Archetype-Based Design: Sensor Network Programming for Application Experts, Not Just Programming Experts

Lan S. Bai† Robert P. Dick† Peter A. Dinda‡
{lanbai, dickrp}@umich.edu pdinda@northwestern.edu
University of Michigan† Northwestern University‡

ABSTRACT

Sensor network application experts such as biologists, geologists, and environmental engineers generally have little experience with, and little patience for, general-purpose and often low-level sensor network programming languages. We believe sensor network languages should be designed for application experts, who may not be expert programmers. To further that goal, we propose the concepts of sensor network application archetypes, archetype-specific languages, and archetype templates. Our work makes the following contributions. (1) We have examined a wide range of wireless sensor networks to develop a taxonomy of seven archetypes. This taxonomy permits the design of compact languages that are appropriate for novice programmers. (2) We developed a language (named WASP) and its associated compiler for a commonly encountered archetype. (3) We conducted user studies to evaluate the suitability of WASP and several alternatives for novice programmers. To the best of our knowledge, this 56-hour 28-user study is the first to evaluate a broad range of sensor network languages (TinyScript, TinySQL, SwissQM, and TinyTemplate). On average, users of other languages successfully implemented their assigned applications 30.6% of the time. Among the successful completions, the average development time was 21.7 minutes. Users of WASP had an average success rate of 80.6%, and an average development time of 12.1 minutes (an improvement of 44.4%).

1. INTRODUCTION

Wireless sensor networks have the potential to allow inexpensive, real-time, and broadly-distributed data collection and analysis for the first time in history. Application experts, such as geologists, entrepreneurs, civil engineers, and biologists have the most to gain. However, these application experts generally have little experience with, and little patience for, the general-purpose, node-level languages available for sensor network programming. As a result, application experts are forced to rely on embedded system experts to implement their ideas. Almost all existing sensor network deployments are implemented by embedded system experts. This approach is costly. Separating design and implementation in this way can also lead to errors due to miscommunication between application experts and embedded system experts. Application experts generally have limited awareness of the constraints on sensor network capabilities imposed by hardware and software limitations. On the other hand, embedded system experts know little about the application requirements, which are tightly related to the measured objects and the working environments. In addition, since application experts' and embedded system experts' domain languages differ significantly, this can cause confusion and misunderstandings that lead to incorrect implementations. Consequently, a collaboration between application experts and embedded system experts requires a large amount of communication, negotiation, redesign, and reimplementation. Many potential users of wireless sensor networks consider them because they have the potential to save time and money. When these potential benefits are outweighed by substantial increases in implementation complexity compared to the bulky and expensive, but often easy-to-deploy, sensing solutions already in use, wireless sensor networks will remain unused.

We believe that appropriate high-level programming languages and compilers have the potential to make wireless sensor networks accessible to the application experts who have the most to benefit from their use. We propose designing sensor network languages with the novice programmer in mind, hence the following language features are desirable.

1. The languages should support specifying application-level requirements, not just node-level behavior.

2. The languages should not expose low-level implementation details, such as resource management, communication protocols, and optimizations, to users. Users

should need only specify application requirements.

3. The languages should be compact and easy to use. People with limited or no programming experience should be able to almost immediately learn and use them to specify correct sensor network applications.

The first step to designing a programming language is to determine the scope of applications it will support. There are two extremes of the range of design philosophies a language designer might adopt: a language might be entirely general-purpose or entirely application-specific. General-purpose languages can be used to specify any application. However, all other things being equal, this flexibility is obtained at the cost of increased language complexity. General-purpose languages have advantages: once such a language is learned, one can write any application with it. However, a novice programmer may never be willing to expend the time to learn it. In contrast, application-specific languages are usually simple and compact, but support one type of application. This makes it more difficult for a novice programmer to select the appropriate language for an application, and requires the design of numerous languages – one for each type of application. Designers need to learn a new language with each new application. We believe the optimal design philosophy for sensor network programming languages is somewhere between these extremes: a moderate number of specialized languages that together cover most of the sensor network application domain. Ideally, each of these languages should be easy to learn and use for novice programmers.

The question remains, "what is the appropriate granularity of the application groups?", i.e., "how specialized should the languages be?" To find the best tradeoff between the complexity of selecting a language and the complexity of the languages, we propose the concept of *sensor network archetypes*. We have categorized sensor networking applications into archetypes based on functional properties that have large impacts on language design. We have examined a wide range of sensor network applications in order to develop a taxonomy of seven archetypes (see Section 3). The language tailored for an archetype is called an *archetype-specific language*.

Once an application's archetype is known, it is possible to provide a program template/example as a starting point. Our studies indicate that the availability of templates improves the success rate for novice programmers implementing sensor network applications from 0% to 8.3% for a node-level language. Knowledge of an archetype further reduces the burden on a novice programmer because only one archetype-specific language needs to be learned, and each such language is simpler than a general-purpose programming language. We have embodied these language design concepts in a language for a frequently encountered sensor network arche-type. In comparison with alternative sensor network programming languages such as TinyScript, TinyDB, and SwissQM, this language results in $1.6\times$ average improvement in success rate and 44.4% average reduction in development time. We plan to extend these language design concepts to other archetypes in our future work.

2. RELATED WORK

A number of researchers have proposed new sensor network languages to improve design productivity. However, most of these languages have been designed with expert programmers in mind. Although they may improve the productivity of embedded system experts, they are unlikely to make the design and deployment of sensor networks accessible to application experts who are often novice programmers. A few languages are proposed for application experts. However, their use by novice programmers has not been experimentally evaluated, making it difficult to draw conclusions about their suitability. In this section, we review these languages and summarize the major differences and contributions of our work.

Node-level programming languages specify the behavior of each single sensor node. To access the state of other nodes, the program must explicity specify data transmission between nodes. NesC [11] and C are widely used node-level programming languages for sensor networks. These languages are too low-level for novice programmers. In addition, concepts such as events and threads are quite difficult for novice programmers to learn. Efforts [14, 24] have been made to raise the abstraction level of these languages.

Network-level programming languages, also called macro-programming languages, treat the whole network as a single machine. Lower-level details such as routing and communication are hidden from programmers. Pleiades [19] extends C to achieve a centralized perspective with access to all the nodes in the network via naming. Some researchers allow designers to treat the sensor network as a database and use query languages to extract data from the network [28, 33]. Regiment [34] lets programmers view the network as a set of distributed data streams. ATaG [4] is based on data-driven program flow and mixed imperative-declarative specification. It lets developers graphically declare the data flow and connectivity of virtual tasks and specify the functionality of tasks using common imperative language. RuleCaster [5] provides a macroprogramming abstraction with a state-based model and uses a high-level language akin to Prolog.

A few researchers have considered the accessibility of sensor network design to application experts. However, we are aware of only one other publication describing experiment evaluation of usability of a sensor network programming language. Some languages [12, 17] are inspired by commercial graphical programming tools such as LabView [21] and Excel [10]. Other researchers made the design of easy-to-use languages tractable by targeting a specific type of applications. NETSHM [7] is a sensor network software system for structural health monitoring applications. In contrast, we did a broad study of existing sensor network applications and proposed a method of identifying classes of applications for specialized language development. BASIC was proposed for use in sensor network programming [30]. The authors implemented BASIC for sensor networks and conducted a user study with novice programmers. Their user study is contemporaneous with ours. Their work targeted a different application domain than ours and focused on node-oriented programming.

More comprehensive reviews and comparisons of existing sensor network programming languages can be found in surveys [46, 31]. Sugihara and Gupta [46] compared the languages using three metrics: energy-efficiency, scalability, and failure-resilience. They acknowledged that ease of programming is a very important criteria but they believed "criteria of easiness is inherently subjective and the complexity of code largely depends on each application". In contrast, we believe that it is possible and important to evaluate the usability of sensor network languages and have designed and executed a rigorous user study to compare a number of languages. Mottola and Picco [31] introduced a taxonomy of wireless sensor network programming models. The taxonomy we introduce is for sensor network applications. Röemer and Mattern [38] have studied and characterized the design space of sensor networks to provide a clearer picture of sensor network applications and their requirements. We also classified sensor network applications but for a different purpose: archetype-based programming language design. We focus solely on dimensions that affect the complexity of specification language.

Our work shares some techniques and principles with that on human–computer interaction [13, 20] and the psychology of programming [49]. Involving users in the design process has been emphasized in the human–computer interface design.

3. TAXONOMY OF WIRELESS SENSOR NETWORK APPLICATIONS

Specialized, high-level specification languages have the potential to open sensor network design to application experts who are novice programmers. Finding the optimal partitioning of the sensor network application domain for the purpose of language design is challenging. This section describes our study of a wide range of sensor network applications in order to build a taxonomy of sensor network archetypes, and thus languages.

3.1 Wireless Sensor Network Applications

We studied 23 sensor network applications and summarized their application-level requirements and functionalities to extract 19 application properties. These applications, most of which have been deployed, span a wide range of domains: environmental monitoring [16, 47, 50, 44, 40], structural health monitoring [8, 6, 26, 45, 35, 18], habitat monitoring [37, 27], target detection and localization [15, 39, 2, 43], residential monitoring [51], active sensing [52], medical care [41], farm management [42, 48], and others [9]. Specifications should focus on the requirements of an application, and avoid implementation details to the greatest degree possible while still maintaining adequate performance. Based on this principle, we identified the following 19 application-level properties (refer to Section 3.2 for definitions): mobility, initiation of sampling process, initiation of data transmission, interactivity, data interpretation, data aggregation, actuation, homogeneity, topography, sampling mode, when sensor locations are known, synchronization, unattended lifetime, mean time to failure, maximum node weight, maximum node size, maximum node volume, maximum node mass, minimum covered area, and quality of service.

3.2 Categorization of Wireless Sensor Network Applications

Among the 19 application properties, only eight affect the complexity of the specification language. Other properties are constraint oriented and have little impact on the specification of sensor network functionality. For example, changing the required lifetime of the system from a month to a year will not change the functional specification, although the implementation may change. Specifying constraints can be uniform and straightforward across many application domains, unlike functional specifications. The syntax *constraint* = *value* is sufficient. Therefore, we ruled out these properties as criteria for placing applications. The following eight properties remain.

- **Mobility** indicates whether the sensor nodes are mobile. Mobile nodes may be wearable devices to monitor or track moving objects such as humans and animals [48, 42]. Sensor nodes might also adjust their positions. For applications with mobile sensor nodes, specifications of node localization and node movement control are usually desired. Therefore, mobile sensor network applications will require more complex specifications.

- **Initiation of sampling** indicates the condition that causes the nodes to start sampling. It can be periodic, event driven or a mix. Periodic sampling requires specification of the sampling period, while event-driven sampling requires the specification of events.

- **Initiation of data transmission** indicates the condition in which nodes send data through the network. It can be periodic, event driven, or both. Applications for event detection usually require data to be sent to a base station under a certain condition.

- **Actuation** indicates whether the sensor network produces signals to trigger or control other hardware components. For example, the autonomous livestock control application [48] generates stimuli to bulls when the sensor network detects two bulls will soon fight. Actuation requires the specification of triggering conditions and actuation actions, and is therefore more complex than specifying only sensing.

- **Interactivity** indicates whether the network is required to respond to commands sent during operation. Interactions are usually required for initial deployment, reprogramming, maintenance, adjusting operational parameters, and on-site visits. Interactivity requires the specification of commands and reactions.

- **Data interpretation** indicates that in-network data processing is carried out on raw sensor data to filter or compute derivative information. Such online data interpretation may support automated decisions or other actions. Support for data interpretation requires specification of the data processing procedures.

- **Data aggregation** indicates whether data should be aggregated across multiple sensor nodes. Data aggregated requires the ability to specify aggregation algorithms as well as the group of nodes the aggregation operation applies to. For this reason, data aggregation complicates specification.

- **Homogeneity** indicates whether the functionality of every sensor node in the network is the same. For a heterogeneous network, the specification language needs to provide the ability of distinguishing among different types of nodes.

The crossproduct of these eight application attributes results in at least 256 unique points in the language design space. The 23 application samples form 20 points, as shown in Table 1. The extreme of designing one language for each point would make it difficult for a user to identify the correct language and increases the burden of language design. Our goal is to find the categorization of sensor network applications that minimizes the complexity of categorizing applications within categories (archetypes) and the complex of using the corresponding language, while also limiting the number of languages required to make the language design process practical.

A good partition should cluster some application types that are adjacent or nearby in the attribute space. In addition, the number of attributes for which multiple dimensions are spanned should be minimized. This suggests using a clustering algorithm for categorization. We adopted the K-Means algorithm to cluster the 23 applications. Dimensions with orthogonal values are treated as sets; dimensions with comparative values are mapped to scalar values with larger values indicate more complex functionality. Choosing the number of clusters involves a trade-off between the complexity of individual languages and the number of languages. The complexity of the specification language corresponding to each application type is hard to quantify precisely ahead of time and the specification language for a potential application category cannot be accurately predicted without language design and evaluation. Therefore, choosing k is a somewhat ad-hoc process based on prior experience with sensor network and language design. The resulting clustering-based archetypes are shown in Table 2. A row in the table corresponds to one archetype. The "size" column indicates how many applications fit into the corresponding archetype. An archetype is defined by its values in the eight application attributes. "*" means any value is accepted. Note that the specification languages may overlap, i.e., an application may be a member of multiple archetypes.

4. ARCHETYPE-SPECIFIC LANGUAGES

The taxonomy of sensor network archetypes described in the previous section guides the design of specialized languages, i.e., *archetype-specific languages*. The concept of archetypes allows templates to be designed to further reduce the programming burden for application experts. In our user study, most test subjects indicated that examples help them to understand a new language. Therefore, we propose the concept of *archetype templates*. These can be generic example programs for specific archetypes or incomplete programs with parameters and lines of code to be modified by programmers according to their needs. An application expert uses an archetype-specific language by reading a short tutorial and using an archetype template to implement an application. We want this procedure to be easy and efficient for novice programmers.

For an application expert using our proposed design flow, the first step in sensor network implementation is to determine the right archetype for one's application. We believe this first step is easy for an application expert for the following reasons. First, the archetypes are determined from eight specific application properties that are meaningful to application designers. Second, the archetype taxonomy allows the design of an interactive tool to assist a user in choosing the correct archetype for an application via a list of multiple-choice questions based on the eight application attributes. Classification should require eight or fewer answers, since an archetype is determined by at most eight attributes.

In short, archetype-specific languages have the following advantages.

1. An application expert only needs to learn the language features that are strongly related to the application of

Table 1: Sensor network applications

Application	Mobility	Sampling	Data transmission	Actuation	Interactive	Data interpretation	Data aggregation	Homo-geneous
Wisden	stationary	periodic	periodic	N	N	Y	Y	Y
Habitat	stationary	periodic	periodic	N	N	N	N	Y
Bridge	stationary	periodic	periodic	N	N	N	Y	Y
FireWxNet	stationary	periodic	periodic	N	N	N	N	Y
Light control	stationary	periodic	periodic	N	N	N	N	Y
ACM	stationary	periodic	periodic	N	N	N	N	Y
Redwoods	stationary	periodic	periodic	N	N	N	Y	Y
Surveillance	stationary	periodic	event-driven	N	Y	Y	Y	Y
VigilNet	stationary	hybrid	event-driven	N	N	Y	Y	Y
SenSlide	stationary	periodic	event-driven	N	N	Y	Y	Y
Vehicle tracking	stationary	periodic	event-driven	N	N	Y	Y	Y
Shooter	stationary	event-driven	event-driven	N	N	Y	Y	Y
Volcanic	stationary	periodic	event-driven	N	N	Y	N	Y
ElevatorNet	mobile	periodic	periodic	N	N	Y	N	Y
ZebraNet	mobile	periodic	event-driven	N	N	N	Y	Y
Active sensing	mobile	periodic	event-driven	Y	N	Y	Y	Y
Animal control	mobile	periodic	periodic	Y	N	Y	N	Y
Farm	mobile	periodic	periodic	Y	Y	N	N	N
ALARM-NET	mobile	periodic	hybrid	N	Y	N	N	N
CodeBlue	mobile	periodic	hybrid	N	Y	Y	N	N
PIPENET	stationary	hybrid	hybrid	N	Y	Y	Y	Y
NETSHM	stationary	event-driven	hybrid	Y	Y	N	Y	Y
Tunnel	mobile	periodic	event-driven	N	N	Y	Y	N

Table 2: Sensor network archetypes

Archetype	Size	Mobility	Sampling	Data transmission	Actuation	Interactive	Data interpretation	Data aggregation	Homo-geneous
1	7	stationary	periodic	periodic	N	N	*	*	Y
2	6	stationary	*	event-driven	N	*	Y	*	Y
3	4	mobile	periodic	*	*	N	*	*	Y
4	3	mobile	periodic	*	*	Y	*	N	N
5	1	stationary	hybrid	hybrid	N	Y	Y	Y	Y
6	1	stationary	event-driven	hybrid	Y	Y	N	Y	Y
7	1	mobile	periodic	event-driven	N	N	Y	Y	N

interest. This results in shorter development time and a shorter learning curve.

2. The simplicity of archetype-specific languages permits short tutorials, simple grammars, high levels of abstraction, and productive use of archetype templates. This reduces development time, improves correctness rates, and increases the satisfaction of novice programmers with the design process.

3. The design of high-level languages is simplified by targeting specific groups of applications.

5. WASP: AN EXAMPLE ARCHETYPE-SPECIFIC LANGUAGE

We selected the archetype with the most existing sensor networking applications as the starting point for archetype-specific language design. This archetype contains seven of the sample applications. It corresponds to applications that periodically sample and transmit raw data, or filter and aggregate data, before transmitting them to a base station from a stationary, homogeneous network. We will refer to this as "Archetype 1". This section presents the proposed language, WASP, as well as its compiler and simulator.

5.1 Language Overview

Among the existing languages, those based on database query languages (e.g., SwissQM and TinyDB) provide the most appropriate high-level abstractions for Archetype 1. However, their support for temporal queries may be difficult to grasp for novice programmers, because the database abstraction represents a snapshot of the network that only contains current data (the default table named *sensors*). In order to use historical data, a *storage point* must be explicitly created in the program. The storage point provides a location to store a streaming view of recent data. For example, in TinyDB

```
CREATE STORAGE POINT recentlight SIZE 8
AS (SELECT nodeid, light
    FROM sensors
    SAMPLE PERIOD 10s)
```

creates a storage point for the most recent eight light samples. For a simple application that compares the current sensor reading with previous readings, developers need to issue a query that joins data from the *sensors* table and the created storage point. Joins are a complex query construction that even experienced database users often get wrong. Our experimental results indicate that many novice programmers have great difficulty using joins correctly (see Section 7 for details). Instead of forcing programmers to explicitly create buffers to store temporal data, WASP makes both historical and current data directly accessible to programmers.

To achieve easy access to both current and historical data, WASP lets programmers view the network as distributed data arrays. Each array corresponds to a node-level variable and stores the stream of a particular type of data. Newly sampled data or computed results are inserted to the top of the array, which is indexed from 0. Older data can thus be referenced by indexing into the array. Another major difference between WASP and existing query languages is that it lets users specify an application at two levels: node-level and network-level. Operations that only use constants and data generated on one node may be specified at node-level. Data transmission and data aggregation are specified at network-level. The two features permit local data processing while retaining the high-level abstraction that hides the mechanics of routing and communication.

5.2 WASP Language Construct

A WASP program is composed of two segments. The node-level code segment, initiated with the keyword "local:", specifies single node behavior. The network-level code segment, initiated with the keyword "network:", specifies how data are aggregated through the network and gathered at the base station.

The node-level code segment specifies two types of functionalities: sampling and data processing. The sampling specification indicates the type of sensor data sampled and the associated sampling frequency. The data processing specification indicates how the raw sensed data are processed to generate other data. It may be used for data interpretation, unit conversion, local event detection, etc. The syntax follows. Keywords are in uppercase. Variables and parameters are in lowercase.

```
SAMPLE sensor EVERY t t_unit INTO buffer
SAMPLE sensor INTO scalar
new_data = arithmetic_expr
new_data = arithmetic_expr EVERY t t_unit
new_data = function(args)
new_data = function(args) EVERY t t_unit
```

`Sensor` describes the type of sampled data. `Buffer`, `scalar`, and `new_data` are user-defined variables. Programmers can view a variable as an infinite array that stores a time series. Data items in the array can be referred to via indexing. Index 0 represents the most recent data, while index n represents the nth most recent data item. A sequence of data items can be referred to using two indices, indicating a range. Fox example, buffer[0:9] returns the most recent 10 data items. Data types of variables are not specified by users, but inferred by the compiler. If a sampling operation or data computation is periodic, "EVERY t t_unit" should be specified at the end of the statement to indicate the period. If absent, this implies that the operation need only be done once. `Function` is selected from a library of built-in functions. These functions are aggregation functions used in node-level code. They aggregate data across time on each individual node. The execution order of the statements is determined by the data dependency. Programmers can write them in any order. The syntax of node-level code is designed to be straightforward and readable by novice programmers. The `SAMPLE` clause is similar to English. The other instructions are based on assignment statements that even novice programmers are likely to be familiar with.

The network-level code segment lets programmers view the entire sensor network as a table and use collective operations to extract the data they want. Instead of containing only the current sensor readings, as in TinyDB, this table contains the most recent data items for all the variables that are defined in the node-level code segment. Although the table represents a snapshot, its columns may contain variables representing or derived from temporal data. Therefore, only one table exists in WASP; programmers need not create tables or query from multiple tables. Network-level code has a syntax that is similar to the TinySQL language used in TinyDB. It consists of *collect-where-group by-having-delay* clause supporting selection, join, projection, and aggregation.

WASP has a DELAY statement for specifying maximum latency of data collection. The syntax is `DELAY t t_unit`. Parameter `t t_unit` indicates the maximum delay from data item generation to arrival at the base station. Programmers without delay constraints should eliminate this statement. The syntax of the clause for network-level code is more constrained than TinyDB. The data following the *select* keyword can either be a node-level variable or an aggregation function. Expressions are not allowed. In contrast with TinyDB, WASP network-level code does not specify sampling frequency. Frequency should always be specified in node-level code segment, together with variable definitions. The data transmission frequency can be inferred from the periods at which data are collected.

5.3 WASP Programming Template

A template for WASP programs is given below. Uppercase words are commands. Lower-case words are descriptions of parameters at the corresponding locations; they will be replaced with the variables, functions, and expressions by programmers.

```
LOCAL:
SAMPLE sensor EVERY t t_unit INTO buffer
SAMPLE sensor INTO scalar
data1 = function(args) EVERY t t_unit
data2 = function(args)
data3 = arithmetic_expr EVERY t t_unit
data4 = arithmetic_expr
NETWORK:
COLLECT field1, field2, ...
WHERE node-selection-conditions
GROUP BY node-variable-list
HAVING group-selection-conditions
DELAY t t_unit
```

5.4 Compiler and Simulator for WASP

The WASP compiler translates a WASP program into NesC code. The generated NesC code is then compiled to executables with ncc, the NesC compiler for TinyOS. The parser is written with PLY [36], a python implementation of the compiler construction tools lex and yacc. The implementation of Archetype 1 requires the use of modules for timing, communicating, synchronization, and routing, which we implement as a library that is automatically accessed by the generated code. We constructed a NesC template for Archetype 1 that embodies the partial implementation that is required for any application in the archetype. The Collection Tree Protocol (CTP), implemented as TinyOS components, is used for the routing and data collection. In the template, application-dependent code segments are marked with special symbols, which are replaced with NesC statements generated by the WASP compiler. The replacement is automated with a Python script. During compilation, variables in the WASP program are converted to arrays or scalars with explicit data types, the minimum size of the arrays are computed. The sampling instructions are converted to NesC instructions to control the sensor components. Other node-level instructions are converted to tasks. The period specification in the WASP program is converted to instructions to set timers. The network-level code is converted to data transmission and in-network data aggregation instructions in NesC. The compiler has been tested with the three applications from our user study (refer to Section 6 for more details). The generated code was run on a multi-hop network composed of four TelosB nodes. We did not yet work on compiler optimization of performance and power.

To support our user study, we also implemented a discrete event simulator for WASP in Python. The parser is modified to generate Python code that creates sampling, processing, and data collection events. The simulator is only used to check functional specification, not implementation or reliability. Therefore, it emulates a perfect network: every operation is instantaneous; there is no node failure or communication loss. The sensor data are randomly generated in the range from 0 to 1024. An user interface was also developed for WASP, providing a simple programming environment in which WASP code is edited, saved, compiled, and simulated.

6. USER STUDY

To test the suitability of WASP and to assess the value of archetype-specific languages, we conducted a user study that tested 28 novice programmers using five different programming languages. This section describes the protocol of our user study. The materials used in the study are available on our project website [1].

The user study was designed to address these questions:

1. What impact does the use of specialized languages have on programmer productivity, as quantified via development success rate and time?
2. What impact does the use of programming templates have on productivity?
3. Is the node-level or the network-level programming model better for novice programmers?
4. Can novice programmers efficiently and correctly use WASP?
5. What is the most appropriate language for the most frequently encountered archetype?
6. What difficulties do novice programmers have in using existing programming languages?

We used the following criteria when selecting languages for testing and comparison: (1) the language is designed to simplify sensor network programming and it provides high-level abstractions; (2) it was designed to support applications that carry out periodic data sampling and transmission; (3) is has been implemented and the associate tool chain is publicly available. Five programming languages were selected for comparison: TinyScript, TinyTemplate, TinySQL, SwissQM, and WASP. Three of them (TinyScript, TinySQL, SwissQM) are from existing work with released software tools.

TinyScript [24] is a general-purpose, node-level, event-driven programming language used for the Maté virtual machine [23]. Programmers write imperative code for event handlers. We made two major changes to create a specialized version of TinyScript, called TinyTemplate, for the most frequently encountered archetype. First, we pruned the library and handlers of TinyScript to only contain functions and events that are related to the target archetype. Second, we provided a programming template. The template is an parameterized example program that implements periodic sampling and data aggregation; comments in the program indicate the variables and instructions that should be replaced for different applications. Expecting that it will be extremely difficult for novice programmers to implement multi-hop communication within reasonable a amount of time, we let the test subjects of TinyTemplate and TinyScript assume a one-hop network structure, in which every node can directly communicate with the root node. Even so, the success rate was extremely low for these two languages.

TinySQL [28] is the SQL-like language used in TinyDB. Programmers view the whole network as a table, with each row indexed with node identification number. User-defined storage points are used in this language to buffer temporal data. SwissQM [33][1] is a programming interface for a query virtual machine. The query language for SwissQM is similar to TinySQL, but instead of letting users write textual code, SwissQM provides a graphical interface. The interface makes composing queries convenient, but it also constrains the supported applications; temporal queries cannot be supported by this interface.

In our study, it was our goal to compare languages and minimize the effect of other factors such as documentation and programming environment. We therefore rewrote the tutorials for these languages (these tutorials are available at our project website [1]). The published documents [32, 29, 22] were generally written for programming experts and proved to be very difficult for novice programmers to understand. Programming templates are provided for WASP, TinySQL, and TinyTemplate. The graphical interface of SwissQM is considered to be a template.

In practice, system design and programming are interactive and iterative processes. Hence, feedback to test subjects is necessary for the user study to approximate real-world circumstances. Asking users to work with a collection of sensor nodes has the potential to introduce problems that are orthogonal to language design and thereby reduce the discerning power of the study. In order to focus on measuring the impact of the language on productively writing functionally correct code, we associated each language with a simulator. A network composed of four nodes was simulated for each language. These simulators run in real-time, and emulate ideal sensor networks without delay or failure[2]. The TinyOS simulator, TOSSIM [25], was used for TinyScript, TinyTemplate, and SwissQM. We implemented a simulator in Python for TinySQL[3]. The TinySQL code is translated into iterative database queries that are passed to a database server. The creation of storage points is converted to creation of view points. We implemented a discrete event simulator for WASP in Python. Though the implementation of the simulation environments of these languages differ, the user interfaces are quite similar.

Our study procedure is designed to permit fair comparisons among languages while maintaining short duration studies. First, the test subjects are introduced to wireless sensor networks via a short description. This gives test subjects a basis for understanding the programming languages and tasks. Next, the test subjects are given 30 minutes to read a tutorial for the language under test, and to familiarize themselves with the programming environment.

After that, they are given the description of two sensor network programming tasks; 40 minutes are permitted for each one. The description of the second task is given after the first is complete, or after 40 minutes have elapsed. The test subjects may notify the test administrator when they think they have a correct solution. Finally, test subjects answer a survey to provide feedback on the language, tasks, programming environment, etc. The screen is recorded during the study, allowing us to examine the interaction between the programmers and the programming environments. During the study, the test subjects are permitted to ask the test administrator questions regarding the tutorials or the task descriptions. However, the administrator does not answer questions related to the implementation of the tasks. Though we used three tasks and five languages for the user study, each test subject was asked to complete two tasks in one language. This is due to the requirement of maintaining a study duration short enough for participants to tolerate. The selection of language and tasks for each test subject was random. To eliminate ordering effects, we randomized the order of the two tasks.

We selected three tasks that we believe are representative of the target archetype and span different levels of difficulty. These tasks are closely related to the real deployed sensor network applications, cited earlier. Task 1 is a basic environmental monitoring application that transmits raw sensor data to a base station. Task 2 requires node grouping and data aggregation. Task 3 requires temporal processing. SwissQM is inherently unable to support Task 3. The descriptions of the tasks follow.

- Task 1: Sample light and temperature every 2 seconds from all the nodes in the network. Transmit the samples with their node identification numbers to the base station [37, 16, 8].

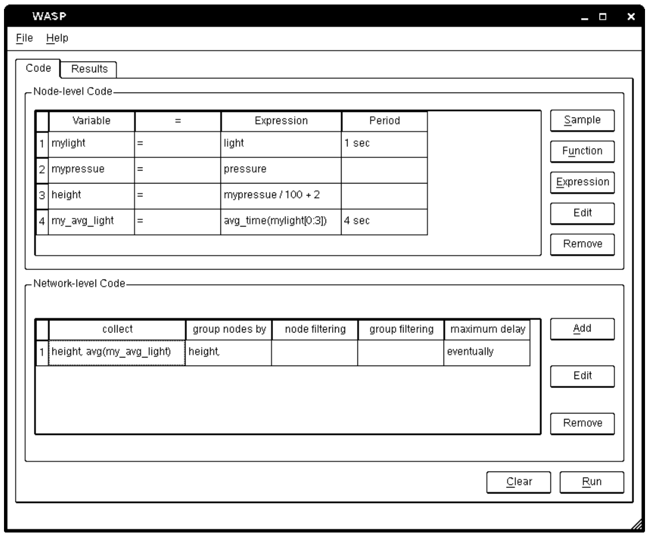
- Task 2: Sample light and temperature every 3 seconds from all the nodes in the network. Collect average temperature readings from nodes that have the same light level. Light levels are computed by dividing raw light readings by 100 [47].

- Task 3: Sample temperature every 2 seconds from all

[1]The new version had not been released at the time we designed our experiments; version 1.0 was used. This is unlikely to have significantly effected results. Although the new version of SwissQM allows users to write query code, temporal queries are not supported.

[2]We intend to consider the interactions between language design, performance, and reliability in future work.

[3]We could not get TinyDB working and the release tool does not support the semantics for temporal query, so we wrote a new simulator for it.



	Variable	=	Expression	Period
1	mylight	=	light	1 sec
2	mypressue	=	pressure	
3	height	=	mypressue / 100 + 2	
4	my_avg_light	=	avg_time(mylight[0:3])	4 sec

	collect	group nodes by	node filtering	group filtering	maximum delay
1	height, avg(my_avg_light)	height,			eventually

Figure 1: User interface of WASP2.

the nodes in the network. Transmit the node identification numbers and the most recent temperature readings from nodes where the current temperature exceeds 1.1 times the maximum temperature reading during the preceding 10 seconds.

7. EXPERIMENTAL RESULTS

The user study evaluated five programming languages when used by 28 novice programmers. This section presents and analyzes the study results.

7.1 Results of User Study

Each of the five languages, except SwissQM, is evaluated based on use by five novice programmers. SwissQM cannot support Task 3 so it was only tested with three test subjects. Our test subjects are from a variety of fields: science, engineering, arts, etc. Ten of them have no programming experience. The others, mostly students in engineering fields, have different levels of experience with Fortan, C, and Matlab. We claim that the level of programming experience for this population is representative of wireless sensor network application expert. By randomly assigning languages to participants, each language was tested by a combination of participants with different background and programming experience.

We used *success rate* and *time-to-success* to quantify programming productivity. Table 3 shows the success rate and the average time-to-success for each language and task. The success rate is shown in the form of n/m, meaning n participants out of m succeeded within 40 minutes. The last column shows the average time-to-success of the n participants. The success rates of TinyScript and TinyTemplate were extremely low; only one test subject completed the simplest task with TinyTemplate. TinySQL and WASP have similar productivity for Task 1 and Task 2, but differ for Task 3. None of the test subjects completed Task 3 with TinySQL, while two out of four succeeded at Task 3 with WASP. The average success rate for TinySQL is 47.2%. The average success rate of WASP is 66.7%. SwissQM has 100% success rate and the shortest completion time for Task 1 and Task 2. However, it does not support Task 3.

The failure of test subjects to complete any tasks with TinyScript suggests that TinyScript may be less appropriate for novice programmers than the other types of languages evaluated. When reviewing mistakes programmers made during the study, we observed two significant obstacles encountered by TinyTemplate users. First, users had trouble with the event-driven programming model; most participants ended up using the wrong event handlers. In addition, many users fell back to programming in an imperative manner. They attempted to specify periodic events with a for loop. Second, novice programmers had trouble implementing node communication. None of the TinyScript users wrote the piece of code required for communication between two nodes (this requires calling the *bcast* function to send the data in a buffer, and calling the *broadcast* function in the broadcast handler to retrieve the data). These issues hindered most programmers from further understanding the programming template provided in TinyTemplate. The difficulty of using TinyScript for novice programmers was also demonstrated in the BASIC study [30] by Miller et al., which used simpler tasks. The increased success rate with TinyTemplate relative to TinyScript implies the benefit of a specialized language with a programming template over a general-purpose language. It is intuitive for WASP and TinySQL to have similar results for Task 1 and Task 2, as the solutions for these tasks are similar in both languages. WASP improved the success rate of Task 3 from 0 to 50% because it allows simpler semantics for accessing temporal data. The short development times of SwissQM for Tasks 1 and 2 can be attributed to two features: (1) SwissQM is well specialized to these tasks and (2) SwissQM provides an easy-to-use graphical user interface for composing a program.

Although it is difficult to interpret the results of statistical tests on small data sets, we did apply the t-test and the rank sum test to evaluate the significance of the differences in success rates and completion times among the languages. The results suggest that, with high confidence, we can conclude (1) WASP has a higher success rate than TinyScript and TinyTemplate; (2) the mean completion time for Task 1 using SwissQM is longer than that using WASP2; and (3) the mean completion time for Task 2 using WASP is longer than that using SwissQM. The details of this analysis and its caveats can be found in the extended technical report version of this paper [3].

In addition to the objective metrics of success rate and time-to-success, we also asked participants to provide their feedback on the study-related factors. They were asked to use a number, on the scale from 1 to 7, to indicate their agreement with the following statements (1 means strongly disagree and 7 means strongly agree):

1. The tutorial is easy to understand.
2. The descriptions of the tasks are clear.
3. The programming environment is friendly.
4. I understand this programming language.
5. The programming language is easy to learn and use.

Statements 1, 2, and 3 help us to identify problems caused by difficulty in understanding the manual or the task descriptions. They also provide additional evidence that may help to better understand the success rate and time-to-success results. For example, if users of one language had difficulty understanding the given tasks but users of another language understood the tasks well, we could not conclude that the difference in productivity is due to the accessibility of these languages. We tried our best to avoid bias caused by factors other than the languages, although we could not totally eliminate biases resulting from differences in the released tool chains of existing languages.

Table 4 shows the average ratings for each language. The five rightmost columns correspond to the statements listed above. According to the results, all participants understood the tutorials and tasks. TinyScript and TinyTemplate have less friendly programming environments, but this is not the major reason for their low success rates. Screen recordings indicated that the programming interface did not introduce any trouble for the participants editing the code or running the simulation. Statements (4) and (5) focus on user experience with the languages. In contrary to our expectations, the TinyScript users think they understand the language better than TinyTemplate users. This may seem counter-intuitive because the TinyTemplate language is a simplified version of TinyScript. The simplest explanation we have found for this irregularity is that TinyScript users are over confident in their understanding of the language. There is some evidence for this conclusion; several TinyScript users made the error of using the *uart* function to send data to the base station. On the contrary, the TinyTemplate users were able to see that the correct implementation for data collection is more complicated, thanks to the template. The ratings for the last statement order languages by perceived difficulty. This ordering is consistent with actual productivity: in general, the harder a language is perceived to be, the lower its success rate.

7.2 Enhanced Version of WASP

From the user study, we observed that, although appropriate programming idioms for Tasks 1 and 2 are similar for SwissQM and WASP, the average completion time for WASP is almost 3× that of SwissQM. By studying screen recordings, we found that test subjects spent a significant amount of time locating and correcting syntax errors in WASP. This was not the case for SwissQM because its interface prohibits many syntax errors; part of the programming is done via selecting from lists and checking radio buttons. The WASP interface provides a text window into which arbitrary text may be entered to compose a program. Program errors are not detected until the syntax check or simulation is started by the users by clicking the associated button. To investigate the impact of user interface on sensor network programming by novice programmers, we designed WASP2. WASP2 is linguistically similar to WASP, but the user interface has a number of enhancements.

Instead of letting users input arbitrary code, WASP2 provides dialogs for composing different types of instructions. The dialogs are equipped with lists containing already-defined variables, validators, auto-completers, syntax checkers, and pop-up warning messages to accelerate instruction editing, prevent user error, and detect syntax errors as early as possible. The widgets in the interface are tagged with pop-up windows displaying appropriate descriptions and help so that programmers do not need to frequently refer to the tutorials. The WASP2 interface is shown in Figure 1. We repeated the original study protocol for WASP2 with another five novice programmers. The results are shown in Table 3 and Table 4. Compared with WASP, WASP2 improves success rate by 20.8%, and reduces average development time by 58.2%. WASP2 results in better productivity than SwissQM: 20.8% improvement in success rate and 25.3% reduction in average development time for Tasks 1 and 2 (recall that SwissQM does not permit the temporal queries required for some tasks in Archetype 1).

Table 3: Results of user study

Language	Success rate			Develop time (min)		
	T1	T2	T3	T1	T2	T3
SwissQM	3/3	3/3	N.A.	5.7	11.3	N.A.
WASP	2/2	2/4	2/4	16	31	29.5
TinySQL	3/4	2/3	0/3	17.7	27.5	N.A.
TinyTemplate	1/4	0/3	0/3	34	N.A.	N.A.
TinyScript	0/3	0/3	0/4	N.A.	N.A.	N.A.
WASP2*	3/3	3/4	2/3	3	9.7	23.5

*WASP2 is presented in Section 7.2.

Table 4: Feedbacks from test subjects

Language	Tutorial	Task	Env.	Understand lang.	Lang. easy
SwissQM	5.7	5.7	5	5.7	6
TinyScript	4.4	5.4	4.2	3.8	3.2
TinySQL	4.6	5.8	5.6	4	4.8
TinyTemplate	5.2	5.6	4.6	2.8	3.2
WASP	4.4	5.4	5.8	4.2	4.6
WASP2*	4.4	6.2	5.8	5.2	4.8

*WASP2 is presented in Section 7.2.

7.3 Lessons Learned

Conducting user studies allowed us to test our hypothesis, evaluate and improve our design, and develop new ideas that would not occurred to us if we were isolated from users. Unfortunately, user studies quickly consume budgets and time, so it is desirable to carefully design them. This section summarizes the lessons we learned and our observations.

- Conduct a small-scale test study first to minimize unexpected problems in the later, large-scale study. A test study needs not be as strict as the formal study. We tested several tasks with EECS students and our collaborators before the study, though the final study requires randomly recruited strangers who are not programmers. Our test study revealed some bugs in the programming tools and helped us determine a reasonable time limit for the large-scale study.

- Observe the user and record the study without interrupting or intervening. Having a record of the user's behavior allows later analysis of anomalies. During our study, the screen was recorded, the observer took notes on the questions users asked, and users could sketch on draft papers and or mark anything in the language manual. These records allowed us to determine where users become stuck and what mistakes they make.

- Try to consider all factors that may affect your study results. We were interested in the differences among programming languages, so we eliminated the effects introduced by the editor and working environment by using the same editor and similar simulators for every test.

- End user behavior can be very different from designers expectations.

- Programming examples are helpful to novice programmers. Many test subjects indicated that they found the templates helpful and most of them suggested including more examples in the language tutorials.

- Event-driven models, explicit programming data communication, and table joins are hard for novice programmers to grasp.

8. CONCLUSIONS

Application-level programming languages that are accessible to novice programmers have the potential to open sensor network design to application experts. We have proposed the concept of sensor network archetypes and built an archetype taxonomy based on 23 applications. This concept supports the design of compact archetype-specific languages. We developed a high-level language for the most frequently encountered archetype. Our user study of 28 novice programmers using five programming languages indicates that archetype-specific languages have the potential to substantially improve the success rates and reduce programming times for novice programmers compared with existing general-purpose and/or node-level sensor network programming languages. Our language, WASP, increased the success rate by $1.6\times$ and reduced average development time by 44.4% compared to other languages. This work is the first step towards our ultimate goal of developing a set of archetype-specific programming language for wireless sensor network applications. We are also working on an optimized implementation of WASP2.

9. ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under awards CNS-0721978 and CNS-0347941. We would like to acknowledge J. Scott Miller, Lawrence Henschen, and David Bild for their helpful suggestions on this work. Lan Bai and Robert Dick were affiliated with Northwestern University during much of this work.

10. REFERENCES

[1] http://absynth-project.org/userstudy.

[2] Arora, A., Dutta, P., Bapat, S., Kulathumani, V., Zhang, H., Naik, V., Mittal, V., Cao, H., Demirbas, M., Gouda, M., Choi, Y., Herman, T., Kulkarni, S., Arumugam, U., Nesterenko, M., Vora, A., and Miyashita, M. A line in the sand: A wireless sensor network for target detection, classification, and tracking. *J. Computer Networks 46* (2004), 605–634.

[3] Bai, L. S., Dick, R. P., and Dinda, P. A. Toward archetype-based design: Opening sensor networks to application experts. Tech. rep., Michigan University, 2009.

[4] Bakshi, A., Prasanna, V. K., Reich, J., and Larner, D. The abstract task graph: a methodology for architecture-independent programming of networked sensor systems. In *Proc. Int. Conf. Mobile Systems, Applications, and Services* (June 2005), pp. 19–24.

[5] Bischoff, U., and Kortuem, G. A state-based programming model and system for wireless sensor networks. In *Proc. Int. Conf. on Pervasive Computing and Communications Wkshp.* (Mar. 2007), pp. 261–266.

[6] Chintalapudi, K., Fu, T., Paek, J., Kothari, N., Rangwala, S., Caffrey, J., Govindan, R., Johnson, E., and Masri, S. Monitoring civil structures with a wireless sensor network. *J. Internet Computing 10*, 2 (Mar. 2006), 26–34.

[7] Chintalapudi, K., Paek, J., Prakash, O., Fu, T., Dantu, K., Caffrey, J., Govindan, R., and Johnson, E. Structural damage detection and localization using NETSHM. In *Proc. Int. Symp. Information Processing in Sensor Networks* (Apr. 2006), pp. 475–482.

[8] Dowding, C. H., Ozer, H., and Kotowsky, M. Wireless crack measurment for control of construction vibrations. In *Proc. Atlanta GeoCongress* (Feb. 2006).

[9] Elson, J., and Parker, A. Tinker: a tool for designing data-centric sensor networks. In *Proc. Int. Symp. Information Processing in Sensor Networks* (Apr. 2006), pp. 350–357.

[10] Excel home page. http://office.microsoft.com/en-us/excel.

[11] Gay, D., Levis, P., Culler, D., and Brewer, E. nesC 1.1 language reference manual, May 2003.

[12] Ghercioiu, M. A graphical programming approach to wireless sensor network nodes. In *Proc. Sensors for Industry Conf.* (Feb. 2005), pp. 118–121.

[13] GOMOLL, K. Some techniques for observing users. In *The Art of Human-Computer Interface Design*, B. Laurel, Ed. Addison-Wesley, 1990, pp. 85–90.

[14] GREENSTEIN, B., KOHLER, E., AND ESTRIN, D. A sensor network application construction kit (SNACK). In *Proc. Int. Conf. Embedded Networked Sensor Systems* (Nov. 2004), pp. 69–80.

[15] GU, L., JIA, D., VICAIRE, P., YAN, T., LUO, L., TIRUMALA, A., CAO, Q., HE, T., STANKOVIC, J. A., ABDELZAHER, T., AND KROGH, B. H. Lightweight detection and classification for wireless sensor networks in realistic environments. In *Proc. Int. Conf. Embedded Networked Sensor Systems* (Nov. 2005), pp. 205–217.

[16] HARTUNG, C., HAN, R., SEIELSTAD, C., AND HOLBROOK, S. FireWxNet: a multi-tiered portable wireless system for monitoring weather conditions in wildland fire environments. In *Proc. Int. Conf. Mobile Systems, Applications, and Services* (June 2006), pp. 28–41.

[17] HOREY, J., NELSON, E., AND MACCABE, A. B. Tables: A table-based language environment for sensor networks. Tech. rep., The University of New Mexico, 2007.

[18] KIM, S., PAKZAD, S., CULLER, D., DEMMEL, J., FENVES, G., GLASER, S., AND TURON, M. Health monitoring of civil infrastructures using wireless sensor networks. In *Proc. Int. Symp. Information Processing in Sensor Networks* (Apr. 2007), pp. 254–263.

[19] KOTHARI, N., GUMMADI, R., MILLSTEIN, T., AND GOVINDAN, R. Reliable and efficient programming abstractions for wireless sensor networks. In *Proc. Programming Languges Design and Implementation* (June 2007), pp. 200–210.

[20] KUNIAVSKY, M. *Observing the User Experience: A Practitioner's Guide to User Research*. Morgan Kaufmann, 2003.

[21] NI LabVIEW. http://www.ni.com/labview.

[22] LEVIS, P. The TinyScript language, July 2004.

[23] LEVIS, P., AND CULLER, D. Mate: A tiny virtual machine for sensor networks. In *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems* (Oct. 2002).

[24] LEVIS, P., GAY, D., AND CULLER, D. Bridging the gap: Programming sensor networks with application specific virtual machines. Tech. rep., UC Berkeley, Aug. 2004.

[25] LEVIS, P., LEE, N., WELSH, M., AND CULLER, D. E. TOSSIM: accurate and scalable simulation of entire tinyOS applications. In *Proc. Int. Conf. Embedded Networked Sensor Systems* (Nov. 2003), pp. 126–137.

[26] LI, M., AND LIU, Y. Underground structure monitoring with wireless sensor networks. In *Proc. Int. Symp. Information Processing in Sensor Networks* (Apr. 2007), pp. 69–78.

[27] LIU, T., SADLER, C. M., ZHANG, P., AND MARTONOSI, M. Implementing software on resource-constrained mobile sensors: experiences with Impala and ZebraNet. In *Proc. Int. Conf. on Mobile systems, Applications, and Services* (June 2004), pp. 256–269.

[28] MADDEN, S., FRANKLIN, M., HELLERSTEIN, J., AND HONG, W. TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. Database Systems 30*, 1 (Mar. 2005), 122–173.

[29] MADDEN, S., HELLERSTEIN, J., AND HONG, W. TinyDB: In-network query processing in TinyOS, Sept. 2003.

[30] MILLER, J. S., DINDA, P., AND DICK, R. GOTO considered helpful: A BASIC approach to sensor network node programming. Tech. rep., Northwestern University, Jan. 2009.

[31] MOTTOLA, L., AND PICCO, G. P. Programming wireless sensor networks: Fundamental concepts and state of the art. Tech. rep., University of Trento, 2008.

[32] MÜLLER, R. SwissQM query interface. http://www.swissqm.inf.ethz.ch/querygui.html.

[33] MÜLLER, R., ALONSO, G., AND KOSSMANN, D. SwissQM: Next generation data processing in sensor networks. In *Proc. Conf. on Innovative Data Systems Research* (Jan. 2007), pp. 1–9.

[34] NEWTON, R., MORRISETT, G., AND WELSH, M. The regiment macroprogramming system. In *Proc. Int. Symp. Information Processing in Sensor Networks* (Apr. 2007), pp. 489–498.

[35] PAEK, J., CHINTALAPUDI, K., GOVINDAN, R., CAFFREY, J., AND MASRI, S. A wireless sensor network for structural health monitoring: performance and experience. In *Proc. Wkshp. on Embedded Networked Sensors* (2005), pp. 1–9.

[36] PLY. http://www.dabeaz.com/ply.

[37] POLASTRE, J., SZEWCZYK, R., MAINWARING, A., CULLER, D., AND ANDERSON, J. Analysis of wireless sensor networks for habitat monitoring. In *Wireless Sensor Networks*, C. S. Raghavendra, K. M. Sivalingam, and T. Znati, Eds. Springer US, 2004, ch. 18, pp. 399–423.

[38] RÖMER, K., AND MATTERN, F. The design space of wireless sensor networks. *J. of Wireless Communications 11*, 6 (Dec. 2004), 54–61.

[39] SHARP, C., SCHAFFERT, S., WOO, A., SASTRY, N., KARLOF, C., SASTRY, S., AND CULLER, D. Design and implementation of a sensor network system for vehicle tracking and autonomous interception. In *Proc. European Workshop on Wireless Sensor Networks* (Jan. 2005), pp. 93–107.

[40] SHETH, A., TEJASWI, K., MEHTA, P., PAREKH, C., BANSAL, R., MERCHANT, S. N., SINGH, T. N., DESAI, U. B., THEKKATH, C. A., AND TOYAMA, K. Senslide: a sensor network based landslide prediction aystem. In *Proc. Int. Conf. Embedded Networked Sensor Systems* (Nov. 2005), pp. 280–281.

[41] SHNAYDER, V., RONG CHEN, B., LORINCZ, K., THADDEUS, AND WELSH, M. Sensor networks for medical care. Tech. rep., Harvard University, Apr. 2005.

[42] SIKKA, P., CORKE, P., VALENCIA, P., CROSSMAN, C., SWAIN, D., AND BISHOP-HURLEY, G. Wireless adhoc sensor and actuator networks on the farm. In *Proc. Int. Symp. Information Processing in Sensor Networks* (Apr. 2006), pp. 492–499.

[43] SIMON, G., MARÓTI, M., LÉDECZI, Á., BALOGH, G., KUSY, B., NÁDAS, A., PAP, G., SALLAI, J., AND FRAMPTON, K. Sensor network-based countersniper system. In *Proc. Int. Conf. Embedded Networked Sensor Systems* (Nov. 2004), pp. 1–12.

[44] SINGHVI, V., KRAUSE, A., GUESTRIN, C., JAMES H. GARRETT, J., AND MATTHEWS, H. S. Intelligent light control using sensor networks. In *Proc. Int. Conf. Embedded Networked Sensor Systems* (Nov. 2005), pp. 218–229.

[45] STOIANOV, I., NACHMAN, L., MADDEN, S., AND TOKMOULINE, T. Pipenet: a wireless sensor network for pipeline monitoring. In *Proc. Int. Symp. Information Processing in Sensor Networks* (Apr. 2007), pp. 264–273.

[46] SUGIHARA, R., AND GUPTA, R. K. Programming models for sensor networks: A survey. *ACM Trans. on Sensor Networks 4*, 2 (Mar. 2008), 1–29.

[47] TOLLE, G., POLASTRE, J., SZEWCZYK, R., CULLER, D., TURNER, N., TU, K., BURGESS, S., DAWSON, T., BUONADONNA, P., GAY, D., AND HONG, W. A macroscope in the redwoods. In *Proc. Int. Conf. Embedded Networked Sensor Systems* (2005), pp. 51–63.

[48] WARK, T., CROSSMAN, C., HU, W., GUO, Y., VALENCIA, P., SIKKA, P., CORKE, P., LEE, C., HENSHALL, J., PRAYAGA, K., O'GRADY, J., REED, M., AND FISHER, A. The design and evaluation of a mobile sensor/actuator network for autonomous animal control. In *Proc. Int. Symp. Information Processing in Sensor Networks* (Apr. 2007), pp. 206–215.

[49] WEINBERG, G. M. *The Psychology of Computer Programming*. Van Nostrand Reinhold, 1971.

[50] WERNER-ALLEN, G., JOHNSON, J., RUIZ, M., LEES, J., AND WELSH, M. Monitoring volcanic eruptions with a wireless sensor network. In *European Workshop on Wireless Sensor Networks* (2005).

[51] WOOD, A., VIRONE, G., DOAN, T., CAO, Q., SELAVO, L., WU, Y., FANG, L., HE, Z., LIN, S., AND STANKOVIC, J. ALARM-NET: Wireless sensor networks for assisted-living and residential monitoring. Tech. rep., University of Virginia, Jan. 2006.

[52] YANG, P., FREEMAN, R., AND LYNCH, K. Distributed cooperative active sensing using consensus filters. In *Proc. Int. Conf. Robotics & Automation* (Apr. 2007), pp. 405–410.