Hindawi Publishing Corporation
International Journal of Distributed Sensor Networks
Volume 2012, Article ID 171738, 12 pages
doi:10.1155/2012/171738

Research Article

Logic Macroprogramming for Wireless Sensor Networks

Supasate Choochaisri, Nuttanart Pornprasitsakul, and Chalermek Intanagonwiwat

Department of Computer Engineering, Chulalongkorn University, Bangkok 10330, Thailand

Correspondence should be addressed to Chalermek Intanagonwiwat, intanago@yahoo.com

Received 3 October 2011; Accepted 17 December 2011

Academic Editor: Tai Hoon Kim

Copyright © 2012 Supasate Choochaisri et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

It is notoriously difficult and tedious to program wireless sensor networks (WSNs). To simplify WSN programming, we propose Sense2P, a logic macroprogramming system for abstracting, programming, and using WSNs as globally deductive databases. Unlike macroprograms in previous works, our logic macroprograms can be described declaratively and imperatively. In Sense2P, logic macroprogrammers can easily express a recursive program or query that is unsupported in existing database abstractions for WSNs. We have evaluated Sense2P analytically and experimentally. Our evaluation result indicates that Sense2P successfully realizes the logic macroprogramming concept while consuming minimal energy as well as maintaining completeness and soundness of the answers.

1. Introduction

Wireless sensor networks (WSNs) have been widely used for collecting data from environments [1–4]. However, sensor nodes are resource constrained and distributed all over the monitored area. Programming WSNs to acquire such data is notoriously difficult and tedious. Traditional WSN programming requires system programming in low-level details (e.g., wiring nesC [5] components, coordinating the program flow among nodes in a distributed manner, routing, discovering resources, accessing, and managing remote data) while maintaining low energy consumption and memory usage [6].

Several programming abstractions have been proposed to simplify WSN programming with high-level languages and to hide the low-level details from programmers [6–12]. The WSN programming abstractions have been divided into two classes: local-behavior class and global-behavior class (also called *macroprogramming* class). The abstraction in the former class simplifies the programming task of specifying the local behavior of each node for distributed computation. Local-behavior abstractions include abstract regions [11, 12] and DSN [6]. These local-behavior abstractions can efficiently hide some of the above low-level programming details but the programmers still need to write a distributed code for routing, coordinating the program flow among nodes, accessing, and managing remote data.

Conversely, the abstraction of the macroprogramming class enables expressing the global behavior of the distributed computation by programming the WSN in the large [7]. These macroprogramming abstractions can hide even more low-level programming details than the local-behavior abstractions do. In a sense, macroprogrammers take a centralized view of programming a distributed system rather than a distributed view. The macrocompiler is responsible for translating the macroprogram into a distributed version for execution.

There are two subclasses of macroprogramming abstractions: node dependent and node independent. In the node dependent subclass, a WSN is abstracted as a collection of nodes that can be simultaneously tasked within a single program. Examples of the node-dependent subclass include Kairos [7], Regiment [10], Split-C [13], SP [14], and DRN [8].

By contrast, in the node-independent subclass, a WSN is abstracted and programmed as a whole or a unit instead of several interacting nodes. Low-level programming details are completely abstracted out in this subclass as there are no longer networks or nodes in the programmer's view. Examples of this subclass include TinyDB [9] and Cougar

[15]. Both have abstracted WSNs as relational databases that are programmed or queried in a SQL-like language. This abstraction is reasonable because WSNs have also been queried for data in relation [16]. Given this database abstraction, WSN programming is reduced to database querying.

However, SQL is a pure declarative programming language for specifying what the programmer wants, not how to algorithmically obtain the desired result. Despite its simplicity, declarative programming may not be applicable to several WSN applications, especially complex tasks or queries. Undoubtedly, imperative programming (or procedural programming) is more appropriate for such complex tasks where efficient algorithmic details are application specific, unobvious, or difficult to generate automatically. Declarative and imperative programming approaches function well within their domain and complement one another. Integration of both approaches can form a powerful programming paradigm suitable for both domains.

Widely considered such integration, logic programming is the use of logic as both a declarative and imperative representation language [17]. A logic program consists of declarative sentences in the form of implications. Based on a backwards reasoning theorem prover, logic programming treats the implications as goal-reduction procedures. Logic programmers can exploit the problem-solving behavior of the theorem prover to achieve efficiency. This is similar to how imperative programmers use programs to control the behavior of a program executor. However, unlike pure imperative programs, the correctness of logic programs can be ensured with their declarative and logical interpretation.

In this paper, we propose Sense2P, a logic macroprogramming system for abstracting and programming WSNs as globally deductive databases. Unlike macroprograms in previous works, our logic macroprograms can be described declaratively and imperatively. As a result, Sense2P is highly expressive and efficient, compared to SQL-based systems.

Another advantage of logic macroprogramming is its capability to easily express a recursive program. Even though one can express a recursive query in SQL, the recursive SQL query is rather verbose (see appendix A) and unsupported in existing systems for WSNs.

Our evaluation result indicates that Sense2P can realize the logic macroprogramming concept while consuming minimal energy and maintaining completeness and soundness of the answers.

The remainder of the paper is described as follows. Section 2 reviews related work about macroprogramming and logic programming in WSNs. Section 3 describes the logic macroprogramming approach to WSNs. Then, we explain Sense2P in Section 4. Sections 5 and 6 cover our programming model and system architecture, respectively. We mathematically analyse the communication cost of our approach in Section 7 and experimentally evaluate the performance of our system in Section 8. Finally, Section 9 concludes the paper.

2. Related Work

Various macroprogramming abstractions have been proposed for several years. However, no abstraction fits all domains. We discuss the differences of our abstraction from those existing ones in this section.

Of a particular interest are Kairos [7], Regiment [10], and DRN [8]. Kairos presents the programming model that computes a set of sensor devices in parallel and provides a facility to sequentially access remote variables. Unlike Kairos, Regiment is the *spatiotemporal* macroprogramming system that is based on the concept of functional reactive programming. However, Regiment is designed for long-running queries (not well-suited for short-lived queries).

DRN is a hybrid approach between imperative programming and declarative programming. Resources and nodes are declaratively named whereas the core algorithm is imperatively programmed. Similar to DRN, Sense2P is also a hybrid approach, given that logic programming is an integration of imperative programming and declarative programming. Kairos, Regiment, and DRN are node dependent but Sense2P is node independent.

Semantic Stream [18] is a macroprogramming framework with logic programming features that allows users to pose declarative queries over semantic interpretations of sensor data. However, Semantic Stream focuses on finding available services and providing the quality of services instead of problem solving. Furthermore, it is not designed specifically for wireless sensor nodes with limited resources.

Chu et al. [6] have further developed the concept of logic programming into Snlog for programming WSNs and enabling recursive queries. Snlog, however, is designed for low-level programmers, not for application-level programmers. Unlike Sense2P programmers, Snlog programmers must write rules by focusing on local behaviors of each sensor node (instead of the global behavior as a whole). Therefore, Snlog does not support a join between different nodes. Additionally, the Snlog programmers must deal with networking details and protocols, such as routing, query disseminating, and data collecting. In summary, Sense2P is a macroprogramming approach but Snlog is not.

TinyDB [9] and Cougar [15] are probably the most cited node-independent abstractions for macroprogramming WSNs. Those approaches abstract a WSN as a relational database. Consequently, WSN programming is reduced to database querying. However, there are several limitations in the mentioned approaches.

First, supported queries in the previous works are quite limited. For example, there is only one table accessible at a time. This may not work in networks of heterogeneous sensors. In other words, their queries do not support a join between different sensor nodes. In addition, only conjunctive comparison predicates are supported, and arithmetic expressions are limited to operations of an attribute and a constant. As a result, tuple selection is inflexible. Furthermore, subqueries and column aliases are not allowed either.

Second, each sensed data item is kept as a tuple associated with each node. Constraints in the query are applied only to attributes in the same tuple as well as the same node

Table 1: Characteristics comparison.

Approach	Characteristic				
	Programming model	Abstraction level	Node dependency	Communication transparency	Recursive query
Kairos	Imperative (procedural programming)	Network level (global)	Node dependent	Yes	No
Regiment	Declarative (functional programming)	Network level (global)	Node dependent	Yes	Yes
DRN	Declarative and imperative (procedural programming with resource variable)	Network level (global)	Node dependent	Yes	No
Cougar	Declarative (SQL)	Network level (global)	Node independent	Yes	No
TinyDB	Declarative (SQL)	Network level (global)	Node independent	Yes	No
Semantic Stream	Declarative (logic programming)	Network level (global)	Node independent	Yes	No
Snlog	Declarative and imperative (logic programming)	Node level (local)	Node dependent	No	Yes
Sense2P	Declarative and imperative (logic programming)	Network level (global)	Node independent	Yes	Yes

[9]. Therefore, the constraints are local, not global. It is not designed for deriving data that is related with other data from different nodes. As a result, they cannot support a join operation.

Third, they do not support recursive queries. It is well documented that the recursive queries can improve the capability of a database [19, 20].

Finally, previous systems with a relational-database abstraction do not support a logic-based query frequently used in deductive databases and expert systems.

Unlike TinyDB and Cougar, our approach abstracts a WSN as a globally deductive database that can be logically programmed. As a result, our approach does not suffer from the above limitations.

We summarize the characteristic differences of related works in Table 1.

3. Logic Macroprogramming

Logic programming is a logic-based declarative approach to knowledge representation that allows recursive programming. Logic programming is widely used in many artificial-intelligence applications such as knowledge-based systems, expert systems, and smart information-management systems, and so forth. Prolog [17] is a de facto language for logic programming in traditional systems. Logic programming can be combined with relational databases in order to construct "deductive database" systems that support a powerful formalism and operate quickly even with very large data sets. Their powerful features include a capability to process recursive queries and their superior expressiveness over relational databases. For example, a Prolog system can be loosely coupled with a relational database system [21] to become a deductive database system (i.e., a relational database system with an inference engine). Our early work in abstracting WSNs as deductive databases has been presented in [22].

Given this deductive-database abstraction, tasks can be logically macroprogrammed in a Prolog-like language. In WSNs, each node senses the environmental data periodically or reactively. The sensed data is locally stored and viewed as a fact in our system (see Section 4 for more details). These facts are available to logic macroprogrammers as if the facts are on the centralized database. Logic macroprogrammers simply focus on what data they need (declaratively) and how to process the data (imperatively) but not on how to retrieve those data. The macroprogrammers can create facts and rules as well as inject queries into our network-transparent system as if the network is a deductive database.

Furthermore, the macroprogrammers can also write rules for deducing new facts from existing facts and rules recursively.

4. Sense2P

Sense2P is our prototype for logic macroprogramming WSNs in a Prolog-like language. Our system allows programmers to write recursive and nonrecursive rules (programs) without being concerned with low-level programming details. Additionally, Sense2P is sufficiently simple for application-level users who only want to query the system for interested data. Our programming model and system architecture are described as follows.

5. Programming Model

Briefly, our programming language in Sense2P is Prolog like. The language consists of predicates, facts, rules, and queries.

5.1. Predicate. Predicates are relations of data (or tables in the relational-database terminology). For example, a predicate temperature (NodeID, Temperature Value) describes a relation between a node identification number and a temperature value. From this example, temperature is a predicate name while NodeID and Temperature Value are

variable arguments. In general, an argument is a variable if it begins with the capital letter. Conversely, an argument is a constant if it begins with the small letter or it is a number. Predicates in Sense2P are divided into 3 categories: user defined, built-in, and sensor specific. The first predicate type includes arbitrary predicates defined by programmers. The second type includes general predicates that are already built in the system. Examples of built-in predicates are *sum* (for computing the summation of two values) and *abs* (for computing the absolute value). The last type includes only built-in predicates that are specific to sensors. The previously mentioned temperature predicate is such a sensor-specific predicate. In Sense2P, the sensor-specific predicates are processed differently from other types. We describe our processing methodology in the next subsection.

5.2. Fact. A fact is a predicate whose arguments are all constant. One may consider facts as already-existing data in the system. Facts can be instantiated in three forms: user defined, sensor generated, and rule deduced. Users can define known facts in the program such as *location* (1, 33, 45). This fact indicates that a node ID 1 is located at coordinate (33, 45). Some facts are data-sensed from sensors. For example, temperature (3, 25) is a fact that indicates a temperature of 25°C sensed by a node ID 3. Finally, facts can also be deduced from existing facts and rules.

5.3. Rule. Rules are clauses that deduce new facts from existing facts. Rules are represented as Horn clauses that contain head and body parts. An example of a rule is shown in Listing 1.

Specifically, an area has a hot spot if an arbitrary node in that area senses temperature with a value over 50 degrees. The left-hand side of a clause is called head and the right-hand side is called body. In Listing 1, the head is *hotSpotArea(AreaID)* and the body is *temperature(NodeID, Temp), Temp* > 50*, area(NodeID, AreaID)*. A rule will be satisfied only if every predicates in the body are satisfied or matched by at least one fact.

5.4. Query. Queries are questions that a user asks to retrieve data from the system. A query is represented by *?-*followed by a predicate. For example, *?-hasHotSpotArea(X)* is a query to retrieve IDs of all areas that has a hot sensor node.

Queries can be classified into 4 groups. The first group is a fact-checking query that is intended for checking the existences of certain facts. This query type is expressed by a predicate with no variable argument, for example, *?-temperature(2, 5)*. The second group includes fact-retrieving queries that are designed for retrieving all data that satisfy the fact types and constraints in the queries. This query type contains at least one variable in a predicate, for example, *?-temperature(X, Y)*. The third group consists of queries for checking whether existing rules can deduce a certain fact. It is almost like the fact-checking query except that the predicate matches with a rule (instead of a fact), for example, *?-hotSpotArea(5)*. The final group is composed of deductive queries for retrieving all data that satisfy the rules

```
hotSpotArea(AreaID):- temperature(NodeID, T)
                    , T > 50
                    , area(NodeID, AreaID)
```

Listing 1: Example of a rule.

```
danger(AreaID):- temperature(NodeID,T)
               , T > 80
               , area(NodeID, AreaID).
danger(AreaID):- humidity(NodeID, H)
               , H < 40
               , area(NodeID, AreaID)
               , adjacent(AreaID, AdjAreaID)
               , winddir(AdjAreaID, AreaID)
               , danger(AdjAreaID).
Query: ?-danger(X).
```

Listing 2: Example of recursive query rules.

in the queries, for example, *?-hotSpotArea(X)*. A query will be recursive if its predicate matches with a recursive rule whose body contains the same predicate name as that in its head. For example, one can write recursive rules for detecting sensor nodes in danger as shown in Listing 2.

The first rule is a base case of a recursive program that describes properties of areas in danger. An area will be in danger if there is at least one node (in that area) whose sensed temperature is greater than 80 degrees (which may be on fire). The second rule is a recursive case. Basically, an area will be in danger if there is at least one node (in the area) whose sensed relative humidity is lower than 40 % (the weather is dry) and there is wind from the adjacent area in danger. When the Sens2P program starts, a user can define facts and rules before injecting a query into the system.

6. System Architecture

Sense2P consists of two major components: the query processing engine and the data-gathering engine (Figure 1). The query processing engine resides on the base station while the data-gathering engine resides on each wireless sensor node.

6.1. Query Processing Engine. The query processing engine is crucial for logic macroprogramming WSNs. The main tasks are to interpret a user program (consisting of facts, rules, and queries) and to process queries to find satisfying answers.

Sense2P query processing engine consists of three main components: a compiler unit, a run-time processing unit, and a network interface unit. The compiler unit parses a logic macroprogram into a compiled code that runs on the run-time processing unit. The run-time processing unit is required to treat sensor-specific facts differently from those of other ordinary facts. Sensor-specific facts are data

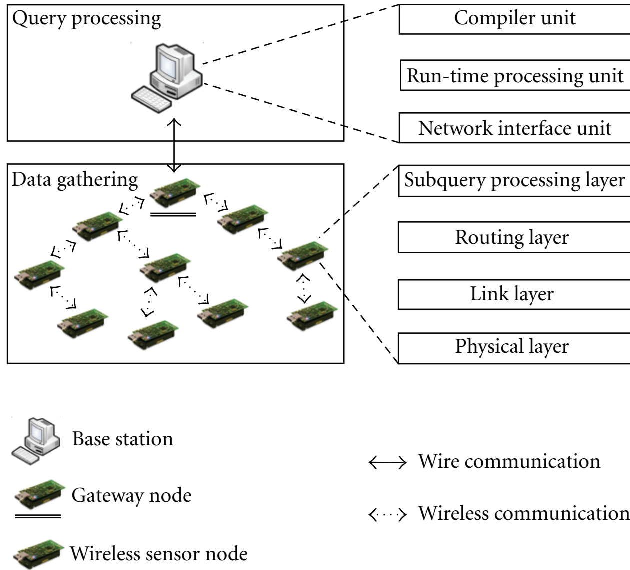


Figure 1: System Architecture.

locally sensed and stored by sensor nodes in the network. Processing queries related to these facts requires special attention because unnecessary data transmissions are costly in WSNs.

In this paper, we consider three previously proposed schemes for query processing in deductive databases. These schemes include top-down, bottom-up, and Prolog-style evaluation approaches [21].

Prolog-style systems (coupled with database systems) are similar to the top-down systems in a sense that their execution starts from the goal and a query can be solved by executing each subgoal until the deduced facts match the goal. However, Prolog-style systems produce answers one tuple at a time whereas top-down methods produce one set at a time without in-order execution of subgoals.

Conversely, the bottom-up methods start from existing facts and attempt to deduce new facts from rules that are related to the query. Only facts that match with the goal of the query are selected as the answers. We refer to [21] for more information of each implementation scheme.

Many works suggest that the bottom-up methods have many advantages over top-down methods in traditional deductive database systems [20, 21, 23]. However, in wireless sensor networks, we argue that top-down and Prolog-style approaches are more appropriate.

Understandably, facts in a wireless sensor network are data that are sensed from an environment. They are locally kept within sensing nodes and sent to the base station only when requested. To process a query in a bottom-up manner, we need facts from all relevant nodes so that new facts can be globally deduced. Therefore, each node may be required to send its data to a rendezvous point (e.g., a base station, an inference engine) for such a deduction. Undoubtedly, the mentioned mechanism consumes excessive energy. To reduce this energy consumption, only relevant data should be delivered.

```
hotObject(Obj, AreaID):- detect(Obj, AreaID)
                       , temperature(Obj, T)
                       , T > 50.
```

Listing 3: Example of a rule that two predicates related to each other with Obj.

However, it is not easy for a node to selectively send relevant data without knowing priori what all other nodes have. A fact in a node may be relevant simply because another fact from another node happens to have a certain value.

Conversely, the top-down approach can use information from a query to suppress irrelevant facts from being sent. For example, a predicate *detect(ObjectID, AreaID)* in a system means a sensor node can detect an object with the identification number *ObjectID* in the region *AreaID*. When a user injects a query *?-detect(oiltank,X)*, only sensor nodes that can detect an object named *oiltank* will send answers back. Other nodes are suppressed.

Furthermore, we can even suppress unnecessary query forwarding and redundant answering. The benefit is evident in fact-checking queries, such as *?-detect(oiltank, area70)*. In our system, only the first node detecting *oiltank* in area 70 will reply, although there may be other nodes (in the same area) that detect the same event. This is reasonable, given that one's reply about the fact existence is sufficient to satisfy the query. Therefore, the first detecting node does not need to forward the query further. Thus, there is no other replier (see Section 6.2 for more details).

In addition, we can use an answer set from the previous subgoal to filter out (or suppress) the irrelevant facts of the next subgoal. For example, consider the rule in Listing 3.

When a user injects a query *?-hotObject(X, area70)*, the system will match the query with the above rule. Therefore, the variable *AreaID* in the rule will be bound with the constant *area70*. Then, the system will attempt to match each predicate in the body of the rule. Each body predicate becomes a subquery that needs to be satisfied.

In this example, the first subquery is *detect(ObjectID, area70)*. This subquery is disseminated into the network. Only eligible repliers are nodes with facts or rules that match the subquery. Others are suppressed. Consequently, only objects in *area70* will be bound to the variable *ObjectID*. Then, each *ObjectID* will be used to bind *temperature(ObjectID, Temp)* predicate and can be used to filter out or suppress irrelevant facts from being sent. Furthermore, we can also use a constraint *Temp > 50* as another filter before injecting a subquery *temperature(ObjectID, Temp)* to the network.

Due to these filtering techniques, this top-down approach can significantly reduce the consumption of energy that is limited in wireless sensor networks [16]. Therefore, the Prolog-style top-down approach is used and combined with our filtering techniques in this paper.

In our system, most relevant facts are pulled from the network except the persistent ones that do not change over

time. The persistent facts can be cached or kept in the backend database of the inference engine for future uses.

Additionally, we propose a *superset-caching* technique to reduce even more energy consumption. With this technique, Sense2P will cache the answer set of the first-timer query (that requires sensor-specific predicates) for future use. If a user injects a new query, Sense2P will check whether one of the previous queries is a superset of the current query or not. If a previous query is the superset, Sense2P will search the satisfying answers in the cache rather than in the network. Otherwise, the query will be disseminated into the network.

For example, an answer set of a query *?-light(X,Y)* is a superset of a query *?-light(3,X)* because *light(X,Y)* contains every possible answers in the network. Conversely, an answer set of a query *?-light(5,X)* or *?-light(X,20)* is not a superset of *?-light(3,X)*. However, this caching technique is not perfect, especially when the data in the environment is volatile. Therefore, a timer to flush cache may be needed. Nevertheless, the flushing period is a trade-off between the data freshness and the energy consumption.

Finally, the network interface unit is responsible for disseminating queries or subqueries into the network. The queries are transformed into a format known in sensor networks, serialized, and sent into the network. The unit is also responsible for receiving answers from the network. This requires deserialization and transformation of messages back into the Prolog-like predicates.

Our approach works well on both recursive and nonrecursive queries. The query-processing flow is illustrated in Figure 2.

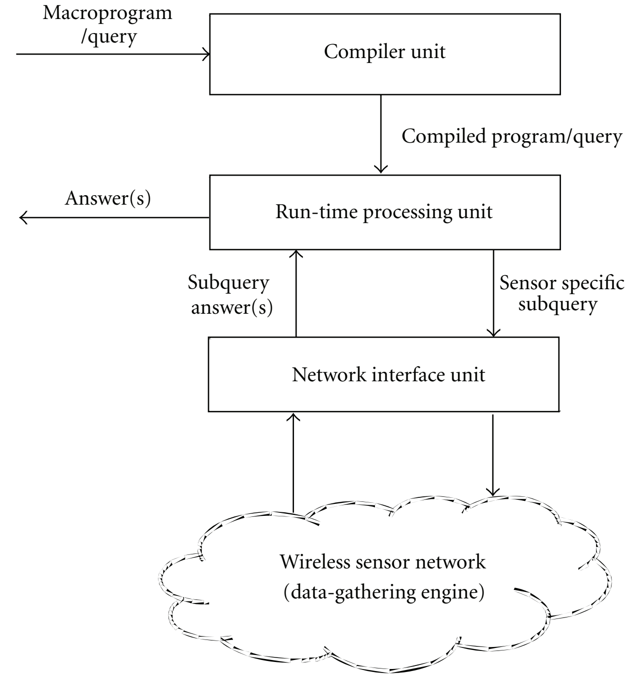


Figure 2: Query processing flow.

6.2. Data-Gathering Engine. Data-gathering engine is responsible for finding answers that are relevant to injected queries. This engine consists of the routing layer, the query-processing layer, the link layer, and the physical layer. However, our work simply focuses on the routing layer and the query processing layer. Both mentioned layers are handled by our LogicQ sub-system.

LogicQ is the underlying subsystem for subquery processing in Sense2P. Running on each sensor node, LogicQ is implemented in TinyOS [24], the operating system for the sensor mote platform. The functionality of LogicQ is to find answers for each subgoal that needs data from wireless sensor networks and minimizes the energy consumption.

When Sense2P starts up, LogicQ constructs a routing tree for disseminating subqueries from the base station to sensor nodes and for collecting answers that satisfy the subqueries. We use a drain routing tree of TinyOS as our routing tree. A root of the tree is the gateway node connected to the base station. Each node can have many child nodes but only one parent node. When disseminating the queries, we simply forward queries along the drain tree except suppressible queries (i.e., no longer necessary to be forwarded because the queries have been satisfied). Then, when answers are ready, each node sends its answers back along this routing tree.

Subgoal predicates that LogicQ is responsible to support are sensor-specific built-in predicates. Such predicates include predicates that are related to specific functions of sensor nodes' capabilities, such as *temperature(NodeID, Temp)* for sensing the temperature and *connect(NodeID1, NodeID2)* for checking connectivity, and so forth. These built-in functions are defined prior to the system installation. The Sense2P's inference engine will solve the subgoals that require sensor-specific predicates by injecting subqueries that correspond to the subgoals into the network.

In the programming model subsection, we classify queries into 4 groups. However, in this lower layer, there are only facts in sensor nodes. As a result, sensor-specific subqueries are classified into two types, one for existence checking and another for retrieving all satisfied predicates. To check an existence of a fact, every argument in this first query type is constant and the answer is only true or false (e.g., *detect(oiltank, area 70)* whereby *oiltank* and *area 70* are constant). Therefore, this type of query is not necessarily disseminated to all sensor nodes. If only one node has a fact that satisfies the query, the system does not need answers from other nodes. Conversely, the second query type requires at least one variable as an argument. For example, *detect(ObjectID, area 70)* contains a variable *ObjectID* and a constant *area70*. Hence, the system will find every possible answer of ObjectID that is detected in *area70*. It is necessary to disseminate this type of queries to all nodes.

The subquery processing algorithm for each sensor node can be written as a pseudocode in Algorithm 1. Once receiving an existence-checking query (Line 1), a sensor node checks its facts locally first whether it has a fact that satisfies the query or not (Line 2). If a sensor node has a satisfying fact, it will send an answer *true* to its parent immediately (Line 3). Given that one answer is sufficient for this query

```
if subquery is checking existence then
    if have local satisfied fact then
        send answer up to parent;
    else
        forward query to children;
    end
else if subquery is asking all Satisfied value then
    forward query to children;
    if have local satisfied fact then
        send answer up to parent;
    end
end
```

Algorithm 1: Subquery processing algorithm.

type, the replying node does not further forward the query. Otherwise, it will forward the query to its children (Line 4-5).

If the query type requires all satisfied answers (Line 6), a sensor node will forward the query immediately (Line 7). Regardless of the local existence of the satisfying facts, the system still needs satisfying answers from all sensor nodes. After the query is forwarded, the node checks for local satisfying answers. If it has one, it will send the answer up to its parent (Line 8-9).

7. Cost Analysis

In this section, we analyse the communication cost of bottom-up and top-down schemes. In all cases, we assume uniform distribution of facts in the network of M nodes. There are n subgoals in a query. $|G_i|$ represents a number of all available facts of the ith subgoal.

7.1. Cost of Bottom-Up Processing Scheme. Understandably, in the bottom-up scheme, all facts in the network must be sent to the central base station. Therefore, the total communication cost consists of rule dissemination cost, query dissemination cost, and facts retrieval cost.

The cost of rule dissemination depends on a number of subgoals because many subgoals increase the message size. For simplicity, we assume that overhead incurred by one subgoal equals to one message. Therefore, the dissemination cost of a rule with n-subgoal to M nodes in the network is

$$C_{\text{rule}} = nM. \tag{1}$$

The cost of query dissemination is obvious that a query is disseminated to all M nodes in the network. Therefore, the dissemination cost of a query is

$$C_{\text{query}} = M. \tag{2}$$

Finally, the cost of facts retrieval equals to the cost of sending all facts of each subgoal in the network to the base station. Let $|G_i|$ be a number of facts related to ith subgoal. Therefore, the facts retrieval cost is

$$C_{\text{fact}} = \overline{D}\sum_{i=1}^{n}|G_i|, \tag{3}$$

where $\overline{D}$ is an average distance from arbitrary node to the base station.

Therefore, from (1), (2), and (3), the total communication cost of the bottom-up processing scheme is

$$\begin{aligned} C_{\text{bottom-up}} &= C_{\text{rule}} + C_{\text{query}} + C_{\text{fact}} \\ &= nM + M + \overline{D}\sum_{i=1}^{n}|G_i| \\ &= (n+1)M + \overline{D}\sum_{i=1}^{n}|G_i|. \end{aligned} \tag{4}$$

7.2. Cost of Top-Down Processing Scheme. In our top-down processing scheme, we perform the join-computation process at the central base station and distributively collect only needed facts from the network.

In retrieving all satisfying answers, our scheme incurs broadcasting a subquery for each subgoal. However, only the selected facts (that satisfy the constraints caused by all previous subgoals) for that subquery are sent back to the base station. In other words, previous satisfied subgoals can suppress many unrelated facts in the network.

A rule, in this scheme, is not necessary to be disseminated into the network because the base station only disseminates a subquery of a subgoal into the network at a time. This kind of subquery is certainly a predicate that each node priori knows before deployment. Therefore, the total communication cost consists of subqueries dissemination cost and fact retrieval cost.

The cost of subqueries dissemination depends on answers of previous subgoals. These answers are used to filter irrelevant facts of the next subgoal.

Let σ_{i_1,i_2} be a selectivity factor to select facts of i_2th subgoal after solving 1st to i_1th subgoal. For example, if $\sigma_{2,3}$ equals 0.05, after solving the 1st and 2nd subgoal, only 5 percent of facts related to the 3rd subgoal are sent back to the base station. $\sigma_{-1,0}$ and $\sigma_{0,1}$ equal to 1.

The number of subqueries for the ith subgoal equals to a number of all distinct answers from the $(i-1)$th subgoal. Therefore, the number of subqueries for the ith subgoal equals to $\mu_{i-1}(\sigma_{i-2,i-1}|G_{i-1}|)$, where μ_i is a distinct factor for answers from ith subgoal and μ_0 equals to 1. For example, if there are 10 answers from several nodes but there are only 2 distinct values, the distinct factor equals to 0.2 in this case. Each subquery is disseminated to M nodes in the network. That is

$$C_{i\text{th subquery}} = M(\mu_{i-1}(\sigma_{i-2,i-1}|G_{i-1}|)). \tag{5}$$

For n subgoals, the cost of n subqueries is

$$\begin{aligned} C_{\text{subqueries}} &= \sum_{i=1}^{n} C_{i\text{th subquery}} \\ &= M\sum_{i=1}^{n}(\mu_{i-1}(\sigma_{i-2,i-1}|G_{i-1}|)). \end{aligned} \tag{6}$$

The cost of subgoal fact retrieval is similar to (3) except only a portion of facts are selected and sent back to the base station. Therefore, the subgoal fact retrieval cost is

$$C_{\text{subgoal-fact}} = \overline{D}\sum_{i=1}^{n}\sigma_{i-1,i}|G_i|. \quad (7)$$

From (6) and (7), we derive the total communication cost of the top-down processing scheme,

$$\begin{aligned} C_{\text{top-down}} &= C_{\text{subqueries}} + C_{\text{subgoal-fact}} \\ &= M\sum_{i=1}^{n}(\mu_{i-1}(\sigma_{i-2,i-1}|G_{i-1}|)) \\ &\quad + \overline{D}\sum_{i=1}^{n}\sigma_{i-1,i}|G_i|. \end{aligned} \quad (8)$$

Noticeably, our approach already includes the cost of producing all answers and the cost of sending all answers to the base station.

Note that the cost of top-down processing scheme is lower than the cost of bottom-up processing scheme when the selectivity factor is low whereas there are superfluous facts in the network. These characteristics of selectivity factor and number of facts are norm in anomaly detection applications that values exceeding defined constraints are rarely found. In the next section, we evaluate the performance of our implemented system.

8. Evaluation

To evaluate our system performance, we write and inject various queries into Sense2P that is connected to TOSSIM [25], a TinyOS simulator. For viability testing, the result is compared with that of 3 other approaches: TinyDB, bottom-up, and simplified Sense2P (a simple integration of an existing Prolog system and LogicQ without other Sense2P features). This experiment shows the impact of Sense2P features that are specifically designed for WSNs. These features include query suppressing, data filtering, and superset caching.

8.1. Performance Metrics. In this section, we use 3 metrics for performance comparison: completeness, soundness, and communication cost. Completeness is the ratio of retrieved answers to the total existing answers in the networks. This indicates whether our system can successfully retrieve all existing answers or not. Similarly, we measure the soundness by the ratio of the relevant answers retrieved to the total retrieved answers. The soundness metric is for assuring that our system does not retrieve irrelevant answers. In this evaluation, the communication cost is measured by the number of sent messages in the system. Communication cost indicates the amount of energy consumed. For a system to be viable for WSNs, the communication cost of that system must be minimized.

8.2. Simulation Environment. We implement Sense2P on TinyOS and simulate each sensor node on TOSSIM. We assume reliable communication (i.e., no packet loss because of bit errors or collisions) to discard the problem caused by radio transmissions. In our simulation, each sensor node can sense the temperature of its environment. The temperature values are randomly assigned between 20 and 80°C. We set up the simulation such that 5 percent of nodes sense the temperature value over 50°C.

8.3. Top-Down versus Bottom-Up. In this subsection, we conduct two experiments in order to compare the communication cost between the top-down evaluation of Sense2P and the traditional bottom-up evaluation. In the first experiment, we inject three temperature queries into the network of temperature sensors. The first query is for checking the existence of a predicate. All arguments in the query are constant. The second query contains one constant argument and one variable argument. In the third query, all arguments are variable. This experiment is quite simple, given that these queries are satisfied by facts, not rules.

Our message counts in answering three mentioned queries are compared with that of the bottom-up method. Regardless of the argument types, the bottom-up approach always incurs a certain amount of messages sent because all facts must be delivered to the base station (see Figure 3(a)).

Sense2P will significantly reduce the communication cost if the query contains at least one constant argument to suppress irrelevant answers. However, Sense2P will incur the communication cost similar to that of the bottom-up approach if all arguments in the query are variable. Understandably, all facts are required in order to answer such a nonconstant query under our investigated scenarios.

Nevertheless, in our simulation, we assume all nodes are equipped with the same sensor type. If the sensor nodes are heterogeneous, Sense2P will still reduce the communication cost significantly even with the nonconstant query because only relevant nodes with the matched sensing capability will send back the data, unlike the bottom-up approach that needs all facts and filters out by the inference engine at the base station.

In the second experiment, we program the rule in Listing 1 on Sense2P and inject two queries into the system: constant type and nonconstant type. Both queries must be satisfied by the mentioned rule. Expectedly, the result in Figure 3(b) indicates that our system outperforms the bottom-up approach regardless of the query types (including the nonconstant type).

However, our savings in nonconstant queries can still be improved. In our implementation, each answer from the previous subgoal is used to bind the variable in the current subgoal. The number of subqueries for the current subgoal depends on the number of answers from the previous subgoal because each answer may bind the variable with a different value.

As the number of nodes is increased, the number of answers for the previous subgoal is also increased. Consequently, the number of subqueries for the current subgoal is unavoidably increased. This causes more messages sent into larger networks. However, this problem can be solved

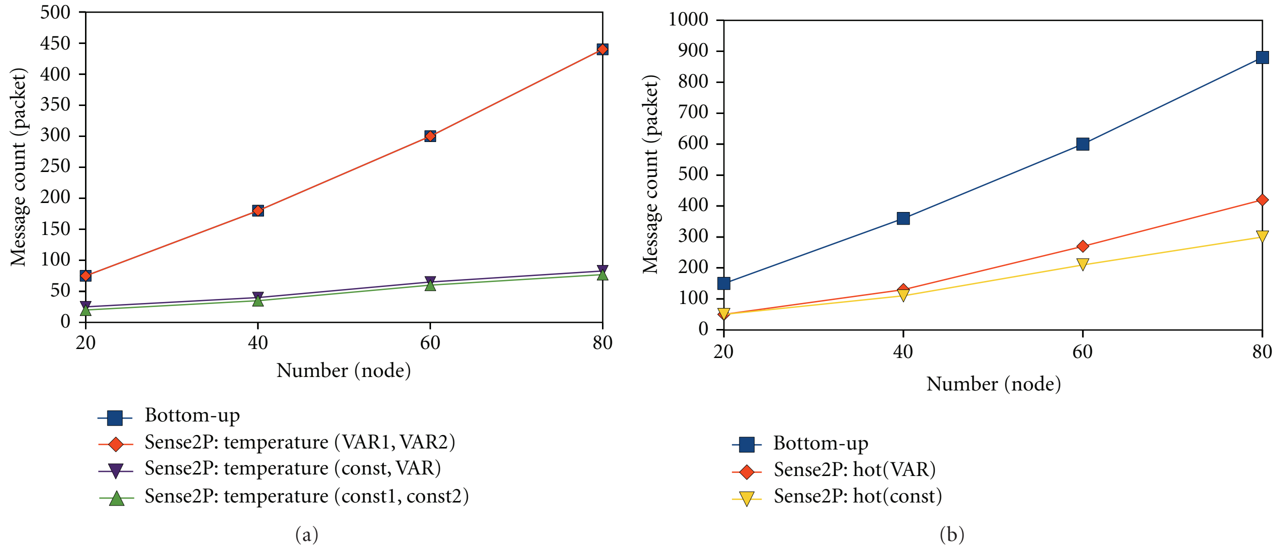


Figure 3: Comparison between traditional bottom-up evaluation and Sense2P top-down evaluation. (a) Simple predicate, temperature(NodeID, T). (b) Rule, hot(AreaID): temperature(NodeID,T), T > 50, area(NodeID, AreaID).

by sending only one subquery for the current subgoal with a list of different values that are bound with the variable. Nevertheless, we have not yet implemented this optimization in this paper. We intend to further explore this technique and other optimization approaches in our future work.

8.4. Query Suppression. In this subsection, we conduct an experiment for comparing Sense2P with TinyDB because TinyDB is also a node-independent macroprogramming paradigm. Given that TinyDB does not support many query types that Sense2P can (see Section 2), we only focus on queries that both approaches can perform.

We inject two temperature queries into the network. One is with two variable arguments for retrieving all temperature facts. Another is with two constant arguments for checking the existence of a temperature fact.

Expectedly, the message count of Sense2P is significantly smaller than that of TinyDB, especially in the constant query (Figure 4). Understandably, the efficiency of Sense2P is due to its query suppression. Sense2P suppresses (does not forward) queries that are already satisfied whereas TinyDB always sends queries to all nodes. The efficiency of Sense2P will be more evident if we measure the byte count instead of the message count, given that the message size of Sense2P is also smaller than that of TinyDB.

However, we are surprised that TinyDB sends more query messages when there are more conditions in the WHERE clause of the query. Our query with two constant arguments corresponds to a SQL query with two conditions in the WHERE clause. Evidently, TinyDB sends more query messages in the constant query than in the nonconstant query.

8.5. Data Filtering. In this subsection, we analyze the impact of data filtering on the performance of Sense2P. We program the rule in Listing 1 and inject two queries into the Sense2P system: a constant type and a nonconstant type. After that, we disable the data filtering feature in Sense2P and repeat the experiment.

Not surprisingly, Sense2P with data filtering performs better than Sense2P without data filtering in both query types (Figure 5). Data filtering is undoubtedly beneficial to the constant query. However, one may wonder how the filtering technique improves the performance of the nonconstant query. Such improvement is possible in Sense2P when the nonconstant query must be satisfied by a rule, especially the rule whose body contains a constraint to a predicate's variable argument. For example, the rule hot(AreaID), in Listing 1, contains a constraint, *Temp > 50*. In the body of the rule, the temperature predicate is the first subgoal whereas the above constraint is the second subgoal. Traditionally, a Prolog system resolves this rule from left to right. Therefore, the traditional system must retrieve every possible value of the temperature predicate before filtering the irrelevant answers with that constraint. However, in Sense2P, our runtime processing engine binds that value constraint to the temperature predicate before sending the first subquery into the network. Therefore, the number of answers for the first subgoal is reduced. Consequently, the number of messages sent in the system is also reduced.

8.6. Superset Caching. To study the impact of superset caching, we inject two set of temperature queries into Sense2P with and without superset caching. A temperature query is injected every 20 seconds in our experiment. The first 3 queries of the first set are in the form of temperature (const, VAR) where const is different for each query. The remaining queries of the first set are in the form of temperature (const1, const2). Some remaining queries are the subsets of the first 3 queries. When they are, they do not incur any radio transmission because Sense2P can search their answers in the superset caching (Figure 6(a)).

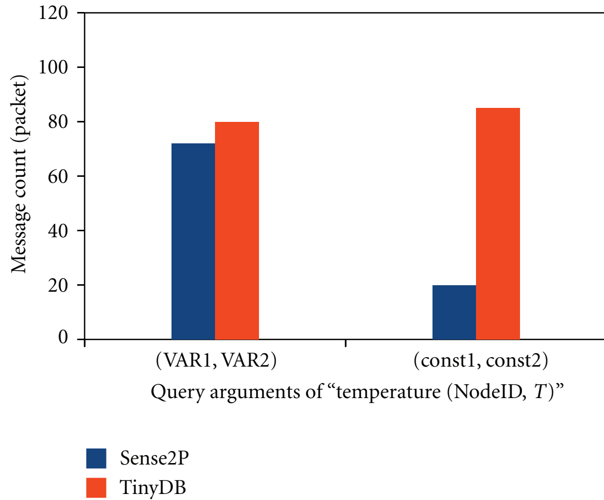


Figure 4: Comparison of Sense2P with query suppression and TinyDB with no query suppression.

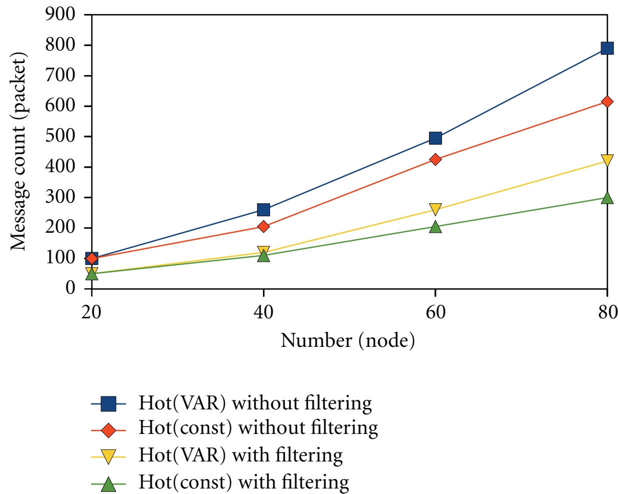
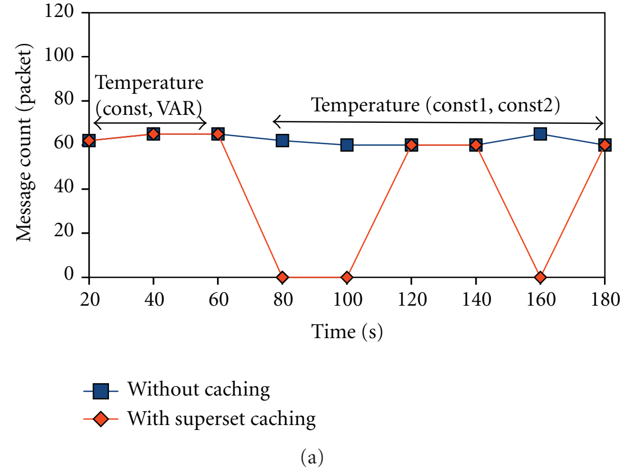


Figure 5: The impact of data filtering technique.



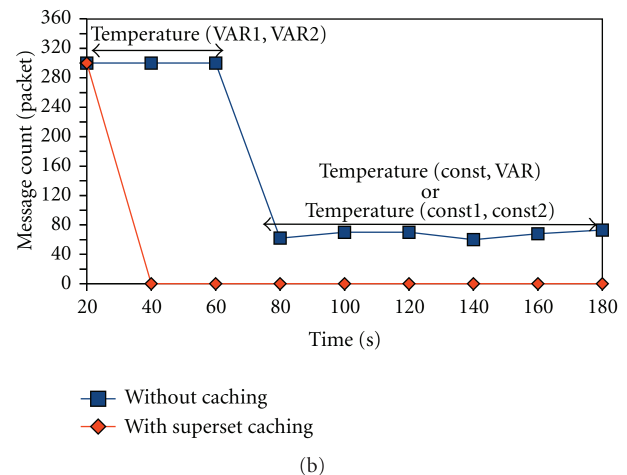


Figure 6: The impact of superset caching.

The impact of the superset caching is more significant when the superset is larger. This is evident in the second set of queries. The first 3 queries of the second set are in the form of temperature (VAR1, VAR2). The remaining queries contain at least one constant argument. Given that the first query is the superset of all remaining queries, there is no radio transmission necessary for answering these queries (Figure 6(b)). Consequently, the energy consumption is significantly reduced.

However, the superset caching has a trade-off issue. If the sensed data of the environment is frequently changed, the cached answer will be stale and useless. Thus, similar to most caching techniques, the superset caching is associated with an application-specific expiration timer or data popularity for flushing stale cached data. Understandably, the flushing rate is a trade-off between data freshness, storage size, and energy consumption. The cache replacement policy is out of scope of this work.

8.7. Completeness and Soundness. Due to the reliable communication in our experiment, there is no packet loss. Sense2P can achieve 100% completeness and 100% soundness under investigated scenarios because of our filtering and suppressing techniques. In practice, there would be some loss due to interference, collision, and congestion. However, to handle loss in the network is out of the scope of this work which mainly focuses on the programming language perspective.

9. Conclusion

This paper proposes a logic node-independent macroprogramming approach for abstracting, programming, and using WSNs as globally deductive databases. Unlike macroprograms in previous works, our logic macroprograms can be described declaratively and imperatively. Furthermore, logic macroprogrammers can easily express a recursive program or query that is unsupported in existing database abstractions for WSNs.

To efficiently process queries and their subqueries (either recursive or nonrecursive), the top-down approach is more appropriate than the bottom-up approach. This is due to

```
WITH RECURSIVE ancestor(anc, desc) AS (
  ( SELECT par AS anc, child AS desc FROM parent )
  UNION
  ( SELECT ancestor.anc, parent.child AS desc
    FROM ancestor, parent
    WHERE ancestor.desc = parent.par ) )
SELECT anc FROM ancestor WHERE desc="John"
```

Listing 4: SQL programming to solve Ancestors problem and Listing.

```
ancestor(anc, desc):- parent(anc, desc).
ancestor(anc, desc):- parent(anc, X), ancestor(X, desc).
?-ancestor(anc, "John").
```

Listing 5: Logic programming to solve Ancestors problem.

its capability to bind subgoal arguments that can be used to reduce communication cost by suppressing irrelevant answers and already satisfied subqueries from being sent or forwarded.

Finally, our evaluation results indicate that Sense2P can significantly reduce energy consumption while maintaining 100% completeness and soundness under our investigated scenarios.

Appendix

Recursive Query in SQL Language and Logic Programming Language

In this section, we describe the verboseness of a SQL language in expressing a recursive query. We also show the conciseness of a logic programming language in expressing the same recursive query for comparison.

To simplify the comparison, we use a well-known recursive query example: the Ancestors problem. Given a set of parent(par, child) relations, we would like to find all ancestors of "John".

The SQL program in Listing 4 demonstrates how to solve this problem.

Concisely, we can solve the same problem in logic programming as in Listing 5.

The logic program is so much shorter and easier to express. The advantage will be even more if the problem is more complex.

Acknowledgments

The authors would like to thank Rawin Youngnoi and UbiNET research group members for their valuable suggestions and effort to improve this work. This work was supported by the Thailand Research Fund (TRF) under Grant MRG5080449 and the CU CP Academic Excellence Scholarship from Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University.

References

[1] C. Gui and P. Mohapatra, "Power conservation and quality of surveillance in target tracking sensor networks," in *Proceedings of the 10th Annual International Conference on Mobile Computing and Networking (MobiCom '04)*, pp. 129–143, ACM, New York, NY, USA, 2004.

[2] T. He, S. Krishnamurthy, J. A. Stankovic et al., "Energy-efficient surveillance system using wireless sensor networks," in *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services (MobiSys '04)*, pp. 270–283, ACM, New York, NY, USA, 2004.

[3] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson, "Wireless sensor networks for habitat monitoring," in *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications (WSNA '02)*, pp. 88–97, ACM, New York, NY, USA, 2002.

[4] N. Xu, S. Rangwala, K. K. Chintalapudi et al., "A wireless sensor network for structural monitoring," in *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys '04)*, pp. 13–24, ACM, New York, NY, USA, November 2004.

[5] D. Gay, P. Levis, E. Brewer, R. Von Behren, M. Welsh, and D. Culler, "The nesC language: a holistic approach to networked embedded systems," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '03)*, pp. 1–11, ACM, New York, NY, USA, June 2003.

[6] D. Chu, L. Popa, A. Tavakoli et al., "The design and implementation of a declarative sensor network system," in *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems (SenSys '07)*, pp. 175–188, ACM, New York, NY, USA, 2007.

[7] R. Gummadi, N. Kothari, R. Govindan, and T. Millstein, "Kairos: a macro-programming system for wireless sensor networks," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pp. 1–2, ACM, New York, NY, USA, 2005.

[8] C. Intanagonwiwat, R. K. Gupta, and A. Vahdat, "Declarative resource naming for macroprogramming wireless networks of embedded systems," in *ALGOSENSORS, Lecture Notes in Computer Science*, vol. 4240, pp. 192–199, Springer, 2006.

[9] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: an acquisitional query processing system for sensor networks," *ACM Transactions on Database Systems*, vol. 30, no. 1, pp. 122–173, 2005.

[10] R. Newton, G. Morrisett, and M. Welsh, "The regiment macroprogramming system," in *Proceedings of the 6th International Conference on Information Processing in Sensor Networks (IPSN '07)*, pp. 489–498, ACM, New York, NY, USA, 2007.

[11] M. Welsh and G. Mainland, "Programming sensor networks using abstract regions," in *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation (NSDI '04)*, p. 3, USENIX Association, Berkeley, Calif, USA, 2004.

[12] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler, "Hood: a neighborhood abstraction for sensor networks," in *Proceedings*

of the 2nd International Conference on Mobile Systems, Applications, and Services (MobiSys '04), pp. 99–110, ACM, New York, NY, USA, 2004.

[13] A. Krishnamurthy, D. E. Culler, A. Dusseau et al., "Parallel programming in split-C," in *Proceedings of the ACM/IEEE Conference on Supercomputing (Supercomputing '93)*, pp. 262–273, ACM, New York, NY, USA, November 1993.

[14] C. Borcea, C. Intanagonwiwat, P. Kang, U. Kremer, and L. Iftode, "Spatial programming using smart messages: design and implementation," in *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS '04)*, pp. 690–399, IEEE Computer Society, Washington, DC, USA, 2004.

[15] Y. Yao and J. Gehrke, "The cougar approach to in-network query processing in sensor networks," *SIGMOD Record*, vol. 31, no. 3, pp. 9–18, 2002.

[16] C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed diffusion: a scalable and robust communication paradigm for sensor networks," in *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking (MobiCom '00)*, pp. 56–67, ACM, New York, NY, USA, 2000.

[17] I. Bratko, *Prolog Programming for Artificial Intelligence*, Addison-Wesley Longman Publishing, Boston, Mass, USA, 1986.

[18] K. Whitehouse, F. Zhao, and J. Liu, "Semantic Streams: a framework for composable semantic interpretation of sensor data," in *Wireless Sensor Networks*, vol. 3868 of *Lecture Notes in Computer Science*, pp. 5–20, Springer, 2006.

[19] F. Bancilhon and R. Ramakrishnan, "An amateur's introduction to recursive query processing strategies," *SIGMOD Record*, vol. 15, no. 2, pp. 16–52, 1986.

[20] Y. K. Hinz, "Datalog bottom-up is the trend in the deductive database evaluation strategy," Tech. Rep. INSS 690, University of Maryland, 2002.

[21] K. Ramamohanarao and J. Harland, "An introduction to deductive database languages and systems," *The VLDB Journal*, vol. 3, no. 2, pp. 107–122, 1994.

[22] S. Choochaisri and C. Intanagonwiwat, "A system for using wireless sensor networks as globally deductive databases," in *Proceedings of the IEEE International Conference on Wireless & Mobile Computing, Networking & Communication (WIMOB '08)*, pp. 649–654, IEEE Computer Society, Washington, DC, USA, 2008.

[23] R. Ramakrishnan and S. Sudarshan, "Top-down vs. bottom-up revisited," in *Proceedings of the International Logic Programming Symposium*, pp. 321–336, MIT Press, 1991.

[24] P. Levis, S. Madden, J. Polastre et al., "Tinyos: an operating system for sensor networks," in *Ambient Intelligence*, Springer, 2004.

[25] P. Levis, N. Lee, M. Welsh, and D. Culler, "Tossim: accurate and scalable simulation of entire tinyos applications," in *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys '03)*, pp. 126–137, ACM, New York, NY, USA, 2003.