Fall 2013

Dependence-Based Source Level Tracing and Replay for Networked Embedded Systems

Man Wang
Purdue University

Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Man Wang

Entitled
Dependence-Based Source Level Tracing and Replay for Networked Embedded Systems

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Zhiyuan Li
Chair

Jan Vitek

Xiangyu Zhang

Dongyan Xu

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): Zhiyuan Li

Approved by: Sunil Prabhakar / William J. Gorman 11/26/2013
Head of the Graduate Program Date

DEPENDENCE-BASED SOURCE LEVEL TRACING AND REPLAY FOR NETWORKED EMBEDDED SYSTEMS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Man Wang

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2013

Purdue University

West Lafayette, Indiana

ACKNOWLEDGMENTS

I would like to take this opportunity to express my deepest appreciation to all the people who have helped and supported me during my doctoral study.

I thank Professor Zhiyuan Li for being an ideal advisor. He has provided me with the great combination of guidance and freedom, perspectives on research, technical direction and material support. Without him, the dissertation would not be possible.

I thank Professor Jan Vitek and Professor Xiangyu Zhang for being on my advisory committee, and Professor Dongyan Xu for being an external member on both my preliminary and final exam. Their valuable and constructive comments have greatly improved this dissertation.

I thank my colleagues and friends at Purdue: Lixia Liu, Situ Yingchong, Pengxuan Zheng, Hou-Jen Ko, Ye Wang, Lei Zhao, Yunhui Zheng, Matthew Tan Creti, Hongtao Yu, Feng Li, Guiqin Li, Yalin Dong and Shuxian Jiang. Their collaboration and friendship have made my life as a graduate student so much easier and more enjoyable.

I thank my parents, without whose understanding and unconditional support I would never have started this study.

Finally, I thank my husband Lin Dong, who is always there to provide me with love and support.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Wang, Man Ph.D., Purdue University, December 2013. Dependence-Based Source Level Tracing and Replay for Networked Embedded Systems. Major Professor: Zhiyuan Li.

Error detection and diagnosis for networked embedded systems remain challenging and tedious due to issues such as a large number of computing entities, hardware resource constraints, and non-deterministic behaviors. The run-time checking is often necessitated by the fact that the static verification fails whenever there exist conditions unknown prior to execution. Complexities in hardware, software and even the operating environments can also defeat the static analysis and simulations. Record-and-replay has long been proposed for distributed systems error diagnosis. Under this method, assertions are inserted in the target program for run-time error detection. At run-time, the violation of any asserted property triggers actions for reporting an error and saving an execution trace for error replay. This dissertation takes wireless sensor networks, a special but representative type of networked embedded systems, as an example to propose a dependence-based source-level tracing-and-replay methodology for detecting and reproducing errors. This work makes three main contributions towards making error detection and replay automatic. First, SensorC, a domain-specific language for wireless sensor networks, is proposed to specify properties at a high level. This property specification approach can be not only used in our record-replay methodology but also integrated with other verification analysis approaches, such as model checking. Second, a greedy heuristic method is developed to decompose global properties into a set of local ones with the goal of minimizing the communication traf-

fic for state information exchanges. Each local property is checked by a certain sensor node. Third, a dependence-based multi-level method for memory-efficient tracing and replay is proposed. In the interest of portability across different hardware platforms, this method is implemented as a source-level tracing and replaying tool. To test our methodology, we have built different wireless sensor networks by using TelosB motes and Zolertia Z1 motes separately. The experiments' results show that our work has made it possible to instrument several test programs on wireless sensor networks under the stringent program memory constraint, reduce the data transferring required for error detection, and find and diagnose realistic errors.

1 INTRODUCTION

Networked embedded systems [1], including wireless sensor networks (WSNs), distributed control systems, and so on, have gained increasing attention due to their wide range of applications. To improve the reliability of such systems, techniques ranging from static program analysis to model checking and so on, have been proposed. However, due to their attributes such as large number of computing entities, hardware resource constraints, and non-deterministic behaviors, runtime error detection and diagnosis for networked embedded systems is needed.

Run-time checking [2–4] is often necessitated by the fact that static verification fails whenever there exist conditions unknown prior to execution. Complexities in hardware, software, and even the operating environment can also defeat a static analysis. Deterministic replay (or record-replay) is an error diagnosis method which has long been proposed for distributed systems to enhance the programmer's ability to find complex software errors. Under this method, nondeterministic events are recorded throughout the system operation. When an error is reported, the program can be re-run with the recorded events. The replayed program can reproduce the detected error and allow the programmer to inspect the executed statements. Therefore, the source of the error, namely the incorrectly written statements or unexpected events causing the error, can be located by the programmer with the understanding of the incorrect execution.

Although numerous replay techniques [5–9] that differ mainly in how to handle nondeterministic events and reduce runtime overhead have emerged, two main obsta-

cles prevent the practical use of deterministic replay in networked embedded systems for error detection and diagnosis.

The first obstacle regards the method of detecting run time errors that are not as obvious as the system crash. Both the deterministic replay and software verification tools assume the availability of specifications of correct behaviors (or inversely, incorrect behaviors). The most natural way to specify the correct behaviors of a networked embedded system is to define a set of global properties that must be satisfied by quantities collected from multiple nodes, since most of the time, nodes in the entire system work together to perform a task. It is commonly known that, in most cases, a global property cannot be verified in a static way because there will likely be quantities that are unknown until the system is either deployed or simulated. One way to verify global properties at run time is to perform a centralized checking [10], i.e., to select a single node (e.g. base-station) so as to collect all the quantities from the nodes involved in the predicates. Such a centralized approach, however, requires all the nodes whose states are involved in the global property to send their execution traces to the base station, which incurs bookkeeping and networking cost that are excessive in most cases. Furthermore, the program execution, including message passing, on all the involved nodes must be replayed in order to diagnose the error. In contrast, by using a global property that can be decomposed into a conjunction of several sub-properties such that each involves only the quantities from a single entity, the ideal goal of distributed local checking by multiple entities can be attained. In general, a networked embedded system reacts to events whose timing is difficult to predict or to specify during the program development, and the data transmission is complex and dynamic. Due to these reasons, it is often a challenging task for the application programmer to decide what sub-properties to check for each node.

The second obstacle is how to run deterministic replay efficiently in networked embedded system. On the one hand, even for resource-rich distributed and parallel systems, the high runtime overhead is a traditional barrier for deterministic replay. Some recent works try to reduce the runtime overhead of production time in the cost of increasing debugging time. However, these approaches may fail to expose the real root of an error [11]. For the networked embedded systems, this overhead is more critical. In addition, in the resource-rich systems, compared with the original program size and the available memory, the instrumented code size is too small to be a problem. However, the limited program memory on embedded devices (e.g., 48 KB on the popular TelosB motes used in wireless sensor networks) has forced most existing schemes for run-time tracing on networked embedded systems to record only coarse information, which is far from sufficient for deterministic replay. This makes it difficult to pinpoint the source of the errors which are detected at run time. On the other hand, an error could propagate along with the message transmission. As a consequence, an error may be caused by the incorrect execution on one node but later detected on another node. Distributed replay can faithfully reenact the runtime execution, but if the system has a large number of nodes and complex message transmission, the programmer may not be able to reason logically about the recorded logs and to find the errors. From our experiments, which will be discussed in Chapter 5, not all the nodes need to be analyzed for locating the error, especially in a large networked embedded system. Replaying all the nodes together will generate redundant information which would actually hamper the programmer's ability to inspect the execution trace efficiently. Therefore, trimming redundant traces without losing the accuracy of the diagnosis also has an important role in deterministic replay.

By focusing on the two main challenges stated above, this dissertation investigates how to detect and diagnose errors efficiently in networked embedded systems. As

one special type of the networked embedded systems, wireless sensor networks [12] are gaining an increased attention for their possible wide use in applications such as military applications, structural health monitoring, environmental surveillance, scientific observation, industrial monitoring, and so on [13–17]. A WSN can consist of many small sensor nodes, each of which is an embedded system and communicates with others via a radio transmitter. Severe resource constraints represent one of the distinctive features of wireless sensor networks, which makes the conventional error diagnosis techniques derived from wired networks not be suited for WSNs [18]. In particular, small memory (Table 1.1 lists several popular commercial motes/sensor nodes) makes applying tracing and replay approach on WSNs more challenging as we discussed above. That is why, we use WSNs, a special kind of networked embedded systems, as our experiment target object. We believe that our proposed methodology and the developed tools can be extended to other types of networked embedded systems.

Table 1.1 Sensor nodes

Name	RAM	Flash Memory	Micro-controller
Mica2 [19]	4KB	128KB	ATMEGA 128L
MicaZ [20]	4KB	128KB	ATMEGA 128
TelosB [21]	10KB	48KB	Texas Instruments MSP430 micro-controller
Zolertia Z1 [22]	8KB	92KB	Texas Instruments MSP430F-2617
TinyNode [23]	8KB	512KB	Texas Instruments MSP430 micro-controller

The main contributions of this dissertation are as follows:

(1) We design a domain specific language SensorC and develop the corresponding SensorC compiler to specify the global properties which must be satisfied when the system and its application software are deployed. The SensorC language allows programmers to specify properties using propositional logic with bounded time variables. It also provides programmers more flexibility to describe a fine system behavior by introducing the network topology and routing information. This property-specification approach can be not only used in our record-replay methodology, but also integrated with other verification analysis, such as model checking.

(2) We design a Global Property Decomposition (GPD) algorithm to decompose global properties automatically into local ones, which can be detected on single sensor nodes. This decomposition can (a) reduce the communication traffic for state information exchanges caused by centralized checking and (b) reduce the collected trace used for replay. This GPD algorithm is integrated with our developed SensorC compiler.

(3) We present a dependence-based source level tracing and replay method for error diagnosis. This scheme lends an effective solution for the memory size problem. We develop a source-level tracing and replaying tool which is independent of the hardware platforms and the cross compiler (except for a system library call to make certain memory accesses atomic). The source-level tracing, compared with the assembly-level tracing, offers a high portability of the tool. It also enables the user to take advantage of the many existing source-level debuggers, such as GNU's gdb, when replaying on a desktop machine.

(4) To improve the efficiency of replay, especially when an error propagates along with messages in a large WSN, we design a new program analysis which identifies program sub-traces that can be skipped for replay without losing the accuracy of the diagnosis.

(5) We build WSNs by using TelosB motes and Zolertia Z1 motes separately to test our methodology. The results of our experiments demonstrate that our work has made it possible to instrument several test programs on wireless sensor networks under the stringent program memory constraint, reduce the data transferring required for error detection, and find and diagnose realistic errors.

The rest of the dissertation is organized as follows. Chapter 2 defines the problem model addressed by this dissertation and gives an overview of our work. Chapter 3 first presents the SensorC specification language and the corresponding SensorC compiler, and then discusses the Global Property Decomposition algorithm. Chapter 4 discusses the record and replay methodology for a single node. Chapter 5 proposes the ways to extend this methodology to the entire system. Chapter 6 summarizes related works on wireless sensor network debugging, deterministic replay, and system behavior synthesis. Finally, Chapter 7 offers the main conclusions and proposes future work.

2 OVERVIEW

In this chapter, we define the error detection and diagnosis problem discussed in this dissertation, and then we propose the methodology and system framework at a high level.

2.1 Error and Error Source

First, we discuss the detected error and the error source targeted in this dissertation.

Error The types of errors targeted by our scheme go beyond the system crash. In order to verify if a program is implemented correctly at run-time, the application programmer may specify a set of correctness properties, e.g., sensor data must be reported from each mote to the base station within a certain time limit. Such properties are specified using predicates defined over a list of program variables under a certain system of logic, e.g., temporal logic [24]. The program is required to satisfy this set of properties within a specified program scope, e.g., the entire program (as long as all the variables in the predicate are global), individual functions, individual program segments, or any point between two specific program statements. In this dissertation, *an error* is detected if any violation of certain properties can be detected at run time.

Error Source Assuming that the property correctness predicates themselves are composed correctly, when a predicate becomes violated, we know that at least one of its variables has obtained an incorrect value through some point in the

program where the variable was updated. That value may be the result of earlier operations that used incorrect operands. Eventually, the violation must be traced back to its source through a chain of data/control dependence and/or message send/receive relationship. Although the properties violation, or errors, caused by hardware, such as messages delay/loss in the network, can be detected at run-time by our error detection method, locating the origin of the violation requires debugging not only the program implemented by programmers but also the hardware equipments. Our error diagnosis work focuses on software bugs. Therefore, for the purpose of this work, we consider the possibility of error sources: one or more program statements are written incorrectly. This possibility can cause certain unexpected events to occur, e.g., messages are sent with incorrect contents. Therefore, the error source can be located in the same sensor node where the error is detected, or it can also be located in a different node due to error propagation along with message transmission.

2.2 Global Properties vs. Local Properties

The properties considered in this dissertation are those that can be formally specified by propositional logic [25]. Time variables are allowed in the predicates as long as arithmetic expressions containing such variables are bounded by constants. For example, one may require the network routing protocol to establish a path from any arbitrary sensor mote to the base station within t time units. As another example, the clause "event $E1$ must happen before event $E2$" is also allowed as long as both events can be timed during the operation. One can then simply state that "$E1.time < E2.time$." However, clauses such as "event E eventually happens" are by nature not suitable for error detection because the fact that E has not happened yet never indicates an error. Hence, such clauses are not considered in this dissertation.

Based on the nodes involved, we categorize properties targeted by this dissertation into two classes, global properties and local properties, which are defined as follows:

Definition 1 *Let W be a wireless sensor network that contains n nodes (also known as motes). The programs executing on different nodes do not need to be the same. If a property P concerns only the state of the program execution on node i, $1 \leq i \leq n$, we call P a local property on node i. Any property that concerns the program states on more than one node is called a global property.*

In the distributed approach stated in the introduction, the global property is first rewritten as a conjunction, such that each conjunctive clause P_{i} is assigned to a node i. If P_{i} contains a data item, d_{i}, that is not local to node i, then message passing must be inserted in the programs such that the up-to-date copy of d_{i} is made available on node i, perhaps after a certain delay. We say that d_{i} becomes local. After all the remote data items become local, P_{i} becomes a local property that can be verified on node i.

Formally, let P_{G} represent a global property and L_{i} $(1 \leq i \leq n)$ a local property of node i. If we have

$$P_{\mathrm{G}} = L_1 \wedge L_2 \wedge \ldots \wedge L_{\mathrm{n}}, \tag{2.1}$$

and then the violation of P_{G} can be represented by

$$\neg P_{\mathrm{G}} = \neg L_1 \vee \neg L_2 \vee \ldots \vee \neg L_{\mathrm{n}}. \tag{2.2}$$

With Equation 2.2, we can insert operations to evaluate each predicate L_{i} on node i. If the truth value is false, node i reports the error to the base station. Assuming there to be no constraints on the placement of message passing, there will be, in general, more than one way to decompose any given global property. The minimization of message passing is used in this dissertation as the metric of optimality.

2.3 Deterministic Tracing and Replay

If the entire sequence of executed instructions and operands are recorded, one could follow the dynamic use-def chain (in a node) or send-receive chain (between nodes)backward to inspect the program statements along the way until the origin of the error is found. The cost of such a complete recording is prohibitive in both time and space. Under the record-and-replay scheme, however, we only need to record all the nondeterministic events on each mote, which mainly includes all external messages, task scheduling decisions, hardware register status, and internal interrupts in a wireless sensor network.

2.3.1 Source-Level Instrumentation

To record run-time logs, we can use either of the two types of instrumentation compared in Table 2.1 [26]. In our work, we have adopted source-level instrumentation tracing and replaying methodology due to the following reasons.

(1) The source-level instrumentation offers a high portability across different devices. There are a variety of embedded system platforms. However, C is the most common language for embedded systems [27]. In addition, it is easier for programmers to modify the software rather than the hardware.

(2) The objective of our work is not only to detect errors but also to help programmers locate error sources in the original program. Therefore, the logs and replayed traces must be readable and tractable. The source-level instrumentation allows execution behaviors to be linked back to source code. It is obvious that reading source code is easier than reading binary code, and the original data/control dependence can be observed.

(3) By applying source level instrumentation, our approach enables the user to take advantage of many existing source-level debuggers, such as GNU's gdb, when replaying on a desktop machine.

Table 2.1 Source-level instrumentation vs. binary instrumentation

	Source-Level Instrumentation	Binary Instrumentaion
Portability	Good across devices	Good across languages
Source Correlation	Possible	Generally impossible
Generated Replay Code	Source level, easily understood by programmers	Binary level

After an error is found, we still leave the instrumented code in the program. This is because the program may still have other hidden errors. Moreover, removing instrumentation may cause certain timing-dependent errors to resurface, and we lose the means to record the trace.

2.3.2 Main Assumptions

Since the program on each sensor node may run indefinitely, the length of the trace is unbounded. With limited storage for the trace, generally one retrieves only a tail of the full trace. Replay is therefore often partial in practice. In order to enable deterministic replay corresponding to the retrieved trace tail, we require the considered program to satisfy the following assumptions:

Assumption 1: The infinite running of the program is controlled by one or more infinite loops which are recognized at compile time.

Assumption 2: The source code of each application is available. If some functions such as system call is unreachable, their side effects are predicable.

Under Assumption 1, we insert in each infinite loop an *anchor checkpoint*(or anchor point) at which we record the values of all variables needed to enable replaying the program starting from this program point. The function containing an anchor point is called *boundary function*. The local variables of the callers of a boundary function are not recorded at the anchor point.

Assumption 2 guarantees that the source-level instrumentation is applicable.

2.4 System Framework

Figure 2.1 is the framework of the whole system, which contains three parts:

Part I: Given a global property, the system automatically decomposes it into several local properties which could be inserted into different nodes. The design and implementation details of this part are discussed in Chapter 3.

Part II: Using the error checking invariants as the slicing criteria, a compiler is used to do backward slicing and dependence analysis to decide and instrument code for run-time logging. If the error checking invariants are different on each node, the instrumented program running on each node might be different. For this part, Chapter 4 focuses on a single node and Chapter 5 extends the work to the entire system.

Part III: When an error is reported, the trace of each node is retrieved, and the replayed program, which will execute on a desktop machine and reproduce the reported error, is generated. The work of this part is presented in both Chapter 4 and 5.

We use TinyOS [28], which is written in nesC [29], as our testing environment for the developed tool discussed in the following chapters. TinyOS is one of the most popular operating systems for sensor network applications. It has been used by more

than 100 research groups worldwide. TinyOS supports an event-driven concurrency model, the core of which consists of tasks and interrupt handlers [30]. Tasks are run to completion, and they are not preempted by each other. However, tasks are not atomic to interrupt handlers which can be triggered at any time. Typically, a TinyOS application consists of a group of components which are wired together by a top-level configuration. The nesC compiler first converts the application into a C program, which is then compiled by the cross compiler into machine code executable on the specific hardware. Although we use TinyOS application as our test cases, our work focuses on the intermediate C program generated by nesC compiler. Therefore, our methodology and implementation can be easily extended to other networked embedded systems whose applications are written by C or can be transferred to C language as nesC does.

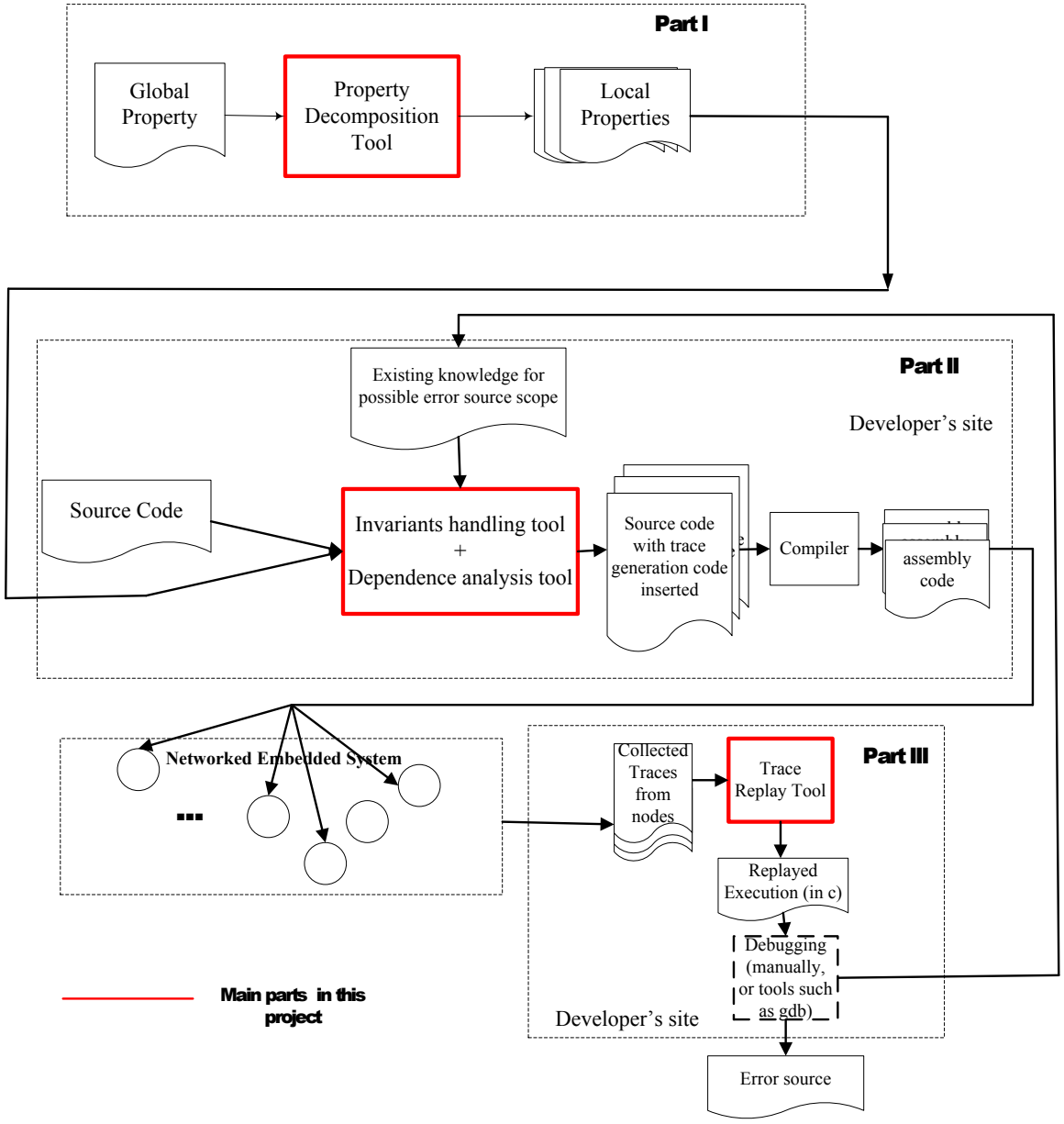


Figure 2.1. Framework of the proposed system

3 GLOBAL PROPERTY VIOLATION DETECTION

In this chapter, we discuss how to define a property using propositional logic and how to convert a defined global property into local invariants inserted into programs installed on motes for runtime checking. In Section 3.1, we design a domain specific language, called SensorC, to specify global properties for WSN applications. Section 3.2 discusses the design of a source-to-source compiler, simply called the SensorC compiler to decompose global properties into conjunctions of local properties. The framework is illustrated in Figure 3.1.

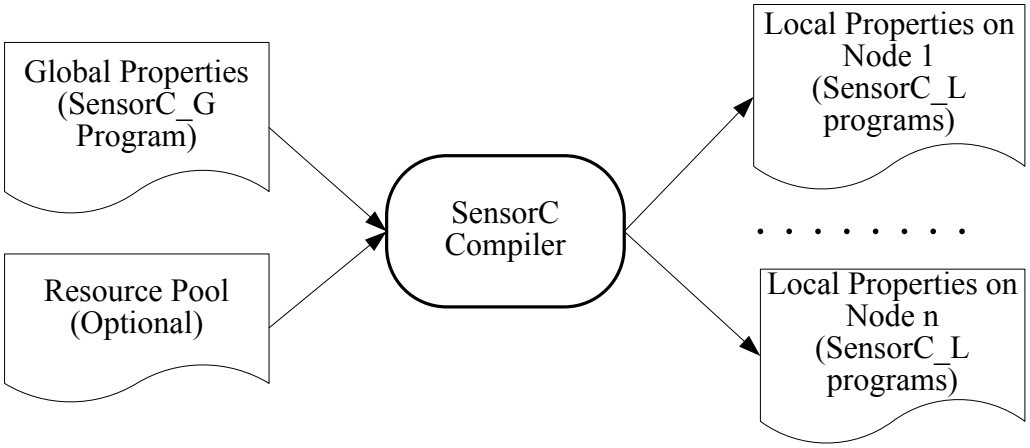


Figure 3.1. Framework of property decomposition tool

3.1 SensorC: How to Specify Properties

SensorC is a domain specific language which can be used to specify WSN global properties and local properties. It is designed for programmers who want to specify properties for detecting their own WSN programs. The global property specification

written in SensorC is called a SensorC_G program, which refers to the variables in the given application program to specify the desired property. Illustrated in Figure 3.1, a sensorC_G program will be translated by the SensorC compiler into a set of SensorC_L programs after decomposing the global properties into the conjunctions of local properties. Each SensorC_L program, which is responsible for checking the local properties, is integrated into the application program for recording the trace needed for replay in case an error is detected. The integrated program will be deployed and run on a certain node (A set of nodes may run copies of the same program). Programmers need to write their own SensorC_G program and the corresponding SensorC_L programs will be generated automatically by SensorC compiler.

As an example, consider the Loop_Free property required for the AODV routing protocol [31]. This property states that the routing tables of all nodes must not form a routing loop. This property can be specified by the SensorC_G program shown in Figure 3.2(a), which is then translated by the SensorC compiler into SensorC_L programs. Figure 3.2(b) shows the SensorC_L program that contains the local property to be checked at run time on node 7.

In the following, we present the syntax forms of SensorC_G and SensorC_L programs and describe how to define global properties checking using such syntax forms.

3.1.1 SensorC_G

A SensorC_G program begins with a pragma #GLOBAL and is followed by four program segments: two essential segments #DEFINITION and #PROPERTY, and two optional segments #NETWORK and #ROUTING. The main production rules are listed in Figure 3.3, where the bold words represent terminal symbols. This section only discusses the essential segments and the optional ones will be discussed in Section 3.3.

```
#GLOBAL
#DEFINITION
GLOBAL var rt = AODV_M__route_table_;
TEMP P1,P2;
NODE n = [ALL];
NODE d = [0];
NODE m = [ALL];

#NETWORK
#ROUTING

#PROPERTY
P1 = (rt[n].dest == d) AND (rt[n].next == m) ;
P2 = (rt[n].seq < rt[m].seq) OR
       (rt[n].seq == rt[m].seq AND
        rt[n].hop>rt[m].hop);
IF ( NOT (P1->P2)) ERROR ;
```

(a)

```
#LOCAL 7
#DEFINITION
GLOBAL var rt = AODV_M__route_table_;
NODE m = [ALL];

#PROPERTY
IF ((rt.dest==0) AND (rt.next==m))
    IF ((rt.seq >= rt[m].seq) AND
              (rt.seq <> rt[m].seq OR
               rt<=rt[m].hop))
         ERROR;
```

(b)

Figure 3.2. Examples of SensorC programs. (a) SensorC_G; (b) SensorC_L

```
segment1→ #DEFINITION
          declList
declList  → ε | declList decl
decl      → GLOBAL globaldecl; | TEMP declname;  | NODE declname = [indices];
globaldecl →alias_event | alias_var
alias_even → event declname = app_expression
alias_var  → var   declname = app_var
indices    → ALL | nodeIDlist
```

(a)

```
segment2 → #NETWORK
           file
file     → ε | file_name;
```

(b)

```
segment3 → #ROUTING
           protocol
protocol → ε | protocol_name;
```

(c)

```
segment4    → #PROPERTY
              propertyDef
propertyDef → ε | propertyDef assignment | propertyDef prochecking
assignment  → declname = propositional_formula ;
prochecking → IF (NOT property) ERROR ;
property    → propositional_formula
propositional_atom → boolean_expression | wsn_expression | #(event)
wsn_expression → declname SEND data TO declname  cond
                 | declname RECEIVE data FROM declname  cond
                 | declname BROADCAST data cond
cond        → ε| condition| time
time        → LESS_THAN x | EVERY x | NEVER
```

(d)

Figure 3.3. Basic grammar for writing SensorC_G programs

#DEFINITION

As illustrated in Figure 3.3(a), the #DEFINITION segment contains a list of declarations. There are three types of declarations (symbol *decl*): GLOBAL, TEMP and NODE, which are explained as follows:

GLOBAL This group of declaration establishes correspondence between the variables used to specify properties (in the later #PROPERTY segment) and the ones appearing in the WSN program. There are two kinds of GLOBAL declarations. The first kind, alias_event, corresponds to events recognized and handled by the WSN application, and the second kind, alias_var, corresponds to program variables. In the third line of Figure 3.2(a), rt corresponds to application variable AODV_M__route_table_, which points to the routing table used in the AODV protocol. As shown in Figure 3.2(b), the declaration of rt is copied to the generated local property specification.

TEMP If a variable is declared as TEMP, then it is a temporary variable used to build a property and does not correspond to any WSN program variable. After global property decomposition, TEMP variables will disappear from the SensorC_L programs.

NODE A NODE variable is used to identify a node in the WSN. In the next program segment, NODE variables will be mapped to real node IDs referenced in the WSN application. In the grammar, the non-terminal symbol nodeIDlist can represent (i) a single integer; (ii) a list of integers which are separated by commas; or (iii) a list of consecutive integers in the form of "first_integer ... last_integer". In Figure 3.2, variable d can only represent the node whose id index is 0, while n and m represent all nodes in this WSN.

#PROPERTY

This segment specifies properties in terms of variables declared in the #DEFINITION segment. Each property is represented as a propositional formula [25]. The propositional atom used in the formulas can be any Boolean expression or wsn_expression as defined in Figure 3.3(d). The production rules define three main types of constructs: (i) productions that specify assignments of propositional formulas to TEMP variables; (ii) productions that define properties in the form of propositional formulas; (iii) the "prochecking" production which explicitly claims that, if a certain property is violated, then the ERROR condition must be raised. The programmer can make as many claims as desired. At run time, as soon as any property is violated, an error is reported.

Unlike in the #DEFINITION segment, all GLOBAL variables used in the #PROPERTY segment are followed by a pair of square brackets (i.e. "[]") which encloses a NODE variable (e.g. [n]) or an integer (e.g. [0]). This indicates the node from which the data will be collected to check the property.

Consider the "Loop_Free" example in Figure 3.2 again. Every reference to GLOBAL variable rt is followed by a NODE variable. The property defined here means that, if node m is the next hop of node n towards the destination d ($P1$), the corresponding sequence number in $n's$ routing table should be less than the one in $m's$ routing table. If, instead, they are the same, then the number of hops from n to d should be greater than the number of hops from m to d ($P2$) [32].

3.1.2 SensorC_L

Different from SensorC_G programs which are written by programmers, a SensorC_L program is generated by the SensorC compiler. It shares most of the gram-

mars with SensorC_G and its main differences from the SensorC_G counterpart are listed as follows.

(1) A SensorC_L program starts with the "#LOCAL" pragma which is followed by a node ID to indicate on which node this local property will be checked at run time.

(2) Compared with SensorC_G programs, a SensorC_L program contains only two essential segments: #DEFINITION and #PROPERTY, whose grammars are similar to that for a SensorC_G program. In the #PROPERTY segment, if a GLOBAL variable represents the data from this local node, the index does not have to be added. For example, in Figure 3.2, the "*rt.dest*" means the value of *rt.dest* collected from node 7.

(3) A SensorC_L program allows another form of property checking (represented by the production for nonterminal prochecking2) in the #PROPERTY segment. The production rule is shown in Figure 3.4. The reason for introducing this type of production will be explained in the next sub-section.

```
prochecking2 → IF (property_1_violation)
                   IF (property_2_violation)
                      ERROR ;
```

Figure 3.4. A production rule for property checking in SensorC_L

3.2 How to decompose a global property

As discussed in Section 2.2, if a global property can be rewritten as Equation 2.1, the violation can be represented by Equation 2.2. We can then follow Equation 2.2 to insert local invariants into different nodes for error detection. We now present a

property decomposition algorithm to find a set of L_{i} which satisfy Equation 2.1. In order to describe the decomposition algorithm, some definitions are made as follows.

Definition 2 *Consider a property P specified in the #PROPERTY segment. If it contains any NODE variables, it is called a template property. By replacing each NODE variable with any integer belonging to its assigned indices, the generated new property is called an instantiated property of P.*

For example, suppose a WSN has 10 nodes. The property defined in Figure 3.2 has 10*10=100 instantiated properties (without explicitly declaration, different NODE variables used in one property may represent the same node) , an example of which being the following:

$$((rt[1].dest == 0) \wedge (rt[1].next == 2)) \rightarrow$$
$$(rt[1].seq < rt[2].seq \vee (rt[1].seq == rt[2].seql \wedge rt[1].hop > rt[2].hop))$$

Obviously, if any instantiated property is violated at run time, an error must be reported.

Definition 3 *For a property assigned to node i, let V (L_i) be the set of GLOBAL variables in L_i. For each variable ($v \in V(L_i)$), if the index followed by v is equal to i, v is a local variable of node i. If the index followed by v is equal to $j(j \geq 0, j \neq i)$, then v is a remote variable of node i. If all variables used in L_i are local variables, then L_i is a local property.*

3.2.1 A Global Property Decomposition (GPD) Algorithm

We follow Algorithm 1 to decompose the global properties. First of all, for each global property P, we substitute each TEMP variable with its defined expression(step

1), which can be achieved by backward substitution. The substitution does not change the truth value of P.

Algorithm 1 *(GPD algorithm) Decompose a global property P into local properties*

1: Replace TEMP variables used in P by their assigned values recursively, until there are no TEMP variables any more.
2: If P is a template property, enumerate all the possibilities of each NODE variables in P to generate S, a set which contains all t instantiated properties of P:

$$S = \{P_1, P_2, \ldots, P_{\mathrm{t}}\}$$

3: For each P_{i} , let P'_{i} be the formula converted from P_{i} by replacing wsn_expression (if there is any included in P_{i}) based on the rules listed in Table 3.1 and Table 3.2.
4: **for** $i = 1 \rightarrow t$ **do**
5: Convert P'_{i} to its equivalent Conjunctive Normal Form:

$$P'_{\mathrm{i}} = P'_{\mathrm{i,1}} \wedge P'_{\mathrm{i,2}} \wedge \ldots \wedge P'_{\mathrm{i,ki}}$$

6: For each $P'_{\mathrm{i,j}}$, find the node *loc* that, if property $P_{\mathrm{i,j}}$ is checked on node *loc*, $P'_{\mathrm{i,j}}$ contains the smallest number of remote variables.
7: **end for**
8: Let $L_{\mathrm{i}} = \wedge(P'_{\mathrm{i,j}}|P'_{\mathrm{i,j}}$ will be checked on node $m_{\mathrm{i}}, 1 \leq i \leq t, 1 \leq j \leq ki)$. If there is no any such $P'_{\mathrm{i,j}}$, $L_{\mathrm{i}} = NULL$. $\diamond$

In the second step, all instantiated properties of P are generated. The reason for generating instantiated properties is to accurately calculate the number of remote variables in the next steps and to reduce the data transmission during run time error detection. In Algorithm 2 presented later, we will regroup suitable instantiated properties and reintroduce NODE variables to make the local properties compact, which will reduce the program size and the run time checking cost. According to Definition 3, we get Equation 3.1.

$$P = P_1 \wedge P_2 \wedge \ldots \wedge P_{\mathrm{t}} \tag{3.1}$$

Table 3.1 Property checking description

Property Checking Function	Property Checking Description
Send_rule(n, m, *data.type*, *cond*)	On node n, check: node n must successfully send message[src:n, dest:m, type:*data.type*] under the condition *cond*
Recv_rule(m , n, *data.type*, *cond*)	On node n, check: node n must successfully receive message [src:m, dest:n,type:*data.type*] under the condition *cond*
Routing_rule(*src*, *dest*)	Based on WSN domain knowledge, for any node on the path from src to dest, once it receives a message with destination *dest*, it must forward the msg to its next hop.

Table 3.2 Conversion rules

wsn_expression	Property Checking Function	
	Without #NETWORK and #ROUTING	With #NETWORK and #ROUTING
n SEND *data* TO m *cond*	Send_rule(n, m, *data.type*, *cond*), Recv_rule(m , n, *data.type*, *cond*)	Send_rule(n, m, *data.type*, *cond*), Recv_rule(m, n, *data.type*, *cond*), Routing_rule(n, m)
n RECEIVE *data* FROM m *cond*	Recv_rule(n, m, *data.type*, *cond*), Send_rule(m , n, *data.type*, *cond*)	Recv_rule(n, m, *data.type*, *cond*), Send_rule(m , n, *data.type*, *cond*), Routing_rule(m, n)
n BROADCAST *data cond*	Send_rule(n, $n's$ *neighbour*, *data.type*, *cond*)	Send_rule(n, n's *neighbour*, *data.type*, *cond*), Recv_rule($n's$ *neighbour*, n, *data.type*, *cond*)

In WSN applications, the wsn_expression defined by SensorC grammar is used as a propositional atom and can be checked locally without requiring any extra information. However, to further reduce the replay workload, step 3 applies the substitution rules in Table 3.2, which introduce redundant error detection, to replace each wsn_expression in a property by pre-defined property-checking functions whose semantics are defined in Table 3.1. To be consistent with the result after step 2, all variables used in Figure 3.2 belong to the same node. Each replaced wsn_expression involves only data from a single node and it can be checked locally. To explain the reason of introducing redundant error checking, we consider a property "After the first 10 seconds, base-station must receive messages from other nodes every 5 seconds" which should be satisfied. Assume node 1 fails to send any message to base-station. If the property is only detected on the receiver node, once an error is detected, we must first replay the receiver node. Since no error is found in the program executing on the receiver, we need to replay all the nodes on the possible paths from the sender

to the receiver. In contrast, with redundant error detection inserted in the sender node (step 3 of Algorithm 1), the error can be detected before it is propagated from where it originates. Hence, only the sender needs to be replayed. The redundant error detection is performed locally and no extra message transmission is required.

All logical formulae can be converted into an equivalent formula in conjunctive normal form by utilizing logic equivalence laws [25]. Step 4-7 converts each $P'_{\mathrm{i}}(1 \leq i \leq t)$ into its equivalent conjunctive normal form, and we can get Equation 3.2.

$$P = \begin{array}{r} P'_{1,1} \wedge P'_{1,2} \wedge \ldots \wedge P'_{1,\mathrm{k1}} \wedge \\ P'_{2,1} \wedge P'_{2,2} \wedge \ldots \wedge P'_{2,\mathrm{k2}} \wedge \\ \ldots \\ P'_{\mathrm{t},1} \wedge P'_{\mathrm{t},2} \wedge \ldots \wedge P'_{\mathrm{t,kt}} \end{array} \tag{3.2}$$

Step 6 is responsible for finding a node to check each sub property $P'_{\mathrm{i,j}}$. If we decide to check $P'_{\mathrm{i,j}}$ on node i, we have the local property $L_{\mathrm{i}} = P'_{\mathrm{i,j}}$. We obtain L_1, L_2 ,..., L_{n} by changing the node on which $P'_{\mathrm{i,j}}$ is checked. The equation $P'_{\mathrm{i,j}} = L_1 = L_2 = \ldots = L_{\mathrm{n}}$ is established in terms of the checking result (or truth table). However, as stated by Definition 3, the number of remote variables may vary for each $L_{\mathrm{i}}(1 \leq i \leq n)$. Every time when a local property is checked, the larger number of remote variables will cause more data transmission. Therefore, we try to find the way which can minimize the message passing.

To formalize the optimization problem, we suppose that there are n nodes in a WSN and m global properties to check. Further we suppose that, after converting every global property into CNF we have t clauses in total. Each clause must be assigned to one and only node for checking. Let R be a matrix with t rows and n columns. Each element in the matrix is a set $R[i][j]$ $(1 \leq i \leq t, 1 \leq j \leq n\ )$ that contains distinct remote variables in clause C_{i} if it is assigned to node j to check.

Otherwise, $R[i][j]$ is an empty set. Given an assignment decision, we define the message passing cost for checking all clauses assigned to node j as the cardinality of the union of $R[i][j]$, $1 \leq i \leq t$. Adding such costs for all j, $1 \leq j \leq n$, we have the total cost. Our minimization problem is to find an assignment decision such that the total cost is minimized, i.e., $\sum_{j=1}^{n} |\bigcup_{i=1}^{t} R[i][j]|$ is minimized. We name the minimization problem stated above as clause-assign problem. Restating this optimization problem as a decision problem, we wish to determine whether a matrix R has an assignment of a given cost k. The formal definition is

CLAUSE-ASSIGN=$\{< R, k >$: Matrix R has an assignment, which allows $MIN = \sum_{j=1}^{n} |\bigcup_{i=1}^{t} R[i][j]| =$ k. $\}$

We can proof the CLAUSE-ASSIGN is a NP-hard problem.

Theorem 3.2.1 *CLAUSE-ASSIGN is NP-hard.*

Proof It is known that the VERTEX-COVER problem is NP-Complete [33]. We will show that VERTEX-COVER $\leq_{\mathrm{p}}$ CLAUSE-ASSIGN.

The reduction algorithm takes as input an instance $< G, k >$ of the VERTEX-COVER problem. Let $G =< V, E >$. We generate a $|E| * |V|$ matrix R as follow:

1. If edge e_{i} is covered by a vertex v_{j} in G, $R[i][j] = \{j\}$.

2. If edge e_{i} is not covered by a vertex v_{j} in G, $R[i][j] = \phi$.

The time complexity of this matrix generation is $O(|E||V|)$. Now we prove: the graph G has a vertex cover of size k if and only if the matrix R has an assignment, which let MIN be k.

$\Rightarrow$: Suppose that G has a vertex cover V' with $|V'| = k$. Then we select k columns each of which is corresponding to a vertex in V'. Since V' covers all edges, in the selected columns, for each row, there is at least one element that is not empty. If there are more than one non-empty in a row, we just keep the one with the smallest

j (i.e. the column index). Since for each column, all the elements are the same, the MIN of matrix R equals to the number of columns, i.e. k.

$\Leftarrow$: Suppose matrix R has an assignment which makes $MIN = k$. For each column j, since $R[i][j](1 \leq i \leq |E|)$ is either $\{j\}$ or ϕ, $|\bigcup_{i=1}^{|V|} R[i][j]|$ can be only 1 or 0. Let V be the set of columns, where $V = \{j||\bigcup_{i=1}^{|V|} R[i][j]| = 1\}$. According to how MIN is calculated, we get $|V| = k$. In addition, R has an assignment which means there is one and only one non-empty element in each row of R. Therefore, the column index of each non-empty element must belong to V. As a result, in graph G, all edges are covered by the k vertices. ■

Therefore, we take a greedy approach that assigns $P'_{\mathrm{i,j}}$ to L_{i} (such that $P'_{\mathrm{i,j}} = L_{\mathrm{i}}$), where L_{i} contains the fewest remote variables.

No $P'_{\mathrm{i,j}}$ is modified in step 6. The only operation is that it is assigned to one and only one node. Consequently, we can deduce that, after step 6 is executed,

$$L_1 \wedge L_2 \wedge \ldots \wedge L_{\mathrm{n}} = \begin{array}{r} P'_{1,1} \wedge P'_{1,2} \wedge \ldots \wedge P'_{1,\mathrm{k1}} \wedge \\ P'_{2,1} \wedge P'_{2,2} \wedge \ldots \wedge P'_{2,\mathrm{k2}} \wedge \\ \ldots \\ P'_{\mathrm{t},1} \wedge P'_{\mathrm{t},2} \wedge \ldots \wedge P'_{\mathrm{t,kt}} \end{array} \tag{3.3}$$

From Equation 3.2 and Equation 3.3, we get:

$$P = L_1 \wedge L_2 \wedge \ldots \wedge L_{\mathrm{n}} \tag{3.4}$$

From the above discussion, we conclude that, the GPD algorithm correctly decomposes a global property into a conjunction of local properties.

3.2.2 Local Property Simplification

After decomposing global properties, the SensorC compiler generates SensorC_L program for each L_i if L_i is not assigned as "true" (i.e. no runtime property checking is necessary on node i). There are two main steps.

(1) The generation of instantiated properties in step 2 of the GPD algorithm has accurately calculated the number of remote variables and reduced data transmission. Nonetheless, this step may increase the size of the decomposed local properties. As a consequence, it may increase the size of the error detection code. The limited program memory on individual nodes motivates improvement of the basic algorithm. Hence, we apply Algorithm 2 to reorganize the local property.

Algorithm 2 *Combine formulas in a local property*

Require: Let $L = P_1 \wedge P_2 \wedge \ldots \wedge P_S$ is a local property generated by Algorithm1.

1: Let P'_i be the formula obtained by removing all indices associated with GLOBAL variables from P_i .
2: Separate P_i into groups such that, if P_i and P_j in the same group, we have $P'_i = P'_j = G_k$, where k is group id.
3: In each group k, define NODE variables and assign them possible indices, and match the GLOBAL variables to the corresponding NODE variables in G_k.
4: Obtain $L = G_1 \wedge G_2 \wedge \ldots \wedge G_q$, where q is the number of groups. $\diamond$

After Algorithm2, all formulas which differ only with the node IDs are represented by only one formula. Figure 3.2 uses NODE variables m to stand for node $1 \ldots 9$.

(2) In Figure 3.4, property_1_violation is a propositional formula defined over only local variables, while propery_2_violation contains also remote variables. This implies that, only when property_1_violation is satisfied, the remote value needs to be obtained in order to check property_2_violation. Therefore, during the

detection phase, the number of messages containing the remote value can be further reduced. Taking Figure 3.2 as an example, node 7 does not need to obtain data from node m to check the expression in the second *IF* until after it sends a message to node 0 and finds the next hop to be m.

Local properties generated thus far cannot be directly inserted into the application code. Instead, another tool should be used to analyze the SensorC_L programs and instrument the error checking code into the application. The basic idea is described as follows.

First, for properties which should be satisfied all the time when the program is executing, instrument property checking code (or assertions) at every place where the variables used in the predicates are loaded. Otherwise, based on the scope, the property checking code is inserted. This step is rather straightforward and will be skipped in this dissertation.

Second, additional SEND/RECEIVE operations are instrumented for getting remote values. If node i needs a remote value x from node j, two steps are executed: (1)in the program running on node i, before every assertion where x is used, a call to the routine that is responsible for requesting x and receiving x is added; (2)in the program running on node j, a call to the sending routine that is responsible for sending value x after receiving the request is added. Note that a remote variable may appear in more than one assertion. If a recently obtained value of the variable remains valid for multiple assertions, then no additional SEND/RECEIVE routines need to be inserted before verifying such assertions. This, however, often depends on the nature of the specific application and is therefore an optimization opportunity for the programmer to exploit. Furthermore, opportunities may exist to aggregate several remote data pieces (used in the same or different assertions) into a single message. In order to let the programmer take advantage of these opportunities, we

allow the programmer to explicitly mark the program points for SEND/RECEIVE operations and the data items to be combined.

3.3 Improving Decomposition by Using #NETWORK and #ROUTING Segments

Besides the two essential segments #DEFINITION and #PROPERTY in a SensorC_G program, programmers are allowed to specify #NETWORK and #ROUTING segments optionally if the following two assumptions are satisfied.

Assumption 3-1 The WSN deployment information, including the number of nodes, the noise floor for each node, and the gain for each link, can be pre-acquired.

Assumption 3-2 The routing protocol is known and when a route is established, the routing table for each node is stationary.

#NETWORK The #NETWORK segment includes the name of a file that contains the network topology information. The topology is described in the same way as in TOSSIM [18], which is a widely used simulator for TinyOS-based WSNs. The topology file contains the noise floor for each node and the gain for each link. The network is abstracted as a directed graph, in which each vertex is a node and each edge is a link. Each node has a private piece of state representing what it hears on the radio channel. In our work, we assume that there are no transmission errors at the radio level. When parsing this segment, SensorC compiler reads the file content and builds a network graph described above.

#ROUTING The Resource Pool shown in the diagram in Figure 3.1 is a library that contains WSN routing algorithms implemented in C language. After generating the WSN model according to the #NETWORK segment discussed above, we automatically generate a routing table for each node according to the WSN

model. Alternatively, the global property specification may include a #ROUTING segment that names the routing protocol used in the application. If the SensorC compiler recognizes the protocol name, it looks in the Resource Pool for a matching function. By invoking that function at compile time, routing tables are generated.

With the assumptions and domain specified information provided in #NETWORK and #ROUTING segments, the conversion rules discussed in Section 3.2 can be extended such that a more refined decomposition can be attained.

Refinement of global property decomposition based on these two additional segments concerns wsn_expressions only. In step 3 of Algorithm 1, a wsn_expression is converted using the third column in Table 3.2. In contrast to the second column, this conversion checks each relay nodes behavior. Take ERROR#3 in Table 3.3 for example. Without #NETWORK and #ROUTING, the error is detected on the receiver node, and we replay the program execution on 7 nodes before we locate the origin of the detected error. However, with the Routing_rule being checked on the relay nodes, the error can be detected on a relay node and we can locate the origin of the error by replaying this relay node alone. On the one hand, since Routing_rule needs to be checked on relay nodes, Assumption 2 is required. If routing table is changed during the run time, a false positive (i.e., a violation is reported but there is no error) may be triggered, because the new local properties added to the relay node are implied by the global property (or more specifically, by the wsn_expression) only if the relay node is on the true routing path. On the other hand, any changes to the routing at run time will never introduce a false negative, because the routing rules checked on a wrong relay node simply widens the error definition.

3.4 Implementation and Experiments

We have implemented the proposed tool targeting WSN applications based on TinyOS 2.1.2. The tool is built upon the GNU Compiler Collection [34]. A SensorC front end is implemented and integrated in GCC-4.7. Two new passes are added to the GCC middle-end for decomposing global properties and for generating local properties, respectively.

We pick two WSN applications, whose objectives are familiar to us, as our test cases in the preliminary experiments.

TC1 (AODV) – This is a published code which can be download from the website [35]. It implements the basic functions, such as the route discovery, of Ad-hoc Ondemand Distance Vector (AODV) routing protocol for TinyOS-2.x.

TC2 (Multihoposcilloscope) This program is included in TinyOS-2.1.2 directory to test CTP(Collection Tree Protocol) [36].

Table 3.3 lists the global properties checked in our experiments and the errors found using our approach. Both programs were tested under TinyOS 2.1.2 installed on wireless sensor networks with different size. The WSNs were deployed in a 4-story building. These WSNs consisted of up 150 Z1 motes, each having 8KB RAM, 92KB ROM, and 2 MB external flash memory. We found three errors in TC1 that had not been reported prior to this experiment, but we found no violation of the checked properties in TC2. We note that TC2 has been published for over 5 years and has been subjected to several debugging studies, e.g., the T-Check study [37]. Therefore, it is not too surprising if previously existing errors have been fixed already. In this section, we only discuss the experiment results related to error detection, and the details of the source of each detected error will be discussed in Chapter 5.

Table 3.3 Global properties under detection

Property	# of Nodes	Description	Detected Error
1. TC1 Link Creation	30/150	During the first 20 seconds, Destination node 0 must receive an RREQ message from source node 1 and node 1 must receive a RREP message from node 0	ERROR#1 ERROR#3
2. TC1-Message Transmission	30	After the first 20 seconds, node 0 must receive message from node 1 periodically with the interval of less than 5 seconds	ERROR#2
3. TC1-Loop_Free	30/150	The routing tables of all nodes do not form a routing loop	N/A
4. TC2-Link Creation	150	During the first 10 seconds, base-station must receive at least one message from other nodes.	N/A
5. TC2-Message Transmission	150	After the first 10 seconds, base-station must receive messages from other nodes every 5 seconds	N/A

To check the TC1-Link Creation property listed in Table 3.3, we first deployed 30 Z1 motes with fixed topology. The SensorC_G program and the decomposed SensorC_L programs are shown in Figure 3.5. The SensorC_L programs on mote 3 to mote 29 are almost identical to that on mote 2, with the only difference being in the parameters in Routing_rule. Hence, we do not list those SensorC_L programs. Next, we continued the experiment with a wireless sensor network consisting of 150 Z1 motes deployed without specifying the network topology. In this case, #NETWORK and #ROUTING segments can not be defined and only mote 0 and mote 1 have SensorC_L programs generated, as we explained in Sections 3.2 and 3.3.

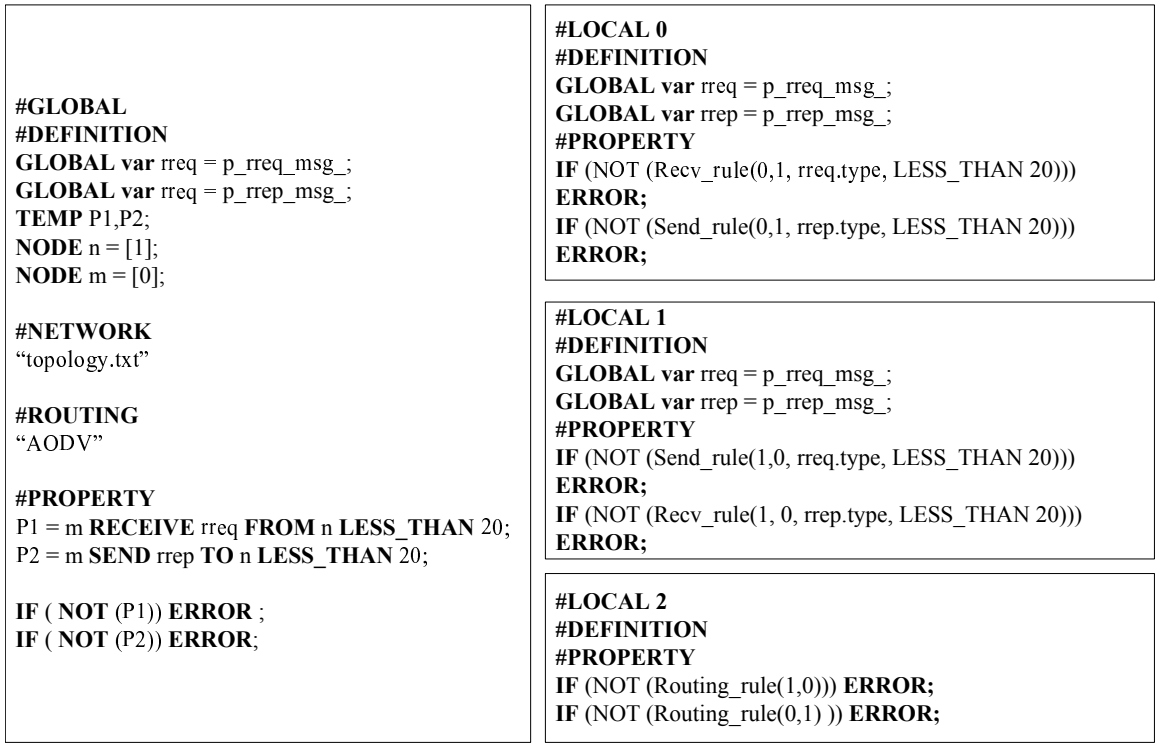


Figure 3.5. TC1-Link Creation in SensorC

The TC1-Loop_Free global property specification is previously given in Figure 3.2 (a). In this example, all nodes have local properties after decomposition. The re-

maining global properties in Table 3.3 are similar to the TC1-Link Creation property. Hence, we omit the decomposition result.

Our GPD algorithm can also be applied to simulation tools. After the errors were detected in real wireless sensor networks, we also run the comparison experiment with/o GPD-based distributed checking on the TOSSIM simulator [38] to further show that our methodology can also be used for software testing prior to the deployment. TOSSIM can simulate the wireless network behaviors with a high fidelity while scaling to thousands of nodes. Table 3.4 compares the error detection time and number of network messages with and without applying GPD-based distributed detection.

According to GPD algorithm, in our test cases, extra messages were required for local property checking only when TC1-Loop property was checked, but so far there is no error found. To compare the number of extra messages, we injected an error, called ERROR#4 by exchanging the sequence of increasing a sequence number and comparing it with an existing one, which would cause local property violation. The modified AODV program was tested on TOSSIM as well. With the centralized checking approach, we designated a server node whose only responsibility was to collect information required for error checking. The server node can be reached by all other nodes through multiple hops, but it didn't relay the application data. Furthermore, the server node must require information for error checking from every node every 5 seconds. That is, every 5 second, the routing table of each node must be sent to the server node. Each message carried exactly one row of the routing table. After decomposition, the local property still required the source node to send data every 5 seconds. The result shows, when the error was reported, the number of extra messages under our approach was 37 while the number was 94 under centralized checking. The reason is, for each node, since we only consider destination node (called

node 0) and source node (called node 1), each time, at most two messages were required from its next hops (in two directions). Moreover, in each routing table, the row with node 0 or node 1 did not exist at the very beginning. Based on Section 3.2, if no such rows exist in a node's routing table, it is unnecessary to require data. For centralized checking, it required data every time, which contained redundant data.

Table 3.4 Data collected from a 20-node WSN on TOSSIM

Error	Detection Time with Decomposition applied (s)	Detection Time without Decomposition applied (s)	# of extra transmitted message with Decomposition applied	# of extra transmitted message without Decomposition applied
ERROR#1	20.000	20.000	0	0
ERROR#2	25.000	25.000	0	0
ERROR#3	2.169	25.000	0	0
ERROR#4	4.062	4.068	37	94

Considering the time when the errors were detected, ERROR#3 shows great advantages while other do not. The reason is, ERROR#3 was caused by duplicated increment in the number of hops during the forwarding of a RREQ message (the diagnosis details will be discussed in Chapter 5) . With our SensorC and property-decomposition-based tool, we inserted error detection in every node on the expected message transmission path according to pre-defined wireless sensor network domain knowledge. Consequently, an error was reported on a relay node at 2.169 second, shown in Table 3.4. Without applying the decomposition approach, the property checking was inserted into the base-station which did the centralized checking. The property was not checked until the timer (20 seconds) was fired on the base station. For other errors, due to the property specification, they were detected when the timer was fired with and without applying our GPD decomposition approach. Under this circumstance, our approach did not lengthen the error detection time.

In Chapter 5, we will also show that this decomposition can reduce the number of nodes being replayed for error diagnosis.

4 DEPENDENCE-BASED TRACING AND REPLAY METHODOLOGY FOR A SINGLE NODE

In this chapter, we discuss how to trace and replay a property violation when the error and error source are on the same node. The approach will be extended to a general case in Chapter 5. First, we propose the dependence-based tracing and replay methodology by assuming that the trace storage is sufficiently large such that, when an error is detected, the stored trace will contain at least one anchor point prior to the source of the error. This assumption guarantees the replay tool to capture the source of the error. If it is unsatisfied, then either the trace cannot be replayed (because of the lack of any anchor point) or the replay will not lead to the source of the error (because the error source falls off the trace). This unfortunate case is discussed in Section 4.4, resorting to multi-level tracing which instruments a subset of the functions but yet permits the trace to be replayed.

4.1 What to Record

The benefit of reducing runtime logging is two-fold. First, a longer execution history can be replayed with the same amount of data storage for the trace. The time to execute the annotated program that is being traced is reduced. Second, the number of instrumented operations to perform tracing is reduced, which leads to a smaller code size.

If a function never has any effect on the kind of errors we monitor, i.e., on any of the variables appearing in the predicates (also called the invariants) which specify the correctness properties, then such a function does not need to be traced at run-

time. To exclude such functions from tracing, we first compute the backward slice [39] using the given set of invariants as the slicing criteria. The result of this computation is a set of control/data dependence chains which include all operations (such as assignments, branching decisions and function calls) having an effect on the set of invariants. Each function that contains any of these operations will be instrumented to obtain the runtime execution log. Obviously, the main function of the program is always instrumented.

This set of functions, however, does not yet include those interrupt handlers which may have an effect on the invariants. In microcontroller execution, interrupts are the basic source of non-determinism. For example, as we mentioned in Chapter 2, TinyOS adopts an event-driven execution model, that the events can occur at any time and interact with the ongoing computation. Until interrupts occur, the scheduler sequentially schedules the tasks from a FIFO (i.e. TinyOS standard scheduling policy) queue for execution. As soon as an interrupt occurs, the current task is preempted until the interrupt handler is finished and no other pending interrupts exist. If any variables which have an effect on the invariants are modified by the interrupt handler, then obviously the interrupt handler may have an effect on the invariants as well. Since it is infeasible to predict when a particular interrupt may happen, we instrument all those interrupt handlers whose execution may modify global variables on which the invariants depend.

After we determine the set of functions to instrument, we insert operations into the source code of these functions to record the following pieces of information. Figure 4.1 gives an example of a function after instrumentation for recording. We shall prove in this section that this set of information is sufficient for accurate replay.

LOG type 1 (Function entry/return) A function always has a single entry but may have multiple return points. We use N_RET_{i}, where i is an integer, to

```
inline void SchedulerBasicP$Scheduler$init()
{

 __write_function_entry(8329);
 memset(SchedulerBasicP$m_next, 255, 2U);
 __start_atomic_control();
 SchedulerBasicP$m_head = 255U;
 __gv_update();
 __stop_atomic_control();
 __start_atomic_control();
 SchedulerBasicP$m_tail = 255U;
 __gv_update();
 __stop_atomic_control();
 __write_function_return(12425);
 return;
}
```

Figure 4.1. An example of instrumented code for recording

indicate which return statement is executed. If this is a function entry, it marks whether it is an interrupt handler and, if so, the name of the function. This type of information is needed for efficiently replaying the correct instance of execution of the function, which will be explained later.

LOG type 2 (Global variable update count) In order to prepare for replaying interrupt routines, when an interrupt routine is invoked at run time, a global-variable reference counter, denoted by #gv_reference, is written to the log, after which the count is reset to zero. For any other functions, #gv_reference is reset to zero both at the entry and at the exit. Every reference (read or write) to a global variable is followed by an increment of #gv_reference. This count will be used during the replay to help determine where in the program to replay specific interrupt routines. The reference to the global variable and the increment of #gv_reference are made a single atomic operation by calling a system library function to disable and re-enable interrupts. Without atomicity, it would be impossible to exactly determine whether an interrupt happens right before the global variable reference or between the reference and the increment of #gv_reference.

LOG type 3 (Task scheduling) If task scheduling order is random, then we need to record the task that is scheduled to next. The standard scheduling policy in TinyOS is FIFO, in this case, as long as the invocations of the interrupt routines are recorded and replayed accurately, this type of information does not need to be recorded.

LOG type 4 (Anchor points) As discussed previously, at each anchor point, we record all variable values which are needed in order for the program to replay from here.

LOG type 5 (Non-deterministic inputs) It is necessary to record the non-deterministic input for future replay. In TinyOS, the messages received from radio communication and the sensor data arriving from the bus belong to this type. Note that the interrupt handlers export such input by writing it to a variable. Since the interrupt handlers which take external input are explicitly marked, we add operations in such handlers to save the variables' value to the trace. In addition, some global variables, which are always defined as "volatile" type, are used to store a hardware register value. The value of reading from a volatile variable should also be recorded.

After the run-time information is recorded, various existing methods [40] such as using a different radio on the same node, storing logged information on a nearby node, and so on, can be used to store and transfer logs. The technique details of retrieving the logged information from a node is out of scope of this dissertation.

4.2 How to Replay

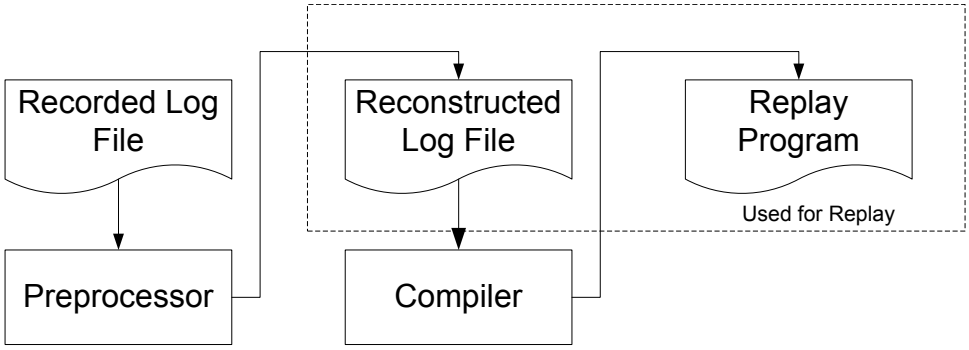


Figure 4.2. The replay scheme

Most existing replay schemes either simulate the machine code or interpret an intermediate code, taking the run-time log as input. Our replay scheme is unique in that it instruments the source code by adding log-reading operations based on

the run-time log, which makes it possible to recompile it for direct execution on any desk-top machines, instead of interpretation or simulation. This approach makes the replay tool more portable since it does not depend on the existence of a simulator for the motes hardware and it is not tied to any intermediate code design. Moreover, direct execution is well known to be faster than interpretation or simulation by at least an order of magnitude.

Figure 4.2 shows a diagram for our replay scheme. The Preprocessor reorganizes Recorded Log File, which contains the raw log information recorded from the motes, into Reconstructed Log File. The latter file consists of a data section, which is to be fed to the replay program later, and an interrupt table, which is used for the creation of the replay program. The data section simply lists all the <variable, type, value> tuples and the task scheduling log (LOG type 3) recorded by the motes program, all kept in the same order as they were recorded. The <variable, type, value> tuples may either be from the anchor points or from the nondeterministic inputs. The interrupt table is composed by examining the interrupts recorded and the associated #gv_reference values. Based on LOG type 1 and LOG type 3, by traversing Recorded Log File, we can find for each interrupt (1)in which function it was triggered and (2) how many times that function was executed before the interrupt arrived. If an interrupt, say interrupt_x with #gv_reference = y occurs in the m-th instance of function_n, then a tuple of the form < function_n, m, interrupt_x, y> is added to the interrupt table.

The replay program is generated at the source level automatically by the compiler based on both the interrupt table and the code instrumented for mote execution. In conventional replaying, after every instruction (or some intermediate-level statement) is simulated or interpreted, the tool checks to see whether an interrupt handler should be replayed at this point (based on the logged information such as the PC, iteration

count and recursive call depth). For our source-level replay, which is directly executed on a desktop after compilation, the replay of an interrupt is triggered by the match between the #gv_reference value observed during replay and that recorded by the interrupt handler. According to each item < function_n, m, $\text{interrupt}_\text{x}$, y> in the interrupt table, we need to instrument a matching operation only in function_n to check whether $\text{interrupt}_\text{x}$ must be triggered, instead of checking for every interrupt in every function. Although function calls are deterministic, without LOG type 1, every update to #gv_reference will trigger a matching operation, which is obviously much more time consuming.

The main program is transformed such that it starts by calling processLOG(type 4), which searches the data section for the earliest anchor point recorded(The original beginning of the program is an anchor point by default which, however, may have been pushed off the log at run time). The main program reads all the <variable, type, value> tuples for the anchor point before executing from the anchor point. After this, the replay program simply executes the original C program statements until it meets the next processLOG library calls. If Recorded Log File shows that, for some reason, the execution returns from the boundary function containing the anchor point, then the execution goes back to the main function which looks for the next anchor point. For each operation inserted to the instrumented mote program which writes LOG type i to the trace, the compiler inserts a corresponding operation, processLOG(type i) in the replay program. For each log type, the processLOG function executes according to the following description.

processLOG(type 1) - This is encountered either at the beginning of a function or right before a return. The routine resets #gv_reference to 0. If it is encountered at the function entry, it also keeps a counter to indicate which instance of the

function is being executed. This counter will be used in processLOG(type 2) to check the conditions that trigger interrupt handlers.

processLOG(type 2) - The replay program updates #gv_reference just like in the mote-executed program except that the atomicity control is no longer necessary because we have sufficient information about when interrupts occur. The current function ID is passed as another parameter to processLOG(type 2) which, at each time #gv_reference is increased during replay, checks to see whether the current instance and #gv_reference value meet the interrupt triggering condition. If so, the corresponding interrupt handler is called. The interrupt handler may not be invoked at exactly the same program point as in the original run, but its effect on the control and data dependence will be exactly the same and therefore does not alter how the error may be propagated.

processLOG(type 3) - If the tasks are scheduled randomly, then the replay program reads LOG type 3 in order to determine which task to execute. Otherwise, if LOG type 3 is not recorded due to the FIFO scheduling policy, this routine is skipped as well.

processLOG(type 4) - A flag indicates whether an anchor point is encountered. If so, according to the pre-determined format, this processLOG routine reads in all variable values before starting to execute the first statement at the anchor point.

processLOG(type 5) -This processLOG routine reads in the external input from the log at the same program point as in the mote-executed code which records the information.

We have two alternatives for handling hardware-dependent code, the operations to hardware registers, to be specific. Our first option is to remove all hardware

dependent code for replay. The impact of interrupts will be on the values of certain global variables(Similar handling is performed in certain TinyOS simulators [37,38]). This however misses the opportunity to trace the error source further when a message containing wrong contents is received and saved to a hardware register by a low-level interrupt handler. Only when the second interrupt handler, posted by the first one, copies the wrong contents from a hardware register to a global variable will the error be located by backward tracking from a violated invariant. A remedy for this omission is to write a preprocessor customized for the hardware platform which converts references to hardware registers to global variables.

Statements which do not affect the invariants are deleted from the replayed program as described in the literature [41]. After these treatments, the resulting code for replay is compiled and executed on an ordinary desktop machine.

Note that the bookkeeping on #gv_reference to enable source-level tracing and replay does not cost much more than the operations to save the loop counts in the assembly code in order for the replay program to be able to continue correct execution after an interrupt handler exits. Recording the return address in the trace alone is insufficient. As a matter of fact, if the function contains irreducible cycles in its control flow graph, it is not obvious how to count loop iterations so the replay can continue correctly after returning from an interrupt handler. The correctness of our replay scheme is formally stated by the following theorem.

Theorem 4.2.1 *Suppose an incorrect program statement causes an invariant to be violated at run time. Under the record-replay scheme described above, the same incorrect program statement will cause the same invariant to be violated in the replayed program.*

Proof The LOG type 3 ensures that the order in which tasks are scheduled from the task queue is exactly the same when executed by the replay program as by the

mote program. We just need to prove that interrupts do not cause the programmer to observe incorrect use-def chains during replay.

First, suppose the incorrect statement execution S and the invariant violation Inv are both outside any interrupt routine. As illustrated by Figure 4.3(a), the $\#gv\_reference$ value at the time S must be the same in the mote program and the replay program. If no interrupts occur between these two at run time, then the replay program will find the last interrupt routine prior to Inv before it replays S.

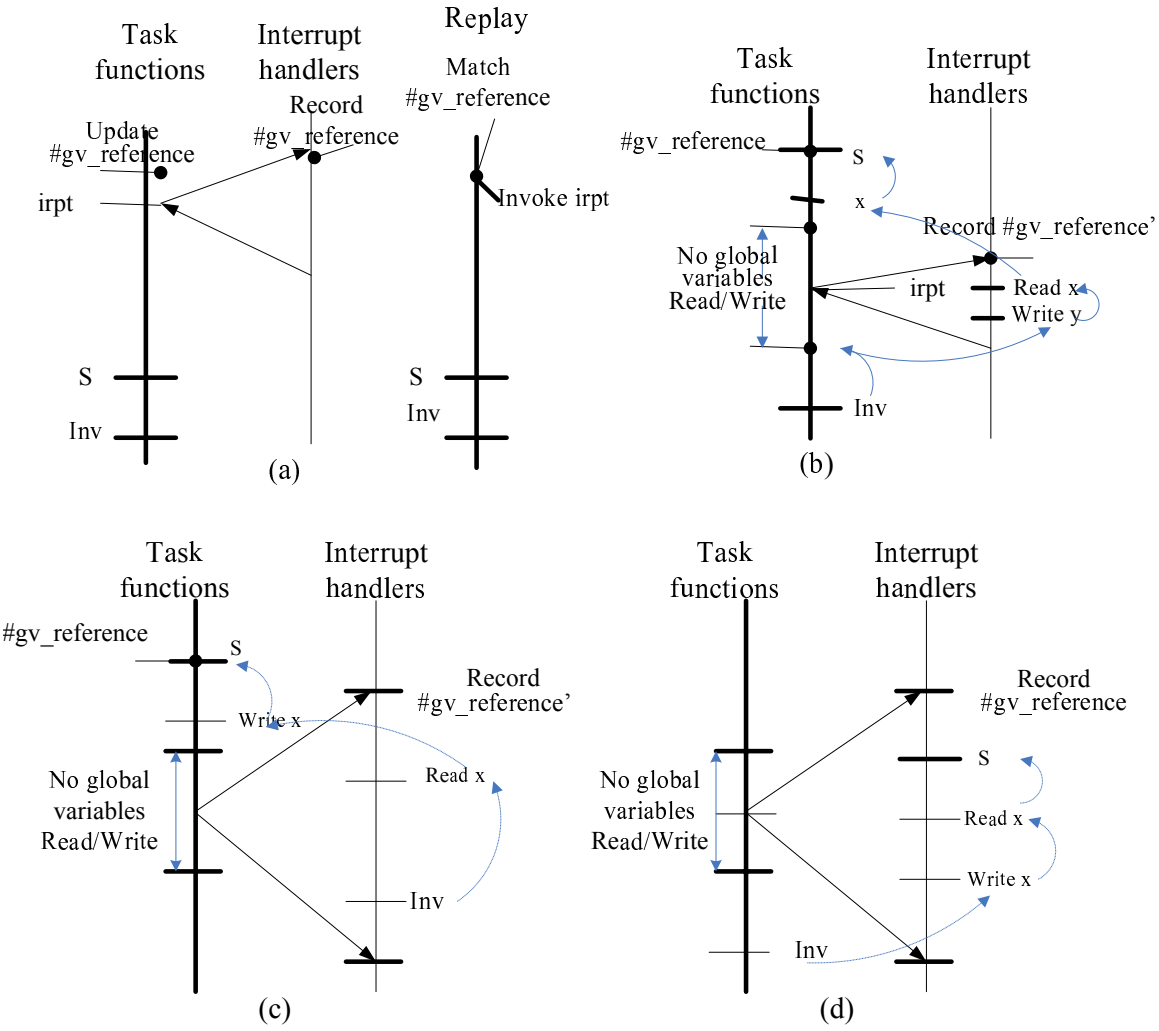


Figure 4.3. An illustration for proof of Theorem 4.1

Conversely, as illustrated by Figure 4.3(b), if an interrupt, *irpt*, occurs between S and Inv, then the programmer must pay attention to *irpt* only if it is part of the use-def chains between S and Inv. This, however, is possible only if *irpt* first reads a global variable, x, computed outside *irpt* such that x depends on S and then writes to a global variable y on which Inv depends(Both dependences are by transitivity, and x may be the same variable as y). Consider two possibilities:

(1) $\#gv\_reference$ recorded by *irpt* is greater than the value at the time of S. S will be replayed before *irpt* in this case.

(2) $\#gv\_reference$ is reset to zero due to other functions called between S and *irpt*. The replay program will replay S before these called functions and therefore before *irpt*. Furthermore, the $\#gv\_reference$ match ensures that the replay program invokes *irpt* between the correct pairs of consecutive references to any global variables.

In both cases above, the correct use-def chains will be observed.

Next, consider two other possibilities:

(3) S is outside any interrupt routine but Inv is inside an interrupt routine *irpt*, as Figure 4.3(c) shows. There must be a global variable, x, which, by transitivity, depends on S, is read inside *irpt* and eventually leads to the violation of Inv.

(4) Illustrated in Figure 4.3(d), S is within an interrupt routine *irpt*, and Inv is outside any interrupt routine. There must be a global variable, x, written between S and the end of *irpt* such that x depends on S, by transitivity, and x is read after the exit from *irpt* which eventually leads to the violation of Inv.

For both (3) and (4), by reasoning about $\#gv\_reference$ and LOG type 1, we can prove that the order of executing S, writing to x, and invoking Inv will be preserved

in replay regardless whether the write to x is inside any interrupt routine or not. The order will also be preserved no matter whether the write to x happens to yet another interrupt routine.

Finally, suppose S is in an interrupt routine $irpt1$ and Inv is in another interrupt routine $irpt2$. There must be a global variable, x, written in $irpt1$ and another, y, read in $irpt2$ such that the value of y depends on x by transitivity and x is depends on S by transitivity. (It is possible for x and y to be the same variable.) Again, by reasoning about $\#gv\_reference$ and LOG type 1, it can be proven that the order between S, write to x, read of y, and Inv will be preserved during replay regardless whether other interrupt routines are invoked. ■

4.3 Decision on Inlining a Function

To further reduce the code size after tracing instrumentation, we notice that we can reduce the number of logs of LOG type 1 if we inline function calls(Of course, interrupt handlers cannot be inlined). However, if a function is called in more than one place in the program, then inlining may increase the program size due to duplication of the function body. Fortunately, the inlining decisions for different functions are independent and the cost model is simple. For each function, let S_{original} be the code size before instrumentation and $S_{\text{Instr\_func}}$ be the increased code size due to inserted operations to write LOG types 2, 3 and 4 (Interrupt handlers are never in-lined). Further, let S_{call} be the increased code size due to inserted operations to write LOG type 1. For inlining to be beneficial for the function under consideration, by assuming this function is invoked $n + 1(0 \leq n)$ times, we must have

$$(S_{\text{original}} + S_{\text{Instr\_func}})n < S_{\text{call}} \tag{4.1}$$

4.4 Multi-level Tracing

Theorem 4.2.1 uses the assumption that the trace storage is sufficiently large enough to include the error source and at least on anchor point before the error source. If that assumption is not satisfied, when an error is detected, we either cannot find an anchor point to replay the program or cannot find the error source during replay. This can happen if the storage for logging is small or the error happens long time before it is detected (through the violation of a property). To enable replay under such a circumstance, we propose multi-level tracing. Rather than instrumenting the whole program, we divide the program functions into different levels based on how "far away" (as defined bellow) they are from the invariants being checked. For each iteration, only the code residing in certain level(s) are tracing and replayed. If the error source can be located in this iteration, the procedure stops. Otherwise, pieces of error-propagation path can be collected in this iteration, and in the next iteration, the tracing level is increased. Nonetheless, with multi-level tracing, we no longer have the guarantee that the error source will be found, but at least we have partial traces to narrow the search.

4.4.1 An Iterative Tracing and Replay Procedure

For the purpose of defining the levels of tracing, we build a graph based on the dependence information computed previously. For convenience of implementation, we wrap each invariant-checking operation in an *invariant-checking function* and insert a call to this function wherever the invariant must be checked.

Definition 4 *Given a set of invariants, the invariant-based Program Function Dependence Graph (PFDG) for a program is a set of nodes, each representing a function whose execution directly or indirectly affects whether the invariants holds, and a set*

of edges of two kinds, namely the calling edges and the dependence edges. A calling edge <f_1,f_2> is drawn if f_1 is directly called by f_2. Dependence edges are drawn according to the construction rules in Algorithm 3.

Algorithm 3 *Dependence edges construction*

1: Suppose operation u in function f_1 has a direct control/data dependence on another operation d in function f_2 and this dependence is a link in a dependence chain originating from an invariant. We draw a directed dependence edge from f_1 to f_2, denoted by $f_1 \rightarrow f_2$ if one of the following is true:

(1) Function f_1 calls f_2 (u takes place after f_2 returns to f_1)

(2) Function f_2 calls f_1 (d takes place before f_1 is called)

(3) Both f_1 and f_2 are directly called by a third function g.

2: If none of the above is true, then f_1's dependence on f_2 is passed through a number of function calls and returns. For the purpose of our tracing algorithm, we draw a chain of dependences to make it clear how this dependence is propagated through a call chain. This is described below.

(1) If there is a call chain, C_1, from g to f_1 and another, C_2, from g to f_2 such that no other node belongs to both call chains, we say g is a closest common ancestor of f_1 and f_2. We find all closest common ancestors of f_1 and f_2 in the call graph.

(2) Next, for each closest common ancestor of f_1 and f_2, say g, we find two of its immediate callees, g_1 and g_2, one in the path from g to f_1 the other in the path from g to f_2. We draw a chain of dependence edges connecting f_1 all the way to g_1 following C_1. Next we draw another chain of dependence from g_2 to f_2, following C_2 in its opposite direction. Finally, we connect these two chains of dependences by the edge $g_1 \rightarrow g_2$.

By following call edges and dependence edges, all dependences can be found in this graph by transitivity. Unless specified otherwise, functions mentioned in the rest of the Chapter refer to those in the invariant-based PFDG, and all variables mentioned will be those used in the invariants or those affecting the variables in the invariants.

Figure 4.4(a) shows a piece of program and its invariant-based PFDG (Figure 4.4(b)). Here the function Inv_fun() is an invariant-checking function and function

f3() and f4() both modify some variables used in the invariants. Solid arcs represent call edges and dotted arcs represent dependence edges.

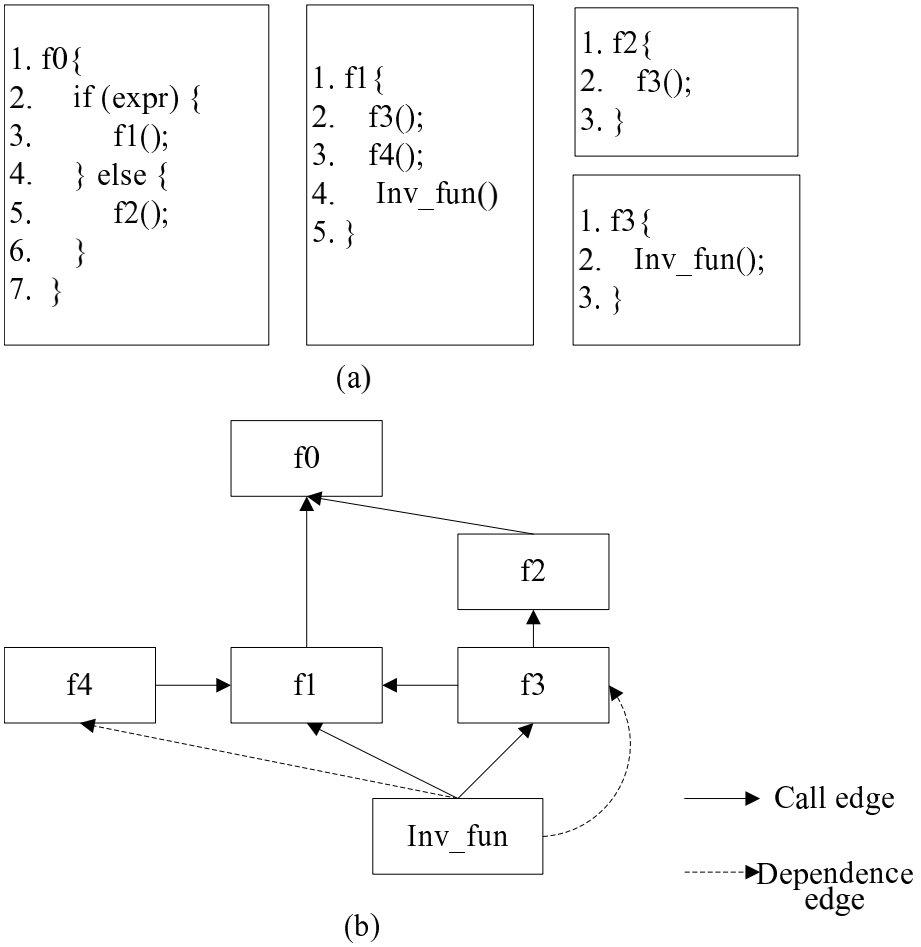


Figure 4.4. An example of invariant-based PFDG

Definition 5 *In an invariant-based PFDG, a sequence of connecting edges is called a canonical path if the sequence originates from an invariant-checking function inv and is composed by a* prefix $\langle inv, f_1\rangle, \langle f_1, f_2\rangle, \ldots, \langle f_{m\text{-}1}, f_m\rangle$ *, with calling edges only, and a* postfix $f_m \rightarrow g_1, g_1 \rightarrow g_2, \ldots, g_{n\text{-}1} \rightarrow f_n$*, with dependence edges only. The prefix or the postfix may be empty, but not both.*

Definition 6 *In an invariant-based PFDG, a function f is said to be at the level n($n \geq 1$), if, among all canonical paths ending with f, the shortest path has the length n.*

With the prefix and postfix clearly separated for each canonical path, we can define set of functions in which variable values are recorded for replaying. In order to make replay possible, in addition to the five types of logs discussed in the previous section, we need to record additional information for boundary functions as Definition 7 defined.

Definition 7 *In an invariant-based PFDG, a function f is said to be a boundary function for level-n tracing if there exist an n-long canonical path ending with f which consists of call edges only.*

In our iterative debugging procedure, what to be included in level-n tracing depends on the result of tracing and replay at the lower levels. Our iterative procedure can start with any level m, as long as all functions at levels m or lower are all included for instrumentation. Without loss of generality, we assume the procedure starts at level 1. The functions to be instrumented include all level-1 functions and all interrupt handlers which may modify any global variables used by any level-1 functions.

Obviously, for level-1 tracing, all immediate callers of an invariant-checking function are boundary functions. At the entry of each boundary function we record the entire calling context at run time, i.e. all global variable values and the arguments passed to the function. For all non-boundary level-1 functions, i.e. those non-interrupt functions connected by dependence edges from an invariant-checking function only, logs of LOG types 1-3 are recorded but not the entire calling context. For all non-deterministic inputs, logs of LOG type 5 are also recorded.

If an instrumented function calls a higher-level function g (which is not instrumented), $g's$ return value (if any) and the global variables written during $g's$ execu-

tion are recorded right before g returns. This allows the instrumented function to continue the execution correctly. Nothing else in g is recorded no matter what non-instrumented routines are called within g. At replay, the program statements in g are not replayed, but its return value and modified global variables are used to continue the execution of $g's$ caller. This way, we limit the size of the instrumented code and the recorded trace. This multi-level tracing is different from existing partial-replay schemes which either replay all callees of any replayed function or estimate the call effect based on certain statistic assumptions.

Note that, during replay, the level-1 functions may be executed multiple times while the program statements belonging to higher-level functions are skipped in between.

Since the invariant-checking functions are always replayed, violation of invariants will always be detected. The programmer, using debugging tools such as GNU's gdb, can follow the program execution and produce a replayed execution trace. The statements along the trace leading to the error can be examined, which will have one of the two outcomes: the faulty statements (or the unexpected events) which cause the error are found, or such statements (or events) lie outside the level-1 trace. In the former case, debugging is done. In the latter case, the execution path extends beyond the level-1 trace. Mapping this non-ending path back to the invariant-based PDFG, we obtain a subset of canonical paths which are called error-hiding paths from level-1 tracing.

Next, we inductively assume that level-(n-1) tracing has not led to the discovery of the source of the error but has marked all parts of error-propagation paths that are found during all level-m tracing ($m < n$). We present Algorithm 4 for level-n tracing.

Among all functions in S, we find the boundary functions for level-n tracing according to the invariant-based PDFG. We add recording operations in these functions

Algorithm 4 *Determine which functions to be instrumented for level-n tracing*

1: Let S be the set of functions to be instrumented.
2: Add all functions in the error-propagation paths found in level-m tracing ($m < n$) to S.
3: Add every level-n function which is immediately reachable from any error-propagation path (i.e. can be connected by a single edge from a node in the path) to S.
4: Add all invariant-checking functions to S.

to record the entire calling context. The rest of the instrumentation follows the same discussion in the case of level-1 tracing. In practice, one can be flexible when using our iterative tracing procedure. If the original program size is too large for even level-1 tracing described above, one can choose a subset of level-1 functions as long as the side-effect of their callees are recorded to allow replay to continue. The invariant-checking functions must always be executed for tracing, so that the error can at least be detected. If the subset chosen for level-1 tracing does not lead to the discovery of the error source, another subset is chosen, and so on. On the other hand, if the size of the original program is small, one can start with level-m tracing for some $m > 1$. The relationship between the original code size, the available program memory and the choice of m is not explored further in this work.

4.4.2 Termination of the Iterative Tracing Procedure

If the replay for the level-n tracing does not lead to the discovery of the error source and neither does it repeat any of the previous execution paths, then the execution paths used for the next level tracing will accumulate further. The tracing may also lead to the violation of a different invariant. The level-1 tracing for the new violation will then be mixed with tracing for the previous violations. All these may theoretically cause the instrumented code size to exceed the available program memory.

However, if we assume that the error-hiding path found in level-m tracing always repeats itself in level $m + 1$ tracing, then, obviously, the iterative tracing and replay will eventually expose the error source by replay, as long as the instrumentation of all functions in the error-hiding paths always fit in the program memory. Note that the program memory required in this case will usually be significantly less than full instrumentation, because we instrument along a single path. Also note that, even though under nondeterministic external inputs the program may take different

execution paths in each deployment or each tracing, the function call/dependence paths leading to the violation of the invariant, i.e., the error-hiding path, may still be the same. Our assumption here, therefore, accommodates nondeterministic behavior to a certain degree, even though it is not ideal.

4.5 Experiments

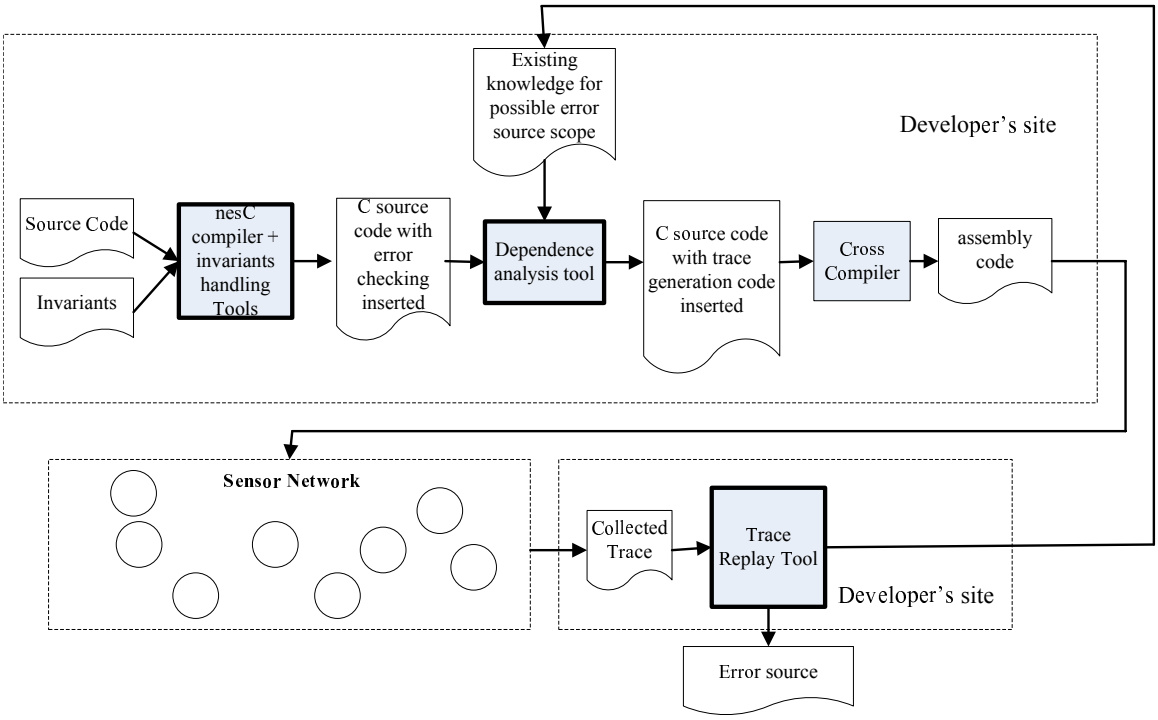


Figure 4.5. Framework of tracing and replay tool for a single WSN node

We have implemented the proposal tool for WSN applications on TinyOS 2.1.2. Figure 4.5 shows its framework. For each TinyOS application, the nesC compiler first converts the application into a C program which is then compiled by the cross compiler into machine code executable on the specific hardware. We use the same cross compiler to compile the C program instrumented by our GCC-based tool before loading it on the sensor mote for normal execution with tracing. When an error

is detected, we retrieve the trace and feed it to our replaying C program which is executed on a desktop machine and the GNU gdb is used to to help us isolate the source of the error. Since all non-deterministic inputs, including received messages, have been recorded, the replay program can be run independently by feeding each with its own retrieved logging information.

The tool is implemented on the top of GCC-4.7. Two new passes are added to the GCC middle-end for tracing/replay code instrumentation and dumping IR to C source code respectively.

We use TelosB motes on which the WSN applications are installed since this kind of motes has more restricted hardwire resources according to Table 1.1. A TelosB mote has 48KB program memory and 1MB external flash memory.

Currently, a trace buffer of the size of 2KB in the RAM is used for LOG recording. The log is transferred to the external flash memory when the buffer is full. We use Blink (TC1), EasyCollection(TC3), and TestSerial(TC2) as our testcases. The first two are the sample programs in TinyOS, and the last one is what we used to collect CO2 data.

TC1 (BlinkC) This is a published TinyOS 2x application. We insert an invariant which requires that the frequency of three LED's blinking must follow a user specified pattern. We then add a long running task which increases the latency of Timer.fire(), causing a violation of the invariant.

TC2 (TestSerialCO2) This application monitors indoor CO2 data in multiple locations inside a building. We require that, from each mote, the base station must receive new CO2 reading with a period of two seconds or less. This property is specified by two invariants. The base station must make sure that it receives a new piece of CO2 reading from each mote every two seconds or less. Each mote must make sure that, within a period of two seconds it receives at

least one piece of data from its own sensor and sends it to through the radio channel.

TC3 (EasyCollectionC) This is a published code which collects data using implemented Collection Tree Protocol. We insert invariants require that the data must be sent in sequence.

Table 4.1 compares the number of the functions traced using the dependence information against those without such information. The data indicate that, with a single invariant consisting of fewer than 3 variables, the dependence information allows between 40% and 85% of the functions (not including interrupts handlers) to be skipped for tracing. However, the number of functions to be traced remains to be large for test cases TC2 and TC3.

Figure 4.6 shows the effect of inlining. Over 70% of the functions are called only once and, based on the simple cost model, can be inlined. Table 4.2 lists the code size under different instrumentation schemes in comparison with its original size S_{original}. For the baseline code size S_{baseline}, we include the inserted operations to record all types of log information without taking advantage of dependence information. The data show that the baseline size is too large for the program memory on TelosB motes. The column $S_{\text{no-inlining}}$ shows the remaining code size if we do not trace functions which have no effect on the invariants. It is much smaller than the baseline size, but still large. Take TC2 for example, the size of its $S_{\text{no-inlining}}$is 50534, which exceeds the TelosB memory boundary size 48K. The column S_{inline} shows the code size after selective inlining. After inlining, the code size is decreased further. Of course, if many invariants are checked in the same program or some invariants involve many variables, then the use-def chains may cover more functions and the instrumented program size may increase. In the worst case, the code size may be too large to fit in the program memory, in which case multi-level tracing will be needed.

Table 4.1 Functions instrumented using dependence information as a fraction of the total functions

Test case	# of traced functions	# of total functions	Percentage
TC1	46	299	15.38
TC2	605	1499	40.36
TC3	604	1385	43.61

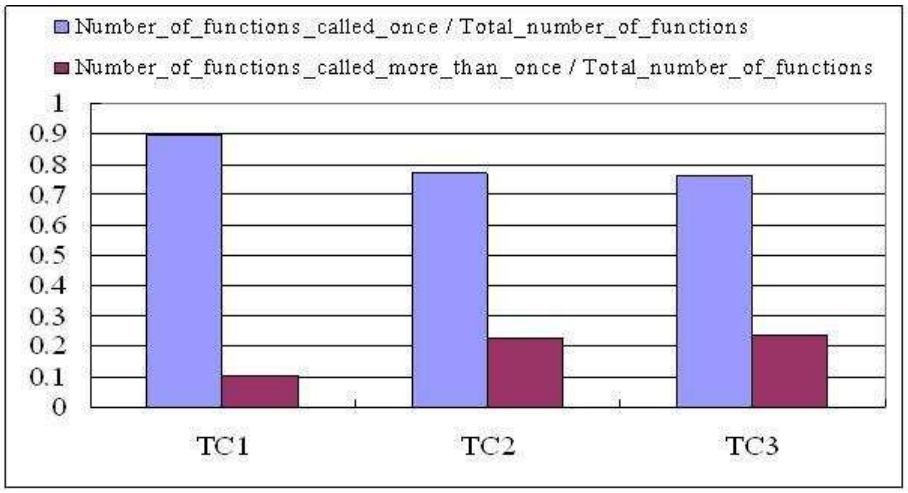


Figure 4.6. Inlined functions as a fraction of the total

Table 4.2 Code size (bytes)

Test case	S_{original}	S_{baseline}	$S_{\text{no-inling}}$	S_{inline}
TC1	2650	18760	13760	13296
TC2	24302	80058	50534	42878
TC3	18670	73214	45090	37778

Table 4.3 Instrumentation overhead

Test Case	Log Size (bytes)	Execution Time		Energy	
		Overhead (10^{-6}s)	Percentage (%)	Overhead (10^{-3}Joule)	Percentage (%)
TC1	56	70	1.01	0.039	2.21
TC2	1496	negligible	negligible	2.11	20.2
TC3	838	1970	13.32	0.129	15.5
TC3	838	15050	101.76	0.364	36.7

Table 4.3 shows the overhead due to instrumentation (with slicing and inlining optimization) . For each call to a task function which does not contain an anchor point, the storage used for log trace is 4 bytes (2 bytes for function entry and 2 bytes for function return). At each anchor, each saved variable requires a record of 5 bytes, including 2 bytes for the variable name, 1 byte for the variable type, and 2 bytes for the variable value. To save a nondeterministic input or the current #gv_reference value, each variable also takes 5 bytes. For TC1 overhead is measured from the beginning of the program to the first time when the error is caught. For TC2, we measured the overhead during each sampling period. For TC3, overhead is measured between the start to send a message till the message is sent, and we call the measured time interval " sending period" for short. For TC3, we compares the overhead with and without counting the cost to write the log to the external flash, listed in two rows respectively. Since the buffer size we set is 2KB, writing external flash is called about every 2.5 sending periods. Therefore, in the second row of TC3, the added overhead of writing external flash is the average overhead for each sending period. It is quite common that there is an idle period between two message sends long enough to be used for writing logs to the flash, as in TC2. Hence we marked the execution time overhead as negligible. When an error occurs, if the log buffer in RAM is large enough to store the entire log, the mote can directly send the log to the base station, rather than reading it from the external flash first.

5 DEPENDENCE-BASED TRACING AND REPLAY METHODOLOGY FOR THE ENTIRE WSN SYSTEM

In Chapter 4, we have discussed how to trace and replay if the error source is located on the same node where an error is detected. Illustrated in Figure 5.1(a), after replaying the node's execution and generating the replayed trace, the error source can be researched by programmers tracing back along with the use/def chain from where the error is detected.

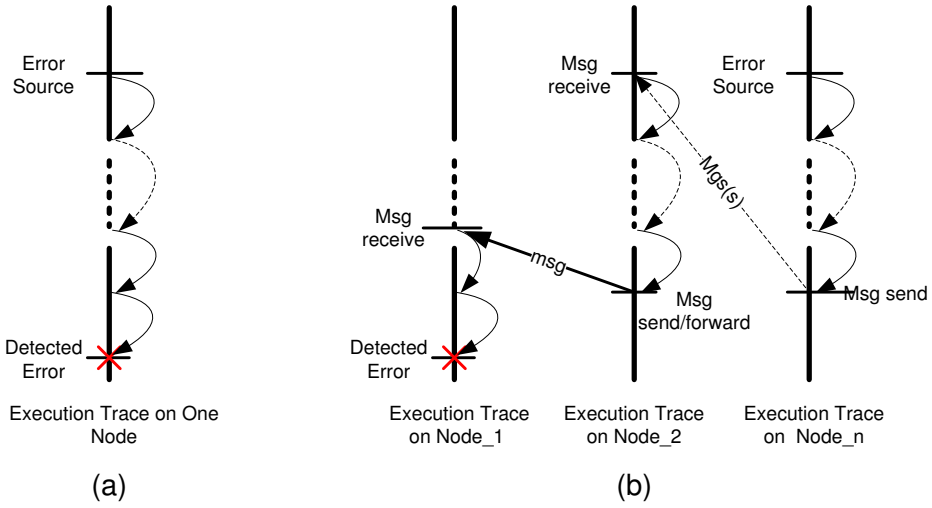


Figure 5.1. Error propagation

However, illustrated in Figure 5.1(b), an error could propagate along with the message transmission. As a consequence, an error may be caused by incorrect execution on node i but later detected on node j. Generally speaking, there are five ways in which an error can be propagated between different nodes:

C1 A node generates an incorrect message and sends it over the network.

C2 A node receives an incorrect message and forwards it.

C3 A node receives a message and modifies it in an incorrect way before forwarding it.

C4 A node receives an incorrect message which does not need to be forwarded.

C5 A node does not forward a message as required.

The following chapter discusses how to extend record-replay methodology to the entire system and deal with situations listed above by tracing messages passing.

5.1 How to Log

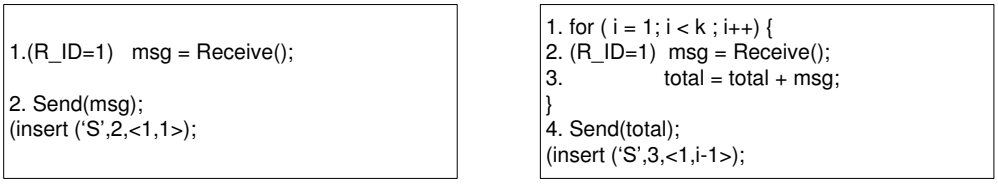
It is necessary to trace the messages with which the error propagate along to make sure the error source is included in the trace used for off-line replay. However, as stated in C3, since an error may be generated when a program uses the content from a received message to generate a new one, it is insufficient to just record the routing information of the messages. The information on the modification history or the def/use chains of the payload should also be acquired. To solve this problem, we record SEND/RECEIVE operations and their dependence relationship. As a result, besides LOG type 1-5 in Chapter 4, one more log type is added to record error propagation.

LOG type 6 (SEND/RECEIVE dependence) - SEND/RECEIVE operations are recorded to preserve how a message is transmitted and modified. First, during the compile time, each statement which indicates a message is successfully received is given a unique identifier, R_ID. Second, right after the statement indicating a message is successfully sent, we insert a SEND record $<$ 'S', MID, R_OP $>$, where 'S' means "sending", and MID is the unique ID of the sent

message. R_OP is a set of tuples <R_ID, x>. Each tuple means that the content of the sent message may be affected by at most of x ($x >= 1$) consecutive messages received at statement R_ID. If none of the receive operation exists, R_OP is NULL. Figure 5.2 shows two pseudo code examples. For each x in $<$ R_ID, x $>$, its value can be calculated by identifying the boundary and the corresponding induction variables of each loop.

Similarly, a record $<'$I$'$,NULL,R_OP $>$ is inserted right after the assertion, where $'I'$ means invariants. R_OP=$\{ <$ R_ID,x $> \}$ means that the variables used in the invariant may be affected by at most of x ($x >= 1$) consecutive messages received at statement R_ID.

Right after the statement indicating a message is successfully received, a RECEIVE record $<$ $'$R$'$, MID, R_ID, SENDER $>$ is inserted, where $'R'$ means "receiving", MID is the unique ID of the received message, R_ID is the receive statement identifier, and SENDER is where this message is sent from. By additional inserting those records, if an error is caused by the incorrect message, we can trace it back during the replay.



(a) R_OP = {<1,1>}. For R_ID = 1, x =1. Because R_ID is only executed once before SEND operation(line 2)

(b) R_OP = {<1,i-1>}. For R_ID = 1, x =i -1. Because R_ID is executed i -1 times before SEND operation(line 4). Each time, the received msg has effect on the SEND operation(line 4)

Figure 5.2. Example of setting $<$ R_ID, x $>$

5.2 How to Replay

Since all the non-deterministic events including the received messages are recorded for each node, it is doable to replay each node independently by applying the methodology discussed in Chapter 4. It is possible to ask programmers manually finding out the message transmission path by checking each pair of sent/received messages in the replay trace, but it is a really daunting work. In addition, in some cases, not all recorded traces for each node are useful for replay errors. Therefore, the Replay Preprocessor discussed in Section 4.2 is extended to firstly identify the sub-traces which may have effect on the detected errors.

5.2.1 Replay Preprocessor

The extended function of Preprocessor is to recognize which parts of Recorded Log Files are effective for replay and discard the rest parts. For example, in Figure 5.3, the entire trace retrieved from node 10, the sub-trace from AP (anchor point) to (S, 7_1, <1,1>) on node 1, and the sub-trace from AP to (S, 7_1, <0, 1>) on node 7 are used for replay. The detail is described in Algorithm 5, which identifies the useful traces by tracing back message transmission paths in WSNs. If an error is caused by a received message, the algorithm will search where the message comes from, and trace back all possible ways until the node where the message is initialized is reached. Therefore, all possible error propagation paths can be abstracted, and programmers are able to replay the program on each node in the order of message transmission.

Steps 1-3 in Algorithm 5 refer to trace. This is the original trace on node e that includes the detected error and therefore is first analyzed. All messages that are received on node e and that may affect the invariants are included in $trace'e$. Such messages are identified by looking up the records < 'R', MID, R_ID, SENDER

Algorithm 5 *Identify Traces for Replay*

Require: INPUT: $trace_1, \ldots, trace_n$, where $trace_i$ means the retrieved trace file of node i. We let e denote the node on which the error is detected.

1: Let set T be empty. Let $trace_1, \ldots, trace_n$ be empty.
2: In $trace_e$, consider the trace segment from earliest Anchor Point recorded to the recode where error is detected. Based on the inserted record $\langle 'I', NULL, R\_OP \rangle$ right after the assertion, mark all RECEIVE records which may affect the invariant. Save this part to $trace'_e$.
3: $T = T + trace'_e$.
4: **while** $T \neq \phi$ **do**
5: Let t be the first element in T, and k be the corresponding node.
6: **for** any marked RECEIVED record in t **do**
7: Analyze where message comes from (SENDER) and what the message id is (MID).
8: In $trace_{\text{SENDER}}$, find the SEND record which indicates MID is send to node k.
9: Let $trace''_{\text{SENDER}}$ be the trace segment from the earliest Anchor Point recorded to the SEND record in $trace_{\text{SENDER}}$.
10: **if** $R\_OP \neq NULL$ **then**
11: For any $\langle R\_ID, x \rangle$ in R_OP, mark x RECEIVE record whose ID is R_ID backward from the SEND record.
12: $T = T + \{\ trace''_{\text{SENDER}}\ \}$
13: **end if**
14: $trace'_{\text{SENDER}} = trace'_{\text{SENDER}} \cup trace''_{\text{SENDER}}$
15: **end for**
16: remove t from T.
17: **end while**
18: $trace'_1, \ldots, trace'_n$ are what we want, where $trace'_i$ means the set of trace segments reelected from $trace_i$ for replay on node i.

>. From parameter SENDER, we can determine the previous node the message comes from, and then the algorithm checks the trace of node SENDER to find the SEND record < 'S' , MID, R_OP > with the same MID (step 4-17). If the message is initialized by node SENDER, the path is completed. Otherwise, we continue searching where the message sent by node SENDER comes from. Based on the dependence relationship analyzed in Section 5.1, the algorithm checks all R_OP in records from the first Anchor point to the SEND record, and finds all RECEIVE Records which may have influence on the sent message. By repeating this process, the entire path is generated. Since a node may be included more than once on the possible messages transmission paths, sub-traces found out each time have to be taken into consideration together to guarantee that there are no missing messages which are possibly related to the error.

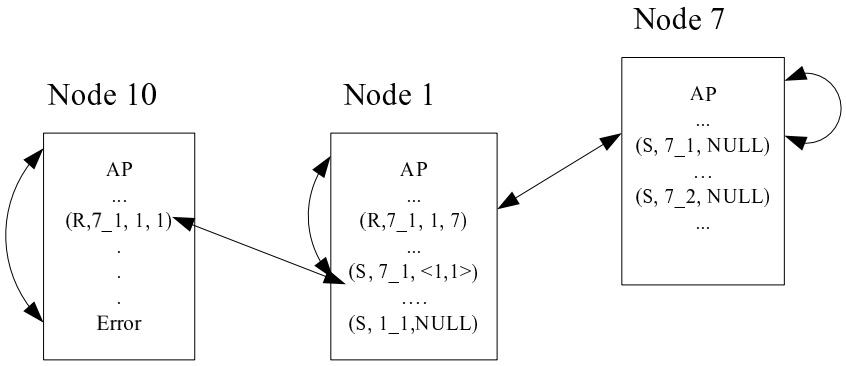


Figure 5.3. Decide which part of trace used by replay based on message matching

Based on the correctness of replay in a single node discussed in Chapter 4, we now prove the correctness of Algorithm 5.

Theorem 5.2.1 *After Algorithm 5 is applied, the rest of the trace segments are still enough for replaying errors.*

Proof Based on Algorithm 5, each output $trace'_{\mathrm{i}}$ contains the earliest anchor point. Based on the correctness of the single node replay, each trace can be independently

replayed. Therefore, if the location where the error is detected is on the same node where the error source is, the proof is accomplished.

Next we prove this situation: an error is detected on node i, and the error source is on node j, where $i \neq j$. Without loss of generality, we assume the error is propagated along with a path $P = j \rightarrow i_1 \rightarrow i_2 \rightarrow \ldots \rightarrow i_{\mathrm{n}} \rightarrow i$. For node i, it must have at least one RECEIVE record before the error detection. For each RECEIVE record, there is one corresponding SENDER, and i_{n} must be one of them. Then the algorithm can find node i_{n}. If i_{n} does not modify the message received by node i, then the algorithm can find a SEND Record by matching MID, and continue the process. Otherwise, R_OP is checked and all related RECEIVE records will be processed, which must contain the messages from node $i_{\text{n-1}}$ to i_{n}. By repeating this procedure, all possible paths are built in the reverse order, and P is included. Therefore the result is enough for replay. Furthermore, by reversing the searching path generated by Algorithm 5, a replay order can be found. ■

5.2.2 Independent Replay

Once the trace segments of each node are decided, we can replay each node independently, by feeding each with its own retrieved trace segments. The Replay procedure is similar to Chapter 4, and only the log process functions for LOG type 6 is added.

processLOG(type 6) - The LOG type 6 is only responsible for maintaining the message transmission. It is used in replay Preprocessor and this is not added into Reconstructed Log Files. Consequently this is not processed in single node replay.

Algorithm 5 can find all traces segments under the conditions C1-C4 listed at the beginning of this section. For condition C5, however, we need to reverse the

search order among the trace segments. Take "n RECEIVE data FROM m cond" in Table 3.2 for example. If no WSN domain knowledge is available, the property will be decomposed and checked only on the sender and the receiver. If a sender successfully sends a message but the message is lost by a relay node, an error will be reported on the receiver. However, since there is no message received, the replay will stop at the receiver. When this happens, we reverse the search order by starting from the beginning trace segment so we can find out exactly how the specific message is originated and forwarded before it finally gets lost. We can then replay the trace from the sending end. ERROR#3 in Table 3.3 shows such an example.

There may be redundant traces that escape the removal by Algorithm 5. The reason is that the dependence information is collected statically and, therefore, conservatively. Spurious dependence may cause certain traces to be mistaken as having an effect on the detected error. For future work, dynamic dependence analysis may be considered, such that more redundant traces can be removed.

5.3 Experiments

The tool discussed in Chapter 4 is extended by adding the code handling LOG type 6 and processLOG(type 6). We still use the two test cases: (a) TC1 (AODV) and (b) TC2 (Multihoposcilloscope) and the same deployed wireless sensor networks (details are described in Section 3.4), as our experiment environment. Similarly, the global properties checked in our experiments and the errors found using our approach are listed in Table 3.3, in Section 3.4.

Table 5.1 lists the number of nodes involved in our tracing and replay approach during the diagnosis after the error is detected. The fifth column shows the maximum number of nodes that may need to be replayed until the faulty program location is located, based on the worst-case scenario. In contrast, the sixth column shows the

actual number of nodes replayed to find the faulty location in our experiment. The result shows that although execution trace is recorded on every node, only a few nodes need to be replayed. The details of each detected error will be discussed in next section.

Table 5.1 Global properties under detection

Error	# of Nodes Recorded	Node Error Detected on	Hops to Trace	Max # Nodes Replayed	Actual # of Nodes Replayed
#1	30	node 1	1	4	2
#2	30	node 0	0	1	1
#3	150	node 0	5	69	7

5.3.1 Test Case Study

In this section, we analyze the error source by replaying the recorded trace.

```
bool add_route_table( uint8_t seq, am_addr_t dest, am_addr_t nexthop,
uint8_t hop ) {
   ...code omitted...
   if( id != AODV_ROUTE_TABLE_SIZE ) {
     if( route_table_[id].next == nexthop ) {
       //BUG – if source node is only one hop from dest node
       if( route_table_[id].seq < seq || route_table_[id].hop > hop ) {
       ...code omitted...
         return TRUE;
       }
     }
   }
    ...code omitted...
   return FALSE;
}
```

Figure 5.4. Source code of Error#1

The TC1-Link Creation property requires that a message transmission path must be established between Node 1 and Node 0 (i.e. the basestation). ERROR#1 (shown in Figure 5.4) was detected on Node 1 during the experiment. However, the error source was located on Node 0 (the destination node). The replay on Node 1 showed that the RREQ message had been sent over and over. By following the message propagation path, four nodes that are one hop away from Node 1 (including Node 0) were replayed. The replay of Node 0 showed that, although the RREQ message had been received, it did not properly update the routing table and send a reply message. Moreover, the replays show that this error occurred when there was only one hop between Node 1 (the node that is responsible for sending a message) and Node 0 (the node that is supposed to receive a message). Under this circumstance, the condition route_table_[id].hop > hop (c.f. Figure 5.4) could not hold. Due to this error, the Node 0 would not send a RREP message and would fail to establish the path, which is a violation of the TC1-Link Creation property).

```
/***************** SubReceive Events ****************/
 event message_t* SubReceive.receive( message_t* p_msg,
                                      void* payload, uint8_t len ) {
  ...code omitted...
   if( aodv_hdr->dest == call AMPacket.address() ) {
  ...code omitted…
    //BUG -- incorrectly changing memory address of received message
    p_msg = signal Receive.receive[aodv_hdr->app]( p_app_msg_,
p_app_msg_->data, len - AODV_MSG_HEADER_LEN );
   } ...code omitted…
 }
```

Figure 5.5. Source code of Error#2

ERROR#2 (shown in Figure 5.5) was detected on Node 0. The replay for Node 0 alone showed that the error source was also located on Node 0: the memory address of the received message was incorrectly changed and the message type was read in an incorrect position. After successfully receiving and delivering the first message

from the source node, the base-station failed to deliver the received message to the application layer, which is a violation of the TC1-Message Transmission property.

```
 event message_t* ReceiveRREQ.receive( message_t* p_msg, void* payload,
uint8_t len ) {
   ...code omitted...
   if( !rreq_pending_ && aodv_hdr->src != me && cached ) {
    // forward RREQ
    ...code omitted...
   //BUG – duplicated increased the number of hop
    rreq_aodv_hdr->hop  = aodv_hdr->hop + 1;
   ...code omitted...
 }
```

Figure 5.6. Source code of Error#3

ERROR#3 was detected on Node 0 when we increased the size of WSN to 150 Z1 motes. On the one hand, replaying of Node 0, however, found no error in the program. On the other hand, there was no message to back trace to other nodes. Hence we have C5 discussed in Section 5.2, and message tracing must start from Node 1, where a RREQ message is sent. The error source was found when we replayed a node that was 5 hops away from Node 1. In this AODV implementation, the maximum hops for a broadcasted RREQ message is set to 10. Based on our deployed WSN size, the number of hops between Node 1 and Node 0 (i.e. the base-station) is less than 10. If the hop number is increased correctly (i.e., one for each hop), then the RREQ message that is to search for a routing path, can be received by Node 0 and the path can be established successfully. However, as Figure 5.6 shows, the number of hops was mistakenly increased one more time when the RREQ message was forwarded. As a result, the RREQ message expired at the fifth hop, which caused the link establishment to fail. Note that ERROR#3 is successfully diagnosed with 7 nodes replayed, which is a much smaller number than the maximum replays that may be needed in the worst replay order. In the worst possible order of the replay

sequence, all 67 nodes that are less than 5 hops away from the source node might be replayed. Adding Node 0 and the node on which the error source is located, a total of 69 nodes would be replayed in such a worst case.

6 RELATED WORK

6.1 Wireless Sensor Networks Software Debugging

Next are discussed the methods for error diagnosis and debugging for wireless sensor networks.

Program analysis [37, 42–44] is widely used for wireless sensor network error detection. Model checking [45–47] is one of the main approaches for error detection in distributed systems. The essential idea is that given a system design/implementation and a property which should be satisfied by that system, by systematically checking all possible execution paths, a model checker either outputs YES if the property can be maintained by the system or generates a counterexample otherwise. Theoretically, model checking should explore the entire state-spaces in a controlled environment. However, for large programs the state-explosion problem becomes a fundamental problem in applying model checking. To combat this problem, researchers have investigated reducing techniques to control the state explosion, such as symmetry reduction [48] and partial-order reduction [49]. Moreover, to explore fully the behavior of large programs using practical resources, e.g., time and memory, heuristics are also introduced in model checking, particularly depth-bounding and context-bounding. Depth-bounding [37, 50] limits the state search within a pre-defined number of steps, while context-bounding [51] distinguishes between preempting and non-preempting context switches and bounds the number of preempting context switches. However, model checking has two main limitations. On the one hand, a networked embedded system reacts to events whose timing is difficult to predict or specify at the time of the program development. Furthermore, the errors that have been detected in the labora-

tory may significantly differ from the errors emerging in the real deployment because of the different conditions. Due to the dynamic and complicated WSN application, it is difficult and sometimes infeasible to let the controlled environment precisely reflect the real running situation, and errors often exist in the network after deployment. On the other hand, even by applying the reducing or heuristic techniques, the scalability of model checking still becomes an issue. In general, model checking handles the WSN with small number of sensor nodes. However, in the real application, tens of hundreds of sensor nodes are used and some errors can not manifest themselves when the WSN is trivial. For example, one of our experiments shows that an error related to calculate the number of hops incorrectly was detected in a WSN with 150 sensor nodes while was hidden in a WSN with only 30 sensor nodes.

Simulation/emulation [38,52–55] offers considerable flexibility but often takes significantly more time than the direct execution. Another factor to consider is that the simulated cases may not be sufficiently extensive to catch errors that may happen during the real operation. The simulated operation environment may also be quite different from the operational environment.

Interactive debugging [40,56,57] allows programmers to interact with sensor nodes by sending commands. The set of commands usually includes those which set break points, watch points, and initiate step-by-step tracing. This methodology works particularly well if the programmer already knows what kind of errors will happen and where the places to look are. Otherwise, the step-by-step execution can be quite slow and tedious, with no guarantee that the anticipated error will surface in the debugging mode. In other circumstances, especially when the number of motes to be debugged simultaneously is large, it seems much more convenient to have execution traces ready when an error is detected.

Run-time logging [2, 3, 58–62] has gained increased importance recently. The critical questions encountered when adopting this approach include what kind of errors should be monitored, where and how to log information for later debugging, and how to analyze the logged information necessary to find out the error cause. Among recent efforts, Sympathy [60] focuses on data-collection applications. The metrics generated by each node are sent to a data sink, and a decision tree is applied to the collected data to find the failures. TinyTrace [61] implements an efficient approach to trace intraprocedural and interprocedural control-flow of all interleaving concurrent events. Dustminer [3] is a tool for uncovering bugs in networked sensing applications due to nondeterministic and incorrect interactions between different nodes. This tool collects a sequence of events and uses data mining techniques to recognize abnormal behaviors. PAD [2] is a light-weight packet marking scheme for collecting necessary hints, and it uses a probabilistic inference model residing at the sink to capture unique features of the sensor networks. Passive Distributed Assertions [59] allows the programmer to define certain properties of a distributed system. The state information of each affected node is collected and analyzed through a separately-deployed sniffer network. PD2 [58] focuses on the data flows generated by an application. It relates the poor application performance to significant data losses or latencies of certain data flows (called problematic data flows) as they go through the software modules on individual nodes and through the network. However, these methods either focus on only a single node or the logging information is too coarse to reproduce errors. Besides software-only approaches, hardware-supported approach [63–65] is also used for runtime logging. AVEKSHA [64] provides a hardware-software approach to trace events at runtime in a sensor node without slowing down the application. The hardware-supported approach can provide a low-overhead logging. However, it is generally designed for a particular platform and for monitoring limited type of events. Com-

pared to our source-level tracing, the collected trace is hard to read and re-matched to the source code.

6.2 Record and Replay

The methods for record and replay can be loosely classified into three categories: software-only, hardware-only, and a hybrid approach. Given that our work is software-only, we briefly survey the current software-only record-and-replay techniques in distributed and parallel systems.

One typical methodology is to record all possible factors (i.e., non-determinism) that affect the program's execution and then re-execute the program. Although this approach is capable to replay the original execution perfectly , the overhead is huge. For example, iDNA [66], developed by Microsoft, logs the memory instruction input values and maintains a copy of user-level memory to identify system-call side-effects. PinPlay [7], developed by Intel, is a framework for deterministic capture and reproducible analysis of parallel programs. In addition, numerous works have made an effort on to reduce the overhead of space and execution time [67, 68]. To lower the production-run recording overhead further, another replay method, only recording partial replay information, has been provided in recent year. PRES [8] records only partial execution information called sketching. Based on the recorded sketching, it navigates a non-deterministic execution space several times trying to reproduce errors. After several replay attempts, PRES can then reproduce the error with 100% probability on every subsequent replay for diagnostic purpose. ODR [6] addresses the output-failure replay problem by using output-determinism rather than value-determinism. That is, it generates a run that exhibits the same outputs as the original rather than an identical replica in order to achieve a low-overhead recording of multiprocessor runs.

However, the literature discussed above focuses on multiprocessors/multi-cores, and their nature of non-determinism is quite different from that arising from distributed systems. The latter are due to factors such as interrupts, network delays, and unreliable communication. Moreover, the replay techniques mentioned above have mainly been used on resource-rich platforms, and cannot be used practically on networked embedded systems which generally have severe resource limitations. For networked embedded systems, Sundmark and Thane [45] took a snapshot of the of the execution context checksums when an interrupt occurs during the recording phase. Gracioli and Fischmeister [46] have adopted hierarchy approaches to record interrupt behavior. Moreover, they have used the observed principle of return address clustering and a formal model for quantitative reasoning about the tracing mechanism to tune their tracing mechanism. However, the above works only consider errors caused by interrupts, and the result is not always accurate.

Additionally, with respect to instrumentation, the record-and-replay techniques can also be separated into the source-level and the binary-level instrumentation. On the one hand, most of the works discussed above used binary level instrumentation due to the lack of source code for system or commercial packets. On the other hand, source-level instrumentation offers a high portability across different devices and an easy correlation to the original program, which provides an easier way to help programmers locate error source. Wu et.al [47] have used an execution flow chat to decide the non-deterministic information for distributed systems, and generated a record program and a replay program at the same time. However, their work assumes that the side effects of an unreachable function can be pre-acknowledged, which is not always true in most distributed embedded systems.

6.3 System Behavior Synthesis

Our GPD algorithm utilizes WSN domain knowledge to obtain a message-efficient decomposition. To our best knowledge, there are no prior studies or tools which employ a similar approach. Our property definition benefits from prior work on automatic synthesis and macro programming. With automatic synthesis [69–72], the behavior of the overall system is specified in a global manner, and then it is automatically synthesized to obtain a distributed implementation from the specification. The previous study focuses on the design of distributed systems, whose main difficulty resides in the extremely high number of possible interactions between the concurrent components of the system. In a macro-programming [73, 74] system, the user writes a single program that specifies the global operations for the entire system, and the framework automatically decomposes this into a set of micro-programs for each node. Unfortunately, the results from these cited prior work cannot be directly utilized for our purpose due to the several reasons presented next.

First, the automatic synthesis is applied in the system design phase. The decomposition result describes local behaviors which can be implemented in different ways (including different algorithms, data structures, variables, and so on) by different programmers. Our approach must deal with an existing application program and must take domain-specific information in order for error detection to be feasible.

Second, in our work, the global property is defined at a high level by using abstractions, a method which is similar to macro programming. But instead of generating a program running on individual nodes (which is the purpose of macro programming system), our decomposition tool produces a piece of code to represent properties to be checked.

In summary, our decomposition framework and its supporting tools represent a new approach shown to be effective for error detection in wireless sensor networks.

7 CONCLUSION AND FUTURE WORK

Error detection and diagnosis for network embedded systems remain challenging tasks due to their large number of computing entities, hardware resource constrains, and inherited nondeterminism. In this dissertation, we take wireless sensor networks, a special but representative type of network embedded systems, as a concrete example, to investigate error detection and dignosis.

We have presented a domain specific language SensorC to specify properties and a Global Property Decomposition (GDP) algorithm intergraded with our developed SensorC compiler, which is responsible for decomposing the given properties and for finding nodes to detect those properties locally. As our experiments have illustrated, the approach can (a) reduce the error detection time; (b) reduce the communication traffic for state information exchanges used in centralized error detection; and (c) narrow down the range of collected trace used for off-line replay. In addition, the global specification can also be intergraded with other verification approaches such as model checking.

To help programmers reproduce and diagnosis errors, we have presented a dependence-based source-level method for memory-efficient tracing and replay. The tool developed based on our method is independent of the hardware platforms and the cross compiler (except for a system library call to make certain memory accesses atomic) and has been applied on WSNs consisting of TelosB motes and Z1 motes separately. The experiments results show that our work has advanced a way to instrument several test programs on WSN under the stringent program memory constraints by using this proposed method, and we found and diagnosed realistic errors.

Our current experiments are performed on TinyOS-based WSN applications; however, the proposed methodology and tool can be applied to other networked embedded systems if (a) the system domain information is acquired; and (b) the applications satisfy the assumptions made in Chapter 2.

Despite the above efforts, there are still several problems to be solved in the future research. First, in our current work, the routing information is useful for property decomposition only if the WSN is stationary, and we have yet to define the decomposition rules in a fine-gained for dynamic WSN. In addition, although we have detected and located some realistic errors, we need to conduct more experiments so as to apply the algorithm to additional WSN applications and other types of networked embedded systems.

LIST OF REFERENCES

LIST OF REFERENCES

[1] Nicola Bombieri, Franco Fummi, and Davide Quaglia. System/Network design-space exploration based on TLM for networked embedded systems. *ACM Transactions on Embedded Computing Systems*, 9(4):37:1–37:32, April 2010.

[2] Kay Romer and Junyan Ma. PDA: Passive distributed assertions for sensor networks. In *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, IPSN '09, pages 337–348, Washington, DC, USA, 2009. IEEE Computer Society.

[3] Mohammad Maifi Hasan Khan, Hieu Khac Le, Hossein Ahmadi, Tarek F. Abdelzaher, and Jiawei Han. Dustminer: Troubleshooting interactive complexity bugs in sensor networks. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, pages 99–112, New York, NY, USA, 2008. ACM.

[4] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D3S: Debugging deployed distributed systems. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '08, pages 423–437, Berkeley, CA, USA, 2008. USENIX Association.

[5] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Partial replay of long-running applications. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 135–145, New York, NY, USA, 2011. ACM.

[6] Gautam Altekar and Ion Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 193–206, New York, NY, USA, 2009. ACM.

[7] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 2–11, New York, NY, USA, 2010. ACM.

[8] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 177–192, New York, NY, USA, 2009. ACM.

[9] H. Thane, Daniel Sundmark, J. Huselius, and A. Pettersson. Replay debugging of real-time systems using time machines. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, page 8, 2003.

[10] Darren Dao, Jeannie Albrecht, Charles Killian, and Amin Vahdat. Live debugging of distributed systems. In *Proceedings of the 18th International Conference on Compiler Construction: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, CC '09, pages 94–108, Berlin, Heidelberg, 2009. Springer-Verlag.

[11] Cristian Zamfir, Gautam Altekar, George Candea, and Ion Stoica. Debug determinism: The sweet spot for replay-based debugging. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 18–18, Berkeley, CA, USA, 2011. USENIX Association.

[12] K. Romer and F. Mattern. The design space of wireless sensor networks. *Wireless Communications, IEEE*, 11(6):54–61, 2004.

[13] Ning Xu, Sumit Rangwala, Krishna Kant Chintalapudi, Deepak Ganesan, Alan Broad, Ramesh Govindan, and Deborah Estrin. A wireless sensor network for structural monitoring. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, pages 13–24, New York, NY, USA, 2004. ACM.

[14] Makoto Suzuki, Shunsuke Saruwatari, Narito Kurata, and Hiroyuki Morikawa. A high-density earthquake monitoring system using wireless sensor networks. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys '07, pages 373–374, New York, NY, USA, 2007. ACM.

[15] Andreas Hasler, Igor Talzi, Christian Tschudin, and Stephan Gruber. Wireless sensor networks in permafrost research - concept, requirements, implementation and challenges. In *9th International Conference on Permafrost, NICOP 2008*, 2008.

[16] Kenan Casey, Alvin Lim, and Gerry Dozier. A sensor network architecture for tsunami detection and response. *International Journal of Distributed Sensor Networks*, 4(1):28–43, January 2008.

[17] K. K. Tan, S. N. Huang, Y. Zhang, and T. H. Lee. Distributed fault detection in industrial system based on sensor wireless network. *Computer Standards & Interfaces*, 31(3):573–578, March 2009.

[18] Arunanshu Mahapatro and Pabitra Mohan Khilar. Fault diagnosis in wireless sensor networks: A survey. *Communications Surveys Tutorials, IEEE*, 15(4):2000–2026, 2013.

[19] http://www.eol.ucar.edu/isf/facilities/isa/internal/CrossBow/DataSheets/mica2.pdf.

[20] http://www.openautomation.net/uploadsproductos/micaz_datasheet.pdf.

[21] http://www.willow.co.uk/TelosB_Datasheet.pdf.

[22] http://www.zolertia.com/products/z1.

[23] http://www.tinynode.com/.

[24] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[25] G.S. Tseitin. On the complexity of derivation in propositional calculus. In JrgH. Siekmann and Graham Wrightson, editors, *Automation of Reasoning*, Symbolic Computation, pages 466–483. Springer Berlin Heidelberg, 1983.

[26] Seth Koehler, John Curreri, and Alan D. George. Performance analysis challenges and framework for high-performance reconfigurable computing. *Parallel Computing*, 34(4-5):217–230, May 2008.

[27] Michael Barr and Anthony Massa. *Programming Embedded Systems*. O'Reilly Media, Inc, second edition, 2006.

[28] http://www.tinyos.net/community.html.

[29] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. *ACM SIGPLAN Notices*, 38(5):1–11, May 2003.

[30] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. TinyOS: An operating system for sensor networks. In *Ambient Intelligence*, pages 115–148. Springer Berlin Heidelberg, 2005.

[31] C.E. Perkins and E.M. Royer. Ad-hoc on-demand distance vector routing. In *Mobile Computing Systems and Applications, 1999. Proceedings. WMCSA '99*, pages 90–100, February 1999.

[32] Madanlal Musuvathi. *CMC: A Model Checker for Network Protocol Implementations*. PhD thesis, Stanford University, 2004.

[33] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[34] GCC, the GNU compiler collection. http://gcc.gnu.org.

[35] http://www2.engr.arizona.edu/~junseok/AODV.html.

[36] The collection tree protocol. http://www.tinyos.net/tinyos-2.x/doc/html/tep123.html.

[37] Peng Li and John Regehr. T-check: Bug finding for sensor networks. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN '10, pages 174–185, New York, NY, USA, 2010. ACM.

[38] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. TOSSIM: Accurate and scalable simulation of entire tinyos applications. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, SenSys '03, pages 126–137, New York, NY, USA, 2003. ACM.

[39] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[40] G. Tolle and D. Culler. Design of an application-cooperative management system for wireless sensor networks. In *Proceeedings of the Second European Workshop on Wireless Sensor Networks*, pages 121–132, 2005.

[41] David Binkley. Precise executable interprocedural slices. *ACM Letters on Programming Languages and Systems*, 2(1-4):31–45, March 1993.

[42] Nguyet T. M. Nguyen and Mary Lou Soffa. Program representations for testing wireless sensor network applications. In *Workshop on Domain Specific Approaches to Software Test Automation*, DOSTA '07, pages 20–26, New York, NY, USA, 2007. ACM.

[43] Zhifeng Lai, S. C. Cheung, and W. K. Chan. Inter-context control-flow and data-flow test adequacy criteria for nesc applications. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 94–104, New York, NY, USA, 2008. ACM.

[44] Nupur Kothari, Todd Millstein, and Ramesh Govindan. Deriving state machines from tinyos programs using symbolic execution. In *Proceedings of the 7th International Conference on Information Processing in Sensor Networks*, IPSN '08, pages 271–282, Washington, DC, USA, 2008. IEEE Computer Society.

[45] D. Sundmark and H. Thane. Pinpointing interrupts in embedded real-time systems using context checksums. In *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2008*, pages 774–781, September 2008.

[46] Giovani Gracioli and Sebastian Fischmeister. Tracing interrupts in embedded software. *ACM SIGPLAN Notices*, 44(7):137–146, June 2009.

[47] Ming Wu, Fan Long, Xi Wang, Zhilei Xu, Haoxiang Lin, Xuezheng Liu, Zhenyu Guo, Huayang Guo, Lidong Zhou, and Zheng Zhang. Language-based replay via data flow cut. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 197–206, New York, NY, USA, 2010. ACM.

[48] Alice Miller, Alastair Donaldson, and Muffy Calder. Symmetry in temporal logic model checking. *ACM Computing Surveys*, 38(3), 2006.

[49] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[50] Stuart J. Russell and Peter Norvig. *Artificial intelligence: A modern approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[51] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 446–455, New York, NY, USA, 2007. ACM.

[52] Lewis Girod, Nithya Ramanathan, Jeremy Elson, Thanos Stathopoulos, Martin Lukac, and Deborah Estrin. Emstar: A software environment for developing and deploying heterogeneous sensor-actuator networks. *ACM Transactions on Sensor Networks*, 3(3), August 2007.

[53] Ye Wen, Rich Wolski, and Selim Gurun. S2DB: A novel simulation-based debugger for sensor network applications. In *Proceedings of the 6th ACM and IEEE International Conference on Embedded Software*, EMSOFT '06, pages 102–111, New York, NY, USA, 2006. ACM.

[54] J. Polley, D. Blazakis, J. McGee, D. Rusk, and J.S. Baras. Atemu: A fine-grained sensor network simulator. In *First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks, IEEE SECON 2004*, pages 145–152, 2004.

[55] B.L. Titzer, D.K. Lee, and J. Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Fourth International Symposium on Information Processing in Sensor Networks, IPSN 2005*, pages 477–482, 2005.

[56] Jing Yang, Mary Lou Soffa, Leo Selavo, and Kamin Whitehouse. Clairvoyant: A comprehensive source-level debugger for wireless sensor networks. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys '07, pages 189–203, New York, NY, USA, 2007. ACM.

[57] Kamin Whitehouse, Gilman Tolle, Jay Taneja, Cory Sharp, Sukun Kim, Jaein Jeong, Jonathan Hui, Prabal Dutta, and David Culler. Marionette: Using RPC for interactive development and debugging of wireless embedded networks. In *Proceedings of the 5th International Conference on Information Processing in Sensor Networks*, IPSN '06, pages 416–423, New York, NY, USA, 2006. ACM.

[58] Zhigang Chen and K.G. Shin. Post-Deployment performance debugging in wireless sensor networks. In *Real-Time Systems Symposium, RTSS 2009, 30th IEEE*, pages 313–322, 2009.

[59] Kebin Liu, Mo Li, Xiaohui Yang, and Mingxing Jiang. Passive diagnosis for wireless sensor networks. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, pages 371–372, New York, NY, USA, 2008. ACM.

[60] Nithya Ramanathan, Kevin Chang, Rahul Kapur, Lewis Girod, Eddie Kohler, and Deborah Estrin. Sympathy for the sensor network debugger. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, SenSys '05, pages 255–267, New York, NY, USA, 2005. ACM.

[61] Vinaitheerthan Sundaram, Patrick Eugster, Xiangyu Zhang, and Vamsidhar Addanki. Diagnostic tracing for wireless sensor networks. *ACM Transactions on Sensor Networks*, 9(4):38:1–38:41, July 2013.

[62] Veljko Krunic, Eric Trumpler, and Richard Han. NodeMD: Diagnosing node-level faults in remote wireless sensor systems. In *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services*, MobiSys '07, pages 43–56, New York, NY, USA, 2007. ACM.

[63] K. Shankar and R. Lysecky. Control focused soft error detection for embedded applications. *Embedded Systems Letters, IEEE*, 2(4):127–130, 2010.

[64] Matthew Tancreti, Mohammad Sajjad Hossain, Saurabh Bagchi, and Vijay Raghunathan. Aveksha: A hardware-software approach for non-intrusive tracing and profiling of wireless embedded systems. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, SenSys '11, pages 288–301, New York, NY, USA, 2011. ACM.

[65] Travis Goodspeed. Goodfet. http://goodfet.sourceforge.net, August 2011.

[66] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE '06, pages 154–163, New York, NY, USA, 2006. ACM.

[67] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. Triage: Diagnosing production run failures at the user's site. *ACM SIGOPS Operating Systems Review*, 41(6):131–144, October 2007.

[68] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient deterministic multithreading in software. *ACM SIGPLAN Notices*, 44(3):97–108, March 2009.

[69] Alin Stefanescu. Automatic synthesis of distributed systems. http://www.fmi.uni-stuttgart.de/szs/publications/stefanan/thesis-online.pdf, 2006.

[70] Sebastian Uchitel, Greg Brunet, and Marsha Chechik. Behaviour model synthesis from properties and scenarios. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 34–43, Washington, DC, USA, 2007. IEEE Computer Society.

[71] Ilaria Castellani, Madhavan Mukund, and P.S. Thiagarajan. Synthesizing distributed transition systems from global specifications. In *Foundations of Software Technology and Theoretical Computer Science*, volume 1738 of *Lecture Notes in Computer Science*, pages 219–231. Springer Berlin Heidelberg, 1999.

[72] M. Kloetzer and C. Belta. Distributed implementations of global temporal logic motion specifications. In *IEEE International Conference on Robotics and Automation*, pages 393–398, May 2008.

[73] Timothy W. Hnat, Tamim I. Sookoor, Pieter Hooimeijer, Westley Weimer, and Kamin Whitehouse. Macrolab: A vector-based macroprogramming framework for cyber-physical systems. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys'08, pages 225–238, New York, NY, USA, 2008. ACM.

[74] Tamim Sookoor, Timothy Hnat, Pieter Hooimeijer, Westley Weimer, and Kamin Whitehouse. Macrodebugging: Global views of distributed program execution. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 141–154, New York, NY, USA, 2009. ACM.

VITA

VITA

Man Wang was born in Xinxiang, Henan Province, China, in 1983. She graduated from the High School attached to Henan Normal University and entered Fudan University in 2001. In 2005, she received her B.S degree from the Computer Science Department in Fudan University and entered Tsinghua University. After she received her M.S. degree from the Computer Science and Engineering Department in Tsinghua University in 2008, she then joined Professor Zhiyuan Li's research group in Purdue University to pursue her Ph.D. degree. During her Ph.D. program her research focused mainly on compiler techniques, and error detection and diagnosis for distributed networked systems.