SENSID: A Situation Detector for Sensor Networks

MARK KRANZ
and
RACHEL CARDELL-OLIVER
School of Computer Science & Software Engineering and
Cooperative Research Centre for Plant-based Management of Dryland Salinity
The University of Western Australia

Sensor networks can be used to detect the occurrence of complex events patterns with spatial and temporal characteristics. Traditionally, sensor networks communicate time series of sensed data to a central database, where powerful computers can perform offline data mining. In this paper we investigate a novel approach for identifying complex event patterns in-situ. In-situ detection allows the network to react to events immediately at source, and saves energy that would be wasted in transmitting irrelevant data. Our approach, based on situation detection methods first developed for active databases, adapts situation detection to the unique constraints of sensor networks: the speed and memory constraints of individual nodes and the environmental constrains of sensors and radio communication. We have designed a middleware system, SENSID, for sensor network situation detection, and implemented a version of SENSID on Berkeley Mica2 motes using the TinyOS operating system. We evaluate the expressiveness, robustness and performance of SENSID and so demonstrate that in-network situation detection is feasible for sensor networks, with acceptable performance for a wide class of sensing applications.

Categories and Subject Descriptors: C2.1 [Computer-Communication Networks]: Network Architecture and Design; C.3 [Special-Purpose and Application-Based Systems]: Real-time and embedded systems; D1.1.7 [Programming Techniques]: Logic Programming

General Terms: Programming models and languages, In-network processing, Data storage and query processing, Situation Detection, Sensor Networks

Additional Key Words and Phrases: SENSID

1. INTRODUCTION

Sensor networks are a tool for observing real world environments. They can be used in many fields, including environmental monitoring [Wang et al. 2002; Cardell-Oliver et al. 2005], studying animal behaviour [Mainwaring et al. 2002; Sikka et al. 2004], controlling robots [Bergbreiter and Pister 2003], and improving building techniques [Glaser 2004].

Most existing sensor network applications regularly sample and deliver sensor readings and run off-line data mining on a central store [Hellerstein et al. 2003;
Cardell-Oliver et al. 2005; Mainwaring et al. 2002]. The primary limitation with these approaches is that the network does not have the opportunity to react to events immediately and in situ. In addition, there is substantial wastage in the energy and bandwidth of the network, as many unnecessary data samples are transmitted. On-line data aggregation [Madden et al. 2002], and event pattern matching [Römer and Mattern 2004; Li et al. 2004], have been proposed to address these problems. However, data aggregation approaches are usually specific to the type of sensor readings taken, and cannot detect complex patterns, nor react to them. Current event pattern matching models are limited by the type of relations that can be expressed. In particular, they cannot detect when events have not occurred during some period, or count the number of event occurrences during some period. Both approaches have a limited ability to express the temporal or spatial context during which events are of interest.

This paper describes SENSID (SEnsor Network SItuation Detector), a tool designed for in-network detection and reporting of complex temporal and spatial event patterns in sensor networks. SENSID is a middleware service. Each node in the network runs application code that gets sensor readings and sends and receives messages to and from other nodes in the network, as well as running the situation detection service. User applications specify the properties they are interested in by registering event patterns called situations with SENSID. These applications pass events to SENSID as they are observed. When SENSID detects a situation (the looked for pattern of events has occurred) then it notifies the application, which in turn can take appropriate action. Figure 1 shows how SENSID and user applications run together on sensor nodes and how they interact with the environment.

The main contributions of this paper are:

—An intentional language of situations for specifying and detecting complex event patterns; the situation language is richer than, for example, TinyDB SQL, or AMIT’s event operators in supporting both spatial and temporal constraints.

—An efficient implementation of the SENSID service for registering and detecting situations, suitable for running on resource constrained network nodes in a sensor network.
In this paper we describe how the SENSID system performs situation detection. The SENSID data model for events, situations and lifespans is described in Section 2. Two different approaches to situation evaluation are given in Section 3. Section 4 describes the process for managing situation detection at runtime. In Section 5 we evaluate the expressiveness, efficiency, and fault tolerance of SENSID and discuss its use in real applications. Section 6 compares SENSID with related work.

2. DATA MODEL

This section describes the main data types required for situation detection. In order to make the following discussion more concrete, we first outline a simple example of situation detection.

A sensor network for detecting explosions will contain sensors to monitor light, sound and temperature, with events triggered when sensed values exceed given thresholds. Figure 2 shows example traces for four different event types (y-axis) monitored by a sensor network node over time (x-axis). The events are light, heat and noise observed at two different locations: either node A or a neighbouring node B. An explosion is detected, if and only if all three events occur in the order: flash, noise, heat, with the constraints that the noise event must occur less than 3 seconds after the flash. Detecting heat events is somewhat unreliable, and so in order to avoid false positives, we require that at least 2 heat events are observed at different locations: node A and B. These heat events must occur within 4 seconds of the initial flash event.
The boxes in Figure 2 represent lifespans: a set of events that provide the context for detecting explosions. A lifespan is triggered by a start event, in this case a flash at node A, and it is terminated a fixed time, say 4 seconds, after that event. In this example fragment there are 4 lifespans, one for each of the triggering flash events. It can be seen that each lifespan determines a set of candidate events. A situation is a predicate on a lifespan set of events. In this example the two shaded lifespans satisfy the explosion predicate whilst the two unshaded lifespans do not. In lifespans 1 and 4 we see that an explosion situation has occurred: all four necessary events are present and the sequence and timing constraints on these events are satisfied.

We now outline the data types used in SENSID for situation detection. Figure 3 uses class diagram notation to summarise the attributes and operators of these data types.

Events are the foundation for situation detection. An event is characterised by a 4-tuple of event type, a data value, the time the event occurred, and the spatial location of the event. Users are responsible for identifying the types of events that are of interest for a given application. In general, events are abstractions of readings from external sensors (e.g. temperature or light values), internal sensor node state (e.g. battery levels), and packets received from other nodes in the network over the radio (e.g. beacon messages containing local readings transmitted by neighbouring nodes). Typically, a single event captures a special property of an observation: for example, whether an observed value is above (or below) some threshold value or the reception of a neighbouring node's data packet reporting an above threshold temperature reading. Additionally, events can capture properties of more than one observation, such as the amount of change in value between two consecutive sensor readings. In this way, our event abstraction allows us to include properties of continuous environment variables and observations with duration that are not instantaneous.

A situation is simply a predicate on a set of events. For a particular finite set of events, a situation is either true or false. We will return to the specification and evaluation of situation predicates in Section 3. We specify situations using a type signature, a lifespan and a predicate. The type signature of a situation is the set of event types that contribute to the occurrence of that situation. A LifespanSpec is used to characterise the set of events on which situation detection is performed.
Finally, a situation is characterised by a predicate on a set of events. The predicate defines whether the looked for pattern of events has occurred.

The sets of events on which a situation is evaluated are called *lifespans*. Lifespans occur in SENSID in several guises. First, a lifespan is a specific set of events on which a situation can be evaluated. We call this representation of a lifespan a *LifespanList*. Since each event is time-stamped, we shall view all lifespans as time-ordered *lists* of events. As shown in the example of Figure 2, the same event may contribute to more than one lifespan and similarly to more than one situation.

Second, lifespans are described intentionally as part of the specification of a situation. The type for these lifespan specifications is a *LifespanSpec*. A *LifespanSpec* defines the type of events that are allowed in the lifespan, the type of event that starts the lifespan, and by an upper time bound between the first and last event.

Finally, within the situation detection service, a particular set of events is extracted from a store of events using pointers specific to the datastore: times and event identifiers. This storage specific specification of a lifespan is called a *LifespanSelect*.

The association between the situations and lifespans is one to many. That is, each situation has 0 or more lifespans, while a specific lifespan belongs to exactly one situation. The association between lifespans and events is many to many since an event may take part in 0, 1 or more lifespans, and each lifespan is associated with a set of 0, 1 or more events.

Since events identify both the time and location of their occurrence, we can express both temporal and spatial situations. Situations over the temporal domain can be defined using the *time* field associated with each event for temporal comparisons such as \(e_1.time - e_2.time < 3 \text{ seconds}\). For each application, the user can decide on the most appropriate representation of time and the level of time synchronisation between nodes. For example, we have used both coarse grained (day and minute) and fine grained (millisecond) time stamps in different environmental monitoring applications. Situations with a spatial context are defined simply by allowing events to arrive from an external location, and stamping them with some form of location identifier (GPS co-ordinates, node identifiers etc). Predicates over the spatial domain are defined by overloading the operators and relations, such as:

\(-e_1.position \neq e_2.position\), compares node identifiers

\(-e_1.position \leq_{\text{numhops}} e_2.position\), compares the number of communication steps

\(-|e_1.position - e_2.position| < C\), uses vector arithmetic and Euclidean distance to find if \(e_1\) and \(e_2\) are physically near each other.

SENSID’s support for both spatial and temporal relationships between events allow us to write robust situation detection predicates involving multiple nodes. In our current system, a distributed situation is evaluated on a single node, which must receive data from all participating nodes. In future work, we plan to develop mechanisms for performing distributed situation detection, and merging the results. This will improve the efficiency and robustness of evaluating complex temporal and spatial situations.
3. SITUATION PREDICATE EVALUATION

The purpose of a situation detection service is to identify when complex temporal and spatial patterns of events have occurred. A situation predicate is a function that evaluates to either true or false on a specific lifespan. For a situation that is satisfied, we may identify one or more subsets of events from the lifespan that cause the situation to be satisfied. At its most general, any predicate on events can be a situation predicate. However, in order to implement situation detection on sensor network nodes, with limited memory and processing power, it is necessary to restrict the class of predicates that can be used.

Situation predicates can be specified in different ways. Here we examine two possibilities: predicates as user provided plug-ins to a framework and predicates given in a situation predicate specification language. Plug-ins provide a very general solution in which users have strong control over the efficiency and expressiveness of situation detection predicates. However, a disadvantage of this approach is that users are required to implement any new predicates, possibly using existing library predicates, and compile that code into the situation detection system. An alternative approach is to provide a situation detection specification language in which users can specify predicates. The situation detection service must then contain a general evaluation procedure for the language. In this case there is no need for the user to compile new code into the system, but there is necessarily some loss of control in the efficiency and expressiveness of predicates. The following subsections describe each of these approaches.

3.1 Situation Predicate Plug-ins

A situation predicate is specified by the operations test, satisfy or respond. Figure 3 shows the type of these operations. Each inputs a lifespan (list of events). The test operation returns only whether the situation predicate is true or not. The satisfy operation returns one or more subsets of events that cause the predicate to be satisfied. The respond operation returns an event that represents the occurrence of the situation. When a situation is not satisfied then these operations return false, an empty list, or a null event, respectively. SENSID offers a library of implemented situation predicates that can be evaluated efficiently. Users may also to add their own code for situation predicates to the SENSID library, using existing library predicates in combination with new predicates.

Since efficiency is of the highest priority when performing situation detection on sensor nodes, we distinguish between predicates with linear complexity for the number of events in a lifespan, and non-linear predicates.

3.1.1 Linear Situation Predicates. An important class of situation predicates are those that can be evaluated in a single pass through a lifespan set of events. For example, the predicate \( \text{atmost}(n, T) \) is true if and only if its lifespan contains at most \( n \) events of type \( T \). This predicate can be evaluated by a single pass through a lifespan event list \( E \) as follows.
Table I gives a number of linear predicates useful for situation detection in sensor networks. They use counting, the absence or presence of events, and constraints on minimum, maximum or averages of the lifespan list. Users can define compound situation predicates using any propositional combination of these linear predicates, without affecting the complexity of situation detection. The complexity of all these predicates is linear on the size of their lifespan event set.

3.1.2 Backtracking Predicates. Linear situation predicates are efficient to implement, but are not sufficient for expressing situations that compare the values of one or more events. These non-linear predicates generally rely on backtracking for evaluation on a given lifespan. For example, to find two distinct events with the same type, T, and the same value, that occur at least 10 time units apart, we use:

```
count=0;
    for i=0 to E.length()
        if (E(i).type==T)
            then count++;
        end if
    end for
return (count<=n);
```
II: Examples of Non-Linear Situation Predicates

<table>
<thead>
<tr>
<th>MPredicate</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_1.time - e_2.time \leq 2 \land$</td>
<td>Explosion</td>
</tr>
<tr>
<td>$e_3.time - e_2.time \geq 3 \land$</td>
<td></td>
</tr>
<tr>
<td>$e_1.type = \text{flash} \land$</td>
<td></td>
</tr>
<tr>
<td>$e_1.type = \text{noise} \land$</td>
<td></td>
</tr>
<tr>
<td>$e_1.type = \text{heat}$</td>
<td></td>
</tr>
<tr>
<td>$e_1.value &lt; e_3.value \land$</td>
<td></td>
</tr>
<tr>
<td>$e_3.value &lt; e_2.value \land$</td>
<td>Peak in values at point $e_3$</td>
</tr>
<tr>
<td>$e_3.time - e_1.time &lt; \text{Peak interval} \land$</td>
<td></td>
</tr>
<tr>
<td>$e_1.time &lt; e_2.time \land$</td>
<td></td>
</tr>
<tr>
<td>$e_2.time &lt; e_3.time$</td>
<td>Spatial contour</td>
</tr>
<tr>
<td>$e_1.position - e_2.position &lt; \text{diameter} \land$</td>
<td></td>
</tr>
<tr>
<td>$e_1.value - e_2.value &gt; \text{High value} \land$</td>
<td></td>
</tr>
<tr>
<td>$e_1.time - e_2.time &lt; \text{Max time}$</td>
<td></td>
</tr>
</tbody>
</table>

The complexity of evaluating this situation is $n^2$ for a lifespan containing $n$ events. In general, predicates that compare $d$ events have worst case evaluation complexity of order $n^d$. SENSID users are responsible for checking that evaluation of complex predicates is feasible for the expected size and composition of lifespan event sets. Fortunately, there are a number of techniques that can be used to ensure efficient evaluation of even complex predicates and these are discussed in Section 5.

Table II lists a number of useful non-linear situation predicates. Users can define compound situation predicates using combinations of both linear and non-linear predicates.

3.2 Situation Predicate Specification Languages

The plug-in predicates defined above allow users to write and compile any evaluation predicate into the SENSID system. An alternative approach is to provide a language for the user to express situations, and to provide a general purpose evaluation engine within the SENSID system.
In [Kranz 2005] a language is presented for expressing situations in conjunctive normal form, \( C_1 \land C_2 \land C_k \). Each \( C_i \) is an event constraint of the form \( e.f \oplus C \) or \( e_1.f - e_2.f \oplus K \) for events \( e, e_1, e_2 \), comparison operator \( \oplus \), value constants \( K \) and event attributes \( f \): the type, value, time or location of the event. Further details of this language and its implementation can be found in [Kranz 2005]. More generally, we are currently investigating a bi-modal space-time logic for expressing situations together with an evaluation mechanism for this logic [Cardell-Oliver et al. 2007].

4. SITUATION DETECTION PROCESS

As shown in Figure 4, we assume that each node in a sensor network utilises application protocols for gathering events and registering situations of interest, together with a situation detection service for evaluating the occurrence of situations. Application protocols collect events, both from the local node and received from other nodes in the network. These events are forwarded to a SENSID situation detection service. In this section we describe the process of situation detection as performed by SENSID. There are many protocols and techniques available for gathering sensor network events, but this topic is beyond the scope of this paper.

In order to manage situation detection, SENSID maintains two important data structures:

--- SituationRegister: a list of situations that have been registered by applications, each with a list of its currently active lifespans, and

--- EventStore: a database of events that have been forwarded to SENSID by applications.

Applications interact with these data structures using the operations shown in Figure 4. The process of situation detection can be summarised by the following scenario:

1. An application registers the situations it is interested in with SENSID.
2. SENSID waits for the next event to be forwarded by the application.
3. When SENSID receives an event relevant to one or more registered situations, then that event is added to the event store. A new lifespan is opened whenever the start event of a registered situation is received.
4. For any open lifespan whose end time has arrived, the relevant list of events is extracted from the event store and the situation predicate is evaluated. The application is notified whenever a situation predicate is satisfied. SENSID then returns to step 2.

4.1 Situation Registration

An application signals its interest in the detection of particular situations by registering them with SENSID. A new situation is added to a registry, SituationRegister, and its type signature and initial event type are registered with the event dispatcher component. Initially, the list of lifespans (each a LifespanSelect) for a new situation is empty. Applications may also remove situations from the register, for example in order to replace them with a situation that is more relevant to the current context for an active sensor network.
4.2 Event Dispatch and New Lifespans
The event dispatcher component receives all incoming events from user applications, and determines when to add events to the EventStore using StoreEvent which returns a storage identifier id. If the event type is either in the type signature of a situation with open lifespans, or is the start event for a registered situation, then the event is added to the store. In the latter case, a new lifespan is opened for the registered situation(s) using AddLifespanSelect(s, id, e.time, e.time+s.lifespan.timespan) where the third and fourth parameters give the start and end times of the lifespan respectively.

4.3 Situation Detection
Situation detection is triggered as soon as the current time exceeds the end time of an open lifespan. Optionally, we may also allow applications to specify a trigger event type, which triggers the detection during a lifespan, or a termination event, which ends the lifespan instead of waiting until endtime.

The situation detector first gathers the list of events relevant to the expired lifespan from the event store using the operator GetLifespanList. This operator returns a temporally ordered list of events, E, satisfying the type signature of the situation, and the time constraints of the lifespan specification. The situation detector can now check the situation predicate s.predicate.respond(E) returning a notification event if the situation has been detected. Alternatively, situation detector may just return a boolean signal for each detection attempt (s.predicate.test(E)) or may return one or more subsets of events that cause the situation predicate to be satisfied

(s.predicate.satisfy(E)).

The notification events may also be dispatched in the same manner as an external event, allowing nested situations (situations that are dependent on the occurrence of other situations) to be specified and detected.

4.4 Situation Notification

The occurrence of a situation is signalled to the event dispatcher, which in turn signals all user applications that have subscribed to the situation that the situation has occurred. If required, the application also receives either a new event representing the occurrence of the situation, or one or more lists of events, each being one way of satisfying the situation from the given candidate event set. The performance of event detection can be improved by searching only for the first subset of events that satisfies a situation, when all possible satisfying sets are not required.

4.5 Closing Lifespans and Maintaining the Store

After situation detection is completed for a given lifespan, that lifespan select record is removed from the situation registry.

From time to time the event store is purged of events no longer required by any open lifespan. Any event types that are not in the type signatures of registered situations with one or more open lifespans can be removed. Also, all events with time stamps earlier than the earliest start time of any open lifespan can be removed. Event purging can take place periodically, or whenever a lifespan is closed. In practice, very simple storage management strategies can be sufficient. For example, we can simply maintain a large circular buffer as event store, in which old events are eventually overwritten by new ones.

5. EVALUATION

In this section we will discuss a sample implementation of the situation detector (SENSID), and evaluate its performance, fault-tolerance and expressiveness.

Our sample implementation is a proof-of-concept library, written in NesC and run under TinyOS on mica2 motes. This version does not include the spatial operators, and uses a simple representation for storing events and lifespan state. Situation predicates use a simplified notation, while the detection algorithm uses a brute-force/backtracking approach.

The core SENSID library requires 4KB of the AVR Atmega’s code memory, plus additional overhead for the application logic to load situation definitions. While a full-featured implementation, with detection optimisations would require additional space, this is not typically be a problem, as most applications fit well within the 128KB code space of mica2 nodes.

A greater concern is the runtime memory usage of the library, which can be a major limiting factor in wireless sensor network development [Gu and Stankovic 2006]. The performance of the detection algorithm is also a concern, since multi-dimensional predicates are expensive to evaluate, especially for long-running lifespans with large event search spaces.

We investigate the performance of SENSID using situations for a sample data-set of battery, light, temperature, and humidity readings. Our sample data is taken from 4 day trial, with a fixed 10 minute sample rate and with 16 sensor nodes. We
will focus on the data collected from a single node for the purpose of evaluating these temporal (non-spatial) examples. There are characteristics of this dataset that will test the robustness of situation detection. For example, time synchronisation and communication difficulties have resulted in duplicates and missing data, and so in irregularly spaced samples. Noise is also present in the data, as a result of low quality sensors and environmental effects, resulting in sudden spikes or drops against the general trend.

If left unmanaged, these data imperfections may produce false situation detections — either false positives, when a situation is detected that did not occur in reality, or false negatives, when a real situation is not picked up by the detection. We endeavour to address these problems in our examples.

### 5.1 Example 1: Detecting significant change

This situation reflects a common challenge in sensor networks — how often shall data be communicated? Since sending every data sample can have a large impact on a wireless sensor network lifetime, it may be preferable to only communicate significant changes in sample readings. These changes are usually field-specific, occur over some period of time, and may be subject to data noise or missing readings.

#### 5.1.1 Specification and Detection

Consider the problem of looking for significant decreases in the battery levels of a sensor node. In general, the device will lose battery power gradually, or gain power through recharging (e.g., solar panels). We wish to report when there is a significant drop over a short period, possibly indicating a problem with the solar panels.

**Naive Approach.** A simple solution for identifying this situation is to look for three consecutive, non-decreasing battery readings, where the total change is greater than some threshold (Table III).

This approach does not attempt to deal with the imperfections in the dataset, such as noise or missing data. However, it may be implemented easily and efficiently, and without a general purpose situation detection library such as SENSID.

**Robust Approach.** A more robust, situation-oriented approach is to divide up time periods with lifespans. Each lifespan starts on a new reading, and ends on a fixed time limit. The situation has occurred if the total voltage decrease is above some threshold, and that there were not too many ‘noisy’ readings (Table IV).

#### 5.1.2 Expressiveness

The Naive approach is very efficient — it only needs to store 3 events at a time, and perform a simple comparison. However, the second approach is a demonstration of how the use of timed lifespans and event constraints (predicates) can capture a better sense of the situation, and offer better fault-tolerance.

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Battery events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lifespan-start</td>
<td>all Battery events</td>
</tr>
<tr>
<td>Timespan</td>
<td>3 events</td>
</tr>
<tr>
<td>Predicate</td>
<td>((e_1 &gt; e_2 &gt; e_3) \land (e_1 - e_3 &gt; \text{threshold}))</td>
</tr>
</tbody>
</table>

III: Detecting Battery voltage drops: Naive Approach
5.1.3 *Fault Tolerance.* The Naive solution will detect significant changes of a perfect dataset, but may fail when the data is flawed. With the Naive specification, noisy data may cause false-negatives in the situation detection. In addition, only evaluating three points does not give us statistically significant situations — however increasing the number of points will increase the likelihood of a false-negative due to noise. Also, there is an implicit assumption that the data-points are evenly spaced, and that there is one reading at each time interval. If there were missing or extra data-points, there could potentially be false-positives, as the detection would be performed on points over the incorrect time period.

The second specification is much better at handling these errors. Noise in the dataset is handled by allowing a number of readings to go against the trend, provided that the overall change is significant. The number of noisy readings may be tuned to the size of the search space — so if we are evaluating over an extended period, we may allow for a greater number of bad readings, and still have a similar confidence in the situation’s reliability. Also, by using the timespan rather than an absolute ordering, we ensure that missing or extra data will not affect the evaluation, as each comparison is always made over the same time interval.

Figure 5 shows how the second approach detects a significant decrease in the bat-
tery voltage of a sensor node, where $threshold_{\text{value}} = 150\, \text{mV}$ and $threshold_{\text{noise}} = 1$. Note that only one situation is detected — spurious voltage drops have not produced false positives. Also note that the situation is robust to noise — a single reading against the downward trend does not cause a false-negative.

The robust specification captures the essential meaning of what we are looking for — a significant decrease in voltage readings, indicating the occurrence of a real situation, for example shadows across a solar panel.

5.1.4 Performance. Situation Detection will typically be applied to monitoring sensors, which have a fixed sampling rate. Since we also use fixed timespans for evaluating each situation, we can calculate the storage requirements directly from the specification.

In our sample dataset, each node takes readings at 10 minute intervals, meaning that a 1 hour timespan will require storage for 6 events and 6 lifespans. The second algorithm will need to evaluate up to 6 events, every 10 minutes. In our sample system, this requires an average of 120 bytes for runtime state, and approximately 1 millisecond of computation time (on a mica2 node). The first approach uses a similar amount of memory and detection runs slightly faster, as it need only compare ordinality instead of counting the number backward events. Further details and benchmarks are covered in [Kranz 2005]

We are therefore presented with a trade-off — while the second approach is more fault-tolerant, it also requires more computation to evaluate. However, in this simple example, with fixed rate sampling and relatively short lifespans, there is minimal impact on the performance of the system.

This example demonstrates the flexibility of the SENSID situation detector. Whether or not the problem contains large search space, or whether it has powerful nodes, the situation can be specified according to demands of the application. We could choose to add additional information to further improve the detection reliability and performance — such as using light readings to monitor battery voltage during daylight, or closing lifespans as soon as they have become unresolvable. We could also remove events from store as soon as they contribute to a situation, to prevent repeated overlapping situations.

5.2 Example 2: Detecting composite situations

Another typical sensor network situation is looking for temporal patterns of different event types, such as our explosion example. To explore this type of situation, we used a dataset of light, temperature and humidity in order to find occurrences of ‘sunrise’. The point of time where the sunrise is detected by a sensor node is characterised by events such as the light levels increasing above a threshold, a significant increase in temperature, and a significant decrease in humidity. Finding a pattern of all three events gives us strong confidence that sunrise has occurred.

This example demonstrates the use of nested situations through the use of ‘inner’ events. This is when we declare the events participating in a situation to originate from detected situations rather than from external sources. Using inner events allows us to compose complex situations from several simple ones.

5.2.1 Specification and Detection. We use the robust approach for the significant change situation to find the rises in temperature (Table VI), and drops in humidity
### Event Type: Light threshold

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Light threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raw event produced by TSR (total solar radiation) sensor driver, triggers when light readings go above zero.</td>
<td></td>
</tr>
</tbody>
</table>

V: Light threshold event

### Event Type: Temperature

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Temperature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lifespan-start</td>
<td>all Temperature events</td>
</tr>
<tr>
<td>Timespan</td>
<td>60 minutes</td>
</tr>
<tr>
<td>Predicate</td>
<td>$e_n.value - e_1.value \geq 100mV \land</td>
</tr>
</tbody>
</table>

VI: Temperature increase situation

### Event Type: Humidity

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Humidity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lifespan-start</td>
<td>all Humidity events</td>
</tr>
<tr>
<td>Timespan</td>
<td>60 minutes</td>
</tr>
<tr>
<td>Predicate</td>
<td>$e_1.value - e_n.value \geq 50mV \land</td>
</tr>
</tbody>
</table>

VII: Humidity decrease situation

(Table VII). Lifespans start when light levels cross some threshold (Table V), and run for 120 minutes. The situation has occurred if a temperature rise occurs within 60 minutes of the light, and a humidity drop occurs at least 30 minutes after the temperature rise (Table VIII).

Figure 6 shows the results of evaluating this situation over two days of the dataset. The upper and middle graphs show the humidity and temperature traces respectively. Each graph annotation represents a detected inner-situation (either humidity decreases or temperature increases). The lower graph shows the light traces, with the black annotations marking the light crosses the threshold (lifespan initialisations), and the red floating marks indicating a successfully detected ‘sunrise’ situation. Notice that despite some noise in readings, the sunrises on both days are successfully detected, with no false positives or negatives.

5.2.2 **Expressiveness.** The sunrise situation demonstrates how complex situations can be built up from nested, simple situations. It would be difficult to combine the heterogeneous timing predicates and homogeneous change predicates into one situation, and would prove challenging to implement efficiently. The nested notation gives us a neat abstraction, which makes situation definitions easier to define, easier to read, and gives the implementation a logical way to break down the situation into detection-efficient constructs.

5.2.3 **Fault Tolerance.** Using multiple sensors adds a degree of redundancy to help remove false-positives. For example, flashes of light during the night, or temporary changes in temperature or humidity will not trigger the situation, since all events must occur in the correct ordering.

Dealing with false-negatives due to missing data is a more difficult problem. Solutions using spatial redundancy through sharing neighbour data is a topic that
6: Detecting the occurrence of sunrises through changes in light, temperature, and humidity
Event Types

d| Light-threshold, Temperature-increase, Humidity-decrease

Lifespan-start

120 minutes

Timespan

Predicate

\[ \exists e_i, e_j, e_k : e_i.type = \text{daylight} \land e_j.type = \text{sit.tempIncrease} \land e_k.type = \text{sit.humidDecrease} \land (e_j.time - e_i.time < 60 \text{ mins}) \land (e_k.time - e_j.time > 30 \text{ mins}) \]

VIII: Sunrise Situation

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Units</th>
<th>Total (2 days)</th>
<th>Storage Period</th>
<th>Max (/period)</th>
<th>Average (/period)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Light</td>
<td>event</td>
<td>264</td>
<td>1 hr</td>
<td>8</td>
<td>6.1</td>
</tr>
<tr>
<td>Temperature</td>
<td>event</td>
<td>264</td>
<td>1 hr</td>
<td>7</td>
<td>5.9</td>
</tr>
<tr>
<td>Humidity</td>
<td>event</td>
<td>264</td>
<td>1 hr</td>
<td>8</td>
<td>6.1</td>
</tr>
<tr>
<td>Light-Threshold</td>
<td>inner-event</td>
<td>81</td>
<td>2 hrs</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>Temperature-Rise</td>
<td>inner-event</td>
<td>73</td>
<td>2 hrs</td>
<td>10</td>
<td>1.9</td>
</tr>
<tr>
<td>Humidity-Drop</td>
<td>inner-event</td>
<td>70</td>
<td>2 hrs</td>
<td>7</td>
<td>5.3</td>
</tr>
<tr>
<td>Sunrise</td>
<td>situation</td>
<td>2</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

IX: Summary of situation detection metrics over a 2 day trial

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Events</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detection time</td>
<td>12 temp-raw/humid-raw</td>
<td>16 ms</td>
</tr>
<tr>
<td>Detection time (nesting,max)</td>
<td>6 temp-rise, 6 humid-rise</td>
<td>4 ms</td>
</tr>
<tr>
<td>Detection time (nesting,avg)</td>
<td>2 temp-rise, 5 humid-rise</td>
<td>1 ms</td>
</tr>
<tr>
<td>Storage size</td>
<td>12 temp-raw/humid-raw/light</td>
<td>720 bytes</td>
</tr>
<tr>
<td>Storage size (nesting,max)</td>
<td>6 temp-rise/humid-rise, 12 light</td>
<td>480 bytes</td>
</tr>
<tr>
<td>Storage size (nesting,avg)</td>
<td>2 temp-rise, 5 humid-rise, 7 light</td>
<td>210 bytes</td>
</tr>
</tbody>
</table>

X: SENSID Performance Benchmarks for mica2 platform

will be explored in future work.

5.2.4 Performance. The SENSID system uses a simple brute-force backtracking method for its detection algorithm. In the sunrise situation, it firstly looks for a temperature rise event, then tries to find a humidity drop event, where each event matches the timing constraints. If no humidity event produces a match, a different temperature rise event is selected. This type of implementation has a worst case performance of \(O(n_{\text{temp}} \times n_{\text{humid}})\), or more generally, \(O(n^d)\), where \(d\) is the number of dimensions, and \(n\) is the average number of events per dimension.

The speed and memory requirements of the significant change situations are the same as in the previous example where there is an average of 6 light and temperature readings for every successfully detected inner-situation. Occasional duplicates and missing readings occur, but on average each node produces one reading every 10 minutes.

For the composite situation, the worst case scenario involves traversing all orderings of temperature and humidity sub-situations without finding a match. For this node, the longest search involved searching 6 temperature rise events and 6 humidity, which gives a worst case search of 36 event combinations to traverse. On a mica2 running SENSID, this requires less than 4 milliseconds to execute. However,
in most lifespans there are fewer temperature and humidity events, giving this node an average detection time of only 1 millisecond.

We are also able to calculate the worst case memory requirements for this configuration. Assuming we keep all raw events, we have a storage upper bound of 36 events and 36 lifespans, or approximately 720 bytes in the current implementation.

The use of nested situations can improve performance, by allowing raw events to be abstracted to fewer, higher level events. Removing the raw events reduces the storage upper bound from 720 bytes, to 480 bytes, a saving of 5% of the mica2’s memory. We also reduce the number of comparisons that need to be made in the final detection stage, at the expense of periodically performing situation detection for the inner situations. This provides an overall improvement in the latency between observation and notification, which is a worthwhile tradeoff.

A more sophisticated implementation may be capable evaluating the situations on the fly, and be more aggressive at discarding events and minimizing search times. However, even with the simple SENSID implementation, this method achieves reasonable performance and has demonstrated the feasibility of running situation detection on a mica2 node.

6. RELATED WORK

The majority of existing methods for programming sensor networks are extensional: they state how network nodes behave, rather than what the network is required to do. Widely used extensional programming environments include TinyOS/NesC [Hill et al. 2000] and the Mate virtual machine [Levis et al. 2004]. Some more recent extensional approaches provide powerful abstractions for programming including abstract regions [Welsh and Mainland 2004], abstract channels [Sen and Cardell-Oliver 2006a], and global macroprogramming [Gummadi et al. 2005].

SENSID, by contrast, is an intentional approach: it states what the user requires rather than how the network should discover that information. SENSID is a middleware implementation of temporal situation detection designed specifically for sensor networks [Kranz 2005]. There have been a few other intentional programming methods for sensor networks, most notably TinyDB [Hellerstein et al. 2003], which like SENSID provides middleware for sensor network programming. To interact with the sensor network running TinyDB, users write queries in a variant of the traditional SQL query language. Queries are compiled into tasks and injected into the network where the task gathers and returns data to the user. For example, a user can request the average value of all temperature readings above a threshold value from all nodes in a certain area.

The query language of SQL has limited support for both temporal and spatial properties in databases. An alternative and richer approach is provided by the active database model of situations [Adi and Etzion 2004; Li et al. 2004]. AMIT situations are patterns on a set of events that can express properties such as the temporal ordering of events, the absence of an event or the number of times it occurs within an interval of time. The specification and implementation of situations in SENSID differs from that of AMIT because the exceptional memory and processing constraints of sensor network nodes make space and time efficiency a critical design goal for SENSID, but not for AMIT. SENSID’s situation detection language has
features specifically for sensor networks, such as event locations, and a choice of situation detection algorithms.

Space-time logic is a bi-modal requirements logic based on timed propositional temporal logic and a one-hop spatial modal logic [Cardell-Oliver et al. 2007]. This highly expressive requirements logic can express both spatial as well as temporal patterns on sets of events. The problem of designing and implementing efficient algorithms for compiling space-time logic into tasks to be executed in the network is the subject of ongoing research.

Regiment is another intentional approach to sensor network programming [Newton and Welsh 2004]. Regiment is based on the functional programming paradigm. Where SENSID views its environment as sets of events on which logical predicates can be tested, Regiment models the same environment as region streams: spatially distributed, time-varying collections of node state. Regiment’s operators for region streams include mapping, folding and filtering. Both SENSID situation formulae and Regiment functional expressions are macroprogramming approaches, in that they describe operations on the whole sensor network environment. Regiment functional expressions are compiled into efficient node programs based on token machines. SENSID situations are evaluated on individual nodes by a logic program that processes incoming events and searches for satisfied situations. The current scope of SENSID does not include the protocols for gathering events from different locations to a single place for evaluation, whilst Regiment does include such protocols. This functionality could be added to SENSID using, for example, abstract regions [Welsh and Mainland 2004] or abstract channels [Sen and Cardell-Oliver 2006b], and is the subject of ongoing work.

7. CONCLUSIONS AND FUTURE WORK

We have presented a novel approach to the problem of event-detection in sensor networks, using situations to capture spatial and temporal event patterns. Based on concepts explored in active-databases, a situation definition features a richer language than current sensor network approaches, incorporating concepts such as contexts of relevance (lifespans), relation operators (such as sequencing, counting, absence, averaging), and temporal and spatial comparisons. These features make situations uniquely suited to specifying complex real world scenarios that cannot be expressed easily with other sensor network languages. We demonstrate that situations are highly expressive and also flexible, allowing the user to balance robust specification against efficient implementation.

SENSID extends the functionality of AMIT in that both spatial and temporal attributes of events are handled, a richer range of situation predicates such as averaging are provided, and any logical combination of predicates is supported rather than only conjunction. SENSID also restricts some features of AMIT in order to be runnable on sensor nodes. We have a demonstrated a lightweight implementation based on the original AMIT system, restricted to features that are suitable for sensor network applications. For example, rather than multiple (named) fields, we limit event instances to one field, since sensors usually record data of a single type. This, in addition to the careful selection of memory-conserving data structures, has enabled us to substantially reduce the footprint and run-time.
memory requirements of SENSID in comparison to AMIT. Our prototype is capable of storing complex situation definitions in less than 100 bytes of memory, and records state information such as event instances and lifespans in only 10 bytes each. The system is therefore able to handle real-world sensor network problem domains within the tight confines of the 4 kilobyte memory of a Mica2 mote.

We describe a situation detection algorithm that runs on single nodes, albeit using spatial information from other nodes. Ideally, the evaluation of a situation should be distributed amongst nodes. Developing algorithms for doing this is a subject of our ongoing work.

We are also investigating the use of spatial-temporal logics to specify situations [Cardell-Oliver et al. 2007]. Using a logic, rather than the application programmer level approach presented in this paper, will allow us to prove properties of situation requirements such as logical equivalence of situations that have different implementation efficiency. Spatial-temporal logic also provides several avenues for extending the expressiveness of situations, whilst maintaining the property of efficient implementation.

REFERENCES


Received May 2007;