EnviroSuite: An Environmentally Immersive Programming Framework for Sensor Networks

LIQIAN LUO and TAREK F. ABDELZAHER
University of Illinois at Urbana-Champaign
TIAN HE
University of Minnesota
and
JOHN A. STANKOVIC
University of Virginia

Sensor networks open a new frontier for embedded distributed computing. Paradigms for sensor network programming in the large have been identified as a significant challenge towards developing large-scale applications. Classical programming languages are too low-level. This paper presents the design, implementation, and evaluation of *EnviroSuite*, a programming framework that introduces a new paradigm, called *environmentally immersive programming*, to abstract distributed interactions with the environment. Environmentally immersive programming refers to an object-based programming model in which individual objects represent physical elements in the external environment. It allows the programmer to think directly in terms of environmental abstractions. EnviroSuite provides language primitives for environmentally immersive programming that map transparently into a support library of distributed algorithms for tracking and environmental monitoring. We show how nesC code of realistic applications is significantly simplified using EnviroSuite, and demonstrate the resulting system performance on Mica2 and XSM platforms.

1. INTRODUCTION

This paper presents *EnviroSuite*, the first sensor network programming framework for environmentally immersive programming. The need to facilitate software development for sensor networks motivates new high-level abstractions for programming-in-the-large. These abstractions must hide the details of distributed monitoring and

tracking algorithms, capture the unique properties of these networks such as their distributed interactions with a physical environment, and address the issue of scale.

Traditional programming languages such as Java and C, as well as their sensor network adaptations, such as nesC [Gay et al. 2003] and its extension galsC [Cheong et al. 2003; Cheong and Liu 2005], are too low-level. Their basic computation, communication and actuation unit is typically the sensor node. Programmers must think in terms of single node activities and explicitly encode interactions among nodes. For example, programmers are exposed to reading sensing data from appropriate sensor devices, aggregating data pertaining to the same external stimulus, deciding where to send it, and communicating with actuators if needed. If the monitored activity moves in the environment, programmers are responsible for spatial and temporal correlation of measurements obtained about the activity across a changing set of sensor nodes, and associating that data with event progress.

A more desirable approach would be for the programmers to encode only overall network behavior, leaving it to the underlying system to decompose such behavior into node-level algorithms. Examples of higher-level abstractions that address this concern include logical-node-based primitives [Gummadi et al. 2005], token-based languages [Newton et al. 2005], database-centric abstractions [Madden et al. 2003; Yao and Gehrke 2002; Madden et al. 2005], event-based systems [Li et al. 2004], group-based primitives [Blum et al. 2003; Whitehouse et al. 2004; Welsh and Mainland 2004; Liu et al. 2003], state-centric approaches [Liu et al. 2003] and virtual machines [Levis and Culler 2002; Boulis et al. 2003]. These paradigms offer logical nodes, tokens, queries, events, sensor node groups, and logical state, respectively, as the underlying abstractions with which the programmer operates.

EnviroSuite is an object-based programming system. Its abstractions revolve directly around elements of the environment as opposed to sensor network constructs such as regions, neighborhoods, or sensor groups. The existence of the sensor network is thus made more transparent. EnviroSuite is different from other object-based systems in that its objects are representations of elements in the external environment. Dynamic object instances are created automatically by the run-time system when the corresponding external elements are detected and are destroyed when these elements leave the network. A unique mapping between object instances and the corresponding environmental elements is maintained by the system. Object instances float across the network following (geographically) the elements they represent. The execution of object code at the location of the corresponding physical element is ideal for sensing and actuation tasks. Objects encapsulate the aggregate state of the elements they represent, making such state available to their methods. These objects (as opposed to the individual nodes) are therefore the units that encapsulate program data, computation, communication, sensing and actuation. Classical objects (that do not represent any environmental elements) are also supported. We call the above model, *environmentally immersive programming* (EIP).

This paper presents the first comprehensive design and implementation of an environmentally immersive programming framework. EnviroSuite abstractions are supported by an underlying library called *EIPLib*, which is implemented in nesC on TinyOS [Hill et al. 2000], an operating system designed specifically for sen-

sor networks. We evaluate EnviroSuite and several applications written in it on TOSSIM [Levis et al. 2003] as well as on a mote-based sensor network. TOSSIM is an emulator that runs the actual nesC service code, emulating on a PC the behavior of programs on the Berkeley motes [U. C. Berkeley 2005]. The framework extends a previous tracking middleware service by the authors, called EnviroTrack [Abdelzaher et al. 2004], which introduced a network address space where representations of environmental entities are the addressable objects.

Finally, two remarks are in order on what EnviroSuite is *not*. First, EnviroSuite is not a replacement to other emerging programming paradigms such as group-based primitives, database-centric abstractions, event-based systems, and virtual machines. This paper does not argue for a single approach to the exclusion of others. The most appropriate abstractions are often a personal choice that depends on subjective programmer preferences as well as application specifics. Ultimately, it is the availability of multiple programming alternatives that induces more software development. EnviroSuite is therefore presented and evaluated for its own merits, and not as a substitution for other high-level paradigms.

Second, EnviroSuite is not a programming language in itself. EnviroSuite is a framework that extends other programming languages with environmentally immersive programming primitives. This extension takes two different forms. First, the programmer is allowed to define and use variables that summarize elements of a potentially distributed environmental state (such as the average temperature of a region or the current location of a moving target). Second, the programmer may define code that is geographically distributed and associate the time and place of its execution with the occurrence of certain environmental events. Both the aggregate variables and distributed code are encapsulated within simple objects. As with other distributed computing paradigms, remote communication is allowed between objects. The purpose is to abstract the distributed aspects of environmental interactions and computation.

With distribution hidden from the programmer, logical computation can be performed using the native programming language. The current implementation of EnviroSuite extends nesC. However, there is nothing in its design and general abstractions that is nesC specific. The implementation can be easily re-targeted to support other programming languages. nesC was chosen due to its wide popularity in the sensor network community and due to the availability of a compiler for the current mote hardware.

The remainder of the paper is organized as follows. Section 2 introduces the overall architecture of the EnviroSuite framework. Section 3 provides a detailed description of the exported abstractions. Section 4 presents the design of the essential algorithms underlying these abstractions. Section 5 highlights the implementation details. Section 6 presents and analyzes performance evaluation results. Section 7 discusses related work. Section 8 concludes the paper.

2. SYSTEM ARCHITECTURE

EnviroSuite lets the programmer think in terms of objects in the external environment that are relevant to the application. An environmental object may refer to a localized entity (such as a moving vehicle) or a distributed region of the en-
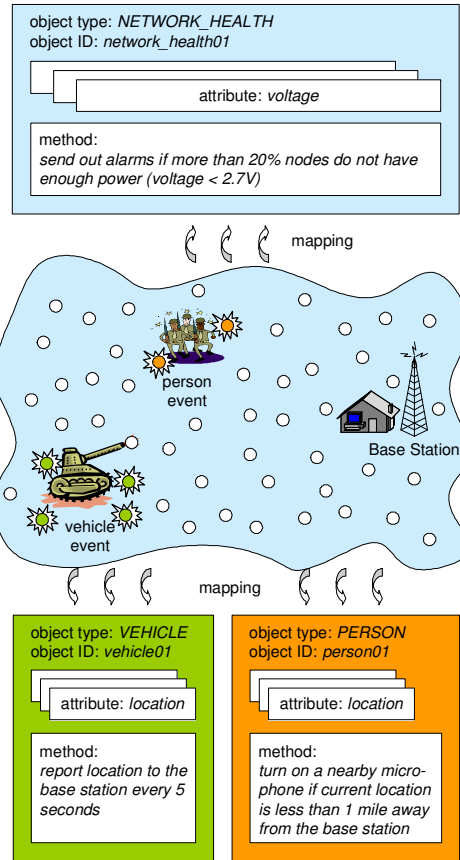
Fig. 1.   One-to-one mapping between physical events and event objects

vironment (e.g., a geographic area or an area specified by some sensory signature such as high temperature). Typically, the system must keep some state or other information about each object. This state is maintained in variables encapsulated within the object that can be accessed using object methods. Hence, each object is given by (i) a sensory or geographic signature that defines its boundaries or location, (ii) a set of data variables to be collected in its vicinity, and (iii) a set of methods that can be performed in its context. The fact that the obtainment of values stored in the encapsulated variables and the execution of local methods may need distributed computation (such as data aggregation) is hidden from the programmer. The programmer may also define regular objects that are not linked to objects in the environment. We call them *function objects*. Environmental objects and function objects seamlessly coexist in the programmers' world and can invoke each other's methods using a remote object invocation mechanism. An example of such mapping is depicted in Figure 1.

The example in Figure 1 represents a surveillance application that monitors vehicle and person movement in a hostile territory (e.g., behind enemy lines). Each tracked vehicle or person is mapped into a dynamically instantiated object with
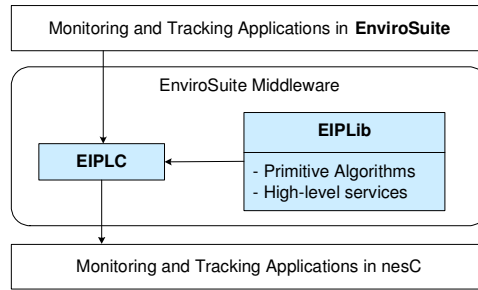
Fig. 2. Relation of EnviroSuite, EIPLC & EIPLib

a unique label, denoted by an object ID in EnviroSuite. Desired event attributes such as location can be returned for the object. This application also periodically monitors the health of the network by collecting information on nodes that are alive and their remaining power. The network is thus mapped into an object that maintains aggregate health statistics. Computation, communication and actuation can be logically attached to these objects. Examples include reporting vehicle location by vehicle objects, turning on microphones in their vicinity for tracking purposes, or sending out alarms if system health fails to meet an acceptable threshold.

These primitives are supported by the environmentally immersive programming library (*EIPLib*), which provides a series of algorithms containing the detailed implementations (such as sensor data processing, object maintenance, and inter-object communication) and some other higher level services. A compiler (*EIPLC*) is introduced to translate EnviroSuite applications into nesC. The relation among EnviroSuite, EIPLC and EIPLib is depicted in Figure 2.

Programmers design and implement environmental monitoring and tracking applications using a combination of EnviroSuite and nesC. Taking such implementations as input, the compiler (EIPLC) configures and restructures services and protocols in EIPLib to automatically produce as output the corresponding implementations in the nesC language. The resulting code can be compiled on TinyOS and uploaded to the motes. In the following sections, we describe in more details the abstractions of EnviroSuite, the services and protocols provided in EIPLib to support these abstractions, and the translation of these abstractions carried out by EIPLC.

3. ENVIRONMENTALLY IMMERSIVE PROGRAMMING SYNTAX

When a programmer develops a monitoring application using EnviroSuite, the programmer creates a virtual world with a set of logical objects, which attempts to reflect the real world with a set of physical objects. Each object is defined by a sensory signature such that contiguous groups of nodes that satisfy that signature will be given a unique object ID. These object IDs constitute a global name space available to the programmer. Various data operations can then be performed on different locales defined by the corresponding object IDs. These operations are typically coded as methods encapsulated in the corresponding objects. Observe that EnviroSuite is only concerned with (i) grouping together nodes that satisfy programmer-defined sensory signatures, (ii) giving global names to those groups,
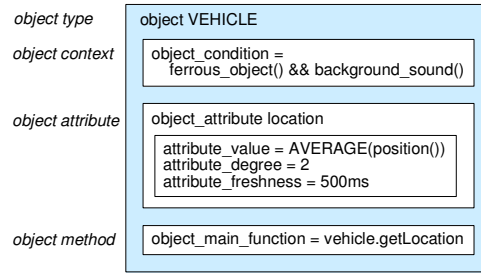
Fig. 3.   Object declaration of object VEHICLE

(iii) executing programmer-defined data operations within each named group, and (iv) storing programmer-defined group state in variables encapsulated within the named objects. The correspondence between the groups and meaningful aspects of the environment is the programmer's responsibility. For example, there is inherent uncertainty regarding whether or not a motion and magnetic signature defined by the programmer truly signals the presence of a vehicle. EnviroSuite is concerned only with tracking the defined signature. The uncertainty in the interpretation of the signature must be handled at a higher-level.

Syntactically, an EnviroSuite program consists of a list of object declarations such as the one shown in Figure 3. Each declaration specifies a user-defined object *type*, an object *condition* statement, declaration of object *attributes*, and the object *methods*.

The object condition statement creates a mapping between the software object and the corresponding environmental element. For example, it can specify the sensory signature of an external tracked entity, or a geographic area defining a physical region. An object is created for each contiguous region where the object condition is true. A contiguous region is one that is not partitioned. In other words, there exists a path between any two nodes in the region that has no intermediate hops outside the region. We call this region the object *context*. A null object condition specifies that this object is not a representation of an environmental element (e.g., a pure computational object), which is called a function object, as mentioned above.

Specifications of object contexts are followed by declarations of encapsulated data to store the state of the object. Such data typically refers to aggregates of sensory measurements or node attributes over the object context. They can be thought of as query results over the context. The declaration specifies the method of aggregation together with confidence and freshness parameters. Finally, as in traditional object-based systems, an object *main* function specifies the name of a default constructor method to be automatically executed when the object is created. Other methods can be defined to be executed when called. Object methods can access the attributes of their encapsulating object and perform remote method invocations on other objects.

Objects are instantiated either statically or dynamically. The former is useful to represent fixed environmental elements such as topological features of the terrain. The latter is useful, for example, to represent dynamically arriving targets in the

Table I. Keywords for basic EnviroSuite object declaration

Object Context	`object_condition`
Object Attributes	`object_attribute`
	`attribute_value`
	`attribute_degree`
	`attribute_freshness`
Object Methods	`object_main_function`
	`object_function`

environment. As described later in this paper, special care is taken to ensure unique representation (i.e., that a single object is instantiated to refer to a single target, even though the target causes multiple sensor hits).

EnviroSuite keywords for basic object declarations are listed in table I. More detailed discussion on EnviroSuite object contexts, attributes, and methods is presented in the following subsections respectively, using the object declaration example depicted in Figure 3.

3.1 Defining the Object Context

In EnviroSuite, the object condition statement defines the object context, which is the continuous region where the object condition is true. EnviroSuite includes a library of sensor data processing algorithms (called the *condition library*) designed by domain experts for purposes of defining object contexts. These algorithms return (possibly) filtered or otherwise processed sensor outputs (e.g., `temperature()`), or identify specific boolean environmental conditions (e.g., `ferrous_object()` or `vehicle_sound()`), or return node attributes (e.g., `position()` or `voltage()`). A boolean expression of such conditions can then define the region of object context. We call it the object condition statement. For example, the following declaration defines the condition that represents the potential presence of a vehicle:

```
object_condition = ferrous_object() && vehicle_sound();
```

In this example, `ferrous_object()` is a function that returns true when the magnetometer output indicates a significant disturbance to the earth magnetic field (consistent with the passage of a large ferrous object), and `vehicle_sound()` indicates microphone output of energy and pitch consistent with the sound of a passing vehicle. Implementation of such functions is described in [Gu et al. 2005].

The idea is to compile a library of such conditions to abstract the specifics of sensor processing away from the programmer in much the same way device drivers abstract the details of device I/O away from application code. The separation between high-level application code and low-level sensor processing comes at the cost of increased condition library size, since many different algorithms need to be written to identify a sufficient range of useful conditions for each sensor type. This is not unlike the proliferation of device drivers (one for each version of every possible device) in contemporary operating system installations. The success of device drivers as a means for separating concerns leads one to believe that the condition library will considerably simplify application development in sensor networks. An object executes when and where the conditions defined in its condition statement become true.

Observe that conditions can also be parameterized. For example, the condition:

```
object_condition = altitude()>500 && temperature()<32;
```

defines the region (i.e., object context) that satisfies freezing temperatures on top of a local hill. The case of `object_condition = NULL` specifies a function object not associated with an environmental element (region or physical entity).

3.2   Defining Object Attributes

The main purpose of objects invoked in response to environmental conditions such as those mentioned above is usually to monitor attributes of environmental events, targets or regions. These attributes are measurements collected and aggregated by nodes in the object context. Specification of attributes requires specification of (i) the sensor measurements in question, and (ii) optionally, their method of aggregation. Aggregation is always performed over all nodes within the object context. The sensor measurements to be aggregated could be any environmental measurements, or node attribute measurements such as remaining battery power or node position, for which a measurement function exists in the condition library described above. A library, called the *aggregation method library*, is supplied, which lists a set of aggregation methods such as `AVERAGE`, `MAX` and `RANGE` on attributes. For example, to define an aggregate attribute, `targetLocation`, EnviroSuite programmers can simply specify the corresponding node measurement, `position()`, from the condition library, and the name of the appropriate aggregation method, say `AVERAGE`, from the aggregation method library, in an object attribute clause, such as:

```
object_attribute targetLocation {
    attribute_value = AVERAGE(position());
}
```

Within the declaration of an attribute, EnviroSuite allows the programmer to specify the minimum aggregation degree, `attribute_degree`. The aggregate attribute is valid only when it is the aggregation result from at least as many nodes as `attribute_degree`. This knob allows programmers to control the confidence in retrieved information. The feature is especially useful in reducing false alarms. Another important property of attributes is freshness. Most monitoring applications have temporal data validity constraints. Usually, stale information is of no use. EnviroSuite allows programmers to define `attribute_freshness`, which determines how often aggregate attributes are to be sampled and updated by the mechanisms that compute them in EIPLib.

3.3   Defining Object Methods

Sensor network applications can have more complex functionality than merely monitoring attributes. In general, computation, communication or actuation could be encapsulated into the definition of an object. EnviroSuite tries to make full use of existing general-purpose languages, such as nesC, and their existing modules, such as those exported by TinyOS, by separating real object method implementation from object declaration. In object declaration, EnviroSuite programmers are required to denote the name of functions implementing in a general language the object methods. Such functions can use the EnviroSuite communication primitives

(using keyword `ES_IOC` and `ES_IOCRESULT`) and read values of encapsulated aggregate attributes of the object (using keyword `ES_GETATTRIBUTE`). The separation of object method declaration and object method implementation retain independence of EnviroSuite abstractions from the underlying programming language.

There are two types of object methods that can be encapsulated within an object. Those object methods specified in `object_main_function` statements are functions which will be automatically executed upon the creation of the corresponding object. In contrast, object methods specified in `object_function` statements will be executed only when they are called by other objects. Assuming programmers choose nesC as the general language to implement object methods, the following clause specifies that the implementation of the main object method can be found in nesC command `getLocation` within interface `vehicle`.

```
object_main_function = vehicle.getLocation;
```

To facilitate communication and coordination beyond the scope of one object, we introduce a RPC-like mechanism in EnviroSuite, called the *Inter-Object Call* (IOC). IOC is different from traditional RPC in several aspects. First, both the caller and the callee of IOC can be migrating across nodes as the location of the external object changes. Such migration is transparent to programmers, who simply specify the callees instance name (to be stated below) and never worry about which physical nodes these objects are located on. Second, IOC is asynchronous. Callers do not block themselves to wait for results. Instead, results declare their arrivals by interrupts. The keyword for IOC is `ES_IOC` and `ES_IOCRESULT`. The former is used for executing an IOC and declaring its handler and the latter for receiving IOC result interrupts. All object methods defined in an object can be remotely called by any other objects by using its reference. The underlying low-level communication protocol and routing extensions to support IOC have been previously published in [Blum et al. 2003] and are thus not described in this paper.

3.4 Defining Static Object Instances and Global Variables

The above discussion covered declaration of object types. Objects that represent fixed environmental elements, such as topological features of the terrain, can be statically instantiated. These static instances can be used, for example, as the destinations of IOCs that invoke object methods. Object types that do not have static instances will be instantiated dynamically at run-time when their object conditions become true. They would have to send their handle to any other objects that need to communicate with them.

EnviroSuite also allows programmers to define globally shared static variables in (static) object declarations and to access defined static variables in object method implementation by using EnviroSuite keyword, `ES_READ` and `ES_WRITE`.

The next section gives a complete tracking application implemented in EnviroSuite, including code samples for static instances and static variables.

3.5 A Tracking and Monitoring Application in EnviroSuite

A typical tracking and monitoring application written in EnviroSuite (and some nesC) is shown in Figure 4. The main function of this application is to estimate the current location of a tracked vehicle, update the estimates every 500 ms and report

Object Declarations

```
1.   object VEHICLE {
2.       object_condition = ferrous_object()&&vehicle_sound();
3.       object_attribute location {
4.           attribute_value = AVERAGE(position());
5.           attribute_degree = 2;
6.           attribute_freshness = 500ms; }
7.       object_main_function = vehicle.getLocation; }
```

```
8.   object NETWORK_HEALTH {
9.       object_condition = TRUE;
10.      object_attribute energyLevel {
11.          attribute_value = voltage();
12.          attribute_freshness = 20m; }
13.      object_main_function = networkHealth.getEnergyLevel; }
14. static NETWORK_HEALTH networkHealthInstance;
```

```
15. object MONITOR {
16.     object_condition = NULL;
17.     object_main_function = monitor.start;
18.     object_function = monitor.reportLocation;
19.     static int vehicleNumber = 0; }
20. static MONITOR monitorInstance;
```

Implementations of Object Methods

object method implementation of object VEHICLE

```
21. Triple_float_t *currentLocation;

22. command result_t vehicle.getLocation() {
23.     call ES_WRITE(monitorInstance.vehicleNumber,
                        monitorInstance.vehicleNumber +1);
24.     return call Timer.start(TIMER_REPEAT, 500); }

25. event result_t Timer.fired() {
26.     currentLocation = call ES_GETATTRIBUTE(location);
27.     ES_IOC report = call monitorInstance.monitor.
                             reportLocation(currentLocation);
28.     return SUCCESS; }

29. ES_IOCRESULT report(bool result) {
        //deal with remote call results here
30.     return; }
```

object method implementation of object NETWORK_HEALTH

```
31. uint16_t currentEnergyLevels[MAX_NODE_NUMBER];

32. command result_t networkHealth.getEnergyLevel() {
33.     return call Timer.start(TIMER_REPEAT, 1200000); }

34. event result_t Timer.fired() {
35.     currentEnergyLevels = call ES_ATTRIBUTE(energyLevel);
        //deal with obtained node IDs and voltage values here
36.     return SUCCESS; }
```

object method implementation of object MONITOR

```
37. command result_t monitor.start() {
38.     return SUCCESS; }

39. command bool monitor.reportLocation(Triple_float_t
        Location) {
        //deal with received target location here
40.     return TRUE; }
```

Fig. 4.   An EnviroSuite application

the estimated location to the base station every 500 ms. The total number of vehicles is counted. At the same time, voltage values for individual nodes are collected every 20 minutes to obtain system health information. This application illustrates the main abstractions supported by the framework, as well as the programming style.

The application declares three object types VEHICLE, NETWORK_HEALTH and MONITOR which refer to a dynamically instantiated object, a geographic region object

and a function object, respectively (lines 1 - 20).

For object type `VEHICLE`, the `object_condition` statement (line 2) specifies its sensory signature as `ferrous_object()` and `vehicle_sound()`. The `object_attri-bute` statements (lines 3 - 6) define an aggregate attribute `location` for which the value is the average of positions of more at least 2 nodes, updated every 500 ms. The `object_main_function` statement (line 7) states that main object method implementation can be found in interface `vehicle` that includes command `getLocation`.

For object type `NETWORK_HEALTH`, the `object_condition` statement (line 9) specifies its object context as `TRUE` to include all nodes in the network. The `object_attri-bute` statements (lines 10 - 12) define an attribute `energyLevel` as the voltage values of individual nodes with an update rate 20 minutes. The `object_main_function` statement (line 13) defines that main object method is command `getEnergyLevel` in interface `networkHealth`, which obtains an array of node IDs and voltage values. Line 14 creates a static instance `networkHealthInstance` for object type `NETWORK_HEALTH` so that it will be instantiated statically in system initialization and IOCs can be made through this reference.

For object type `MONITOR`, the `object_condition` statement (line 16) specifies `NULL` as the object context since the object is not mapped to any environmental element. The `object_main_function` statement (line 17) specifies the command `start` in interface `monitor` as the main object method. Finally, the `object_func-tion` statement (line 18) defines that command `reportLocation` in interface `moni-tor` can be remotely called by any other objects by using IOC and its static instance `monitorInstance` (line 20). Line 19 defines a static variable `vehicleNumber` which is globally accessible by any object through `ES_READ` and `ES_WRITE`.

In the object method implementation of object `VEHICLE`, it is defined that static variable `vehicleNumber` is increased by one whenever a new instance of `VEHICLE` is created (line 23). For each instance, every 500 ms (line 24) the current value of aggregate attribute `location` is fetched (line 26) and sent to the base station by using `ES_IOC` (line 27) to remotely call method `monitor.reportLocation` located in static instance `monitorInstance`. In line 29, `ES_IOCRESULT` keyword is used to receive IOC interrupts of `ES_IOC report`. The interrupt handler name must be the same as `ES_IOC` which is `report` and the parameters should be of the same type as the returned value of remote called method `reportLocation` which is `bool`.

In the object method implementation of object `NETWORK_HEALTH`, it is defined that every 20 m (line 33) the current values of individual voltages are collected (line 35) and analyzed (not included) to monitor system health.

The object method implementation of object `MONITOR` includes the implementation of its constructor method `monitor.start` (lines 37 - 38) and its exported method `monitor.reportLocation` (lines 39 - 40).

This application is used as a running example throughout this paper. It is compiled and evaluated on an actual sensor network as well as on TOSSIM.

4. OBJECT MAINTENANCE ALGORITHMS

To support EnviroSuite abstractions, the main question is how physical state, events, and activities can be uniquely and identically mapped into objects despite of distribution and possible mobility in the environment. This section gives extensive

answers.

While all objects in EnviroSuite have the same declaration syntax and programming interface, underneath the common API, EnviroSuite supports three different implementations of objects, namely, *event objects* (created for mobile events defined as those that dynamically change their geographical locations), *region objects* (mapped to static or slowly moving regions), and *function objects* (not mapped to an environmental element).

To alleviate the programmers burden, EnviroSuite can automatically determine the best category for each object based on the keywords used in the `object_condition` statement. Conditions defined in terms of volatile measurements (such as motion sensing) typically give rise to dynamic contexts with rapidly changing node membership, which are more appropriately implemented as event objects. In contrast, conditions defined in terms of slowly changing measurements (such as temperature) result in more stable groups that can be implemented as region objects. Taking sensor type into account therefore allows the compiler to make intelligent guesses about the most appropriate group management protocols to use for object implementation. The programmer is allowed (although not required) to annotate the object as event or region object, overriding the compilers intelligent guess. An incorrect annotation, however, will result in impaired performance. Function objects are similar to region objects, except that they do not interact with the physical environment. In the following, we describe the three different object maintenance protocols, which determine how and when to form the object context, what group management protocols are involved, where to execute object code, and how to compute object attributes.

4.1   EVENT OBJECT MAINTENANCE

Typically, event objects are created dynamically in response to environmental events that may be mobile and usually fast moving. (A compile-time warning is generated if a static instance is declared for such objects.) In the current implementation and in the discussion below, only localized events are supported. By a localized event, we mean those with a geographically limited sensory signature, such as moving vehicles. We call such localized events, *targets*. Supporting events with a large signature that move quickly is challenging because of the high overhead. However, we do support slowly moving large-signature events as described in region objects.

The core component of our event object implementation is the *multi-target group management protocol* (MGMP). When the condition statement of an event object evaluates to true in a new contiguous region, MGMP creates a new globally unique address, *object ID*, and associates it with the geographically contiguous group of sensor nodes which sense the environmental event. The movement of the contiguous region associated with the event results in dynamic changes to group membership. The protocol ensures that the same object ID is maintained for the event object despite mobility and membership changes, so that it can always be addressed via its uniquely assigned object ID. Dynamically created event objects are aware of their ID and must explicitly send it to other objects if they want to be contacted. Observe that the internal details of MGMP are transparent to the programmer. From the perspective of application code, the only visible effect of MGMP is the
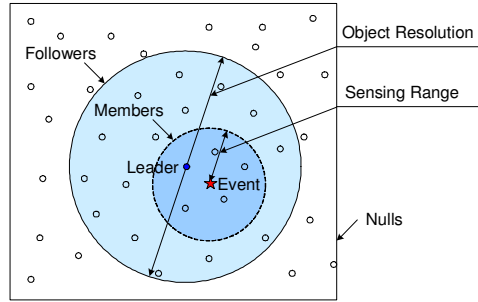
Fig. 5.   States of nodes around an environmental event

dynamic creation and deletion of object instances (in response to environmental conditions). These instances encapsulate aggregate object attributes as defined by the programmer.

Internally, MGMP elects a leader in each object context to maintain a persistent and unique object ID, collects raw data from group members in the context, performs aggregation functions on the leader to compute object attributes, and coordinates computation and actuation tasks as defined in object methods.

In the following, we discuss how MGMP maintains object uniqueness (one-to-one mapping of external events to logical objects) and object identity (immutability of the mapping function) for fast moving targets.

4.1.1   *State Machine Representation.* MGMP treats each node as a state machine. The sensor network around an environmental event might have the state distribution shown in Figure 5. It should be noted that although we use circles to indicate sensing areas, we do not assume sensing areas are circular.

All nodes sensing an event constitute the *member* set. A single *leader* is elected by MGMP among the member set. The leader sends periodic *heartbeats* to nodes within half an `object_resolution` (default half is two times the sensing range) away from itself to claim its leadership and to inform them of the existence of the event. Note that the sensing range can be statically derived from the sensor characteristics, and, if the event is detected by a combination of multiple sensors, the shortest one is used. Heartbeats are disseminated through limited flooding, and later on, members communicate to the leader through reverse paths of flooding. The period of these messages, called the *heartbeat period*, is one of the key parameters of MGMP. As we show in the evaluation section, this period can be chosen automatically by EnviroSuite from a high-level specification of the maximum abject creation latency.

All nodes that cannot sense the event themselves but know of its existence through received nearby leader heartbeats are said to be in the *follower* state. All MGMP control messages are transmitted to nodes within half `object_resolution` away from senders. Thus, half the `object_resolution` must be no less than two times the sensing range, since nodes within the same sensing area must communicate with each other to agree on a single leader. The minimal tolerable object resolution in EnviroSuite is therefore four times the sensing range.

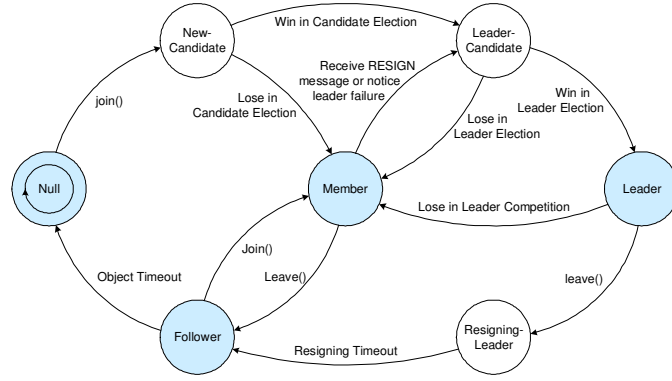At any point of time, a node stays at a single state from a set of states S_s:

Fig. 6.   State machine in MGMP

$$S_s = \{Null, Follower, Member, NewCandidate, LeaderCandidate,$$
$$Leader, ResigningLeader\}$$

To make MGMP suitable for sensor devices with limited computation and storage ability, we allow each node except leaders to maintain only one object/object ID to reduce algorithm complexity both in time and space. For instance, a node cannot act as member of two different objects/object IDs. Figure 6 depicts the general state machine algorithm of MGMP without providing details of associated objects/object IDs.

4.1.2   *Maintaining Object Uniqueness.* Object uniqueness can be compromised in several cases. The first is at the time when a new event causes the creation of a new object. Multiple object IDs for one event may be created since there is no agreement on a single leader initially. To solve the problem we employ a *delayed object creation* mechanism, which delays the creation of a new object by an amount called the *candidate period*, until we are of high confidence that the group of nodes has elected a single leader node. In this mechanism, null nodes, when sensing an event, transit their states to newCandidate and begin to send periodic CANDIDATE messages at the heartbeat period, containing sequence numbers and their own node ID. To prolong system lifetime, instead of using the fixed heartbeat period, we can enhance energy balancing by using a dynamic period inversely proportional to remainder energy of nodes. Hence, nodes with a higher energy will become candidates first and will have a higher chance of being elected. In the case of re-transmissions, candidates with a higher energy can back-off less, hence having a higher chance of successfully claiming leadership. The node with the smaller sequence number or, if sequence numbers are equal, with the bigger node ID is forced to quit from the newCandidate state and transition to state member. This procedure is called *candidate election*, which finally results in only one node at the newCandidate state. After a given delay (namely, the candidate period) this node transits to the leaderCandidate state. The candidate period is measured in the number of periodic CANDIDATE messages sent before one newCandidate node can

transit to the `leaderCandidate` state. The candidate election algorithm ensures a single `leaderCandidate` in the absence of message loss. Even if messages can be lost, by increasing the candidate delay, a single `leaderCandidate` can be generally guaranteed since the possibility of consecutive message loss is small. In the evaluation section, we determine a good choice for the candidate delay, such that the programmer need not be involved in the decision.

The next problem that compromises uniqueness occurs during leader re-election. When tracked events move out of the current leaders sensing ranges, these leaders must handover their leadership to other nodes, which is called *object migration*. Object migration, especially frequent object migration caused by fast moving events, challenges the maintenance of object uniqueness. MGMP solves this problem by introducing the `follower` state. Through heartbeats from leaders, `follower` nodes know in advance the event objects associated with incoming events. When these nodes come to sense these events, they join the existing objects as `member` instead of creating spurious objects. It was shown in [Abdelzaher et al. 2004] that this mechanism is successful in maintaining object uniqueness as long as object velocity is below some maximum limit.

The third case that challenges object uniqueness is when multiple events of same signatures become closer than defined object resolution, or even cross each others path. To simplify the situation, we assume that event crossing does not coincide with event disappearance. In the previous cases without event crossing, the delayed object creation mechanism and the introduction of the `follower` set ensures object uniqueness. Here, we need only to prevent accidental object termination during event crossing, so that object uniqueness is maintained. The leadership handoff mechanism used in MGMP prevents object termination as long as the object is maintained by one `leader` node and at least one `member` node. Thus, the key in maintaining object uniqueness during event crossing is to balance `member` nodes between merging objects to assign at least one `member` for each object, which is detailed below.

To show the *member balancing* mechanism, Figure 7 depicts part of the state machine, which describes how `member` nodes choose their corresponding objects. $Member_x$ denotes `member` state with object ID x. A simple way to balance Member nodes is to divide `member` nodes based on leader position. When a `member` node receives heartbeats from multiple objects, it chooses to join the one with the nearest leader since there is a higher possibility that this node is sensing the same event as that leader. However, such division is not accurate since leader positions are not identical to event locations. It is also possible that a `member` node is actually sensing the same event as the farther leader. For this reason, a new state called `freeMember` is introduced into the state machine. The continuous reception of n continuous heartbeats from `object 2` can transit $member_1$ to `freeMember` and then to $member_2$ even if the last heartbeat was from a closer leader (`object 1`). The introduction of `freeMember` allows wrong choices to be corrected, thus ensuring correctness of member balancing. The member balancing mechanism prevents object termination successfully, therefore enhancing object uniqueness in the third case.

4.1.3 *Maintaining Object Identity.* While object uniqueness refers to maintaining a single object representation for each external target, maintaining object iden-
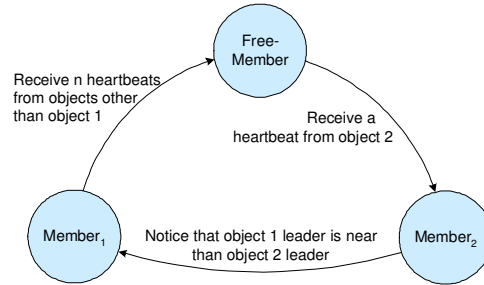
Fig. 7.   Member balancing mechanism

tity refers to keeping the *correct* association between external targets and their representing objects. In the case where events with same signatures are closer than one object resolution, an extra mechanism is required to maintain identity since member balancing only ensures object uniqueness. EnviroSuite makes the default extra assumption that targets tend not to change direction abruptly. This assumption, for example, allows disambiguation of crossing targets based on their path. The default assumption can be customized by programmers if needed.

EnviroSuite keeps a record of the recent trajectory of each target (storing it within its representing object). To reduce system overhead, instead of using position information of group members to estimate target locations, EnviroSuite takes leaders positions as an approximation. When transferring leadership, each leader also transfers its maintained history of the last $n-1$ old leaders positions plus its own. When two events E_1 and E_2 cross each others path, each object leader is able to receive the heartbeat from the other. Each object leader marks itself by the concatenation of the old object ID (O_1) and the new object ID (O_2) as its temporary object ID (O_1O_2). The two leaders exchange their event trajectories such that each remembers both. After separation, a disambiguation algorithm is used, based on recorded history and current locations to chose the ID assignment most consistent the default (straight path) assumption.

4.2   REGION AND FUNCTION OBJECT MAINTENANCE

Region object maintenance differs from event object maintenance since region objects are associated with a relatively fixed set of nodes. What we implement for region object maintenance is a spanning-tree based information collection structure described in [He et al. 2004]. Like event objects, the details of region object maintenance are transparent to the programmer. The application code is only aware of the object and its encapsulated aggregate attributes.

When a region object is initialized (statically at system deployment time or dynamically, depending on whether a static instance is declared), a default leader node disseminates tree construction requests to the object context with a running hop-count initialized to zero. Requests are flooded outward with hop-count incremented at every intermediate hop. After receiving tree construction requests, nodes establish multiple reverse paths towards the sending nodes. As a result, a multi-parent diffusion tree is constructed with the leader residing at the root. Spanning tree construction stops when nodes are reached that do not satisfy the region object

condition statement. Such nodes become the outer boundary of the tree and serve a role similar to followers in event objects. If these nodes ever satisfy the condition statement they become members and recruit other followers for which the statement is not satisfied. Also, if tree leaves cease to satisfy the object condition, they truncate themselves from the tree and become outer boundary nodes. Hence, membership of the tree can change slowly over time. Measurements needed to compute object attributes can flow up the tree from members towards the leader and get aggregated along intermediate hops. We do not provide details of aggregation algorithms here, since similar mechanisms have been described in previous literature such as directed diffusion [Intanagonwiwat et al. 2000] and TAG [Madden et al. 2002]. Our contribution lies in the uniform programming abstractions presented on top of such mechanisms.

One aspect where our region object maintenance algorithm differs from previous work is that we automatically migrate the root of the aggregation tree to the location that minimizes communication and aggregation overhead, as well as to a higher energy node, periodically escaping energy depleted regions. This load balancing flexibility is made possible in our programming model since we implement the program inside the network, alleviating external bottlenecks. After each migration, a (possibly partial) tree reconstruction is done to form a new spanning-tree rooted in the new host node.

The introduction of region objects enables EnviroSuite to support not only tracking functions, but also region monitoring functions such as contour finding and system health monitoring, thus making EnviroSuite applicable to a broader set of applications.

Function objects are quite similar to region objects except that there are no object contexts and object attributes in function objects. There is no need for object context maintenance and object attribute collection since function objects do not interact directly with the physical environment. In EnviroSuite, the leader of a function object always migrates to the gravity center of all other objects which have recently communicated with the function object through IOC or global variable access.

Like in event objects, leaders in region objects and function objects are responsible for object method execution.

5. IMPLEMENTATIONS IN NESC

In this section, we take nesC, the most popular language in sensor network area, as the general language that implements EnviroSuite. EnviroSuite object declarations (defined by programmers) and object methods (assumed to be written in nesC by programmers) are to be automatically translated by EIPLC into a whole nesC application by selecting and integrating primitive algorithms provided in EIPLib. This section describes how we design EIPLib to simplify the work of EIPLC and how we implement the compiler EIPLC with the help of EIPLib. Although the implementation details are specific to nesC, most design decisions we make in this section are portable to other languages.

All nesC applications consist of a set of *components*. A component provides and uses *interfaces*, as defined in the components *provides* and *uses* clauses. An interface

describes the parameters of a set of *commands* and *events*. There are two types of components: *modules* and *configurations*. Modules provide application code, implementing one or more interfaces. Configurations connect interfaces used by components to interfaces provided by other components. The action of connecting component interfaces is called component *wiring*. It is the main mechanism for building large applications from smaller modules. Wiring is done at compile time, and offers no run-time overhead. We use wiring extensively to connect application components to components implemented by our language libraries.

5.1   EIP Service and Protocol Library (EIPLib)

EIPLib contains a series of primitive algorithms to be used by EIPLC to build comprehensive applications in nesC, currently including: sensor data processing algorithms (condition library), aggregation algorithms (aggregate method library), object maintenance algorithms and inter-object communication protocols. It also contains higher level services as potential consumers of primitive algorithms, including: object context determination components, object attribute collection components and object method execution components.

Each condition such as `temperature()` and `vehicle_sound()` is associated with a sensor data processing algorithm in EIPLib, which returns processed sensor outputs either as a meaningful value or a boolean either immediately or in a phase-splitting way. However, the association and ways of accessing are hidden in EIPLC and programmers are only aware of available condition names and their purposes. Also, each aggregation method such as `AVERAGE` is associated with an aggregation algorithm which implements the method. In the current version, object maintenance algorithms contain separate implementations for three object categories: event objects, region objects and function objects. As stated in the beginning of Section 4, the object categories are transparent to programmers and are determined by EIPLC based on the `object_condition` statement. (Advanced APIs are provided for sophisticated programmers to override default rules.) Inter-object communication protocols provide supports for maintaining links between dynamic objects, which is required to implement IOCs and global variable access. All these primitive algorithms are implemented as nesC components with standard interfaces.

Object context determination components determine whether the current node should join some object context based on object declarations. Object attribute collection components collect raw object attributes from member nodes, apply aggregation methods to form aggregate attributes in leader nodes, and support access to aggregate attributes. Object method execution components are responsible for executing object methods in leader nodes whenever corresponding objects exist. These higher level components are also implemented in the form of nesC components, yet differ from usual nesC components in many ways, including:

(1) They are not pre-wired since object declarations and object method implementations are not available until compile time. Wiring is left for the compiler so that primitive algorithm components may be freely selected and wired into higher level components to construct any EnviroSuite applications defined by programmers.

(2) They contain special clauses that are recognizable only by the compiler. In
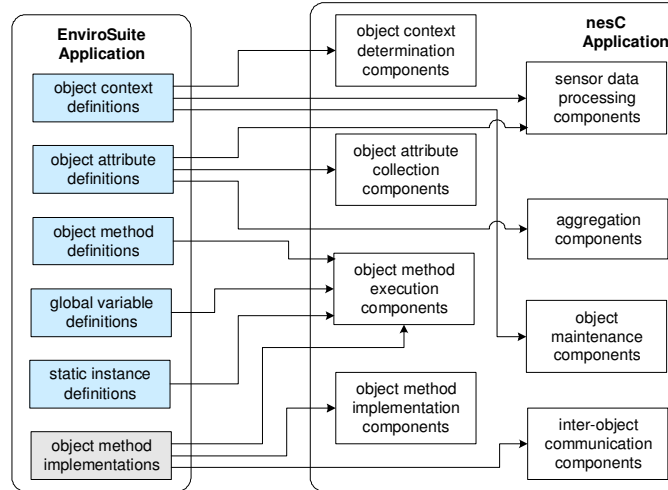
Fig. 8.   Translate an EnviroSuite application into a nesC application

many cases, such clauses are necessary to guide language translation. For example, the configuration of object context determination components may include a special clause ({__ES_COMPONENTS}) which indicates the position where necessary sensor data processing components are to be listed by EIPLC. These clauses greatly simplify the implementation of EIPLC by giving some hints.

The hierarchical structure between primitive algorithms and high level components is also critical. In this structure, various configurations can be achieved by changing only the high level components while other components can remain unchanged, thus reducing the complexity of the compiler.

5.2   EnviroSuite Compiler (EIPLC)

EIPLC is essentially a translator that takes EnviroSuite code as input and outputs desired environmental monitoring applications in nesC, which then can be compiled by a standard nesC compiler and uploaded to the motes. EIPLC is implemented in Perl, a language with powerful built-in support for text processing. Current implementation of EIPLC contains 1533 lines.

EnviroSuite application code consists of two parts, object declarations and object method implementations. The detailed translation of both parts is illustrated in Figure 8.

EIPLC analyzes object declarations line by line, making corresponding configurations and integrations. As depicted in Figure 8, for object context definitions, EIPLC identifies all conditions, locates the corresponding sensor data processing components by searching condition library, which lists all condition names and the corresponding implementations, wires them into object context determination components. A feature of EIPLC is that it automatically determines the best category for each object based on these conditions and integrates the corresponding object maintenance algorithms.

For object attribute definitions, besides identifying conditions and wiring corre-

sponding components into object attribute collection modules, EIPLC also wires
aggregation components. Additional work includes setting attribute refreshing
timers based on `attribute_freshness` definition and validating resulted aggregate
attribute based on `attribute_degree` definition. Based on object method defini-
tions, EIPLC wires the implementations into object method execution components.
EIPLC also copies global variable definitions into object method execution com-
ponents and enables remote access to global variables by implementing local read
and write commands, which respond to received remote calls. For each static ob-
ject instance, EIPLC randomly selects a node as the default leader, which initially
executes the main object function, and migrate the leader to a more power-efficient
position later.

EIPLC also filters object method implementations for keywords, translating
`ES_GETATTRIBUTE` into command calls to object attribute collection components
and `ES_IOC`, `ES_IOCRESULT`, `ES_READ` and `ES_WRITE` into command calls and event
handlers of inter-object communication components.

As seen above, EIPLC successfully bridges between low-level implementations
in EIPLib and high-level abstractions exported by EnviroSuite by making several
intelligent steps that are transparent to the programmers: selecting sensor data
processing algorithms; automatically identifying object categories and applying cor-
responding maintenance algorithms; and automatically collecting and aggregating
attributes from multiple nodes.

Observe that, one clause in an EnviroSuite application may result in multiple
changes in higher level components, and one higher level component from EIPLib
may be changed multiple times by multiple EnviroSuite clauses, which means
EIPLC may need to change the same file in EIPLib repeatedly. Considering such
phenomenon, instead of creating corresponding new code line by line, we store the
resulting changes in a hash of hashes, so that already changed code can be further
changed easily. The hash of hashes stores, for each file and each special clause such
as {`_ET_COMPONENTS`}, their corresponding nesC code. Only after analyzing the
entire EnviroSuite application, EIPLC changes files from EIPLib based on the re-
sulted hash of hashes. The storage space needed by the hash of hashes may be very
large. However, we consider it acceptable since EIPLC runs on a PC and therefore
does not have severe storage constraints.

6. PERFORMANCE EVALUATION

This section provides a detailed quantitative analysis of EnviroSuite. We begin by
evaluating the performance of a series of micro-benchmarks on simulators, which
analyze the primary features of EnviroSuite: object uniqueness and identity main-
tenance, and inter-object communication support. The first set of benchmarks
tests object uniqueness and identity management during object creation, object
migration and object crossing (which is the most challenging case). The second set
of benchmarks tests inter-object communication. We then move to real platforms
to evaluate the performance of a surveillance system built using the EnviroSuite
framework. Both tracking performance and monitoring performance are evaluated
to demonstrate event objects and region objects. The evaluated system is the one
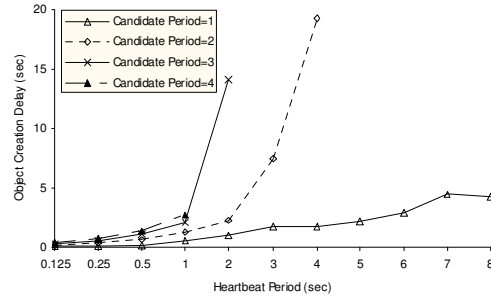described in Section 3. Its abbreviated code is given in Figure 4.

Fig. 9. Object creation delay for varied heartbeat period and candidate period

6.1 Performance of Object Operations

To evaluate the performance of primitive object operations, we choose TOSSIM since EnviroSuite produces real nesC code for motes and TOSSIM can emulate the execution of the real code on the motes without the need for deployment. The radio model simulated in TOSSIM is almost identical to the 40 Kbit RFM-based stack on the motes. To control per-hop message loss at the packet level we added an external program component. We focus on fast-moving objects (event objects), since their real-time maintenance offers the most challenge to the EnviroSuite infrastructure.

In our emulated experiments, we set the sensing range to 100 feet (approximately 30 meters). Current sensor devices such as the micropower impulse radar [Azevedo and McEwan 1996] can detect objects up to 50 meters away. Radio range is set to 300 feet. Current sensor network products such as the Mica2 and Mica2Dot motes [U. C. Berkeley 2005] have a maximum outdoor radio range of 500 feet to 1000 feet under ideal conditions when sending with full power. Sensor nodes are placed on a grid 100 feet apart.

6.1.1 *Experiment 1 - Object Creation.* EnviroSuite associates a logical object with each physical event. It is critical that such association should be done as soon as possible to reduce the inconsistency between the physical world and the logical world exported by EnviroSuite. In the first experiment, we measure object creation delay, defined as the difference between the time the first node senses an external stimulus and the time an object ID is created for the corresponding object. The external entity tracked, in this case, is a vehicle. The tracking code is given in Figure 4.

The delay of object creation is decided by both the candidate period, which indicates how many candidate messages must be sent before creating objects, and the heartbeat period, which determines candidate message intervals. In the following we show the experimental data that allow these parameters to be selected automatically by EnviroSuite from a high-level specification of the maximum tolerable object creation delay. Figure 9 plots object creation delay versus heartbeat period for different candidate periods.

From Figure 9 we observe that object creation delay increases with the increase in both the candidate period and the heartbeat period. The plots show only those points for which a non-zero number of objects are created. A candidate period of 1 performs best in terms of object creation delay. However, it is undesirable since
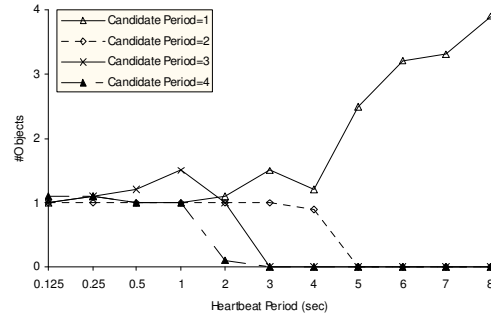
Fig. 10.   Number of objects created for varied heartbeat period and candidate period

it causes spurious objects at higher heartbeat periods as stated below.

The candidate period and the heartbeat period affect not only object creation delays but also object uniqueness. Figure 10 shows the impact of the candidate period and the heartbeat period on object uniqueness by plotting the average number of created objects. Ideally, only one object should be created per experiment, since the only target is deployed.

Figure 10 shows that with shorter heartbeat periods, candidate periods 2, 3 and 4 perform similarly. However, longer candidate periods result in a longer object creation delay, so that when the heartbeat period exceeds a certain threshold, objects cannot be formed in time before the vehicle moves out of their sensing ranges. Figure 10 shows that for candidate period 4, it is difficult to create objects after the heartbeat period exceeds 2 seconds, while for candidate period of 2, objects can be created up to a heartbeat period around 5 seconds. We therefore choose 2 as the default candidate period in EnviroSuite. A longer candidate period should be chosen in the presence of message loss.

Given the default candidate period (of 2), the object creation delay can be chosen anywhere from a small fraction of a second to multiple seconds depending on the choice of heartbeat period, as shown in Figure 9. The programmer should therefore specify a maximum tolerable value of object creation delay. This specification stems easily from application domain knowledge. For example, in a vehicle tracking application, a delay of 1-2 seconds between vehicle entry into the field and the creation of a corresponding event object is quite tolerable. EnviroSuite then uses Figure 9 to compute the corresponding heartbeat period. Observe that a smaller heartbeat period implies more communication, more energy consumption, and consequently a shorter lifetime. Hence, a trade-off exists between system responsiveness (object creation delay) and lifetime.

6.1.2   *Experiment 2 - Object Migration.* The core part of EnviroSuite is to uniquely and identically map physical events to logical objects. In this experiment, we reveal how fast object migration could be performed without breaking object uniqueness and identity. Object migration is caused by the movement of associated events. Hence, from the perspective of applications, the velocity limit of object migration is more meaningfully expressed by the maximum tolerable event velocity. It is defined as the maximum velocity of events, which can be uniquely and
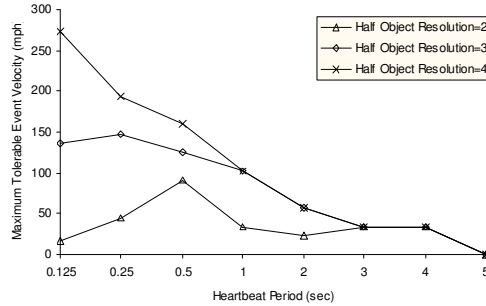
Fig. 11. Maximum tolerable event velocity for varied heartbeat period and object resolution

identically mapped to logical objects. Observe that for a given maximum object migration speed (in hops per second), the corresponding maximum event velocity depends on the radio range (distance per hop). The data presented below is for the range parameters mentioned in Section 6.

We explore several factors, which affect maximum tolerable event velocity, including heartbeat period and object resolution (in multiples of sensing range). As is shown in Figure 11, the maximum tolerable velocity increases when the heartbeat period decreases, since a shorter heartbeat period results in a shorter leader re-election delay and thus a higher trackable velocity. This trend is reversed when heartbeat period becomes short enough to cause message loss or congestion as shown in Figure 11 (for a half object resolution of 2 sensing ranges) when heartbeat period falls below 0.5 s. Increasing object resolution has positive impact on the maximum tolerable velocity since a bigger set of followers allows the vehicle to go farther without causing new object creation. Similar results were reported in [Abdelzaher et al. 2004]. We stress, however, that results reported in [Abdelzaher et al. 2004] were obtained from algorithm simulation in GloMoSim. In contrast, results presented in this paper test the performance of actual nesC code generated by our functional EIPLC compiler for the application in Figure 4.

Next, we evaluate how robust the object uniqueness guarantee is against message loss during object migration. TOSSIM does not provide message loss models at the packet level. Thus, we add a simple external program to control per-hop packet loss ratio. Figure 12 depicts the average number of objects formed per run as a function of target velocity in the presence of different degrees of packet loss. As before, the ideal number should be 1 object per run.

From Figure 12, we see that EnvoroSuite can completely tolerate a 10% loss ratio since we get similar results to those with 0% loss ratio. EnviroSuite can also tolerate a loss ratio of up to 30% when event velocity does not exceed 68 mph. Larger velocities or loss percentages, however, cause spurious objects to emerge. Observe that at a very high event velocity, the number of formed objects decreases again, which might seem like an anomaly. The explanation lies in that very high speed objects do not have enough time to form in the first place.

6.1.3 *Experiment 3 - Object Crossing Performance.* Next, we explore the efficacy of EnviroSuite in maintaining object uniqueness and identity when two objects of the same sensory signature (i.e., fulfill the same object_condition statement)
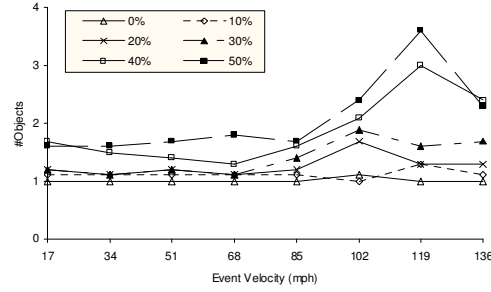
Fig. 12.   Number of objects created for varied event velocity and per-hop packet loss ratio
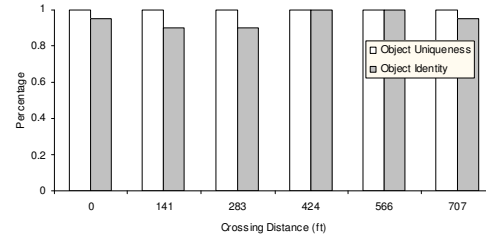

Fig. 13.   Achieved object uniqueness (white) and identity (shaded) for varied crossing distance

cross paths. In this experiment, two vehicles are moving straight along crossing diagonals with the same speed of 24 mph. The diagonals cross in the center of the field. However, these objects may not start at the same time, and hence may not reach the crossing point together. We vary their relative start times to vary the shortest distance reached between the two objects at the crossing point (which we call, the *crossing distance*). We show the percentage of runs where object uniqueness and identity are maintained as a function of crossing distance. As shown in Figure 13, object uniqueness and identity are ensured in most cases even when the two targets cross the center point at the same time (crossing distance is 0).

Each bar in Figure 13 represents the average of more than 10 runs. The tracked trajectory for one run with crossing distance 0 is shown in Figure 14. After passing the center point, although object identity is lost for a while, the system successfully recovers from the confused state after accumulating enough history.

The results prove the relative success of our adopted direction disambiguation algorithm. It also shows that defensive programming is advisable. While we elevate the level of abstraction to that of objects representing environmental elements, the programmer should expect such objects to be occasionally confused. The application code may chose to implement its own disambiguation on top of EnviroSuite object IDs.

6.1.4   *Experiment 4 - Inter-object Communication.* Programming for communication and coordination between objects becomes very simple by using IOC and global variables. In this experiment, we evaluate a vehicle counting application, which counts the total number of vehicles in a global variable, to analyze the performance of inter-object communication. As seen in the code from Figure 4, whenever
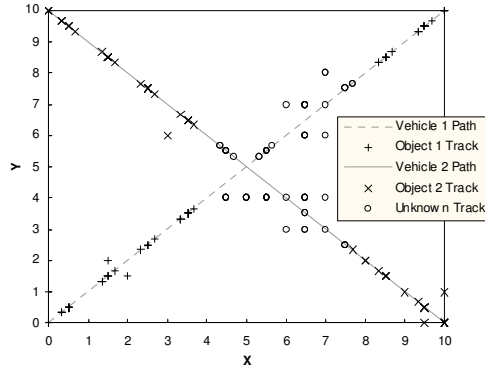
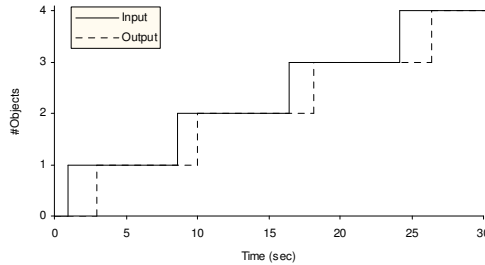Fig. 14. Reported target tracks with crossing distance 0

Fig. 15. Vehicle counting application results

a vehicle appears, the global variable `vehicleNumber` is increased by one through an `ES_WRITE` call from the corresponding `vehicle` object.

In this scenario, four vehicles enter the coverage field one by one, maintaining the same speed of 35 mph and thus the same distance. The first one goes straight from $(-1, 1)$ to $(16, 1)$; the second from $(-5, 5)$ to $(16, 5)$; the third from $(-9, 9)$ to $(16, 9)$; the last from $(-13, 13)$ to $(16, 13)$.

Figure 15 plots the counter values as a function of time in this application. Input represents real numbers of vehicles. Output represents the counting results achieved by the application. Delay between the input and output curves represent the end-to-end performance of remote object invocation. These delays reflect the sum of object creation delay and inter-object communication delay.

6.2 A Surveillance System

Finally, we test the complete surveillance application written in EnviroSuite, described in Section 3. This surveillance system tracks all in-field vehicles, counts their number and monitors system health at the same time. The EnviroSuite code of this application can be translated by EIPLC into a nesC application. Emitted nesC code size of different services in the translated application is listed in Table II.

Table III compares EnviroSuite code and emitted nesC code of the same application in terms of module number, code length and size. The code size of the nesC version gives a good estimation of required programming effort if the whole system

Table II.   Services and code sizes of the nesC application translated from the EnviroSuite application

Service Name	Code Size (KB)
Sensing Data Processing	10.8
Event Object Maintenance	25.6
Region and Function Object Maintenance	28.5
Inter-object Communication	15.0
Other Service (Aggregation, etc.)	25.1
Object Method Components	6.0

Table III.   Code Comparison of EnviroSuite Version and nesC Version

	Module Number	Code Length (lines)	Code Size (KB)
EnviroSuite Version	3	218	5.9
nesC Version	12	3692	111.0

is to be programmed directly in nesC. As is seen from Table III, the code size of the nesC version is more than ten times of that of the EnviroSuite version. Thus, the estimated programming effort with EnviroSuite is roughly an order of magnitude less. The result reflects the efficiency of EnviroSuite compared with node-based languages, such as nesC.

6.2.1   *Tracking Performance.* In this experiment, we evaluate the efficiency of EnviroSuite in terms of tracking performance and power consumption by comparing it to a simple baseline. This baseline is to plot the trajectory of a tracked target at a base station located in $(0,0)$. In the EnviroSuite implementation, members, who are sensing the target, report to the current leader their own positions every 0.5 seconds. The leader aggregates these positions and reports the average to the base station twice per second. The baseline has a simple implementation of the same application. Each node that senses the target sends its own position to the base station every 0.5 seconds. The base station averages received positions twice per second. In both the EnviroSuite version and the baseline, a minimum aggregation degree of 2 is enforced to reduce false alarms.

The actual testbed for this experiment consists 40 Mica2 motes laid out in a 10×4 grid with integer (x, y) coordinates ranging from $(0,0)$ to $(9,3)$. The goal is to track a rectangular object, 1 square grid in size, moving straight along the middle of the longer axis, with a speed of 0.5 grid per second. This testbed does not take into account errors in localization and time synchronization services. To ensure enough tracking accuracy for real applications, we require that localization errors not exceed half grid and time synchronization errors be kept in the order of ms. Many existent techniques support such precision.

Figure 16 compares the target trajectory obtained by the EnviroSuite application to the one resulting from the baseline. Some tracking error is seen because our sensor devices have no notion of proximity to the target. As shown in Figure 16, the EnviroSuite version has a smaller average tracking error compared with the baseline although it introduces a little more variability. The underlying reason is that in the baseline, position reports from nodes may not be in order when they arrive at the base station, since they may have traversed multiple hops, which results in more inaccuracy.

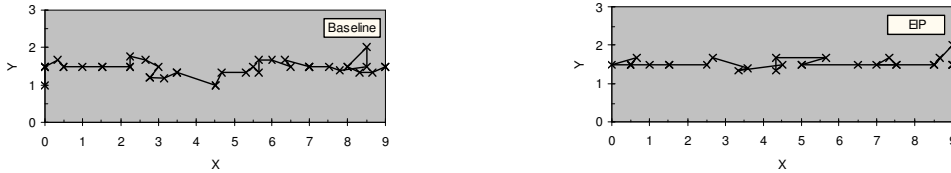Figure 17 depicts the number of packets sent or forwarded by each node in a slice
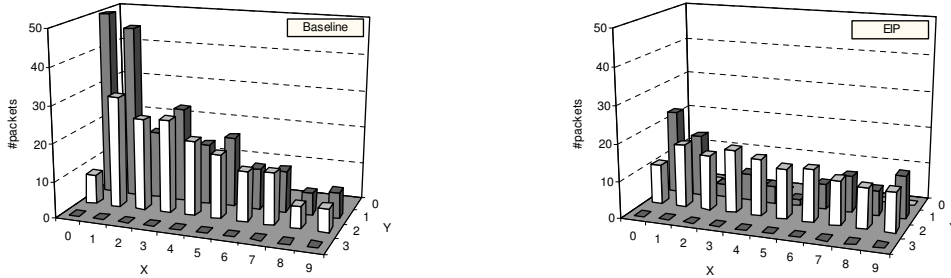
Fig. 16. Tracked target trajectory comparison

Fig. 17. Transmitted packet number comparison

of the network over the duration of the experiment (where X, Y is the coordinate of each node). Each bar in this figure represents the average of 15 runs to ensure a statistical significance at the 0.05 level. The number of packets is important as it is proportional to power consumption. It can be seen that the EnviroSuite version achieves its comparable tracking performance with much less power consumption in terms of the number of transmitted packets. Hence, our tracking algorithms are more energy-efficient.

In the baseline test, most packet transmissions occur on nodes with Y coordinates between 1 and 2 since only these nodes can forward the packets to the base station. The nodes with smaller X coordinates in the baseline send much more packets than those in the EnviroSuite version since each node sensing the target sends packets directly to the base station located in $(0,0)$. Hence, a greater number of packets have to be forwarded by nodes with smaller X coordinates. In the EnviroSuite version, position reports are aggregated locally by leaders, amounting to much fewer packets forwarded to the base station.

6.2.2 *Monitoring Performance.* In this experiment, we utilize the `NETWORK_-HEALTH` object coded in Figure 4 to monitor the health of the network by collecting information on nodes that are alive and their remaining power. Alarms will be sent out if a big portion of the network is dead or lacks power.

We carry out this experiment on a network of 27 XSM motes [Dutta et al. 2005] deployed in a grassy field. The system performs the function of vehicle tracking as well as health monitoring. For system health monitoring, the `NETWORK_HEALTH` object is determined as a region object by EIPLC, thus a multi-parent spanning tree is automatically constructed at object initialization to collect power information of each node every 20 minutes. The system is tested for several hours. Figure 18 depicts the collected power information, where the black bars represent initial voltage
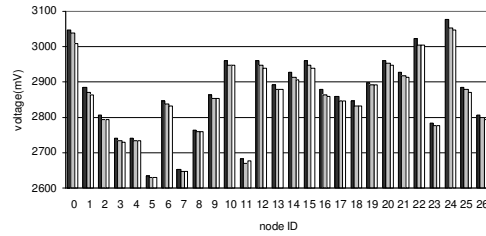
Fig. 18.   Power level of each node for different times

reported by the region object, the grey ones show voltage reported after 20 minutes and the white ones shows voltage reported after 40 minutes. Node 0 is the base node, which consumes the most power.

7.   RELATED WORK

EnviroSuite opens a new category of distributed programming paradigms. It differentiates itself from traditional paradigms such as CORBA [Vinoski 1997], Microsoft's COM [Microsoft 1994], and remote procedure calls [Birrell and Nelson 1984] by combining within its programming abstractions objects and events in the physical world.

Several communication and programming models have been proposed for sensor networks in recent years. These include node-based languages, virtual machines, database-centric abstractions, event-based models, and group-based primitives. EnviroSuite is different in that its abstractions are not centered about computational constructs such as queries or sensor groups. Instead, these abstractions are centered around elements of the physical environment. The aspiration is that at the highest level of abstraction, the existence of the sensor network itself should be entirely transparent.

Node-based languages such as nesC [Gay et al. 2003] and galsC [Cheong et al. 2003; Cheong and Liu 2005] are too low-level since they typically take the sensor node as basic computation, communication and actuation unit. To address this issue, higher-level languages that export logical nodes [Gummadi et al. 2005] were proposed to abstract away from physical sensors. EnviroSuite successfully raises the abstraction level to logical objects mapped from physical elements, thus expedite the procedure of design and programming compared with node-based languages.

Virtual machines such as Mate [Levis and Culler 2002] and SensorWare [Boulis et al. 2003] allow large sensor networks to be reprogrammable frequently by writing application scripts, replicating them through the network and executing them automatically. However, they usually concentrate on issues related to code replication and auto-execution rather than raising programming abstraction levels. For example, to reduce energy cost of code replication, Mate even provides an instruction-like language to shorten code length, which actually puts extra burden on programmers shoulders.

Database-centric abstractions such as TinyDB [Madden et al. 2003] and Cougar [Yao and Gehrke 2002] view sensor networks as databases that allow users to express requirements as queries, and to distribute and execute these queries. Comparatively, our work, instead of providing a specific data collection and aggregation

model, attempts to support a wider range of applications by encapsulating not only computation and communication units but also actuation units into its programming abstractions.

Event-based models such as [Li et al. 2004] are similar with database-centric abstractions except that they view the sensor field as an active entity that automatically push data streams to users when defined events are triggered instead of a passive database which only responds upon queries.

Group-based primitives such as Hood [Whitehouse et al. 2004] and Abstract Regions [Welsh and Mainland 2004], provide neighbor discovery and neighborhood data sharing mechanisms. Compared with EnviroSuite, these abstractions are passive. In contrast, EnviroSuite abstractions are active objects that encapsulate local code and aggregate state, as well as share data across neighborhoods or regions.

Another group-based paradigm, State-centric programming [Liu et al. 2003], described a programming abstraction mostly related to our work. However, it is implemented and evaluated only on Pieces simulator built in Java and Matlab, which can not simulate some critical features of wireless communication including message collision. In contrast, our paper presents a detailed implementation in nesC on TinyOS, an operating system for real sensor network devices, and provides comprehensive evaluation results both in TOSSIM and real sensor devices. Furthermore, the underlying group management protocol [Liu et al. 2003] differs in its mechanisms for object classification and identity management.

An earlier paper by the authors [Abdelzaher et al. 2004] presented a programming paradigm focusing on tracking applications. In this paper, we expand this idea and present programming abstractions that successfully support a broader set of applications including not only event tracking but also regional monitoring applications.

Finally, we should mention that a criticism of current high-level programming languages has been that they are too application specific. Hence, intermediate-level languages such as [Newton et al. 2005] were proposed as a step towards macroprogramming. EnviroSuite attempts to cater to a general application pool by diversifying the supported object types.

8. CONCLUSION

In this work, we describe an environmental immersive programming paradigm for application developers in sensor networks. We present the design, implementation and evaluation of a framework implementing this paradigm. The EnviroSuite framework successfully exports high-level abstractions, such as objects and inter-object calls. It implements low-level distributed protocols such as sensing data processing, group management and inter-object communication in an underlying library EIPLib, transparent to programmers, thus resulting in a considerable potential to reduce development costs of deeply embedded systems. This paper describes the first comprehensive design and implementation of all EIP abstractions including objects, their attributes, methods and inter-object calls (The concept of EIP was described earlier in [Blum et al. 2003]). This paper also presented the first comprehensive evaluation of the performance of real nesC code generated by the EnviroSuite compiler from EnviroSuite source files. This is to be distinguished from

prior initial results, which reported some GloMoSim simulations.

REFERENCES

ABDELZAHER, T., BLUM, B., CAO, Q., EVANS, D., GEORGE, J., GEORGE, S., HE, T., LUO, L., SON, S., STOLERU, R., STANKOVIC, J., AND WOOD, A. 2004. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. In *ICDCS '04: Proceedings of the International Conference on Distributed Computing Systems.*

AZEVEDO, S. G. AND MCEWAN, T. E. 1996. Micropower impulse radar. *Science and Technology Review.*

BIRRELL, A. D. AND NELSON, B. J. 1984. Implementing remote procedure calls. *ACM Trans. Comput. Syst. 2,* 1, 39–59.

BLUM, B., NAGARADDI, P., WOOD, A., ABDELZAHER, T., SON, S., AND STANKOVIC, J. 2003. An entity maintenance and connection service for sensor networks. In *MobiSys '03: Proceedings of the 1st international conference on Mobile systems, applications and services.* ACM Press, New York, NY, USA, 201–214.

BOULIS, A., HAN, C.-C., AND SRIVASTAVA, M. B. 2003. Design and implementation of a framework for efficient and programmable sensor networks. In *MobiSys '03: Proceedings of the 1st international conference on Mobile systems, applications and services.* ACM Press, New York, NY, USA, 187–200.

CHEONG, E., LIEBMAN, J., LIU, J., AND ZHAO, F. 2003. Tinygals: a programming model for event-driven embedded systems. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing.* ACM Press, New York, NY, USA, 698–704.

CHEONG, E. AND LIU, J. 2005. galsc: A language for event-driven embedded systems. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe.* IEEE Computer Society, Washington, DC, USA, 1050–1055.

DUTTA, P., GRIMMER, M., ARORA, A., BIBYK, S., AND CULLER, D. 2005. Design of a wireless sensor network platform for detecting rare, random, and ephemeral events. In *IPSN '05: Proceedings of the Fourth International Conference on Information Processing in Sensor Networks.*

GAY, D., LEVIS, P., VON BEHREN, R., WELSH, M., BREWER, E., AND CULLER, D. 2003. The nesc language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation.* ACM Press, New York, NY, USA, 1–11.

GU, L., JIA, D., VICAIRE, P., YAN, T., LUO, L., TIRUMALA, A., CAO, Q., STANKOVIC, J. A., ABDELZAHER, T., AND KROGH, B. 2005. Lightweight detection and classification for wireless sensor networks in realistic environments. In *Sensys.* San Diego, CA.

GUMMADI, R., GNAWALI, O., AND GOVINDAN, R. 2005. Macro-programming wireless sensor networks using kairos. In *DCoSS.*

HE, T., KRISHNAMURTHY, S., STANKOVIC, J. A., ABDELZAHER, T., LUO, L., STOLERU, R., YAN, T., GU, L., HUI, J., AND KROGH, B. 2004. Energy-efficient surveillance system using wireless sensor networks. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services.* ACM Press, New York, NY, USA, 270–283.

HILL, J., SZEWCZYK, R., WOO, A., HOLLAR, S., CULLER, D., AND PISTER, K. 2000. System architecture directions for networked sensors. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems.* ACM Press, New York, NY, USA, 93–104.

INTANAGONWIWAT, C., GOVINDAN, R., AND ESTRIN, D. 2000. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *MobiCom '00: Proceedings of the 6th annual international conference on Mobile computing and networking.* ACM Press, New York, NY, USA, 56–67.

LEVIS, P. AND CULLER, D. 2002. Mat: A tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems.*

Levis, P., Lee, N., Welsh, M., and Culler, D. 2003. Tossim: accurate and scalable simulation of entire tinyos applications. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*. ACM Press, New York, NY, USA, 126–137.

Li, S., Lin, Y., Son, S. H., Stankovic, J., and Wei, Y. 2004. Event detection services using data service middleware in distributed sensor networks. *Telecommunication Systems, Special Issue on Information Processing in Sensor Networks 26*, 2-4.

Liu, J., Chu, M., Liu, J., Reich, J., and Zhao, F. 2003. State-centric programming for sensor-actuator network systems. *Pervasive Computing, IEEE 2*, 4, 50–62.

Liu, J., Liu, J., Reich, J., Cheung, P., and Zhao, F. 2003. Distributed group management for track initiaition and maintenance in target localization applications. In *IPSN '03: Proceedings of the 2nd International Workshop on Information Processing in Sensor Networks*.

Madden, S., Franklin, M. J., Hellerstein, J. M., and Hong, W. 2002. Tag: a tiny aggregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev. 36*, SI, 131–146.

Madden, S., Franklin, M. J., Hellerstein, J. M., and Hong, W. 2003. The design of an acquisitional query processor for sensor networks. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM Press, New York, NY, USA, 491–502.

Madden, S. R., Franklin, M., Hellerstein, J., and Hong, W. 2005. Tinydb: An acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems 30*, 1 (March).

Microsoft. 1994. Ole2 programmers reference.

Newton, R., Arvind, and Welsh, M. 2005. Building up to macroprogramming: An intermediate language for sensor networks. In *IPSN '05: Proceedings of the Fourth International Conference on Information Processing in Sensor Networks*.

U. C. Berkeley. 2005. the motes. `http://www.tinyos.net/scoop/special/hardware#mica`.

Vinoski, S. 1997. Corba: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine 32*, 2 (February), 46–55.

Welsh, M. and Mainland, G. 2004. Programming sensor networks using abstract regions. In *NSDI '04: Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation*.

Whitehouse, K., Sharp, C., Brewer, E., and Culler, D. 2004. Hood: a neighborhood abstraction for sensor networks. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*. ACM Press, New York, NY, USA, 99–110.

Yao, Y. and Gehrke, J. 2002. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec. 31*, 3, 9–18.