

Robust Data and Event Services in Real-Time Embedded Sensor Network Systems*

Krasimira Kapitanova
University of Virginia, USA
krasi@virginia.edu

Sang H. Son
University of Virginia, USA
son@virginia.edu

Seog Park
Sogang University, Korea
spark@dblab.sogang.ac.kr

Abstract

The majority of event detection in real-time embedded sensor network systems is based on data fusion that uses noisy sensor data collected from complicated real-world environments. Current research has produced several excellent low-level mechanisms to collect sensor data and perform aggregation. However, solutions that enable these systems to provide real-time data processing using readings from heterogeneous sensors and subsequently detect complex events of interest in real-time fashion need further research. We are developing real-time event detection approaches which allow light-weight data fusion and do not require significant computing resources. Underlying the event detection framework is a collection of real-time monitoring and fusion mechanisms that are invoked upon the arrival of sensor data. The combination of these mechanisms and the framework has the potential to significantly improve the timeliness and reduce the resource requirements of embedded sensor networks.

1. Introduction

With the continued miniaturization and growing computation power of wireless sensors, the deployment of real-time embedded systems using such devices has significantly increased. These systems use sensors to monitor the physical world and provide appropriate reaction and control over it. The scale of such interactions is very wide, ranging from resource-constrained sensor devices to global-scale networked monitoring systems, and these systems have the potential to transform the way people interact with and control the physical world.

Real-time embedded sensor network systems are used for variety of applications such as infrastructure monitoring, medical systems and smart healthcare facilities, surveillance applications, and environmental monitoring and control. Regardless of the specific application, these systems should be able to detect when particular events of interest occur. In embedded sensor networks, most events are not binary. Instead, they are

based on sensor data fusion using noisy sensors deployed in complicated real-world environments. Several excellent low-level mechanisms and protocols to collect, transport, and perform data aggregation on the raw sensor data have been developed. However, systematic solutions that allow these sensor network systems to provide real-time data processing using data fusion from heterogeneous sensors and subsequently detect complex events are still lacking. Recent work, [8], addresses confident event detection in sensor networks. The proposed detection mechanism is centralized and nodes are trained offline to recognize specific events. However, such training is often not feasible, e.g. when the detected events present danger (explosions or fires), or cannot be easily reproduced (earthquakes, volcano eruptions).

There are a number of requirements that an event detection service for embedded sensor networks must satisfy. It has to support event specification, real-time data fusion, and real-time stream data management. Since sensor devices typically have limited resources, this service should be light-weight. It must also provide high confidence event detection and minimize false alarms. All these features make building robust event services for embedded sensor network systems very challenging.

The main contribution of our work is that it provides key building blocks for robust real-time data and event services to be used by event detection applications. We have designed novel approaches that allow light-weight data fusion for real-time event detection and do not require significant computing resources. We are developing an event detection framework built around a collection of event specification, transformation, real-time monitoring, and fusion mechanisms. We expect that this framework together with the underlying mechanisms will significantly improve the accuracy and timeliness of event detection, as well as help reduce the resource requirements of embedded sensor network applications.

2. Complex event detection

2.1 Event description

Petri nets are well accepted as a model to describe systems with distributed, concurrent, asynchronous, and

*This work was supported by KOSEF WCU Project R33-2009-000-10110-0

non-deterministic features [1]. A basic Petri net consists of places (circles), transitions (rectangles or bars), directed arcs, and tokens (dots inside places). Arcs represent changes between states and the way in which tokens are created or destroyed. Places represent the states in which the application can be, and transitions are used to model various kinds of actions. Each place can contain zero or more number of tokens. A transition of a Petri net can fire whenever there is a suitable token in each of the input places for that transition. When the transition fires, it consumes these tokens, and injects tokens in its output places. A marking of a Petri net represents the status of the Petri net, i.e. a specific distribution of the tokens.

One of our objectives is to develop an effective specification language for event detection based on Petri nets. This language should address features specific to applications relying on the analysis of data from streaming networks (sensors, video cameras, etc). The foundation of our work is a compact Event Description and Analysis Language (MEDAL) [2]. As a formal method, MEDAL is based on Petri nets, which allows it to rigorously and unambiguously specify complex events. MEDAL attempts to address key aspects of event detection networks such as temporal control, spatial constraints, heterogeneity, and probability issues.

The MEDAL description of a sensor network event system can be given as a 7-tuple structure: $F = (P, T, A, \lambda, \beta, H, L)$, where P is the set of places, T is the set of transitions, A is the set of arcs, λ is a probability/weight function for the arcs, β is a temporal guard function, H is a threshold function for places, and L is a spatial guard function for transitions.

Figure 1 shows the MEDAL model of a complex event detection application. The token at *Temperature event 1* represents the detection of temperature value v at time t by a sensor at location (x, y) with sensing range r . The event detected by this application, event E , is characterized by specific simple events in temperature (e.g., detection of high temperature at a specific location), pressure (e.g., detection of potentially dangerous levels of pressure), and friction (e.g., occurrence of high friction at the joint point). The occurrence of each of these simple events is represented by a token in the corresponding places. When all three tokens are present, transition $T4$ fires and the application reports the detection of the complex event E . Complex sensor network events often require processing of stream data. For those events, tokens can be generated using the query results from stream data processing, as discussed in Section 3.

Complex events are a function of when and where they occur. We address the need of event detection applications for spatial and temporal semantics by incorporating spatial and temporal logic into MEDAL.

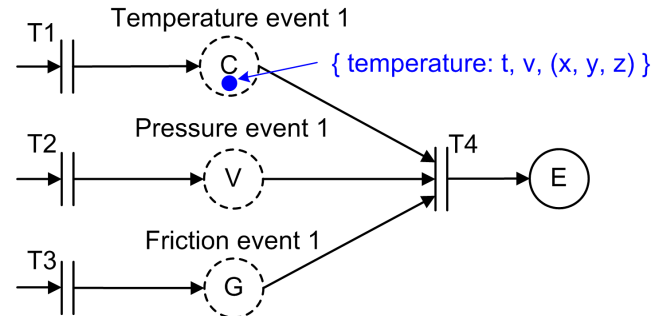


Figure 1: MEDAL model of an event detection system

2.1.1 Temporal logic: Temporal logic refers to the temporal guard function β . It helps specify the temporal concepts “when” and “how long” in MEDAL Petri nets. β guards the transitions to ensure they fire only during the specified temporal intervals. Introducing β in MEDAL has practical importance because, some events can occur only during a particular temporal interval. For example, clustering to form a group can only happen if intensive communication occurs within a relatively short interval. In addition, β can help specify conditions such as “a transition will fire only if the input tokens have been generated within a predefined time interval”. In Figure 1, for example, if the generation times of the tokens entering transition $T4$ are more than 30 seconds away from each other, it is more likely that the events were unrelated or they belong to different groups rather than indicating the occurrence of event E . In cases like this, the network should not report the occurrence of an event even if the necessary input tokens are present.

2.1.2 Spatial logic The geographic semantics of the application are enforced by the spatial function L . As a guard function for a transition T , L ensures that the tokens carried by T 's incoming arcs satisfy the spatial locality conditions. If $L(T) = R$, the effective radius of the higher-level event recognized by T should be equal to or smaller than R . In other words, there should be a circle of radius R encompassing all tokens' locations in order to consider the readings to be caused by one particular event. In Figure 1, for example, if the tokens in the input places of transition $T4$ are generated at a distance larger than R from each other, it is likely that the events are not related. L could help detect such situations and thus decrease the number of false positives in event detection.

2.2 Introducing probabilities

Most previous work on event description in embedded sensor networks uses precise, also called “crisp”, values to specify the parameters that characterize an event. For example, we might want to know if the temperature drops below 5°C . However, sensor readings are not always precise. In addition, different sensors, even if located close

to each other, often vary in the values they register. Consider an example scenario where we want the cooling to be turned on if the temperature goes above 5°C. Two sensors, *A* and *B*, measure the temperature in the room and the average of the values they report is used to determine if an action should be taken. At some point, sensor *A* reports 5.1°C and sensor *B* reports 4.8°C. The average, 4.95°C, is below the predefined threshold and the cooling remains off. However, if sensor *B*'s measurement is imprecise and therefore lower than the actual temperature, we have made the wrong decision. The situation becomes even more convoluted when more than two sensor measurements are involved. This makes determining the precise event thresholds an extremely hard task which has led us to believe that using crisp values might not be the best approach. Fuzzy logic, on the other hand, has a number of properties that make it suitable for describing events in sensor networks:

- It can tolerate the unreliable and imprecise sensor readings;
- It is much closer to our way of thinking than crisp logic. For example, we think of fire as an event described by high temperature and the presence smoke rather than an event characterized by temperature above 55°C and smoke obscuration level above 15%;
- Compared to other classification algorithms based on probability theory, fuzzy logic is much more intuitive and easier to use.

The structure of a general fuzzy logic system (FLS) is shown in Figure 2. First, the fuzzifier converts the crisp input variables $x \in X$, where X is the set of possible input variables, to fuzzy *linguistic variables* by applying the corresponding membership functions. Linguistic variables are “variables whose values are not numbers but words or sentences in a natural or artificial language” [3]. An input variable can be associated with one or more fuzzy sets depending on the calculated membership degrees. For example, a temperature value can be classified as both Cold and Lukewarm. Second, the fuzzified values are processed by *if-then* linguistic statements, called rules, derived from domain knowledge provided by experts. These rules are of the form:

IF premise, THEN consequent

where the *premise* is composed of fuzzy input variables connected by logical functions (e.g. AND, OR, NOT) and the *consequent* is a fuzzy output variable. The inference scheme maps input fuzzy sets to output fuzzy sets. Finally, the defuzzifier uses the output fuzzy sets to compute a crisp output. The crisp output value determines the control actions that need to be taken.

As previously mentioned, sensor readings are generally believed to be unreliable and imprecise. Therefore, to increase our confidence in the presence of an

event somewhere in the monitored area, we often need readings from multiple sensors and/or readings over some period of time. To address this, we instrument the event detection process with spatial and temporal semantics. We believe that including temporal and spatial linguistic variables in the rule-base can significantly improve the detection accuracy. It can also allow us to describe and detect more complex events. To the best of our knowledge, no previous work on applying fuzzy logic to event detection has considered the effects of temporal and spatial semantics on the accuracy of event detection.

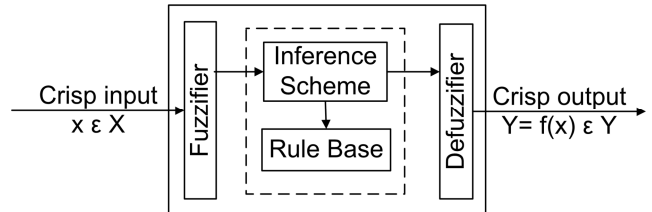


Figure 2: The structure of a fuzzy logic system

2.2.1 Spatial semantics: It is important to understand how including spatial guards affects the accuracy of event detection. Spatial guard variables will allow us to express any spatial requirements the detection application might have. An example requirement could be: “if the readings of two sensor nodes indicate the presence of event *X*, we believe that this event has occurred only if the two sensor nodes are located close to each other”. A disadvantage of including spatial semantics is that sometimes events might not be detected if the spatial guard is too strict. However, we believe that using fuzzy logic will help us alleviate this problem.

2.2.2 Temporal semantics: To further decrease the number of false alarms we also need to take into account the temporal properties of the monitored events. One approach is to include linguistic variables in the rule-base that can act as temporal guards. Adding temporal semantics is especially important for embedded sensor networks because of the nature of sensor communication. It is very possible for messages in a wireless sensor network to be delayed because of network congestions or bad routing. Consequently, a reliable event detection rule-base should take into consideration the generation times of the participating sensor readings.

2.2.3 Decreasing the rule-base: A disadvantage of using fuzzy logic is that storing the rule-base might require a significant amount of memory. The number of rules grows exponentially to the number of variables. With n variables each of which can take m values, the number of rules in the rule-base is m^n . For example, if there are four linguistic variables each of which can be associated with one of five values, the rule-base will contain 625 rules.

Adding spatial and temporal linguistic variables to the rule-base further increases the number of rules. Since sensor nodes have limited memory, storing a full rule-base on every node would be a waste of valuable resources. In addition, constantly traversing a large rule-base might considerably slow down the detection process. Further, the intensified computation caused by frequently going through a larger rule-base will also lead to increase in the power consumption. To address this issue, we have designed a set of rule-base reduction techniques to decrease the size of the rule-base. An important property of these techniques is that they do not negatively affect the accuracy and promptness of event detection.

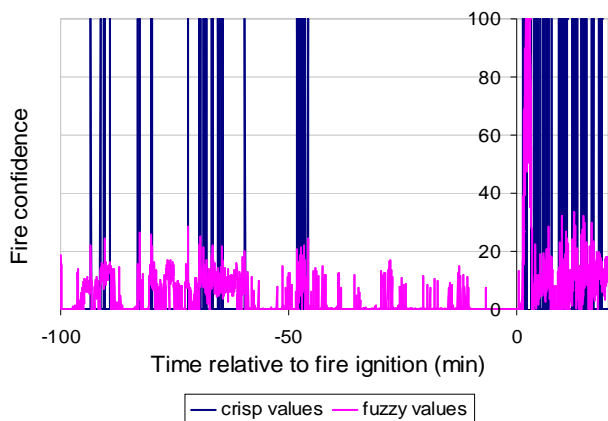


Figure 3: Fire simulation: burning mattress

2.2.4. Experimental results: We have performed preliminary simulation experiments on real fire data publicly available on the National Institute of Standards and Technology (NIST) website [4]. The goal of these simulations was to evaluate how using fuzzy logic affects the accuracy of event detection. The experiments were run with both crisp and fuzzy values. The temperature and smoke obscuration values we used in the crisp logic experiments are threshold values used in commercial smoke and heat detectors [5, 6].

Figure 3 shows the results from one of the crisp-value experiments. The origin of the coordinate system represents the time of fire ignition. As we can see from the figure, using crisp values results in a very large number of false fire detections. In the period prior to fire ignition, there were 40 false fire detections which constitutes about 1.3% of the readings. This considerable number of false positives significantly affect the efficiency and fidelity of an event detection system. These results show that fuzzy logic is suitable for use for event detection in sensor networks. However, we need to perform additional experiments to determine if there are applications for which this does not hold true. It is also important to compare the resource requirements of applications using fuzzy and crisp values.

We used the same scenario to evaluate the efficiency of the rule-base reduction techniques. The rule-base initially had 81 rules. Applying two of our reduction techniques, which take advantage of the similarity between rules as well as the significance of different rules, helped decrease the size of the rule-base by more than 70% without compromising the event detection accuracy.

2.3 Event transformation

Transforming the formal model into code that can be executed on the sensor nodes is the next step an event service needs to perform. Since the process of recognizing the specified events is similar to a DNA transcript procedure, we call the event recognition code generated from the MEDAL model and stored on the sensor nodes, event-DNA. Each event-DNA is an encoded representation of a MEDAL model and just like a MEDAL model, the event-DNA might represent the description of simple or complex events. The sensor nodes have an event detection middleware stored in their memory which can read different event-DNAs and act accordingly.

2.4 Event service framework

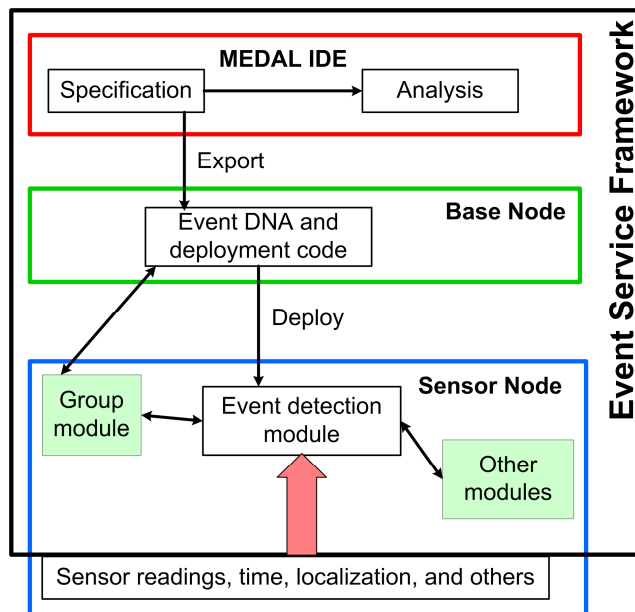


Figure 4: Event service framework

The overall event service architecture we are developing is shown in Figure 4. In the figure, the MEDAL IDE (Integrated Development Environment) is an offline package which resides on a PC and from which specified event semantics can be encoded and exported to a base node. The base node installs a deployment module that deploys all event-DNAs onto their corresponding nodes. For example, if an event-DNA represents a mote-

level event, it will be deployed onto every mote, while if an event-DNA represents a group-level event, only a group leader will have a copy. For dynamic leader schemes, the deployment module needs to interact with the group module to determine what to deploy. Then inside each mote, the imported code generated from the event-DNA is stored in program memory. The detection module achieves its goal by calling other lower primitives in the program section such as reading sensor values, obtaining time, location information, and other data from other modules in the program section.

The framework contains the following modules:

- **MEDAL environment**, which includes both a specification module and an analysis module. It also encodes event specifications into event-DNAs and exports them to the base node.

- **The Base node** contains encoded event-DNAs and has a deployment module in charge of transferring event-DNAs to nodes in the field. The deployment module provides communication, in which the messages are event-DNAs.

- At the **Sensor node** level, the event detection module resides in program memory. The event detection module uses a token vector to communicate with other nodes for collaborative detection of higher level events.

Once the designers have the MEDAL model that describes their application, the next step is to write the code to be run on the nodes. A weakness of this step, however, is that manually translating the formal model into code might introduce bugs as well as lead to divergence of the code from the model. An advantage MEDAL has over other currently used event description approaches is that due to its formal structure and lack of ambiguity, it can be directly and automatically translated into code that can be executed on the sensor nodes. We have been developing a tool to automatically generate the event-DNA code based on the MEDAL model of the application. Currently, we are generating TinyOS code using the formal application model [7]. However, our approach is adaptable so that code in languages other than nesC can be generated as well. This approach will significantly reduce the effort of writing TinyOS code and improve the correctness of the code.

3. Real-time stream data

3.1 Overview

For timely detection of complex events, real-time embedded sensor network systems have to operate on continuous unbounded data streams from multiple sources. The streaming data may come from different types of sensors which need to be integrated and fused with each other as well as with the data already stored in the system. Stream data takes the form of continuous,

ordered, potentially infinite data streams, as opposed to finite, statically stored data sets. In embedded sensor network applications, there exist significant volumes of dynamically changing data in the form of data streams, such as data associated with the current locations and movements, data in the form of simple events detected by other subsystems, and data from other sources reflecting the dynamic and volatile situations. Analysis of stream data poses great challenges to real-time data management, due to the unique features of data streams, such as huge (and possibly infinite) volume, unpredictable changing patterns, and flowing in-and-out in a dynamic order. Due to the high volume of data streams and the timing constraints of the applications, it is often assumed that it is not possible to store a stream in its entirety, nor is it feasible to query the whole stream history. Typically, the queries are executed on a *window* of data. A window on a data stream is a segment of the data stream that is considered for the current query. A lot of stream data resides at the primitive abstraction level in the form of raw sensor data. It is necessary to perform *aggregation and generation of derived data* from the raw data to find interesting patterns or outliers at appropriate levels of abstraction and with appropriate dimension combinations.

3.2 Quality management

To process real-time stream data in embedded sensor networks, the system should support long-running and persistent queries. When a query arrives in the system, it is registered and its instances are executed periodically. All queries are pre-registered in the system and converted to query plans (containing operators, queues, and synopses) before the system starts executing. Queues in a query plan model the incoming data streams and the intermediate results between the operators. A synopsis could be related to a specific operator and it stores state that may be needed for future evaluation of the system. For example, a join operator may be associated with a synopsis for each of its inputs to store some of the items. These items can be probed later by the operator as needed.

Let us consider the following data streams and query associated with it. The query result can be used to detect unusually speeding trucks.

Stream: Speed (int lane, float value, char[8] type);

Relation: Lanes (int ID);

Query: SELECT avg (Speed.value) FROM Speed [range 1 minute], Lanes WHERE Speed.lane = Lanes.ID AND Speed.type = Truck;

Period 10 seconds

Deadline 5 seconds

The query above operates on data streams generated by speed sensors and calculates the average speed of trucks in

particular lanes in the last 1 minute. The query needs to be executed every 10 seconds and the deadline is 5 seconds after the release time of every periodic query instance. The generated query plan is shown in Figure 5. This query plan consists of three query operators (range window operator, join operator, and aggregate operator) and two buffers (one for storing the range window output and the other for storing the output of the join operator).

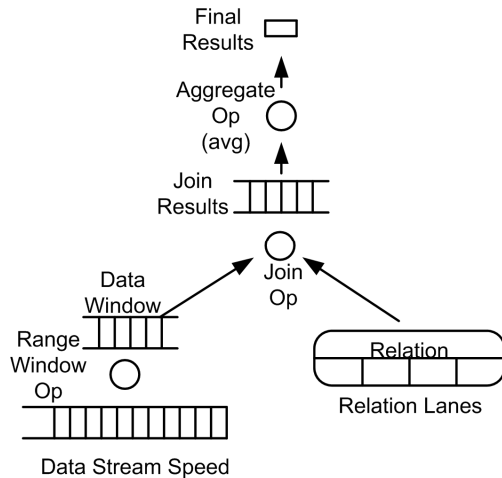


Figure 5: Query plan

One of the main challenges for data stream management with real-time constraints, as in embedded sensor networks, is the unpredictability of the data streams themselves. It is possible that the system may get overloaded as the arrival rates and contents of the incoming data streams change. The system must be able to handle these workload fluctuations. Otherwise some of the queries may miss their deadline. This QoS management can be performed at two levels. The *inter-query QoS management* can allocate resources to different queries in the case of system overload so that they have similar quality in query results. The *intra-query QoS management* is then used to allocate available resources to different operators within the query plan so as to maximize the query quality. For inter-query QoS management, the system needs to estimate the query execution time corresponding to different input data sizes. For intra-query QoS management, the system needs to know the selectivity of different operators corresponding to the current data input and their estimated execution time. Since our main objective is to ensure the timeliness of query results, the query execution time estimation is the key to QoS management. In order to estimate the query execution time, we need three parameters for each query, namely, the input data stream volume, the operator selectivity, and the execution time per data tuple for each operator. In this work, we consider queries that are ready to be executed when performing the QoS management routine. Therefore, the input data volumes for these

queries are known since the incoming data stream segments are already present in the system.

It is beneficial to estimate the execution time of queries which do not have their complete input yet and use these estimations in the QoS management process, since the current ready-to-go queries may overlap with these future queries in their life span. This is a challenging problem as it involves designing effective algorithms to monitor and estimate the volume and contents of data streams.

4. Summary

This work provides the basis for developing robust real-time data and event services that allow sensor network applications to monitor, detect, and react to sophisticated events. We are designing an event service framework which combines several techniques that will help provide light-weight and timely event detection in embedded sensor networks. In addition, in order to improve the performance of these sensor systems, we are developing approaches to manage the QoS. We believe that combining the event service framework and the QoS management mechanisms will significantly improve the timeliness and reduce the resource requirements of event detection in embedded sensor network systems.

5. References

- [1] C. Girault, R. Valk. "Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications" Springer-Verlag New York, Inc. (2001).
- [2] K. Kapitanova, S. H. Son. "MEDAL: A coMcompact Event Description and Analysis Language for Wireless Sensor Networks", *International Conference on Networked Sensing Systems*, 2009
- [3] L. Zadeh. "Outline of a New Approach to the Analysis of Complex Systems and Decision Processes", *IEEE Transactions on Systems, Man, and Cybernetics*, 1973
- [4] Building and fire research laboratory. <http://smokealarm.nist.gov/>.
- [5] J. Geiman and D. Gottuk, "Alarm Thresholds for Smoke Detector Modeling," *Fire Safety Science – Proceedings of the 7th International Symposium 2002*
- [6] WS4916 Series Wireless Smoke Detector. <http://www.alarmsuperstore.com/dsc/ws4916installation.pdf>.
- [7] Y. Wu, K. Kapitanova, J. Li, J Stankovic, S. Son, K. Whitehouse, "Run Time Assurance of Application-Level Requirements in Wireless Sensor Networks", *IPSN 2010*
- [8] M. Keally, G. Zhou, G. Xing, "Watchdog: Confident Event Detection in Heterogeneous Sensor Networks", *RTAS 2010*