

# Finding Symbolic Bug Patterns in Sensor Networks

Mohammad Maifi Hasan Khan, Tarek Abdelzaher, Jiawei Han,  
and Hossein Ahmadi

Department of Computer Science  
University of Illinois at Urbana-Champaign  
201 North Goodwin, Urbana, Illinois, USA  
mmkhan2@uiuc.edu, {zaher,hanj}@cs.uiuc.edu, hahmadi2@uiuc.edu

**Abstract.** This paper presents a failure diagnosis algorithm for summarizing and generalizing patterns that lead to instances of anomalous behavior in sensor networks. Often multiple seemingly different event patterns lead to the same type of failure manifestation. A hidden relationship exists, in those patterns, among event attributes that is somehow responsible for the failure. For example, in some system, a message might always get corrupted if the sender is more than two hops away from the receiver (which is a distance relationship) irrespective of the senderId and receiverId. To uncover such failure-causing relationships, we present a new *symbolic* pattern extraction technique that identifies and symbolically expresses relationships correlated with anomalous behavior. *Symbolic* pattern extraction is a new concept in sensor network debugging that is unique in its ability to generalize over patterns that involve different combinations of nodes or message exchanges by extracting their common relationship. As a proof of concept, we provide synthetic traffic scenarios where we show that applying symbolic pattern extraction can uncover more complex bug patterns that are crucial to the understanding of real causes of problems. We also use symbolic pattern extraction to diagnose a real bug and show that it generates much fewer and more accurate patterns compared to previous approaches.

**Keywords:** symbolic pattern, interactive bugs, wireless sensor network.

## 1 Introduction

Wireless sensor network applications typically implement distributed protocols where multiple nodes communicate with each other to collectively perform a collaborative task. Nodes often assume roles such as cluster heads, sensors, or forwarding nodes. Messages have types, usually defined by the respective applications. Such applications often fail due to some unexpected sequences of (communication or other) events, which are not handled properly by the protocol design and/or due to some implementation oversight that leads to a “bad” state which eventually leads to the failure. In this paper, we generalize from actual observed message exchanges to the underlying relationships defined on

nodes, roles, and message types, that lead to a failure. We call them *symbolic bug patterns*.

Unfortunately, none of the existing debugging tools and techniques available for sensor networks is capable of troubleshooting symbolic bugs. State of the art debugging techniques for sensor network include passive diagnosis [10,16], usage of declarative tracepoints to collect runtime logs for offline analysis [4], discriminative sequence mining for offline analysis of runtime logs [8,9], real-time failure diagnosis [18], traditional breakpoints and watchpoint primitives to probe into hardware at runtime [24,23] and more traditional techniques such as simulation [11,17,21], emulation [7], and testbeds [22,5]. Though source code level debugging tools [24,23] help identify single node programming errors, these are not very effective to diagnose interaction bugs. Dustminer [9] comes closest to our work. It uses frequent sequence mining techniques for bug diagnosis. Though it makes an effort towards using sequence mining for finding interaction bugs, in this paper we show that analyzing sequences of events based on absolute event attribute values to diagnose such bugs is not enough. To make things worse, the resulting patterns can often be misleading and can confuse the application developer.

In this paper, to diagnose complex bug patterns, we introduce the concept of *symbolic patterns* that identify the “culprit” sequences of events responsible for failure by capturing the relationships among different event attributes. In the context of this paper, a symbolic pattern is a pattern where all or a subset of the absolute values of event attributes within the pattern are replaced with *symbols* to generalize the pattern. To perform offline analysis using symbolic pattern extraction, different types of runtime events are logged during program execution and offline analysis is done to identify the discriminative set of frequent symbolic patterns that will contain the “culprit” symbolic patterns that are highly correlated to failure.

The rest of the paper is organized as follows. In section 2 we introduce the model for symbolic patterns. In Section 3, we describe the state of the art in debugging tools and techniques specifically developed for sensor network applications and explain their limitations. In Section 4, we explain the mechanism behind symbolic pattern extraction and the pattern ranking scheme. We compare and evaluate the debugging capability of symbolic pattern extraction to prior related schemes [8,9] in Section 5 using a synthetic bug and a real bug. Finally, Section 6 concludes the paper.

## 2 A Model for Symbolic Patterns

The logged events in our system can include any operations performed at runtime such as message transmission, message reception, and writing to flash storage. Each recorded event can have multiple attributes. For example, a message transmission event can have senderId, senderType, destinationId and msgType as attributes. For the purposes of the discussion below, let us define an event to be the basic element in the event log that is analyzed for failure diagnosis. The

format of a logged event and the definition of sequences of events are similar to those defined in [9]:  $\langle EventType, attribute_1, attribute_2, \dots, attribute_n \rangle$ .

For example,  $attribute_1$  can be  $SenderId$  in case of a  $messageSent$  event. The generated log can be thought of as a single sequence of events. For example, consider the following logged events in a sample log:

$\langle msgSent, senderId = 1, msgType = 0, destinationId = 3 \rangle$   
 $\langle msgReceived, receiverId = 1, msgType = 1, senderId = 3 \rangle$   
 $\langle flashWriteInitiated, nodeId = 1, dataSize = 100 \rangle$

The above log can be considered a single sequence of three events each with multiple attributes. A frequent sequence mining algorithm [1] is used to extract frequent subsequences of events. Events in the subsequence do not have to be contiguous in the original sequence. We use the term “frequent (sub)sequence of events” and “frequent pattern” interchangeably in this paper.

A *discriminative pattern* between two logs is an ordered subsequence of events that occurs with a different *support* in the two logs, where support refers to the number of times it occurs. The larger the difference in support, the better the discriminative power. Before we formally define a *symbolic pattern*, let us consider the following example to illustrate what it means. Say, we have two patterns S1 and S2 where each pattern has two events with multiple attributes as follows:

$S1 = \langle msgSent, senderId = 1, msgType = 0 \rangle$   
 $\langle msgReceived, receiverId = 2, msgType = 0 \rangle$   
 $S2 = \langle msgSent, senderId = 3, msgType = 0 \rangle$   
 $\langle msgReceived, receiverId = 5, msgType = 0 \rangle$

where node 1 is the neighbor of node 2 and node 3 is the neighbor of node 5. On the surface, patterns S1 and S2 are different. Now, if we parameterize the relationship that exists between  $senderId$  and  $receiverId$  and represent it using symbol  $X$  for  $senderId$ , S1 and S2 can be represented as follows:

$S1 = \langle msgSent, \underline{senderId = X}, msgType = 0 \rangle$   
 $\langle msgReceived, \underline{receiverId = neighbor(X)}, msgType = 0 \rangle$   
 $S2 = \langle msgSent, \underline{senderId = X}, msgType = 0 \rangle$   
 $\langle msgReceived, \underline{receiverId = neighbor(X)}, msgType = 0 \rangle$

Interestingly, S1 and S2 now become the same pattern which expresses a more general relationship. Note that, if S1 and S2 each has support 1, the symbolic version has support 2 and hence symbolizing patterns increase the visibility of the pattern in the event log.

More formally, in the context of this paper, *symbolic pattern extraction* is the task of identifying frequent patterns that satisfy certain relationships, specified by the user or selected from a library of common relationships, defined among event attributes of same or different types of events (*e.g., neighborhood relationship, identity relationship, and type relationships*). These relationships are then represented using symbols instead of absolute values where appropriate. In this paper, we present an algorithm for *symbolic pattern extraction* which at first, generates frequent patterns using the Apriori algorithm [1]. Next, it generalizes frequent patterns generated in the first stage by mining for “relationships” in those patterns. In this paper, we also present a hybrid scheme for counting the

support for individual patterns which greatly enhances the chance of identifying “infrequent” events that are correlated with failure. Finally, we propose a pattern ranking scheme exploiting the characteristics of symbolic patterns which increases the usability of the tool.

To analyze the performance of symbolic pattern extraction, we simulated several bugs in TOSSIM to generate log files and analyzed them using our new algorithm. We choose simulation to generate log files as it gives us the flexibility to experiment with bugs of arbitrary complexity. We compare discriminative symbolic patterns generated by our symbolic pattern extraction algorithm with the discriminative patterns generated by the algorithm we presented earlier [9] and show that symbolic patterns greatly enhance the diagnostic capability and the usability of the tool.

### 3 Related Work

State of the art debugging techniques for wireless sensor networks include passive diagnosis [10,16], sequence mining for offline analysis of runtime logs [8,9], and real-time failure diagnosis [18]. To increase the visibility inside the node a recent effort [4] proposed the usage of declarative tracepoints to collect runtime logs for offline analysis.

To facilitate debugging single-node programming bugs (e.g., bad pointer reference, stack overflow etc.), sophisticated tools such as Clairvoyant [24] and Marionette [23] are developed that provide standard debugging primitives such as breakpoints and watchpoints that enable stepping through execution on a line-by-line basis. Though these tools are very useful to debug programming or localized errors, these are not very useful if the cause of the failure is distributed across multiple nodes. Moreover stepping through code execution can cause the bug to disappear or to create new problems, if they are timing-related.

At the other extreme of the spectrum lie debugging tools such as SNTS [10] which try to debug a deployed sensor network by analyzing passively recorded radio communication messages in the network. Although passive diagnosis does not interfere with the operation of the network, the diagnostic capability is rather limited due to unavailability of “critical” run time information. Sympathy [18] performs real-time diagnosis using a classification tree approach and uses a minimal amount of run-time data collected at a central node to perform the diagnosis. The diagnosis is done based on reduced throughput in the network. Dustminer [9] uses flash on the chip to log runtime events and uses sequence mining for offline analysis to diagnose the cause of the problem. As each node records logs locally and does not upload data in real time, it does not compete for the radio to communicate and hence minimizes the interference. In [9], the authors identified several limitations of an existing sequence mining algorithm [1] and extended it to address those issues. None of the above techniques can find *symbolic patterns* automatically.

SNMS [20] presents a more traditional sensor network management service that collects and summarizes different types of measurements such as packet

loss and radio energy consumption. Although laboratory test-beds like Motelab [22], Kansei [5], and Emstar [7] provide the convenience of testing in a controlled environment, they do not provide hints to the developer if something goes “wrong” (e.g., some random nodes stopped after 2 hours) during the testing. Other work [19] shows that erroneous sensor readings such as temperature and humidity can be used to predict network and node failures but it does not provide the answer to the question “Why was the sensor reading bad in the first place?”.

Using machine learning techniques to diagnose failure is not new [3,2,6,12,14,15]. Discriminative frequent pattern analysis [6], software behavior graph analysis [14], a Bayesian analysis based approach [13], and control flow analysis to identify logic error [15] are a few examples. These techniques do not focus on extracting symbolic relationships, however.

## 4 Overview

To answer the question “*Why do we need discriminative symbolic pattern extraction to debug interaction bugs?*”, we provide an example in Section 4.1. We then present the symbolic pattern extraction algorithm in Section 4.2. We conclude the section by presenting a hybrid support count function and a pattern ranking scheme that have significant impact on the quality of the patterns generated and the scalability of the algorithm.

### 4.1 Motivation for Using Symbolic Pattern for Debugging

Let us assume that, in a particular application, each neighbor of node A periodically communicates with node A and is always expected to send messages of type 0, 1 and 2 in a fixed order, where msgType 0 is followed by msgType 1 and msgType 2, respectively. Also, assume that any violation of this message order from a specific sender crashes the system. Now, let us log a few examples of correct execution (Good Log) and execution that leads to a manifestation of error (Bad Log). Consider the log file presented in Table 1, collected from node 1 and node 7, where node 1 did not crash and node 7 crashed. Note that, node 7 crashed as node 8 sent messages violating the required sequence of message types.

If we generate the patterns correlated with failure, a state of the art algorithm would come up with the following: pattern  $seq_1$  with support 2 and pattern  $seq_2$  with support 1 along with other frequent patterns.

$$\begin{aligned}
 seq_1 &= \langle msgReceived, msgType = 0 \rangle \\
 &\quad \langle msgReceived, msgType = 1 \rangle \\
 &\quad \langle msgReceived, msgType = 2 \rangle \\
 seq_2 &= \langle msgReceived, msgType = 2 \rangle \\
 &\quad \langle msgReceived, msgType = 0 \rangle \\
 &\quad \langle msgReceived, msgType = 1 \rangle
 \end{aligned}$$

**Table 1.** Sample Log File

Good Log (Node 1)	1. $\langle msgReceived, receiverId = 1, senderId = 3, msgType = 0 \rangle$
	2. $\langle msgReceived, receiverId = 1, senderId = 3, msgType = 1 \rangle$
	3. $\langle msgReceived, receiverId = 1, senderId = 3, msgType = 2 \rangle$
	4. $\langle msgReceived, receiverId = 1, senderId = 2, msgType = 0 \rangle$
	5. $\langle msgReceived, receiverId = 1, senderId = 2, msgType = 1 \rangle$
	6. $\langle msgReceived, receiverId = 1, senderId = 2, msgType = 2 \rangle$
Bad Log (Node 7)	1. $\langle msgReceived, receiverId = 7, senderId = 6, msgType = 0 \rangle$
	2. $\langle msgReceived, receiverId = 7, senderId = 6, msgType = 1 \rangle$
	3. $\langle msgReceived, receiverId = 7, senderId = 8, msgType = 2 \rangle$
	4. $\langle msgReceived, receiverId = 7, senderId = 8, msgType = 0 \rangle$
	5. $\langle msgReceived, receiverId = 7, senderId = 8, msgType = 1 \rangle$
	6. $\langle msgReceived, receiverId = 7, senderId = 6, msgType = 2 \rangle$

If we inspect the logged events presented in table 1 carefully, we can see that there is also a pattern associated with the senderId, where senderId is the same for  $seq_1$ . For the first and second occurrence of  $seq_1$ , senderId is 3 and 2 respectively. This pattern is missed due to different support. For example, logged event  $\langle msgReceived, receiverId = 1, senderId = 3, msgType = 0 \rangle$  has support 1 and  $\langle msgReceived, receiverId = 1, msgType = 0 \rangle$  has support 3.

On the other hand, if we parameterize the values of receiverId and senderId and replace the identical values with symbols where *receiverId* is replaced with *X* and *senderId* with *Y* in Good Log, the following pattern will be identified with support 2 where for the first occurrence  $X = 1$  and  $Y = 3$  and for the second occurrence  $X = 1$  and  $Y = 2$ .

$\langle msgReceived, receiverId = X, senderId = Y, msgType = 0 \rangle$

$\langle msgReceived, receiverId = X, senderId = Y, msgType = 1 \rangle$

$\langle msgReceived, receiverId = X, senderId = Y, msgType = 2 \rangle$

Similarly, if we extract symbolic patterns, we would be able to identify that the following pattern  $seq_3$  occurs only in Bad log but not in the Good log -

$\langle msgReceived, receiverId = X, senderId = Y, msgType = 2 \rangle$

$\langle msgReceived, receiverId = X, senderId = Y, msgType = 0 \rangle$

$\langle msgReceived, receiverId = X, senderId = Y, msgType = 1 \rangle$

Without symbolic pattern extraction there is no way of identifying  $seq_3$ . A more detailed description of the *Symbolic pattern extraction* algorithm is presented in section 4.2.

## 4.2 Symbolic Pattern Extraction Algorithm

Symbolic pattern extraction is a two step process.

- During the first stage, multiattribute events are converted into single attribute events to reduce the computational complexity. Frequent patterns of events with single attribute are generated using a current sequence mining algorithm [1]. Let us call this set of frequent pattern the *base\_frequent\_set*.
- At the second stage, the candidate symbolic patterns are generated from this *base\_frequent\_set*. If the symbolic pattern  $s_i$  has support  $sup_{s_i}$  which is

generated from the base pattern  $p_i$  with support  $sup_{p_i}$  and  $(sup_{s_i}/sup_{p_i}) > \delta$ , then  $p_i$  is replaced by  $s_i$ .  $\delta$  is the *equivalence threshold* which is set by the user. If  $\delta$  is set to 0, all the symbolic patterns are retained and if it is set to 1 then symbolic patterns with the exact same support as the base pattern are retained. The generation of candidate symbolic patterns is described below.

**Generation of Candidate Symbolic Patterns.** To explain the generation of candidate symbolic patterns, without loss of generality, let us assume that  $Seq_a$  is a frequent base pattern of three events where each event is of different type and includes a single attribute from each event type.

$Seq_a = (\langle E_x, attr_2 = v_i \rangle, \langle E_y, attr_2 = v_j \rangle, \langle E_z, attr_3 = v_k \rangle)$

Say, event  $\langle E_x \rangle$  originally has 3 attributes and  $Seq_a$  includes only the second attribute of  $\langle E_x \rangle$ . Similarly, we assume  $\langle E_y \rangle$  and  $\langle E_z \rangle$  originally have 2 and 3 attributes respectively.

Next, the algorithm reconstructs the equivalent, complete pattern where each event has all the attributes. Now, the equivalent pattern generated from  $Seq_a$  would look like as follows:

$$\begin{aligned} &(\langle E_x, attr_1 = * \rangle, \langle E_x, attr_2 = v_i \rangle, \langle E_x, attr_3 = * \rangle) \\ &(\langle E_y, attr_1 = * \rangle, \langle E_y, attr_2 = v_j \rangle) \\ &(\langle E_z, attr_1 = * \rangle, \langle E_z, attr_2 = * \rangle, \langle E_z, attr_3 = v_k \rangle) \end{aligned}$$

Here “\*” is used for the attributes that are not included in the original pattern which basically says that the “\*” attributes are “*don't care*”. Next, the algorithm replaces a subset of the “\*” attributes with symbols and mine for relationship among those symbolic attributes. The symbolic pattern replaces  $Seq_a$  if the support of the symbolic pattern in the original log is “similar” to the support of  $Seq_a$ .

### 4.3 Challenges

**Meaningful Condition Identification.** “Which subset of “\*” attributes to replace with symbols and what “relationship” to test for?” is one of the key questions in finding meaningful symbolic bug patterns. We need to decide this intelligently to avoid useless checking such as “checking if nodeId and timeStamp are equal or not in a particular event”. Our goal is to automate the process as much as possible. To reduce the user involvement, we provide a list of predefined conditions that are especially applicable for wireless sensor network applications. For example, the common attributes expected for wireless sensor network applications are nodeId, message types, sensor data types, timestamps, etc. We tried to come up with the basic conditions that need to be checked. For example, checking if the “neighbor” condition holds between senderId and receiverId makes sense. The user needs to specify the type of the attribute in a header file. For example, if the type of  $i^{th}$  attribute of event  $E_x$  is “nodeId”, user may specify that information as  $(\langle E_x, attr_i, type : nodeId \rangle)$  from which the tool automatically determines the set of applicable conditions for this attribute. From that information, combinations of conditions of arbitrary complexity such as “Is the senderId always same as the receiverId?”, or “Does the msgType has to be

X and sender has to be the immediate neighbor to crash the receiver?" and so on can be generated automatically.

We realize that there may be conditions which are not provided by us. If the user wants to check for conditions that are not provided by a library function, he/she may implement the desired condition and add it to our library. A user may specify a condition that is not provided by the tool as follows:

( $\langle E_x, attr_i \rangle, \langle E_y, attr_j \rangle, Condition_q$ ) where  $Condition_q$  is defined and implemented by the user for his/her specific application. A pseudocode of the algorithm is given in table 2.

**Table 2.** Symbolic Pattern Extraction Algorithm

---

**Algorithm: Symbolic Pattern Extraction**

---

**Input:** Set of Good Logs (GL), Set of Bad Logs (BL), similarity measure ( $\delta$ )

**Output:** Set of discriminative symbolic pattern

1. PatternSetA=GenerateFrequentPatterns(GL)
2. SymbolicPatternSetA=ExtractSymbolicPattern(PatternSetA, GL,  $\delta$ )
3. PatternSetB=GenerateFrequentPatterns(BL)
4. SymbolicPatternSetB=ExtractSymbolicPattern(PatternSetB, BL,  $\delta$ )
5. DiscriminativePatternSet=DiffMine(SymbolicPatternSetA, SymbolicPatternSetB)
6. output DiscriminativePatternSet

**Function: ExtractSymbolicPattern**

**Input:** Set of Frequent Pattern (FP), Set of Logs (L), similarity measure ( $\delta$ )

**Output:** Set of symbolic pattern (SP)

1. SP=NULL; / \* set of Symbolic pattern \* /
  2. for each pattern  $p$  in FP
    - 2.1 for each checkcondition  $c$ 
      - 2.1.1 CSP=GenerateCandidateSymbolicPattern( $p, c$ )
      - 2.1.2 if(support(CSP)/support( $p$ ) >  $\delta$ ) then SP=SP U CSP
  2. return SP
- 

**Scalability.** One of the problems with symbolic pattern mining is that the number of combinations of conditions to check is exponential. For example, consider the following symbolic candidate pattern -

$$\begin{aligned} & (\langle E_x, attr_1 = * \rangle, \langle E_x, attr_2 = v_i \rangle, \langle E_x, attr_3 = * \rangle) \\ & (\langle E_y, attr_1 = * \rangle, \langle E_y, attr_2 = v_j \rangle) \\ & (\langle E_z, attr_1 = * \rangle, \langle E_z, attr_2 = * \rangle, \langle E_z, attr_3 = v_k \rangle) \end{aligned}$$

Now, assume that the applicable set of conditions that need to be checked for this pattern are:

$$\begin{aligned} c1 : & (\langle E_x, attr_1 \rangle, \langle E_y, attr_1 \rangle, IdentityCondition) \\ c2 : & (\langle E_x, attr_1 \rangle, \langle E_z, attr_1 \rangle, IdentityCondition) \\ c3 : & (\langle E_y, attr_1 \rangle, \langle E_z, attr_1 \rangle, IdentityCondition) \\ c4 : & (\langle E_x, attr_2 \rangle, \langle E_z, attr_3 \rangle, LessThanCondition) \end{aligned}$$

The possible combinations of conditions are  $2^{NoOfApplicableConditions} - 1$  where for the above example  $NoOfApplicableConditions = 4$ .

To reduce the number of combinations to check we apply the following heuristic which is based on the *a priori* property. Informally, *a priori* property states that for a combination of  $n$  conditions to be satisfied, any subset of those  $n$  conditions must also be satisfied. To exploit this property, at first, we check for single conditions and try to reduce the number of applicable conditions. For example, if  $c_1$  is not satisfied, we do not need to check any combination that includes  $c_1$ .

Next, we check for conditions in increasing length. For example, assume conditions  $c_2$ ,  $c_3$  and  $c_4$  are satisfied. We check which combinations of  $(c_2, c_3)$ ,  $(c_2, c_4)$ , and  $(c_3, c_4)$  are satisfied. If all of the length-2 combinations are satisfied, we check if  $(c_2, c_3, c_4)$  is satisfied or not.

**Symbolic Pattern Ranking.** The discriminative pattern extraction algorithm often returns patterns with same or very similar support. In the case of non symbolic patterns, there is no clear way to decide which patterns should be ranked as the more important ones. Fortunately, in the case of symbolic patterns, we have a convenient way to rank the patterns. We applied a simple scheme where we give more importance to patterns that are more specific. To do that, we simply count the number of “\*” in a symbolic pattern. The higher the number of “\*” in a pattern, the lower the rank, as it is likely to be a self-evident generality that does not carry much information. The rationale behind this is due to the fact that “\*” implies “don’t care” and hence patterns having more “\*” are more likely to have higher support but represent a weaker concept. In contrast, patterns with fewer “\*” give more information and should be ranked higher.

#### 4.4 Hybrid Support Count Function

One of the inherent problems with any discriminative pattern extraction algorithm is that the number of patterns generated as discriminant patterns is overwhelmingly large, which can be in the order of thousands. It makes it “easy” to miss the “culprit” pattern which may end up deep down the list of discriminative patterns. As at each stage  $i$ , the candidate set is generated by concatenating the frequent patterns generated at stage  $(i - 1)$  with each of the unique events in the log file, for 100 unique events (e.g. the alphabet equivalent of English language) in a log file, the number of candidate patterns of length 3 is 10,00,000 and so on. To avoid losing crucial events, we have to set the minimum support threshold to 1 (e.g., a single node reboot event may cause a large number of message losses and setting minimum support threshold larger than 1 will discard the “reboot” event). To address this challenge, in [9] we proposed a two stage approach which first identifies symptoms (e.g., message loss) with setting high minimum support threshold and later tries to identify the cause of failure (e.g., reboot) with lower support. Although this addresses the scalability issue to some extent, the patterns returned in this scheme still fail to return the “culprit” sequence at the top of the list if the cause of failure is infrequent (e.g., large number of message lost due to single node reboot).

The cause of the problem lies in the way support for an event is calculated in the frequent sequence mining algorithm in the data mining domain, which is ill-suited

for debugging purposes. The reason is that if we have  $N$  log files and an event  $X$  exists in only one file 1000 times and does not happen in any of the other files,  $X$  will still be considered a frequent event with support 1000. But for debugging purpose, this is “**wrong**”. As the reasoning behind using discriminative pattern extraction for debugging relies on the assumption that an event correlated with failure should “exist” in at least a majority of the “Bad” log files. Event  $X$  in fact violates this assumption and is not a frequent event from a debugging perspective.

To address this problem, we have implemented a support count function that counts the frequency of patterns not only within a single log file but also across multiple log files and uses both estimates to generate support for single attribute events. For example, according to our scheme, if an event  $X$  “exists” in only one of the  $N$  log files, the *across support* for  $X$  is 1 irrespective of how many times it happened in that single file.

For example, though “reboot” event has a lower support in a single file, it has a higher support across the files (“reboot” exists in all the file for cases that crashed). Using this observation, we discard events from the base set that have across support lower than a threshold  $\theta$  which is set by the user (i.e.,  $\theta = 0.6$  implies that for an event to be frequent, it has to “exist” in at least 60% of the files). We have two sets of frequent events (i.e., alphabet set), one for the set of good logs and one for the set of bad logs which are subsequently used to generate longer patterns. This reduces the execution time significantly and helps rank the patterns that are more correlated with failure higher than the other “less” correlated patterns.

#### 4.5 Collection of Logs

To use the tool, one must collect runtime logs from the application nodes. As long as the runtime logs follow the format specification required by the data analysis backend, the source of the logs does not matter. Logs can be collected from simulation, emulation or from real hardware. For example, if the user intends to use TOSSIM, user can log any event inside the application using TOSSIM’s “dbg” statement as in `dbg(“Channel”, “%d : %d : %d : %d...”, NodeId, EventId, attr1, attr2, ...)` as described in [8]. The user can also use the data collection front end designed for real hardware described in [9] to collect runtime logs from real deployment or choose to build his/her own data collection front end.

## 5 Evaluation

To evaluate the diagnostic capability using discriminative symbolic pattern analysis, we used TOSSIM in TinyOS 2.0 to simulate the nesC code where we used a synthesized bug to create the sample log files. We simulated a network of 25 nodes placed on a grid topology with 5 rows and 5 columns. For the simulated bug, we compare the generated symbolic patterns with the non symbolic patterns generated by the algorithm presented in [9]. We choose to compare our result with [9] as [9] is the most related to our work that uses discriminative patterns to diagnose bugs.

## 5.1 Synthesized Bug

In this section we give examples of a synthesized bug to illustrate the strength of symbolic discriminative pattern extraction for debugging.

- **Failure scenario-I: Out of order events, deterministic failure:** Let us assume that in a particular application, each neighbor of node A periodically communicates with node A and is always expected to send the messages of type 0, 1 and 2 in a fixed order where msgType 0 is followed by msgType 1 and msgType 2 respectively. Also assume that message reception in reverse order from a specific sender crashes the system. The discriminative pattern set returned by both algorithms from simulated logs for this failure scenario are given in table 3. The first discriminative symbolic pattern captured the bug perfectly where it expressed the fact that if a particular receiver(X1) receives from a particular sender(X2) messages in reverse order of msgType where msgType 2 is followed by msgType 1 and msgType 0 respectively, there is a problem. Not surprisingly, the algorithm borrowed from [9] generated completely misleading patterns with highest support. Though it returned the pattern “( $\langle \text{msgReceived} : (\text{msgType} : 2) \rangle, \langle \text{msgReceived} : (\text{msgType} : 1) \rangle, \langle \text{msgReceived} : (\text{msgType} : 0) \rangle$ )” at the very end of the list, it failed to identify the crucial condition that this sequence causes a problem only if the messages are received from the same sender.

**Table 3.** Top Patterns for Failure scenario - I

Patterns generated by [9]	
1.	$\langle \text{msgSent} : (\text{msgType} : 2) \rangle, \langle \text{msgSent} : (\text{MsgType} : 0) \rangle, \langle \text{msgSent} : (\text{SenderType} : 0) \rangle$
2.	$\langle \text{msgReceived} : (\text{msgType} : 1) \rangle, \langle \text{msgReceived} : (\text{senderType} : 0) \rangle, \langle \text{msgSent} : (\text{msgType} : 2) \rangle$
Patterns generated by Symbolic Pattern Extraction	
1.	$\langle \text{msgReceived} : (\text{ReceiverId} : X1), (\text{SenderId} : X2), (\text{SenderType} : *), (\text{MsgType} : 2) \rangle$ $\langle \text{msgReceived} : (\text{ReceiverId} : X1), (\text{SenderId} : X2), (\text{SenderType} : *), (\text{MsgType} : 1) \rangle$ $\langle \text{msgReceived} : (\text{ReceiverId} : X1), (\text{SenderId} : X2), (\text{SenderType} : *), (\text{MsgType} : 0) \rangle$

## 5.2 A Real Bug: Directed Diffusion Protocol Bug

We used the bug reported in [8] where a node experiences a large number of message losses after a node is rebooted in the directed diffusion protocol. For a detailed description of the bug, interested readers are encouraged to read [8]. Briefly, in the directed diffusion protocol, each node maintains an interest cache entry to keep track of which way to forward a data packet. There can be multiple paths from a single data source to the destination node. If there is no interest cache entry that matches a received packet’s interest description, the receiver node silently discards the packet assuming it is not on the forwarding path. The problem is if a node gets rebooted for some reason, it wipes out the interest cache completely and causes a large number of consecutive message losses. The problem manifests only if there is a single path from the source node to the destination node. This bug is particularly interesting because in [8] the reported discriminative patterns showed the

manifestation of the problem rather than showing that the “Reboot” event is the one that is actually causing the problem. For the log generated for this bug, the discriminative pattern set returned by the symbolic pattern extraction algorithm is given in table 4. Symbolic patterns identified the real cause of the problem and correctly correlated the cause of failure and the manifestation. It clearly shows that the “Boot” event is followed by the interest cache empty event and message is dropped due to no matching interest cache entry.

**Table 4.** Top Patterns for Directed Diffusion Protocol Bug

Patterns reported in [8]	
1.	$\langle \text{interestCacheEmpty} : \text{NodeId} : 3 \rangle,$ $\langle \text{dataCacheEmpty} : \text{NodeId} : 3 \rangle,$ $\langle \text{dataMsgSent} : \text{TimeStamp} : 20 \rangle$
2.	$\langle \text{interestCacheEmpty} : \text{NodeId} : 3 \rangle,$ $\langle \text{dataCacheEmpty} : \text{NodeId} : 3 \rangle,$ $\langle \text{dataMsgSent} : \text{NodeId} : 4 \rangle$
3.	$\langle \text{interestCacheEmpty} : \text{NodeId} : 3 \rangle,$ $\langle \text{dataCacheEmpty} : \text{NodeId} : 3 \rangle,$ $\langle \text{dataMsgSent} : \text{msgType} : 5 \rangle$
Patterns generated by Symbolic Pattern Extraction	
1.	$\langle \text{BOOT\_EVENT} : (\text{NodeId} : X1) \rangle,$ $\langle \text{interestCacheEmpty} : (\text{NodeId} : X1) \rangle,$ $\langle \text{dataCacheEmpty} : (\text{NodeId} : X1) \rangle$
2.	$\langle \text{BOOT\_EVENT} : (\text{NodeId} : X1) \rangle,$ $\langle \text{msgDropped} : (\text{NodeId} : X1), (\text{ReasonToDrop} : \text{dataWithNoMatchingInterest}), (\text{TimeStamp} : *) \rangle,$ $\langle \text{interestCacheEmpty} : (\text{NodeId} : X1) \rangle$
3.	$\langle \text{BOOT\_EVENT} : (\text{NodeId} : X1) \rangle,$ $\langle \text{msgDropped} : (\text{NodeId} : X1), (\text{ReasonToDrop} : \text{dataWithNoMatchingInterest}), (\text{TimeStamp} : *) \rangle,$ $\langle \text{dataCacheEmpty} : (\text{NodeId} : X1) \rangle$

### 5.3 Performance Comparison

For the log files collected for the directed diffusion protocol bug, we used three good logs and three bad logs and analyzed using symbolic pattern extraction algorithm to generate patterns. Using symbolic pattern extraction, it took less than 1 hour and returned 188 symbolic patterns of length 3. In comparison, when we applied algorithm from [8], it took more than 3 hours and returned over several thousand patterns. This is due to the fact that in our approach, we were able to discard many unimportant events that had low support across multiple log files and thus reduced the number of the base events that were used to generate longer patterns.

## 6 Conclusion

The concept of discriminative *symbolic* pattern extraction is introduced in this paper which is a new concept both in wireless sensor networks and data mining domains. We demonstrated the power of symbolic patterns using several bug scenarios. From a comparison of patterns reported by [8] and [9] with patterns

generated using discriminative *symbolic* pattern extraction, the strength of the *symbolic* approach for debugging purposes is clear. The new algorithm is better in terms of pattern expressiveness, concept generalization and discovery of hidden patterns, some of which are much harder to notice using traditional pattern mining algorithms which mine based on absolute attribute values rather than abstract symbols. Discriminative *symbolic* pattern extraction for debugging adds an invaluable technique to the arsenal of debugging techniques available in the wireless sensor networks domain.

**Acknowledgments.** This work was funded in part by NSF grants CNS 06-26342, CNS 05-53420, and CNS 05-54759.

## References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: Proceedings of the Twentieth International Conference on Very Large Data Bases (VLDB 1994), pp. 487–499 (1994)
2. Aguilera, M.K., Mogul, J.C., Wiener, J.L., Reynolds, P., Muthitacharoen, A.: Performance debugging for distributed systems of black boxes. In: Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP 2003), Bolton Landing, NY, USA, pp. 74–89 (2003)
3. Bodík, P., Friedman, G., Biewald, L., Levine, H., Candea, G., Patel, K., Tolle, G., Hui, J., Fox, A., Jordan, M.I., Patterson, D.: Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In: Proceedings of the 2nd International Conference on Autonomic Computing (ICAC 2005) (2005)
4. Cao, Q., Abdelzaher, T., Stankovic, J., Whitehouse, K., Luo, L.: Declarative tracepoints: A programmable and application independent debugging system for wireless sensor networks. In: Proceedings of the 6th ACM Conference on Embedded Networked Sensor Systems (SenSys), Raleigh, NC, USA (2008)
5. Ertin, E., Arora, A., Ramnath, R., Nesterenko, M.: Kansei: A testbed for sensing at scale. In: Proceedings of the 4th Symposium on Information Processing in Sensor Networks (IPSN/SPOTS track) (2006)
6. Fatta, G.D., Leue, S., Stegantova, E.: Discriminative pattern mining in software fault detection. In: Proceedings of the 3rd international workshop on Software quality assurance (SOQUA 2006), pp. 62–69 (2006)
7. Girod, L., Elson, J., Cerpa, A., Stathopoulos, T., Ramanathan, N., Estrin, D.: Emstar: a software environment for developing and deploying wireless sensor networks. In: Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC 2004), Boston, MA, p. 24 (2004)
8. Khan, M.M.H., Abdelzaher, T., Gupta, K.K.: Towards diagnostic simulation in sensor networks. In: Proceedings of International Conference on Distributed Computing in Sensor Systems (DCOSS), Greece (2008)
9. Khan, M.M.H., Le, H.K., Ahmadi, H., Abdelzaher, T.F., Han, J.: Dustminer: Troubleshooting interactive complexity bugs in sensor networks. In: Proceedings of the 6th ACM Conference on Embedded Networked Sensor Systems (SenSys), Raleigh, NC, USA (2008)

10. Khan, M.M.H., Luo, L., Huang, C., Abdelzaher, T.: Snts: Sensor network troubleshooting suite. In: Aspnes, J., Scheideler, C., Arora, A., Madden, S. (eds.) DCOSS 2007. LNCS, vol. 4549, pp. 142–157. Springer, Heidelberg (2007)
11. Levis, P., Lee, N., Welsh, M., Culler, D.: Tossim: accurate and scalable simulation of entire tinyos applications. In: Proceedings of the 1st international conference on Embedded networked sensor systems (SenSys 2003), Los Angeles, California, USA, pp. 126–137 (2003)
12. Liu, C., Fei, L., Yan, X., Han, J., Midkiff, S.P.: Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering*
13. Liu, C., Lian, Z., Han, J.: How bayesians debug. In: Proceedings of the Sixth International Conference on Data Mining (ICDM 2006) (December 2006)
14. Liu, C., Yan, X., Fei, L., Han, J., Midkiff, S.P.: Sober: statistical model-based bug localization. In: Proceedings of the 13th ACM SIGSOFT international symposium on Foundations of software engineering (FSE 13), Lisbon, Portugal (2005)
15. Liu, C., Yan, X., Han, J.: Mining control flow abnormality for logic error isolation. In: Proceedings of 2006 SIAM International Conference on Data Mining (SDM 2006), Bethesda, MD (April 2006)
16. Liu, K., Li, M., Liu, Y., Li, M., Guo, Z., Hong, F.: Pad: Passive diagnosis for wireless sensor networks. In: Proceedings of the 6th ACM Conference on Embedded Networked Sensor Systems (SenSys), Raleigh, NC, USA (2008)
17. Polley, J., Blazakis, D., McGee, J., Rusk, D., Baras, J.S.: Atemu: A fine-grained sensor network simulator. In: Proceedings of the First International Conference on Sensor and Ad Hoc Communications and Networks (SECON 2004), pp. 145–152, Santa Clara, CA (October 2004)
18. Ramanathan, N., Chang, K., Kapur, R., Girod, L., Kohler, E., Estrin, D.: Symmetry for the sensor network debugger. In: Proceedings of the 3rd international conference on Embedded networked sensor systems (SenSys 2005) (2005)
19. Szewczyk, R., Polastre, J., Mainwaring, A., Culler, D.: Lessons from a sensor network expedition. In: Karl, H., Wolisz, A., Willig, A. (eds.) EWSN 2004. LNCS, vol. 2920, pp. 307–322. Springer, Heidelberg (2004)
20. Tolle, G., Culler, D.: Design of an application-cooperative management system for wireless sensor networks. In: Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN 2005), Turkey, February 2005, pp. 121–132 (2005)
21. Wen, Y., Wolski, R., Moore, G.: Disens: scalable distributed sensor network simulation. In: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP 2007), San Jose, California, USA, pp. 24–34 (2007)
22. Werner-Allen, G., Swieskowski, P., Welsh, M.: Motelab: A wireless sensor network testbed. In: Proceedings of the Fourth International Conference on Information Processing in Sensor Networks (IPSN 2005), Special Track on Platform Tools and Design Methods for Network Embedded Sensors (SPOTS) (April 2005)
23. Whitehouse, K., Tolle, G., Taneja, J., Sharp, C., Kim, S., Jeong, J., Hui, J., Dutta, P., Culler, D.: Marionette: Using rpc for interactive development and debugging of wireless embedded networks. In: Proceedings of the Fifth International Conference on Information Processing in Sensor Networks: Special Track on Sensor Platform, Tools, and Design Methods for Network Embedded Systems (IPSN/SPOTS), Nashville, TN, pp. 416–423 (April 2006)
24. Yang, J., Soffa, M.L., Selavo, L., Whitehouse, K.: Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. In: Proceedings of the 5th international conference on Embedded networked sensor systems (SenSys 2007), pp. 189–203 (2007)