To appear in *Reliable Software Technologies: Ada-Europe 2007*, N. Abdennadher and F. Kordon (eds.), Lecture Notes in Computer Science, vol. 4498, Springer, Berlin (2007)

# Towards the Testing of Power-Aware Software Applications for Wireless Sensor Networks ⋆ ⋆⋆

W. K. Chan[1], T. Y. Chen[2], S. C. Cheung[3], T. H. Tse[4], and Zhenyu Zhang[4]

[1] City University of Hong Kong, Kowloon Tong, Hong Kong wkchan@cs.cityu.edu.hk
[2] Swinburne University of Technology, Hawthorn, Australia tchen@ict.swin.edu.au
[3] The Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong sccheung@cse.ust.hk
[4] The University of Hong Kong, Pokfulam, Hong Kong {thtse, zyzhang}@cs.hku.hk

**Abstract.** The testing of programs in wireless sensor networks (WSN) is an important means to assure quality but is a challenging process. As pervasive computing has been identified as a notable trend in computing, investigations on effective software testing techniques for WSN are essential. In particular, energy is a crucial and scarce resource in WSN nodes. Programs running correctly but failing to meet the energy constraints may still be problematic. As such, testing techniques for power-aware applications are useful; otherwise, the quickly depleted device batteries will need frequent replacements, hence challenging the effectiveness of automation. Since current testing techniques do not consider the issue of energy constraints, their automation in the WSN domain warrants further investigation.

This paper proposes a novel power-aware technique built on top of the notion of metamorphic testing to alleviate both the test oracle issue and the power-awareness issue. It tests the functions of programs in WSN nodes that are in close proximity, and uses the data consolidation criteria of data aggregation in programs as the basis for verifying test results. The power-aware transmissions of intermediate and final test data as well as the computation required for verification



⋆⋆ This work is supported in part by a grant of the Innovation and Technology Commission in Hong Kong (Project No. ITS/076/06), a grant of City University of Hong Kong (Project No. 7200079), a discovery grant of the Australian Research Council (Project No. DP 0771733), and CERG grants of the Research Grants Council of Hong Kong (Project Nos. 612306 and 717506).

of test results are directly supported by the WSN programs. Our proposed technique has been strategically designed to blend in with the special features of the WSN environment.

*Keywords* Wireless sensor network, WSN application, power awareness, test oracle, metamorphic testing, software testing, test automation.

## 1 Introduction

A wireless sensor network (WSN) is an ad hoc computer network formed by many sensor devices interconnected by wireless channels. Each sensor device, known as a sensor node or simply a *node*, is built with sensors to capture data (such as temperature and light intensity) from its physical environment. Because of physical and environmental limitations, such nodes should satisfy various constraints regarding battery power, computation capacity, unforeseeable communication restrictions, and others. A popular class of sensor nodes are those that, once deployed, will be operational until the depletion of their batteries. In other words, the lifetime of the sensor nodes depend critically on power management.

In certain WSN, many nodes are deployed in proximity to perform the same function, such as sensing a change of temperature for the detection of any passing hot object. A few nodes, equipped with relatively plentiful resources, are selected as base stations that bridge other nodes and target client stations. These nodes are normally known as *data aggregators*. They consolidate the data from sensory nodes and send the consolidated results to other nodes. For example, they eliminate duplicated copies of data or compute statistics from the data received according to defined evaluation criteria. In this paper, we refer to such criteria as *data consolidation criteria*. In this way, the data consolidation feature of a WSN application would save transmission energy by propagating the computed results instead of simply relaying received data [10].

Power efficiency is critical to many WSN applications. A power-aware application should "enable users to work for longer periods of time before having to reconnect to recharge the system battery. ... an anti-virus scanner would run in a full-featured fashion, providing file scanning on all files opened and also running periodic system-wide scans. When on battery power, the scanner could defer the system-wide scans until a later time and continue processing safely, analyzing just the open files." [5] For example, according to http://www.tinyOS.net, the de facto site for important news release for tinyOS, researchers advocate to design WSN artifacts to be power-aware.

On the other hand, the nodes in a typical WSN must coordinate among themselves and the transmission of data across nodes in the network is a typical feature in a WSN application — but data communication is the most energy-demanding aspect among communication, sensing and computation. [6]

[5] http://solveit.jotxpert.net/WikiHome/Articles/277366.

[6] M. Srivastava. Wireless sensor and actuator networks: challenges in long-lived and high-integrity operation. ASI Lecture, City University of Hong Kong, December 4, 2006. Available at http://www.cs.cityu.edu.hk/ asi06/program.shtml.

The development of automated testing techniques for WSN applications with the above properties is further complicated by the following:

(a) The outputs from an individual node are often unstable, unreliable, and dependent on hardware quality and unforeseeable physical environmental conditions. One way to overcome these limitations is to test software components on a node simulator, such as TOSSIM [7], which emulates simplified situations of WSN applications. In practice, however, such an approach requires elaborated studies of possible application-specific environments. Testing on a real platform is another alternative. Unfortunately, the results of an application in the unforeseeable WSN environment are determined not only by *what* test cases are selected, but also *where*, *when* and *how* they are executed.
(b) As the power-aware aspect is a key feature of WSN applications, the testing of both functional correctness and power-awareness is necessary for quality assurance. In particular, applications in sensor nodes are often designed for a particular range of workload such as sleeping for 99% of its time and working actively for the remaining 1%. Using excessive resources in sensor nodes to conduct testing activities may change the workload pattern, which directly alters the energy consumption and, hence, the power-awareness test results. The testing of applications in the WSN environment is, therefore, a research and practical challenge.

As such, we identify at least two testing research directions for assuring WSN applications: (1) the formulation of effective testing techniques for power-aware applications, and (2) the development of energy-efficient testing techniques for WSN. This paper is our first step towards these goals.

As we have mentioned earlier, data communication is unavoidable in WSN and consumes most of the energy. It inspires us to use the existing data communication of the application to transmit test results to data aggregators (which have relatively plentiful resources) to conduct the checking of test results. We explore this observation to develop our testing proposal.

This paper proposes a novel test automation strategy to alleviate the test oracle issue associated with power management concerns in a WSN environment. The backbone of our test verification technique is metamorphic testing [7]. Our strategy consists of three elements: (a) The test inputs are sensed data of isotropic physical phenomena of sensor nodes that are in proximity. (b) We then use the data consolidation criteria supported by the data aggregators of WSN applications as a basis for verifying results according to the metamorphic approach [8]. (c) We enhance current techniques of metamorphic testing, such as [8, 9, 17], to address non-functional concerns. For any given test case, apart from verifying the functional output from the sensor node, we also check the energy consumption that has been used to compute the functional output. By comparing and contrasting the energy consumptions, we verify whether an abnormal amount of energy has been consumed.

The main contributions of the paper are two-fold: (i) To our best knowledge, it is the first research to alleviate the test oracle problem for the testing of software applications

[7] http://www.cs.berkeley.edu/ pal/research/tossim.html.

running on top of wireless sensor networks. (ii) It is also the first attempt to address the testing of power-aware concerns for these applications.

The rest of the paper is organized as follows: It first reviews related work in the next section, followed by metamorphic testing in Section 3. In Section 4, we shall develop a software testing model that helps testers alleviate the test oracle problem in the presence of power management concerns. In Section 5, we discuss issues of other potential options, and portray directions for future work. We conclude the paper in Section 6.

## 2 Related Work

To support high-quality automated software testing, a reliable and automatic test oracle should be developed. Binder [6] extensively reviews many popular types of test oracle and casts them in the context of object-oriented testing. A simple and intuitive mechanism for automatic test oracles is program assertion, as supported by JUnit [2]. This approach may also be applied to the testing of programs in sensor nodes. However, sensory computing is intrinsically imprecise. The use of program assertions in imprecise WSN computation warrants further research.

Some practitioners recommend adopting the techniques in Graphical User Interface (GUI) testing to embedded systems testing [11]. Berstel et al. [5] use a formal specification approach to verify GUI specifications. They work at the specification level; whereas we propose a technique that works at the implementation level. Xie and Memon [19] empirically study the effect of a number of selected test oracle approaches on GUI applications. They find in their experiments that, for effective identification of failures, test oracles should be strong, or else the failures for some test cases may not be identifiable by the test oracles. Our approach proposes to use data aggregation criteria of the application to serve as the mechanism to define test oracles. Their work complements ours.

Chen et al. [9] evaluate the failure-detection capabilities of different metamorphic relations for the same applications. They find that different forms of metamorphic relations have diverse strengths in detecting failures even for the same set of source test cases. We also utilize the knowledge of domain experts specified in the data consolidation criteria for eliminating duplicated data entries by the data aggregator of the application.

Tse et al. [16] report on an approach, which is close to metamorphic testing, to tackle the test oracle issue for embedded software in bonding machines. They do not consider the power-awareness issue. Kapfhammer et al. [12] propose to unload not-in-use test components to ease the memory constraints of testing activities. Our approach, on the other hand, directly uses application code to detect failures.

Some researches improve the infrastructure to support testing. For example, testers may use emulators of wireless sensor networks [13, 18] to selectively track whether their applications have executed as intended. These emulators simulate the hardware environments to facilitate the development and checking software applications.

The emulator approach is quite laborious since extensive prior profiling is required. An alternative is to use verification patterns [15] that implement test scenarios

collectively as test templates so that different initializations of test scripts can share the same test infrastructure. There are approaches that propose some kind of conformance testing framework to test embedded software formally [14].

The effectiveness and tradeoffs of these approaches are, however, not examined. Although WSN applications are gaining popularity, their testing problems have not yet been adequately studied.

## 3 Metamorphic Approach

Metamorphic testing [7, 8] was proposed as a property-based approach to alleviate the test oracle problem. It has been applied to unit testing as well as integration testing. These studies illustrate the notion of metamorphic testing via functional testing. Since our proposal is closely related to the notion, we review it in this section.

### 3.1 Concept

A test oracle is a mechanism to determine whether a test case has passed or failed. It is difficult to establish (automated) test oracles for some programs. Examples are software for optimal routing, shortest paths, and partial differential equations. This is known as the test oracle problem. Although human may serve as manual test oracles in some cases, it is costly and error-prone.

In such circumstances, metamorphic testing aims at selecting follow-up test cases based on source test cases, and check whether their results satisfy necessary conditions relating the expected solutions of the source and follow-up test cases. These necessary conditions are known as metamorphic relations [7, 8].

### 3.2 Metamorphic Relation

Informally, a metamorphic relation consists of two components. The first relies on a relation, known as a source relation, which defines the relationship between source test cases, their outputs, and follow-up test cases. This effectively provides a specification for developers or test drivers to select follow-up test cases. The second component defines the relationship among the target functional outputs of the source test cases and follow-up test cases, and the test cases themselves. Such a metamorphic relation can be used to verify test results to alleviate the test oracle problem.

A metamorphic relation is vital to the application of metamorphic testing. We adapt the definition of metamorphic relation from [8] as follows:

> Let $f$ be a target function and let $P$ be its implementation. Intuitively, a metamorphic relation is a necessary condition over a series of inputs $x_1, x_2, \ldots, x_n$ and their corresponding results $f(x_1), f(x_2), \ldots, f(x_n)$ for multiple evaluations of $f$.

**Definition 1 (Metamorphic Relation).** [8] *Let $x_1, x_2, \ldots, x_k$, where $k \geq 1$, be a series of inputs to a function $f$ and let $\langle f(x_1), f(x_2), \ldots, f(x_k)\rangle$ be the corresponding*

*series of results. Suppose* $\langle f(x_{i1}), f(x_{i2}), \ldots, f(x_{im})\rangle$ *is a subseries, possibly an empty subseries, of* $\langle f(x_1), f(x_2), \ldots, f(x_k)\rangle$. *Let* $\langle x_{k+1}, x_{k+2}, \ldots, x_n\rangle$, *where* $n \geq k+1$, *be another series of inputs to* $f$ *and let* $\langle f(x_{k+1}), f(x_{k+2}), \ldots, f(x_n)\rangle$ *be the corresponding series of results. Suppose, further, that there exist relations* $r(x_1, x_2, \ldots, x_k, f(x_{i1}), f(x_{i2}), \ldots, f(x_{im}), x_{k+1}, x_{k+2}, \ldots, x_n)$ *and* $r'(x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n))$ *such that* $r'$ *must be true whenever* $r$ *is satisfied. We say that*

$$\begin{aligned} MR = \{\ & (x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n)) \mid \\ & r(x_1, x_2, \ldots, x_k, f(x_{i1}), f(x_{i2}), \ldots, f(x_{im}), x_{k+1}, x_{k+2}, \ldots, x_n) \\ & \rightarrow r'(x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n))\} \end{aligned}$$

*is a* metamorphic relation. *When there is no ambiguity, we simply write the metamorphic relation as*

$$\begin{aligned} MR{:}\ & \textit{If } r(x_1, x_2, \ldots, x_k, f(x_{i1}), f(x_{i2}), \ldots, f(x_{im}), x_{k+1}, x_{k+2}, \ldots, x_n), \\ & \textit{then } r'(x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n)). \end{aligned}$$

Furthermore, $x_1, x_2, \ldots, x_k$ are known as *source test cases* and $x_{k+1}, x_{k+2}, \ldots, x_n$ are known as *follow-up test cases*.

Consider an implementation of a hash function $f$, which computes the hashed values of input vectors $a_1, a_2, \ldots, a_m$. A necessary condition for its correctness is that the hashed values of the input elements of an input vector should not change even if we rearrange their positions in the input vector. In other words, given a correct implementation, the hash total of the outputs from a source input vector should be the same as that of the outputs from a permuted input vector.

Based on the above concept, a tester may first find a reliable permutation program $\pi$ to rearrange input vectors. The tester then sets up a source relation $\langle b_1, b_2, \ldots, b_m\rangle = \pi(\langle a_1, a_2, \ldots, a_m\rangle)$ to select a follow-up test case $\langle b_1, b_2, \ldots, b_m\rangle$ from any source test case $\langle a_1, a_2, \ldots, a_m\rangle$. Thus, the tester can define a metamorphic relation for the hash function as follows:

$$\begin{aligned} MR_{hash}{:}\ & \text{If } \langle b_1, b_2, \ldots, b_m\rangle = \pi(\langle a_1, a_2, \ldots, a_m\rangle), \\ & \text{then } \textstyle\sum_{i=1}^{k} f(\langle b_1, b_2, \ldots, b_m\rangle)[i] = \sum_{i=1}^{k} f(\langle a_1, a_2, \ldots, a_m\rangle)[i], \\ & \text{where } [i] \text{ refers to the i-th index of the array.} \end{aligned}$$

### 3.3 Metamorphic Testing

We also adapt the definition of metamorphic testing from [8] as follows:

> This [metamorphic] relation must be satisfied when we replace $f$ by $P$; otherwise $P$ will not be a correct implementation of $f$.

**Definition 2 (Metamorphic Testing).** [8] *Let* $P$ *be an implementation of a target function* $f$. *The* metamorphic testing *of metamorphic relation*

$$\begin{aligned} MR{:}\ & \textit{If } r(x_1, x_2, \ldots, x_k, f(x_{i1}), f(x_{i2}), \ldots, f(x_{im}), x_{k+1}, x_{k+2}, \ldots, x_n), \\ & \textit{then } r'(x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n)). \end{aligned}$$

*involves the following steps:* (1) *Given a series of source test cases* $\langle x_1, x_2, \ldots, x_k \rangle$ *and their respective results* $\langle P(x_1), P(x_2), \ldots, P(x_k) \rangle$*, select a series of follow-up test cases* $\langle x_{k+1}, x_{k+2}, \ldots, x_n \rangle$ *according to the relation* $r(x_1, x_2, \ldots, x_k, P(x_{i1}), P(x_{i2}), \ldots, P(x_{im}), x_{k+1}, x_{k+2}, \ldots, x_n)$ *over the implementation P.* (2) *Check the relation* $r'(x_1, x_2, \ldots, x_n, P(x_1), P(x_2), \ldots, P(x_n))$ *over P. If* $r'$ *is false, then the metamorphic testing of MR reveals a failure.*

Let us use the above hash function example for an illustration. Suppose the input vector $\langle 1, 2, 3 \rangle$ is hashed to $\langle -1234, 98, 36 \rangle$, and the input vector $\langle 3, 2, 1 \rangle$, constructed according to the implementation of the above source relation, is hashed to $\langle 36, 98, 1234 \rangle$ by the implementation under test. A test driver will then compute the sum of $-1234$, 98, and 36, which is $-1100$. It will also compute the sum of 36, 98, and 1234, which is 1368. Since the two resultant values differ, the metamorphic relation is violated. The two test cases collaboratively reveal a failure.

## 4 Our Approach

In this section, we first clarify its relationship with metamorphic testing. Next, we discuss the software environment in which the notion of metamorphic testing can be adapted to reveal functional and non-functional problems of an application. Section 4.3 gives an example application scenario, and describes the types of fault. Finally, we shall illustrate the use of our approach to identify these faults in Section 4.4.

### 4.1 Relationship with Metamorphic Testing

Consider a program $P$ under test that supposedly implements a target function $f$. Metamorphic testing, as reviewed in the last section, relies on the formulation of a necessary condition of $f$. It ignores the software environment of $P$. On the other hand, we observe that the deployment of $P$ in different environments has substantial impacts on its non-functional properties. Putting $P$ on a low-voltage platform, for example, may cause the power consumption to be different from that on a standard PC platform. We suggest that the software environment should also be taken into consideration for necessary conditions related to non-functional properties.

In addition, the generic notion of metamorphic testing needs to be enhanced to address at least the following two areas in the testing of WSN applications: (a) How to implement a metamorphic relation in wireless sensor networks? (b) Where to evaluate the test results, since most WSN nodes have very limited storage, memory, bandwidth, and so on?

In the rest of this section and Section 5, we shall investigate and elaborate on our proposal to address these concerns and beyond.

### 4.2 Software Environment

In this section, we present an overview of our test model of WSN software applications. A WSN is modeled as a set of nodes. We assume that one software component is

deployed in each node.[8] Hence, one may refer to a software component by simply referring to a node. In the sequel, we shall use the terms "node" and "software component" interchangeably unless otherwise stated.

Each software component has its own function. We have studied a variety of third-party TinyOS [4] applications as well as the tutorial applications of TinyOS to come up with the following observation. We note that each software component will accept an input, complete the required processing, and output a result before accepting another input. Furthermore, in nesC applications, for instance, there are sub-components within a software component such that the output of one sub-component is piped to another sequentially. We maintain the concept of sequential programs in our model because typical sensor nodes have little resources, and concurrency models found in conventional operating systems are still poorly developed in, say, TinyOS and nesC [3]. Thus, a temperature sensor node may behave as follows: It obtains the current and voltage readings from its temperature sensor and computes the value of the temperature. It then identifies its surrounding nodes and sends the computed value to a destination node such as a data aggregation node or a base station.

We assume the following network property based on the characteristic that nodes in a wireless sensor network are deployed in large scale for a particular application, such as temperature monitoring in a nontrivial area zone.

> Assumption: *Every node has at least one adjacent node whose software component performs the same function as the node itself.*

Since energy is a critical resource for WSN nodes, each node is equipped with a residual energy-level scan function. We model it as a *utility function* that returns the instant energy level of the node. Hence, a test driver may inquire about the energy level of a node before and after the execution of a software component in a software environment and determine the energy $\phi$ consumed by the execution of a test case. From this, software testers can track the energy spent on the execution of one test case in one particular node.

### 4.3 Example Software Application Scenario

We describe an example temperature monitoring application that aims to identify incidents of wildfire and forward alerts to the clients concerned. Figure 1 depicts a blueprint of the application.

***Description of the software application.*** A cluster of wireless sensor nodes is deployed in a fire control zone. Some of them, such as nodes 1 and 5, are deployed with the same software component *P* that computes the ambient temperature. These nodes determine a raw temperature reading as follows: The resistance of the conductor in a sensor normally varies with temperature, causing changes in current and voltage according to Ohm's Law plus or minus detailed practical deviations.

[8] It is not difficult to extend our model to allow more than one software component to be deployed in a node.

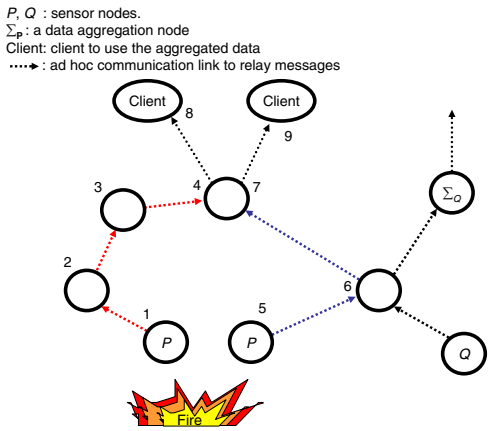


**Fig. 1.** Example scenario of software application of wireless sensor network, which monitors whether there is any wildfire.

For the ease of discussion, let us assume that the change in resistance is directly proportional to the change in temperature under the operational conditions of the sensor nodes. Thus, by comparing the instant current and voltage readings from the sensors with those of reference temperatures, program logic can be implemented to estimate the present temperature from the current and voltage readings.

Suppose a wildfire breaks out near sensor nodes 1 and 5, as shown in Figure 1. The ambient temperature rises and, hence, triggers these nodes to gauge the environment. As described above, the software component *P* will interpret the current temperature based on the readings.

Each of the two nodes will send the computed temperature values to their respective adjacent nodes, which will relay the values to the data aggregation device, node 7, for further data summarization. In Figure 1, the computed temperature value from node 1 is forwarded to node 7 via nodes 2, 3, and 4, while that from node 5 is routed via nodes 6 and 7. (We note that nodes 4 and 7 are aliases of each other.)

Data aggregation is essential for wireless sensor networks [10]. It merges duplicated data from distinct sources to save energy in data transmission. This is mostly application-oriented. In other words, data consolidation criteria should have been implemented in WSN applications to cater for resource constraints. In our example, the data from nodes 1 and 5 are consolidated at node 7, so that clients at nodes 8 and 9 will receive a consolidated result.

***Program.*** Embedded system developers frequently implement their software to meet resource-stringent constraints. They use application-specific heuristics to develop the program logic.

In the example application, the temperature sensors attached to nodes 1 and 5 have the following specific property: In certain temperature ranges, the current or voltage reading has a linear relationship with the temperature. See, for instance, the linear portions of the two sample plots in Figure 2. On the other hand, the temperature range

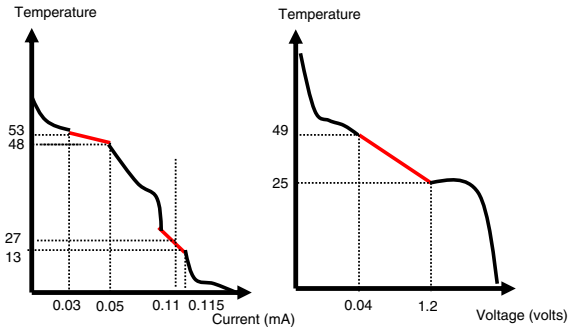


**Fig. 2.** Example plots of variations in temperature against variations in current (left) and voltage. (Not to scale.)

applicable to the linear portion of the current-temperature plot may not overlap with the temperature range applicable to that of the voltage-temperature plot [1]. Furthermore, the relationship among temperature, current, and voltage may be non-linear and much more complex outside these temperature ranges.

Thus, when a program finds that it cannot determine the temperature value using a simple calculation based on the instant current reading, it will try to compute the temperature based on the instant voltage reading. When both approaches are not applicable, it will use a more sophisticated and computationally expensive formula. This three-way policy serves to save energy. There is also some environment-specific calibration in the program to initialize the temperature variable.

Consider the following annotated self-explanatory example code fragment of software component *P* to compute the temperature. [9]

```
P (clb_I, clb_V: integer) { // environmental specific calibrations
                            // for current and voltage.
     V, I, T: integer;     // voltage, current, and temperature
     T = 300;               // environment-specific initialization
     ...
     V = sensor_channel1( );
     I = sensor_channel2( );
     ...
     if (I >= 300 and I <= 500)
s0:     T = 53 - (( I - 300 ) * (53 - 48)) / (500 - 300) + clb_I;
     else if (I >= 1100 and I <= 1150)
        // [Hard] Fault (a): The correct version should be:
        // T = 27 - (( I -  1100 ) * (27 - 13)) / (1150 - 1100) + clb_I;
s1:        T = 27 - (( I -  1150 ) * (53 - 48)) / (1150 - 1100) + clb_I;
```

[9] The data types of the variables are integers instead of floating point numbers, because floating-point calculation is expensive and seldom used in embedded system computations. For the ease of presentation, a current of "$I$ mA" is written as "$10000 \times I$" in the sample code. The treatment of voltage $V$ is similar. We also show numbers in the format of, say, "$53 - 48$" instead of "5", to enable readers to cross-reference with the example code fragment.

```
     // [Soft] Fault (b): The "else" keyword is missing.
     // The correct version should be:
     // else if (V >= 400 and V < 12000) ...
     if (V >= 400 and V < 12000)
s2:     T = 49 - (( V - 400 ) * (49 - 25)) / (12000 - 400) + clb_V;
     else ...
        // the general and sophisticated approach to compute T.
        return T;
     }
```

In the code fragment above, `clb_I` and `clb_V` are the input parameters of the program *P*, denoting calibrations of voltage and current. Since the qualities of the hardware of different nodes may vary, these parameters are used to offset the differences.

***Faults.*** Two faults occur in the above program. **Fault (a)** affects the correctness of the program. For instance, if `clb_I` is 0 and `I` is 1100, the faulty statement `s1` will compute `T` to be 13 instead of the expected value of 27. **Fault (b)** will cause statement `s2` to re-compute `T` (after being computed once by statement `s0` or `s1`) if the voltage reading fulfills the guard condition. Suppose that each of statements `s0`, `s1`, and `s2` consumes the same amount of energy $\omega$, and that the energy for queries and condition checking is negligible. Then, we have different energy consumptions of the sensor node although the functional correctness of the program is not affected. We should add that, in practice, the code is much more complex than the statements `s0`, `s1`, and `s2` in the example.

To help readers follow our illustration, we note that many physical events have life cycles, in which an event initializes, occurs, evolves, and then fades out. Different sensors may simultaneously observe the same event at the same stage in their life cycles, probably with slight differences in reading, which will be used as input parameters to the WSN applications such as the example program above. For similar input parameters, if two executions of the same program consume excessively different amounts of energy, it may indicate an anomaly in the program.

### 4.4 An Illustration of Our Testing Approach

To adapt metamorphic testing to software applications on wireless sensor networks, testers should determine a source relation *r* and the encompassing metamorphic relation. In the sequel, we first illustrate them using the example application scenario. Next, we will describe how to use them to detect faults (a) and (b).

***Source relation.*** The use of isotropic physical conditions as metamorphic relations has been proposed by Tse et al. [17]. For the testing of WSN applications, however, we need to determine how to obtain such physical conditions. Consider again our application scenario, where a wildfire sends heat to sensors nearby. As sensor nodes are typically deployed in a massive scale, readings sensed by adjacent nodes or nodes in proximity, such as nodes 1 and 5 in Figure 3, should be close enough to be considered equivalent. As we shall explain in the next paragraph, a sense of equivalence is determined by the application itself and we propose to use this important feature of WSN applications as the basis for metamorphic relations.

**Table 1.** Example test cases, their test outputs, and MT results that indicate failures.

| Test case | At node | Temperature of environment | Input components of test case | | Computed result of $P(0, 0)$ | Expected result of $P(0, 0)$ | Computed by | Metamorphic testing result that indicates failure |
|---|---|---|---|---|---|---|---|---|
| | | | Sensed $I$ | Sensed $V$ | | | | |
| $t_1$ | 1 | 27 | 0.110 mA | – | ***13*** | 27 | $s1$ | "$MR_A$: If $r(T_1, T_2)$, then $\approx_P(T_1, T_2)$" is violated because $13 \neq 27$. |
| $t_2$ | 5 | 27 | – | 1.15 V | 27 | 27 | $s2$ | |
| $t_3$ | 1 | 49 | 0.047 mA | 0.04 V | 49 | 49 | $s0$, $s2$ | "$MR_B$: If $r(T_3, T_4) \wedge \approx_P(T_3, T_4)$, then $\phi_3 = \phi_4$" is violated because $2\omega \neq \omega$. |
| $t_4$ | 5 | 49 | 0.060 mA | 0.05 V | 49 | 49 | $s2$ | |

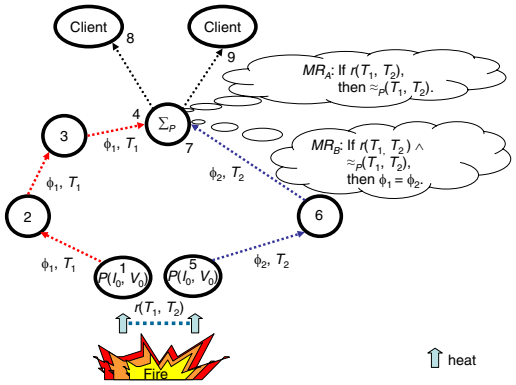


**Fig. 3.** Example metamorphic testing in wireless sensor network.

As shown in the column "Temperature of environment" in Table 1, for example, the temperatures at nodes 1 and 5 are equivalent during the first two scenarios ($T_1$ and $T_2$ as depicted in Figure 3), and again equivalent during the last two scenarios. Testers can control their test cases by controlling the location of the fire and the locations of the sensor nodes. In this way, nodes can be set up to monitor the same physical phenomenon. This source relation is represented by $r(T_1, T_2)$ above the dotted line linking nodes 1 and 5 in Figure 3.

***Metamorphic relations.*** Two metamorphic relations are proposed:

Data aggregation is essential to software applications for wireless sensor networks [10]. As we have explained in Section 4.2, a major property of data aggregators is to eliminate duplicated data. A data aggregator may have a data consolidation criterion $\approx_P(\,)$ to determine whether two data sets resemble each other. In our example, since any two sensory inputs fulfilling the isotropic physical condition $r$ should trigger the program under test to give "equivalent" estimated temperatures, the data consolidation criterion should decide whether two estimated temperatures are close enough to be treated as duplicated data. For the ease of illustration, we shall use simple identity as the data consolidation criterion for the equivalence of temperatures. The resulting metamorphic relation is as follows, and if a pair of test cases and their execution results do not satisfy $MR_A$, it indicates a functional failure.

$$MR_A\text{: If } r(T_1, T_2), \text{ then } \approx_P (T_1, T_2).$$

Let us now consider non-functional testing. In the absence of a test oracle, we cannot isolate functionally passed test cases from failed ones. We resolve to make use of test cases that do not indicate functional failures in terms of metamorphic relations such as $MR_A$. Since equivalent temperature values are determined from the application-specific data consolidation criterion $\approx_P()$ based on an isotropic physical condition $r$, if they need non-equivalent amounts of power for computation, it should indicate a failure. We express this idea as a metamorphic relation.

$$MR_B\text{: If } r(T_1, T_2) \text{ and } \approx_P (T_1, T_2), \text{ then } \phi_1 = \phi_2.$$

***Testing.*** In defining the above metamorphic relations, we recommended to make use of the data consolidation criteria of data aggregation supplied by the application, together with the isotropic physical conditions of sensor nodes in close proximity. A sketch of the scheme is depicted in Figure 3.

We propose to place the task of test result evaluations in the data aggregator component of the application, because the data consolidation criteria constitute part of the logic of the data aggregator. This also relieves sensor nodes from having to evaluate the test results, which would deplete the batteries faster than the original plan of the application designers. Moreover, by utilizing the implementation provided by the application, any failure detected via our approach will indeed indicate a fault in the application. In the sequel, we shall illustrate the usefulness of our proposal in identifying such failures.

To identify a failure due to fault (a), let us consider test cases $t_1$ and $t_2$ in Table 1. [10] Suppose that the input current and voltage calibrations are both zero as indicated in the column captioned "Computed result of $P(0,0)$" in the table. Test case $t_1$ causes $P$ to output 13 via the statement `s1` (as indicated in the column captioned "Computed by") while test case $t_2$ cause $P$ to output 27 via the statement `s2`. Since 13 and 27 are not equivalent, it violates the metamorphic relation $MR_A$ and, hence, reveals a failure. The column captioned "Expected result of $P(0,0)$" shows the expected result of the test case for readers' reference.

To identify a failure due to fault (b), let us consider the test cases $t_3$ and $t_4$ in Table 1. Test case $t_3$ executes both statements `s0` and `s2`, each of which suffices to give the correct result; however, the amount of energy consumed by the test case is $2\omega$. Test case $t_4$ executes statement `s2` only and, hence, the energy consumed is $\omega$. As a result, they consume different amounts of energy. According to $MR_B$, this indicates a failure related to the energy consumption of the temperature monitoring software application.

## 5 Discussions

In our model, we place the implementation of a metamorphic relation in a data aggregation node. There may be other alternatives. In general, one may deploy a piece of software to a resource-stringent existing node (or cluster), to an existing node (or

[10] Source test cases can be generated randomly or via test case selection methods. Discussions on test case generation are beyond the scope of this paper.

cluster) with ample resources, or to a new node (or cluster). Our approach lies with the first option.

The second option further routes the data to a device with an ample amount of resources, such as the server or a client workstation. However, the data communication will consume a lot of energy. Sending all the test verdicts from a massive number of devices to a specific monitoring device may exhaust the batteries of the devices easily. It is, therefore, a less attractive approach.

One may suggest using simulations such as TOSSIM [13] for TinyOS applications, so that simulated batteries can be replenished easily for simulated data communications whenever necessary. However, simulators need profile settings in order to emulate various initial testing conditions. The effort to set up testing profiles to cover different test cases and the respective simulation results would be nontrivial.

The third option requires setting up a new device (or cluster) that may communicate with the existing wireless sensor network. Since existing devices would discover and communicate with the new device (or cluster), this may affect, say, the scheduling and routing of the existing network and, hence, the original configuration to be verified. Our proposal also suffers from a similar limitation, but intuitively to a lesser extent. The impact of these issues warrants further investigation.

Our current proposal does not support streaming data, and has not explored the potential of using other generic characteristics of WSN applications to define metamorphic relations. We are identifying representative applications for case studies and further investigation.

## 6 Concluding Remarks

We share the view of other researchers that there is a mega-trend in computing that many computational units will be shifted to pervasive devices. Software applications running on top of wireless sensor networks are emerging. It is a fast changing field where a mature software development methodology is yet to be defined. This is particularly the case for the testing of such applications, which must be carefully conducted before deployment in real life. Our work is the first research attempt to adapt a testing technique to the wireless sensor network environment.

Unlike their counterparts in conventional computing, wireless sensor networks applications are subject to additional non-functional constraints (such as energy constraint) that may have critical impacts on the behaviors of the software. For example, a functionally correct execution can still be anomalous if it consumes an abnormal amount of energy. In this paper, we have studied the correctness issue in the presence of power-awareness concerns. We have investigated techniques to apply test cases and verify test results related to power-awareness in a WSN environment. In short, our testing technique has been strategically designed to blend in with the special features of the WSN environment.

A sensory node reports readings by gauging the environment. It is, therefore, intuitive to apply test cases in sensory nodes. However, these nodes are typically resource-stringent, such as being equipped with limited amounts of memory and battery power. Executing all the test activities in sensory nodes is impractical. This paper

proposes to execute automated test result evaluations in data aggregation nodes instead. The power-aware transmissions of intermediate and final test data are supported directly by WSN programs. Since the data aggregation feature of WSN programs should be power-aware, it further makes our technique attractive.

Our technique takes advantages of such built-in functionality of data aggregation. We believe in the design rationale held by software engineers when they implement the data aggregation functionality onto WSN nodes in their applications. As such, we propose to use the data consolidation criteria of data aggregators to verify the test results of isotropic physical phenomena of WSN applications in nodes in close proximity to such data aggregation nodes. Since either the original application code or the built-in equality check of the programming language is used to implement metamorphic relations, any failure revealed should be due to a fault in the WSN application under test.

We use a temperature monitoring application scenario to illustrate how to formulate metamorphic relations based on the data consolidation criteria, and to tackle both functional faults and power-related faults. We have taken an initial step to extend the notion of metamorphic testing to deal with non-functional quality aspects. We have also discussed the limitations and alternative approaches in the paper. Future work includes the study of test case selection techniques, further optimization of oracle checks, and the debugging of software applications. We shall report such findings and conduct more empirical evaluations in the future.

## Acknowledgment

We thank the anonymous reviewers for their invaluable advice to improve the paper.

## References

1. FHSST physics electricity: nonlinear conduction. Wikibook. http://en.wikibooks.org/wiki/FHSST_Physics_Electricity: Nonlinear_conduction.
2. JUnit. http://www.junit.org.
3. nesC. http://nescc.sourceforge.net.
4. TinyOS. http://www.tinyOS.net.
5. J. Berstel, S. C. Reghizzi, G. Roussel, and P. San Pietro. A scalable formal method for design and automatic checking of user interfaces. ACM Transactions on Software Engineering and Methodology, 14 (2): 124–167, 2005.
6. R. V. Binder. Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison Wesley, Reading, Massachusetts, 2000.
7. F. T. Chan, T. Y. Chen, S. C. Cheung, M. F. Lau, and S. M. Yiu. Application of metamorphic testing in numerical analysis. In Proceedings of the IASTED International Conference on Software Engineering (SE 1998), pages 191–197. ACTA Press, Calgary, Canada, 1998.
8. W. K. Chan, T. Y. Chen, H. Lu, T. H. Tse, and S. S. Yau. Integration testing of context-sensitive middleware-based applications: a metamorphic approach. International Journal of Software Engineering and Knowledge Engineering, 16 (5): 677–703, 2006.

9. T. Y. Chen, D. H. Huang, T. H. Tse, and Z. Q. Zhou. Case studies on the selection of useful relations in metamorphic testing. In Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC 2004), pages 569–583. Polytechnic University of Madrid, Madrid, Spain, 2004.
10. J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building efficient wireless sensor networks with low-level naming. In Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP 2001), pages 146–159. ACM Press, New York, 2001.
11. J. Kandler. Automated testing of embedded software. International Conference on Software Testing Analysis and Review (STAREAST 2000), Orlando, Florida, 2000. Paper available at http://www.stickyminds.com/s.asp?F=S2049_ART_2.
12. G. M. Kapfhammer, M. L. Soffa, and D. Mosse. Testing in resource constrained execution environments. In Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering (ASE 2005), pages 418–422. ACM Press, New York, 2005.
13. P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: accurate and scalable simulation of entire TinyOS applications. In Proceedings of the 1st ACM Conference on Embedded Networked Sensor Systems (SenSys 2003). ACM Press, New York, 2001.
14. M. A. A. Sanvido, V. Cechticky, and W. Schaufelberger. Testing embedded control systems using hardware-in-the-loop simulation and temporal logic. In Proceedings of the 15th IFAC World Congress on Automatic Control. Barcelona, Spain, 2002.
15. W.-T. Tsai, L. Yu, F. Zhu, and R. Paul. Rapid embedded system testing using verification patterns. IEEE Software, 22 (4): 68–75, 2005.
16. T. H. Tse, F. C. M. Lau, W. K. Chan, P. C. K. Liu and C. K. F. Luk. Testing object-oriented industrial software without precise oracles or results. Communications of the ACM, to appear.
17. T. H. Tse, S. S. Yau, W. K. Chan, H. Lu, and T. Y. Chen. Testing context-sensitive middleware-based software applications. In Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC 2004), volume 1, pages 458–465. IEEE Computer Society Press, Los Alamitos, California, 2004.
18. G. Werner-Allen, P. Swieskowski, and M. Welsh. MoteLab: a wireless sensor network testbed. In Proceedings of the 4th International Conference on Information Processing in Sensor Networks (IPSN 2005), pages 483–488. IEEE Computer Society Press, Los Alamitos, California, 2005.
19. Q. Xie and A. Memon. Designing and comparing automated test oracles for GUI-based software applications. ACM Transactions on Software Engineering and Methodology, to appear.