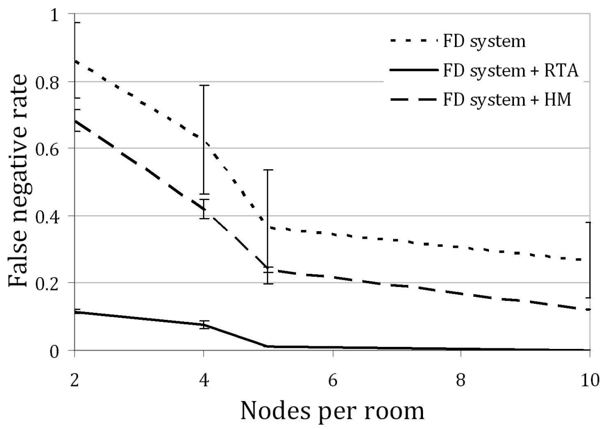


Figure 10: With location errors, the RTA system missed 75% fewer fires than the HM system on average.

a room to less than two should cause an application-level failure.

In this experiment, we change the location of nodes to simulate location errors. We still use a Poisson sequence to generate location errors with the rate of 1/s, and randomly chosen nodes change their locations to adjacent rooms. The false negative rates of our three FD systems are shown in Figure 10. We see that the HM system has a high false negative rate and cannot guarantee the system's robustness. This occurs because when a node is moved to an adjacent room, it can still send heartbeats to its neighbors, so the HM system considers it operational. However, since the RTA system tests the fire detection ability of each room, it can catch such errors without using extra mechanisms to explicitly detect location errors. On average, the RTA system misses 75% fewer system failures. When the node redundancy reaches 5 nodes per room, RTA does not incur any false negatives. Based on these results we can conclude that, compared to an HM system, an RTA system can provide better visibility into the application behavior at run time.

6.2.2 Simulation Results

We have used simulation to demonstrate that RTA can significantly reduce the amount of network maintenance. We simulate a scenario in which we have 25 rooms, and each room has N nodes, where N represents the level of node redundancy in that room. We assume that the lifetime of the sensor nodes follows a Poisson distribution. R is the mean lifetime of the sensors nodes in months and it represents the reliability of the nodes. We generate fire events following a Poisson distribution. The expected occurrence interval of these events is three months, and the simulation period is set to two years. Each data point represents the average of five runs.

We make the following two assumptions when comparing the amount of maintenance work required by RTA and HM: 1) If the RTA tests determine that the nodes in a room are not able to detect the presence of fire, a maintenance worker is dispatched to replace the failed nodes in that room; 2) If the HM determines that a node has failed, a maintenance worker is dispatched to replace the failed node.

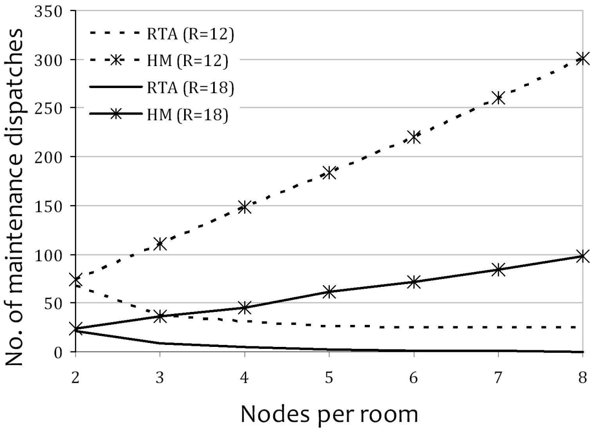


Figure 11: RTA reduces the number of maintenance dispatches to only 0.3%-33.9% of HM.

System	ROM (bytes)	RAM (bytes)	lines of code
Pure FD	18142	898	1767
FD + RTA	24888	3386	3193
FD + HM	22976	1280	2590

Table 3: Compared to HM, RTA has a larger memory footprint.

In order to achieve a fair comparison, we use the same testing overhead (in terms of the number of testing messages sent) for RTA and HM. On average, an RTA test requires K times more messages than HM - in our implementation K≈8. Therefore, in the simulations RTA runs in 1/8 the frequency of HM. In this simulation the RTA test frequency is a test per day for each room.

Figure 11 shows the number of maintenance dispatches under different levels of node reliability R and redundancy (nodes per room). We observe that when the redundancy is increased RTA makes a significant reduction in the number of maintenance dispatches. With 6 or more nodes per room, the number of RTA maintenance dispatches is less than 10% of the dispatches required by HM. Since HM maintains node-level reliability, the number of maintenance dispatches is proportional to the number of node failures. In contrast, RTA maintains application-level reliability and a maintenance dispatch is only necessary when the application is not able to meet its high-level requirements. Thus, by taking advantage of the node redundancy level, RTA can considerably reduce the total number of the required maintenance dispatches.

6.3 Overhead

Table 3 compares the memory usage and footprint of the three FD systems. We can see that, because of its token-flow programming structure and more powerful test execution facilities, our RTA system uses more program memory and RAM. Also, the code for our RTA system is larger than that for the other two systems. However, since most of this code is automatically generated by our RTA framework, users only need to write the SNEDL model specification script, which was less than 50 lines for the experimental FD system.

7. CONCLUSIONS

A major disadvantage of the current reliability and health monitoring techniques used for WSN applications is that they monitor the performance of the low-level components of the system instead of the reliability of the application layer. To the best of our knowledge, this is the first paper to address this issue in WSNs and suggest a methodology to help designers and users verify an application's integrity at run time. We have also implemented a framework that facilitates the use of our RTA methodology. This framework provides automated code generation, automated test generation, and execution support for the RTA tests. To decrease the vast number of generated test cases, we introduce three test suite reduction techniques, two of which are unique to the nature of WSN applications and take advantage of the network's topology and node redundancy.

We evaluated an implementation of a fire detection application with RTA support and found out that RTA performs much better than health monitoring in identifying application-level failures and almost just as good in identifying low-level failures. In addition, our RTA implementation incurs considerably less communication overhead and also decreases the amount of maintenance work that needs to be done in order to keep the system operational.

8. ACKNOWLEDGMENTS

This work was supported, in part, by the National Science Foundation under Grants CNS - 0834299 and IIS - 0931972, and by the Korean Science and Engineering Foundation WCU Project R33-2008-000-10110-0.

9. REFERENCES

[1] Y. Al-Raisi and D. Parish. Approximate wireless sensor network health monitoring. In *IWCMC*, August 2007.

[2] T. Clouqueur, K. K. Saluja, and P. Ramanathan. Fault tolerance in collaborative sensor networks for target detection. In *IEEE Transactions on Computers*, 2004.

[3] CrossBow. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/TelosB_Datasheet.pdf.

[4] M. Dyer, J. Beutel, L. Thiele, T. Kalt, P. Oehen, K. Martin, and P. Blum. Deployment support network - a toolkit for the development of WSNs. In *EWSN 2007*, January 2007.

[5] C. Girault and R. Valk. *Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications*. Springer-Verlag New York, Inc., 2001.

[6] T. He, S. Krishnamurthy, J. Stankovic, T. Abdelzaher, L. Luo, T. Yan, J. Hui, and B. Krogh. Energy efficient surveillance systems using wireless sensor networks. In *Mobisys*, June 2004.

[7] D. Herbert, V. Sundaram, Y. Lu, S. Bagchi, and Z. Li. Adaptive correctness monitoring for wireless sensor networks using hierarchical distributed run-time invariant checking. In *TAAS*, September 2007.

[8] IBM Autonomic Computing, 2008. http://www.research.ibm.com/ autonomic/.

[9] X. Jiang, J. Taneja, J. Ortiz, A. Tavakoli, P. Dutta, J. Jeong, D. Culler, P. Levis, , and S. Shenker. An architecture for energy management in wireless sensor networks. In *SIGBED*, April 2007.

[10] B. Jiao, S. Son, and J. Stankovic. GEM: Generic event service middleware for wireless sensor networks. *INSS*, 2005.

[11] M. Khan, T. Abdelzaher, and K. Gupta. Towards diagnostic simulation in sensor networks. In *DCOSS*, 2008.

[12] M. Khan, H. Le, H. Ahmadi, T. F. Abdelzaher, and J. Han. Dustminer: troubleshooting interactive complexity bugs in sensor networks. In *SenSys*, 2008.

[13] A. Kolawa and D. Huizinga. *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Press, 2007.

[14] Z. Lai, S. Cheung, and W. Chan. Inter-context control-flow and data-flow test adequacy criteria for nesc applications. In *SIGSOFT*, 2008.

[15] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse1, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. TinyOS: An operating system for sensor networks. In *Ambient Intelligence*, 2005.

[16] H. Liu, L. Selavo, , and J. Stankovic. SeeDTV: Deployment-time validation for wireless sensor networks. In *EmNetS 2007*, June 2007.

[17] L. Luo, T. He, G. Zhou, L. Gu, T. Abdelzaher, and J. A. Stankovic. Achieving repeatability of asynchronous events in wireless sensor networks with envirolog. In *Infocom 2006*, April 2006.

[18] N. Nguyen and M. L. Soffa. Program representations for testing wireless sensor network applications. In *DOSTA*, 2007.

[19] L. Paradis and Q. Han. A survey of fault management in wireless sensor networks. In *JNSM*, June 2007.

[20] J. L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, 1977.

[21] N. Ramanathan, K. Chang, L. Girod, R. Kapur, E. Kohler, and D. Estrin. Sympathy for the sensor network debugger. In *SenSys*, November 2005.

[22] J. Regehr. Random testing of interrupt-driven software. In *EMSOFT*, 2005.

[23] M. Ringwald, K. Romer, and A. Vitaletti. SNTS: Sensor Network Troubleshooting Suite. In *DCOSS*, June 2007.

[24] M. Ringwald, K. Romera, and A. Vitaletti. Passive inspection of sensor networks. In *DCOSS 2007*, June 2007.

[25] B. rong Chen, G. Peterson, G. Mainland, and M. Welsh. LiveNet: Using passive monitoring to reconstruct sensor network dynamics. In *DCOSS 2008*, June 2008.

[26] S. Rost and H. Balakrishnan. Memento: A health monitoring system for wireless sensor networks. In *SECON 2006*, September 2006.

[27] L. Ruiz, J. Nogueira, and A. Loureiro. MANNA: A management architecture for wireless sensor networks. In *IEEE Communications Magazine*, February 2003.

[28] L. Ruiz, I. Siqueira, L. Oliveira, H. Wong, J. Nogueira, and A. Loureiro. Fault management in event-driven wireless sensor networks. In *MSWiM*, October 2004.

[29] T. Sookoor, T. Hnat, P. Hooimeijer, W. Weimer, and K. Whitehouse. Macrodebugging: Global views of distributed program execution. In *SenSys*, 2009.

[30] G. Tolle and D. Culler. Design of an application-cooperative management system for wireless sensor networks. In *EWSN*, February 2005.

[31] C. Wan, S. Eisenman, and A. Campbell. CODA: Congestion detection and avoidance in sensor networks. In *SenSys*, November 2003.

[32] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: using RPC for interactive development and debugging of wireless embedded networks. In *IPSN*, 2006.

[33] J. Yang, M. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: A comprehensive source-level debugger for wireless sensor networks. In *SenSys*, November 2007.

[34] M. Yu, H. Mokhtar, and M. Merabti. Fault management in wireless sensor networks. In *IEEE Wireless Communications*, December 2007.

[35] H. Zhang and A. Arora. GS3: scalable self-configuration and self-healing in wireless sensor networks. In *Computer Networks (Elsevier)*, November 2003.

[36] Y. Zhao, R. Govindan, and D. Estrin. Residual energy scan for monitoring sensor networks. In *WCNC*, March 2002.