Programming Paradigms for Networked Sensing: A Distributed Systems' Perspective*

Amol Bakshi and Viktor K. Prasanna

Department of Electrical Engineering,
University of Southern California, Los Angeles, CA 90089, USA
{amol, prasanna}@usc.edu

Abstract. Research in embedded networked sensing has primarily focused on the design of hardware architectures for sensor nodes and infrastructure protocols for long lived operation of resource constrained sensor network deployments. There is now an increasing interest in the programming aspects of sensor networks, especially in the broader context of pervasive computing. This paper provides a brief overview of ongoing research in programming of sensor networks and classifies it into layers of abstraction that provide the application developer with progressively higher level primitives to express distributed, phenomenon-centric collaborative computation. As a specific instance of a macroprogramming methodology, we discuss the data driven Abstract Task Graph (ATaG) model and the structure of its underlying runtime system. ATaG separates the application functionality from non-functional aspects, thereby enabling end-to-end architecture-independent programming and automatic software synthesis for a class of networked sensor systems. A prototype visual programming, software synthesis, functional simulation and visualization environment for ATaG has been implemented.

1 Introduction

Distributed sensor networks allow intelligent, dense monitoring and control of physical environments and have a wide range of applications such as home and office automation, habitat monitoring, intruder detection, etc. Advances in VLSI technology have enabled the integration of sensing, computation, and wireless communication capabilities into small, inexpensive hardware platforms. Ad hoc wireless sensor networks (WSNs) comprised of such untethered nodes provide embedded sense-and-response capability. The unprecedented degree of access to information about the physical world could provide context awareness to other applications, making WSNs an integral part of the vision of pervasive, ubiquitous computing – with the long term objective of seamlessly integrating this fine grained sensing infrastructure into larger, multi-tier systems. A comprehensive overview of state of the art in wireless embedded sensing can be found in [1].

* This work is supported by the National Science Foundation, USA, under grant number IIS-0330445.

A. Pal et al. (Eds.): IWDC 2005, LNCS 3741, pp. 451–462, 2005.

With continuing advancements in sensor node design and increasingly complex applications, interest in automatic synthesis of sensor network applications is inevitable, given the fact that ease of programming is perhaps the single most important determinant of the ubiquity and acceptance of a computing platform. In other words, there should be a well-defined methodology to translate high level intentions of the programmer expressed in a suitable formalism into an executable specification for the underlying deployment.

In this paper, we focus on programming of networked sensor systems from a distributed systems perspective. We assume that protocols and services for the basic communication and collaboration infrastructure are already available for the target platform. The job of the programming model is to suitably abstract these existing services and define a model of computation for the distributed system that is useful for application development. A collection of autonomous sensor nodes passing messages through a communication network fits the definition of a distributed computing system. However, some of the fundamentally new characteristics of networked sensing systems that differentiate them from traditional parallel and distributed computing are as follows:

- **Transformational vs. reactive processing:** Most of the traditional parallel and distributed applications are transformational systems characterized by a function that maps input data to output data. The main purpose of parallelism for such systems is to reduce the overall latency of computation and to provide robustness through replication [2]. A networked sensor system is not transformational but is primarily reactive in that it has to continuously respond to external and internal stimuli. An event of interest in the environment triggers computation and communication in one or more nodes of the network, usually in the immediate vicinity of the event.
- **Nature of input data:** In transformational distributed systems, a given set of input data is statically and/or dynamically distributed among various computing nodes in order to perform the 'transformation' with lowest latency. In sensor networks, however, most the data is continuously created in the network through the act of sampling the sensor interfaces. The time and location of origin of a particular piece of data influences the processing performed on it. The typical untethered wireless sensor node is energy constrained. It is desirable to process the data as close to the source as possible, and collaborative, in-network data processing is hence an important consideration in networked sensing.
- **Spatial awareness:** From the end users' perspective, an embedded sensor network ultimately represents a discrete sampling of a continuous physical space. Instead of specifying applications in terms of sensor nodes and the network connectivity, behaviors can be naturally specified using spatial abstractions. For instance, the exact placement of sensor nodes will probably be of incidental interest as long as the set of sensing tasks mapped onto a subset of those nodes at any given time collaboratively ensure the desired degree of coverage.

Macroprogramming of sensor networks broadly refers to an application development methodology – supported by a suitable programming model, compiler, and runtime support – that liberates the programmer from having to compose the complex control, coordination, and state maintenance mechanisms at the individual node in order to accomplish the desired global behavior.

Low level optimizations especially related to the networking layer are important for long lived operation of untethered resource-constrained networks, and protocols for positioning, time synchronization, etc., provide the basic infrastructure for distributed computing in the sensor network. The challenge in defining high level macroprogramming models is achieving the right balance between long lived operation through low level optimizations and ease of application development by hiding most of the low level details from the programmer.

In the next section, we analyze the layers of programming abstractions that naturally emerge from the ongoing research in programming models in the sensor network community. Section 3 discusses our macroprogramming model called the Abstract Task Graph (ATaG) [17] that builds upon the core concepts of the data driven computing paradigm to allow domain experts to develop sensor network applications. ATaG provides support for reactive processing, mechanisms to concisely indicate distributed, in-network collaborative computation, and support for space awareness both for expressing collaborative computation in terms of spatial neighborhoods, and to express notions such as spatial density of task placement that can be used to provide a desired degree of sensing coverage. Details of the ATaG programming model and the runtime system can be found in [3] and [4] respectively. We conclude in Section 4 with a discussion of our broader vision and related work in the context of design automation for sensor networks.

2 Layers of Programming Abstraction

Figure 1 depicts our view of the emerging layers of programming abstraction for networked sensor systems. Many protocols have been implemented to provide the basic mechanisms for efficient infrastructure establishment and communication in ad hoc deployments. These include energy-efficient medium access, positioning, time synchronization, and a variety of routing protocols such as data centric and geographic routing that are unique to spatial computing in embedded networked sensing. Ongoing research, such as MiLAN [5] is focusing on sensor data composition as part of the basic infrastructure. A sensor data composition framework delegates the responsibility of interfacing with physical sensors and aggregating the data into meaningful application-level variables to an underlying middleware instead requiring its incorporation as part of the application-level logic [6].

2.1 Service-Oriented Specification

To handle the complexity of programming heterogeneous, large-scale, and possibly dynamic sensor network deployments and to make the computing substrate

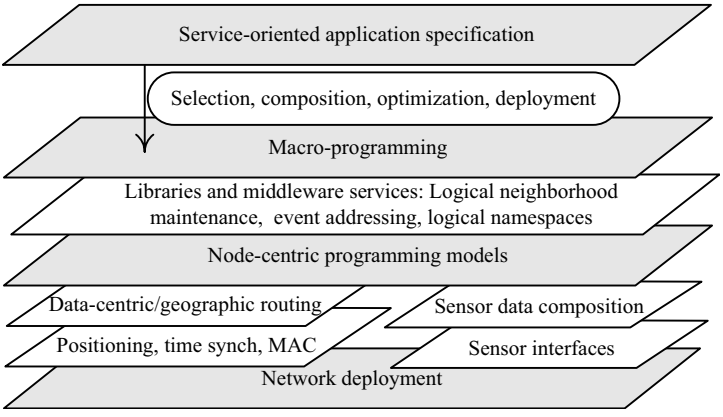

Fig. 1. Layers of abstraction for application development on WSNs

accessible to the non-expert, the highest level of programming abstraction for a sensor network is likely to be a purely declarative language. The Semantic Streams markup and query language [7] is an example of such a language that can be used by end users to query for semantic information without worrying about how the corresponding raw sensor data is gathered and aggregated. The basic idea is to abstract the collaborative computing applications in the network as a set of services, and provide a query interpretation, planning, and resource management engine to translate the service requirements specified by the end user into a customized distributed computing application that provides the result. A declarative, service-oriented specification allows dynamic tasking of the network by multiple users and is also easier to understand compared to low level distributed programming.

2.2 Macroprogramming

The objective of macroprogramming is to allow the programmer to write a distributed sensing application without explicitly managing control, coordination, and state maintenance at the individual node level. Macroprogramming languages provide abstractions that can specify aggregate behaviors that are automatically synthesized into software for each node in the target deployment. The structure of the underlying runtime system will depend on the particular programming model. While service-oriented specification is likely to be invariably declarative, various program flow mechanisms - functional, dataflow, and imperative - are being explored as the basis for macroprogramming languages. Regiment [8] is a declarative functional language based on Haskell, with support for region-based aggregation, filtering, and function mapping. Kairos [9] is an imperative, control-driven macroprogramming language for sensor networks that allows the application developer to write a single centralized program that operates on a centralized memory model of the sensor network state.

ATaG [3] (discussed in more detail in the next section) explores the data flow paradigm as a basis for architecture-independent programming of sensor network applications.

2.3 Node-Centric Programming

In node-centric programming, the programmer has to translate the global application behavior in terms of local actions on each node, and individually program the sensor nodes using languages such as nesC [10], galsC [11], C/C++, or Java. The program accesses local sensing interfaces, maintains application level state in the local memory, sends messages to other nodes addressed by node ID or location, and responds to incoming messages from other nodes. While node-centric programming allows manual cross-layer optimizations and thereby leads to efficient implementations, the required expertise and effort makes this approach insufficient for developing sophisticated application behaviors for large-scale sensor networks.

The concept of a logical neighborhood – defined in terms of distance, hops, or other attributes – is common in node-centric programming. Common operations upon the logical neighborhood include gathering data from all neighbors, disseminating data to all neighbors, applying a computational transform to specific values stored in the neighbors, etc. The usefulness and ubiquity of neighborhood creation and maintenance has motivated the design of node-level libraries [12], [13] that handle the low level details of control and coordination and provide a neighborhood API to the programmer.

Middleware services [5], [14], [15] also increase the level of programming abstraction by providing facilities such as phenomenon-centric abstractions. Middleware services could create virtual topologies such as meshes and trees in the network, allow the program to address other nodes in terms of logical, dynamic relationships such as leader-follower or parent-child, support state-centric programming models [16], etc. The middleware protocols themselves will typically be implemented using node-centric programming models, and could possibly but not necessarily use communication libraries as part of their implementation.

3 Data Driven Macroprogramming with the Abstract Task Graph

The Abstract Task Graph (ATaG) [3], [17] seeks to raise the level of programming abstraction by (a) allowing the architecture-independent specification of application behavior through a mixed imperative-declarative program specification, and (b) transferring the responsibility of low level coordination, communication, and optimization to an underlying runtime system, thereby allowing the application developer to focus on high level behavioral aspects.

Macroprogramming broadly refers to the collaborative tasking of sensor nodes as opposed to configuring individual node behaviors. ATaG support macroness at the application level by allowing the programmer to define and manipulate information at the desired level of abstraction without worrying about how

the information is created. ATaG also supports macro-ness at the architecture level by allowing concise specification of common patterns of in-network distributed processing such as neighbor-to-neighbor, many-to-one, tree-based, etc.

3.1 Objectives and Key Concepts

ATaG is designed to support intuitive expression of reactive processing, spatial awareness, network awareness, architecture independence, and composability. The first three are the functional objectives that allow concise and intuitive expression of the behavior of a networked sensing application. The non-functional objectives – architecture independence and composability – are motivated by software development concerns such as ease of programming and code reusability.

To accomplish these objectives, ATaG employs a *data driven programming model* and *mixed imperative-declarative program specification* for separation of concerns. Tasks are defined in terms of their input and output data objects. An underlying runtime system manages task scheduling and inter-task communication. Availability of operands triggers task execution, subject to firing rules. This model is attractive for computing in distributed systems for programming convenience, and the modularity and extensibility of the programs. Also, a sensor network can be viewed as a system for domain-specific transformation of sensor data and many applications can be naturally expressed as a set of transformations on raw and processed sensor readings.

The mixed imperative-declarative specification separates the 'when and where' of processing from the 'what'. The same program can be compiled for a different network size and topology by interpreting the declarative ('when and where') part in the context of that network architecture, while the imperative ('what') part remains unchanged. The ATaG programmer, who writes only the task implementations, is free to focus on application-level design without being concerned about low level details of the sensor node platform and the specifics of a particular deployment.

3.2 Illustrative Example

Figure 2 is a complete ATaG program for a sensor network application with two distinct behaviors. The first is to periodically sample the temperature at each node (through a temperature sensor), continuously compute the average reading, and log it at a designated root node. The second is to detect an object in the network through acoustic sensors, and report the current location of the target to a designated root node.

The figure shows the declarative part of the ATaG program, which consists of the set of abstract tasks (ovals), abstract data items (square rectangles), I/O dependencies or channels (arrows), and annotations (shaded rectangles). The imperative part consists of the user-supplied code associated with each of the five abstract tasks in the program, and the user-supplied data structures that represent the four abstract data items. Abstract tasks represent the types of processing in the application, abstract data items represent the types of application-specific

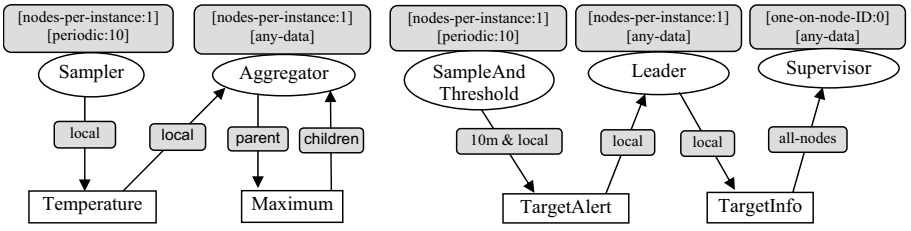

Fig. 2. An ATaG program for temperature monitoring and object tracking

data exchanged between instances of abstract tasks, and the abstract channels denote the I/O relationship between tasks and data. Task annotations govern the placement of task instances and the firing conditions of each instance. Channel annotations govern the scope of dissemination (collection) of instances of a particular data item produced (consumed) by an instance of the abstract task.

The dependencies and annotations in this program specify that the `Sampler` and `Aggregator` tasks are to be instantiated on each node. The `Sampler` periodically produces a `Temperature` reading which is routed to the `Aggregator` task on the same node. The `Aggregator` receives temperature readings from its own node and from its child nodes in a logical tree structure maintained by the runtime. The `Aggregator` is fired whenever an instance of `Temperature` is produced on its node or on its child nodes. The aggregated reading is conveyed up the tree. The object tracking algorithm depicted here is based on the one discussed in [12]. Briefly, each node determines whether the object is in its vicinity by periodically sampling and thresholding the reading from its acoustic sensor. If a target is detected (a `TargetAlert` is produced), all the nodes that detect the target broadcast their readings to all other nodes that might have detected the same target - in this example, the assumption is that a 10m radius includes this set of nodes. The `Leader` task on each node receives all such readings, including its own. The `Leader` task on the node that has the highest reading calculates the target position and transmits the `TargetInfo` to the designated Supervisor on the root node.

The ATaG program has data driven semantics. A particular task instance is scheduled for execution when the firing rules of the abstract task are satisfied. A task can be specified as periodic with a specified period of execution, or its execution can be predicated on the occurrence of either (any-data) or all (all-data) of its input data items. This paradigm intuitively supports *reactive processing* because abstract data items can represent the occurrence of events such as the detection of an intruder, in addition to carrying information about the occurrence such as the location of the intruder. Each execution of a task may not necessarily result in the production of each of its output data items; depending on the (application-specific) semantics of the abstract data items, the output can be produced only when certain conditions are satisfied. For instance, in this example, `SampleAndThreshold` produces a `TargetAlert` only if the sampled reading exceeds a specific threshold and not otherwise.

Architecture independence is evident in the fact that both the task and channel annotations are independent of a particular network deployment. Task

annotations indicate requirements such as density of placement and can be generic (e.g., instantiate task on each node) or specific (e.g., instantiate task on node ID 0). The exact physical node which hosts an instance of the task will be determined when this program is compiled for a particular deployment. *Spatial awareness and network awareness* is also supported through channel annotations that allow a task to control the scope of input and output of data items. For instance, when an instance of the `SampleAndThreshold` task produces an instance of `TargetAlert`, it is disseminated by an underlying runtime system to all nodes within 10m of the producer. Similar annotations can be used to specify the neighborhood in terms of nodes, e.g., 'k-hop'. The application developer need not worry about how the neighborhood information is maintained at that node, what routing protocols are used for the communication, etc.

Finally, the data driven paradigm makes ATaG programs highly composable. The ATaG program in Figure 2 actually consists of two disjoint abstract task graphs, and can be considered as a larger application that is composed by concatenating the ATaG programs of two smaller applications, corresponding to temperature averaging and object tracking respectively. Composability is also enabled by the fact that the only methods available to a task for producing and consuming data items are the `put()` and `get()` methods respectively. Similar to the communication orthogonality of tuple spaces, these methods do not require the producer and consumer to know each other's identity. This enables distributed sharing of data both in space and in time. Also, since tasks are not coupled to each other, there is a high degree of code reuse since a new task can be added to the application without modifying the code associated with existing tasks.

3.3 Application Development Methodology

Figure 3 depicts the process of application development using ATaG. The input to the process is an ATaG program and a description of the target deployment

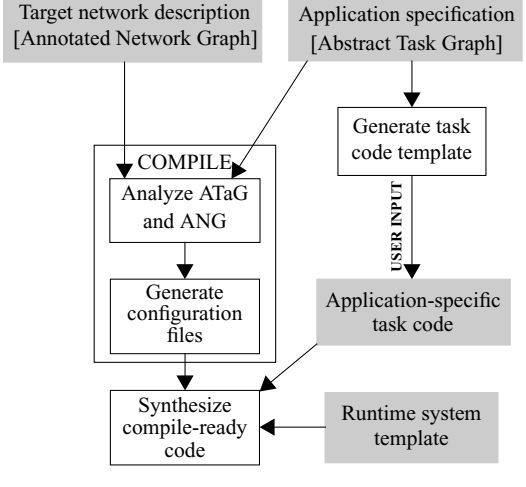

Fig. 3. Application development with ATaG

in the form of an annotated network graph (ANG), which is not discussed in this paper. The ANG contains information such as the number of nodes, the co-ordinates of each node, network connectivity, etc.

The graphical interface to the programming and synthesis environment is through a configurable graphical tool suite called the Generic Modeling Environment (GME) [18]. The declarative part of the ATaG program which consists of the various declarations and their annotations is specified visually. GME stores the model defined by the user in a canonical format. Tools called *model interpreters* can read from and write to this model database. In our case, model interpreters were written for the components represented by unshaded boxes in Figure 3.

4 Towards Design Automation: System-Level Support for Macroprogramming

In the context of programming methodologies for sensor networks, *design automation* refers to the automatic customization of the underlying system level support for a high level language. As depicted in Figure 1, the highest level of abstraction is a declarative specification that expresses the desired semantic information to be extracted from the system. This specification will ideally be translated into a macroprogram after suitable identification, selection, and composition of the individual behaviors that collaborative provide the desired service. The macroprogram in turn is compiled into a distributed software system that includes the application-level functionality as well as the mechanisms for control and coordination within a node and between nodes in the system.

The design of the underlying runtime system is critical for design automation because a well designed runtime system can (i) greatly simplify the compilation and code generation process, and (ii) allow plug and play integration of the various low level protocols and services whose choice could be influenced at compile time by the performance requirements of the end user. The data-driven ATaG runtime (DART) [4] is designed to separate application-independent mechanisms for control and coordination from application-specific configuration information to customize the individual node behavior.

Figure 4 is a high level overview of the modular structure of the data driven ATaG runtime called DART. Each module offers a well-defined interface to other modules in the system, and has complete ownership of the data and the protocols required to provide that functionality. This reduces interactions and dependencies among modules, and hiding the module implementation allows an entirely different set of protocols to be used within a module as long as the interface is not affected. We briefly summarize the purpose of each module - details can be found in [4]. The *ATaGManager* stores the information from the user-specified ATaG program that is relevant to the particular node. This information includes task annotations such as firing rule and I/O dependencies, and the annotations of input and output channels associated with the data items that are produced or consumed by tasks on the node. *Datapool* is responsible

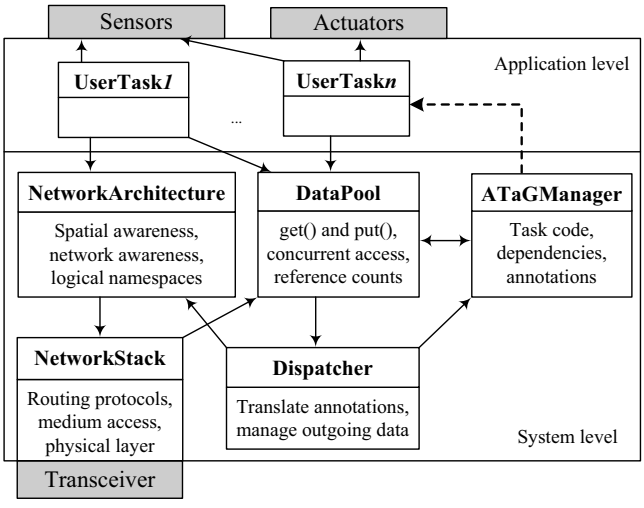

Fig. 4. The structure of the DART runtime system

for managing the instances of abstract data items produced or consumed at the node. *NetworkArchitecture* is responsible for maintaining all information about the real and virtual topology of the network. *NetworkStack* is in charge of communication with other nodes in the network, and manages the routing, medium access, and physical layer protocols. *Dispatcher* is responsible for disseminating data items that are produced on the node to other nodes in the network as specified in the ATaG program. In addition, a *Startup* module is responsible for initializing node-level services such as the transceiver functionality, the protocols for topology discovery, etc., and then starting the initial set of application-level tasks. The remainder of the execution is driven by the side-effects of the `get()` and `put()` calls made by the tasks, and the data items arriving over the network interface for addition to the data pool.

During the normal course of application execution, there are three main events that can occur: (i) a `get()` invocation by a user task, (ii) a `put()` invocation by a user task, or (iii) a `put()` invocation by the receiver thread when a data item arrives from another node. When a `get()` invocation occurs, *DataPool* merely decrements the reference count of the data item in question. When a local task invokes a `put()`, *DataPool* first checks if the corresponding data item is inactive before adding the newly produced data instance to the pool. This check ensures that all currently scheduled tasks that have been triggered by the production of a particular data instance get a chance to consume the data before it is overwritten by the same or different producer. *DataPool* the informs *ATaGManager* about the production of the data. *ATaGManager* determines the list of tasks that depend on this data item, checks their firing rules, and schedules the eligible tasks for execution. *DataPool* then notifies *Dispatcher* and finally returns control to the user task. *Dispatcher* interacts with *ATaGManager*,

NetworkArchitecture, and *NetworkStack* to send the data item to other nodes as indicated by the ATaG program. When the third type of event - an invocation of `put()` by the receiver thread of the *NetworkStack* - occurs, it is handled in much the same way as a local invocation, except that *Dispatcher* is not part of the loop.

There are three classes of APIs available to the ATaG programmer: (i) `get()/put()` calls to the data pool, (ii) the network-awareness and spatial-awareness API that allows a task instance to determine the composition of its neighborhood, and (iii) the API to the sensor interface. Since the runtime system does not know the access pattern of each task to the sensing interface, it cannot optimize resource usage. To enable resource management by the runtime requires, there should be a way for tasks to specify their sensor data requirements at a high level and leave the details of interfacing with sensors to the runtime system. In future work, we plan to extend the ATaG model with a special class of abstract data items to represent readings (scalar values, images, etc.) from the sensing interface(s). A set of annotations will be defined for the abstract 'sensor data' items, to indicate the type of sensing interface and other parameters such as spatial coverage and temporal coverage. This extension will allow the runtime a greater flexibility in task placement and resource management. An important problem in this context is resource allocation in face of conflicting requests from application tasks. The challenge is to develop a robust and scalable mechanism and a common utility scale to arbitrate across disparately developed ATaG libraries that are combined into a larger application. The key challenge in extending the basic model to handle such scenarios is to maintain the core design objectives - especially application neutrality - while enabling the expression of increasingly sophisticated behaviors.

5 Conclusion

The complexity of programming large scale embedded networked sensor systems has stimulated interest in high level programming paradigms that ease the task of application development. Many of the concepts from traditional distributed computing such as different program flow mechanisms (control driven, data driven, and demand driven) and coordination structures (distributed shared memory, tuple spaces, etc.) are applicable at various levels of abstraction in the "programming stack" for sensor networks.

The Abstract Task Graph was discussed in this paper as a demonstration of the applicability of data driven computing for code modularity, reuse, and extensibility, and of mixed imperative-declarative programming for separation of concerns to the programming of networked sensor systems. Programming languages such as ATaG will ultimately act as the intermediate representations that are generated from high level service-oriented specifications and synthesized into deployable software for a target system.

References

1. Iyengar, S.S., Brooks, R.R., eds.: Distributed Sensor Networks. Chapman & Hall/CRC (2004)
2. Bal, H.E., Steiner, J.G., Tanenbaum, A.S.: Programming languages for distributed computing systems. ACM Computing Surveys **21** (1989) 261–322
3. Bakshi, A., Prasanna, V.K., Reich, J., Larner, D.: The abstract task graph: A methodology for architecture-independent programming of networked sensor systems. In: Workshop on End-to-end Sense-and-respond Systems (EESR). (2005)
4. Bakshi, A., Pathak, A., Prasanna, V.K.: System-level support for macroprogramming of networked sensing applications. In: Intl. Conf. on Pervasive Systems and Computing (PSC). (2005)
5. Heinzelman, W., Murphy, A., Carvalho, H., Perillo, M.: Middleware to support sensor network applications. IEEE Network (2004)
6. Cohen, N.H., Purakayastha, A., Turek, J., Wong, L., Yeh, D.: Challenges in flexible aggregation of pervasive data. In: IBM Research Report RC 21942. (2001)
7. Whitehouse, K., Zhao, F., Liu, J.: Semantic streams: a framework for declarative queries and automatic data interpretation. Technical Report MSR-TR-2005-45, Microsoft Research (2005)
8. Newton, R., Welsh, M.: Region streams: Functional macroprogramming for sensor networks. In: 1st Intl. Workshop on Data Management for Sensor Networks (DMSN). (2004)
9. Gummadi, R., Gnawali, O., Govindan, R.: Macro-programming wireless sensor networks using kairos. In: Intl. Conf. Distributed Computing in Sensor Systems (DCOSS). (2005)
10. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesC language: A holistic approach to networked embedded systems. In: Proceedings of Programming Language Design and Implementation (PLDI). (2003)
11. Cheong, E., Liu, J.: galsC: A language for event-driven embedded systems. In: Proceedings of Design, Automation and Test in Europe (DATE). (2005)
12. Whitehouse, K., Sharp, C., Brewer, E., Culler, D.: Hood: a neighborhood abstraction for sensor networks. In: 2nd Intl. Conf. on Mobile systems, applications, and services. (2004)
13. Welsh, M., Mainland, G.: Programming sensor networks using abstract regions. In: First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI). (2004)
14. Liu, T., Martonosi, M.: Impala: A middleware system for managing autonomic, parallel sensor systems. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. (2003)
15. Yu, Y., Krishnamachari, B., Prasanna, V.K.: Issues in designing middleware for wireless sensor networks. IEEE Network **18** (2004)
16. Liu, J., Chu, M., Liu, J., Reich, J., Zhao, F.: State-centric programming for sensor-actuator network systems. In: IEEE Pervasive Computing. (2003)
17. Bakshi, A.: Architecture-independent programming and software synthesis for networked sensor systems. PhD thesis, University of Southern California (2005)
18. Generic Modeling Environment, http://www.isis.vanderbilt.edu/projects/gme