Dynamic Macroprogramming of Wireless Sensor Networks with Mobile Agents

Reza Razavi
University of Luxembourg
FSTC, LUXEMBOURG
razavi@acm.org

Kirill Mechitov, Gul Agha
University of Illinois at Urbana-Champaign, IL, USA.
{mechitov, agha}@cs.uiuc.edu

Jean-François Perrot
Université Pierre et Marie Curie
LIP6, Paris, FRANCE
jean-francois.perrot@lip6.fr

Abstract

Wireless Sensor Networks (WSNs) are a key enabling technology for Ambient Intelligence. Macroprogramming has been proposed as a technique for facilitating programming WSNs, but current solutions do not provide the combination of dynamicity and query specification that would be useful to domain experts. We have implemented the first query engine which provides both these features. Our system leverages AI methods such as multiagent systems and sophisticated meta-level knowledge representation techniques to keep track of the domain knowledge and to enable adaptation in query processing. Such adaptations include dynamic representations, transformations, optimizations and deployment strategies translating queries into a system of automatically generated mobile actors in a WSN.

Keywords: Sensor Networks, Macroprogramming, Mobile Agents, Actor Systems.

1 Motivation and Problem Statement

The infrastructure for *Ambient Intelligence* (AmI) [IST, 2003] will include a massive deployment in everyday environments of *Wireless Sensor Networks* (WSNs) consisting of autonomous, spatially distributed, tiny, low-powered computers, endowed with communication, sensing and actuating capabilities [Whitehouse *et al*., 2006; Luo *et al*., 2006]. To realize this vision, WSNs must provide an omnipresent interactive environment. Our goal is to support the exploitation of an AmI infrastructure by ordinary end-users, not embedded systems programmers.

Macroprogramming has been proposed as a technique for facilitating programming WSNs. Macroprogramming enables the specification of a given distributed computation as a single global specification that abstracts away low-level distribution. The programming environment first automatically compiles this high-level specification into the relatively complex low-level operations that are implemented by each sensor node, and then deploys and executes these operations [Mainland *et al*., 2004].

As explained in [Razavi *et al.*, 2006a], we are interested in situations where both the users' requirements and the WSN environment may be dynamic. This goal matches the diversity of functionalities that end-users need from the ambient infrastructure, further amplified by the unpredictability of the phenomena being monitored and the potential changes in the ambient computing infrastructure. We develop a high-level language which supports WSN macroprograms called *uQueries.* A uQuery is represented and executed using *uQuery Engines.* Thus the key challenge is to support the cost-effective and stepwise development of *uQuery Engines*. In particular, this requires enabling uQuery specifications by multiple concurrent and uncoordinated end-users. It also requires deploying and executing such specifications in a parallel and resource-efficient manner which ensures interoperability with other uQuery engines.

In this paper, we focus primarily on representation (Section 3) and execution of uQueries (Section 4). uQueries result in a system of dynamically created meta-actors which generate actors that are concurrently deployed on a WSN. The meta-actors coordinate to control the execution of these actors on a mobile agent system. Due to limited space, we do not address our implementation of the end-user Web interface for uQueries.

As a motivating example, consider a scenario proposed by Microsoft Research [Woo *et al*., 2006]: a parking garage has been wired with break beam sensors and security cameras. Two ordinary end-users, namely Liz and Pablo, working independently, desire to use the ambient system for their own goals. Liz is a site manager of the garage building and is interested in collecting vehicle arrival time data. Pablo is a security officer in the building who wants to issue tickets to speeding drivers. In the remainder of this paper, we will use this example to illustrate our system.

2 Meta-level uQuery Engine Architecture

A key challenge is to dynamically transform high-level specifications of the users' queries into low-level executable code for WSNs. We need a truly dynamic deployment of independent code segments, which may interact and migrate, running on distributed sensor nodes. This is facilitated by a meta-level architecture for uQuery Engine.

2.1 Knowledge Level Support

The *knowledge level* keeps track of the domain knowledge and controls query processing (representing, transforming, optimizing and deploying). Specifically, it tracks two types of metadata: *Static metadata* consisting of uQuery representations and the business ontology, and *Dynamic metadata* consisting of the contextual information (including computed domain objects) obtained from uQuery execution.

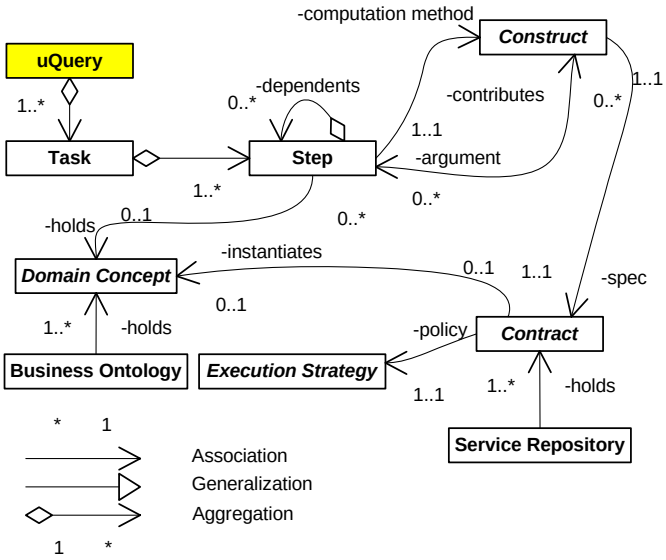

Figure 1: uQuery representation meta-model

In [Razavi *et al.*, 2006b] we describe *Dart*, the first representation meta-model that satisfies requirements of dynamicity, end-users accessibility, as well as extendibility and reusability by programmers. Figure 1 illustrates Dart using the UML notation. A *uQuery* is represented as an aggregation of *tasks*, where each task is a set of interrelated steps. When a *step* is executed, it produces a domain object. A *construct* specifies the computation method for that object. A step bridges a *construct* with the concept of that object. The most common type of construct is the *primitive construct*, which reifies a function call. Each construct refers to (or *instantiates*) a *contract* which holds meta-data about the construct. In the case of *primitive constructs*, a contract incorporates the type specification, arguments, name and a list of properties required for its execution. The *service repository* for a uQuery Engine holds these contracts. The *business ontology* holds the domain concepts, together with their relationships and constraints. In order to execute a construct, we need to make platform-specific decisions, which are delegated to the *execution strategy*.

We assume that both the business ontology and the service repository are given. We further assume a comprehensive service repository which enables all interesting uQueries to be explicitly represented as a composition of its elements.

2.2 Operational Level Support

At the operational level, we require that fine-grained mobile agent applications can be executed on resource-limited, real-time distributed systems. The mobile agent platform is responsible for:

- Deploying and executing actor code dynamically generated at runtime,
- Dynamically discovering and providing access to all sensors in the WSN, and
- Implementing the elements of the service repository.

```
1. ((lambda (migrate)   ; main function
2.  (seq
3.    (migrate       ; migrate to destination
4.      200          ; destination id
5.      111)         ; meta-actor id
6.    (par (extmsg   ; send result back to source
7.      111          ; meta-actor id
8.      (migrate
9.        100        ; source id
10.       (prim      ; execute primitive
11.         1        ; primitive index in library
12.         nil))    ; list of arguments (empty)
13.   ))))
14. (lambda (adrs val)   ; migrate function
15.   (callcc (lambda (cc) (send (list adrs cc
16.     (list quote val)))))))
```

Figure 2: Filled mobile agent code template.

To the best of our knowledge, the only existing system which satisfies our operational level requirements is *ActorNet* [Kwon *et al.*, 2006]. ActorNet agents are based on the actor model of computation [Agha, 1986]. The ActorNet runtime consists of an interpreter running on each sensor node in the WSN along with several supporting services. The runtime enables the actors to execute, communicate, migrate and access sensors and actuators.

The uQuery execution procedure fills a template for an ActorNet agent (see Figure 2). The template contains four areas to be filled by the code generator:

- Meta-actor id on lines 5 and 7,
- Application logic on lines 10-12,
- List of arguments on line 12, and
- Execution location and uQuery Engine server address on lines 4 and 9, respectively.

In Section 4.2, we explain how this template is used by the knowledge level when executing uQueries.

3 uQuery Representation

We now describe the method for representing uQueries in our system. The execution procedure will be explained in the Section 4.

Consider the parking garage example from Section 1. Pablo wants to make a uQuery to take a photo from a specific camera in a sensor network. We assume a business ontology with one concept, *Photo*, and a service repository

with one contract, *Capture Camera*, which returns a Photo object, and a library including an implementation of the above service (*e.g.* in nesC for the Mica2 sensor platform). The representation of this simple uQuery comprises a single task with a single `Capture Camera` step, which specifies a camera sensor id as a property.

More realistic uQueries, however, require a higher degree of concurrency, control (iterative and conditional), and nesting of tasks. For instance, consider Liz's query that asks for a vehicle arrival time histogram for a period of two weeks. The additional ontologies required to specify steps for this uQuery are shown in Figure 3. We denote the return value of tasks by an upward arrow, and the "contributes" relation of Figure 1 by a dashed arrow. The entire box represents a uQuery. The boxes 2, 3 and 4 represents tasks. The rest of the boxes represent steps. Due to shortage of space, we skip the details of the ontology and services required.

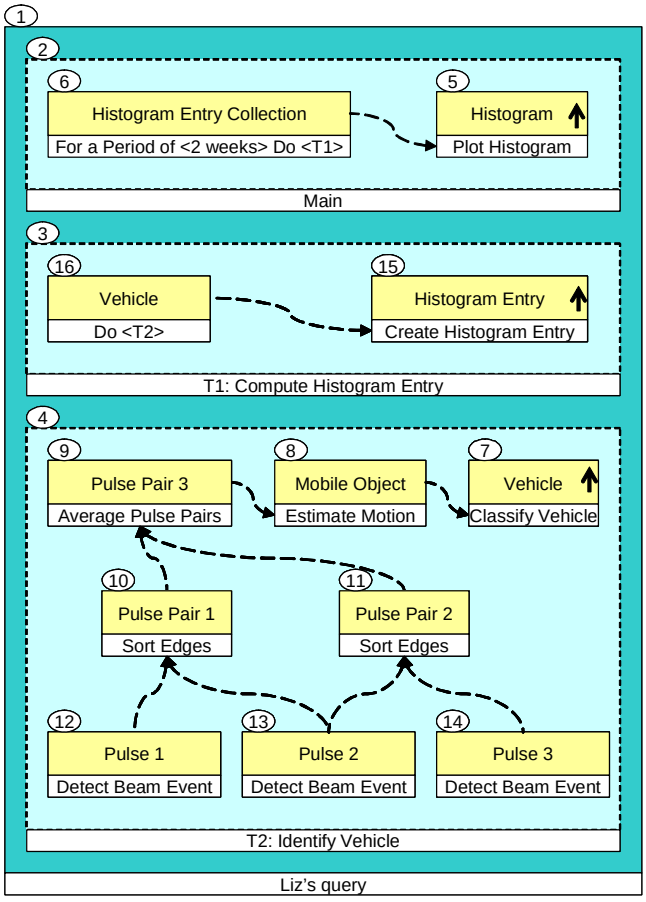

Figure 3: Representation of Liz's query by three tasks (Numbers in ovals are used here for annotation, and are not a part of the query representation).

The corresponding uQuery, called `Liz's query` (1),[1] is represented as three interrelated hierarchical tasks: a main task (2), and two *subordinate* tasks, called `T1: Compute Histogram Entry` (3) and `T2: Identify Vehicle` (4). The `main` task has two steps: `Plot Histogram` (5), which returns a Histogram object to the user, and the control construct `For a Period Do` (6) that is in charge of executing `T1` for a duration of two weeks. `T1` is then subordinate to step (6), which controls its execution. It in turn comprises two steps. The first step instantiates `Create Histogram Entry` (15), which returns a Histogram Entry object (to the main task). This requires as argument a Vehicle object, provided by the second step of `T1`, `Do <T2>` (16), which nests `T2` as computation method. `T2` comprises several steps (7-14), whose representation is analogous to that of the `Capture Camera` step described above. The task `T2` uses inputs from multiple break beam sensors to detect moving objects and classify them as vehicles.

To summarize, this uQuery representation contains occurrences of the following kinds of steps:

- *Primitive calls*: designed as steps that call a primitive as their computation method,
- *Control structures*: designed as steps that control the execution of other tasks, such as the `For a Period Do` step that controls `T1`,
- *Nested tasks*: designed as steps that use a task as their computation method, *e.g.*, the `Do` step that uses `T2`, simplify the specification of complex uQueries by hierarchically modularizing them.

The syntax of Dart is *recursive*, *i.e.*, steps may hierarchically point to tasks. It is also extensible, *i.e.*, *tasks* and *steps* may be used to extend constructs. Arbitrary control constructs can be implemented by uQuery Engine programmers using these properties, thus extending the abstract notion of a construct in Dart. We provide by default an abstraction called *Control Construct*, which has an instance variable called *closures*, which keeps track of a collection of subordinate tasks whose number and semantics depend on the desired functionality, based on the meta-data held by its contract. In the case of the `For a Period Do` control structure, which iteratively executes a given task for a duration specified as a property, the `closures` collection holds one task, here a pointer to `T1`. Its `propertyValues` instance variable, inherited from its superclass, holds the duration value, *i.e.*, 2 weeks in our example.

The representation of nested tasks is close to that of control structures. The `closure` collection contains only one task, which is executed unconditionally upon each activation of the step.

4 Concurrent Execution of uQueries

We now explain how the above representations are concurrently executed by our system. We separate the *actor* at the operational level, which is a thread and associated code actually executed as a mobile agent, from the *meta-actor* at the knowledge level, which is a thread that controls the generation of code and execution of the actor, based on the meta-data contained in the query representation. Once deployed,

[1] In this paragraph, numbers between parentheses, *e.g.*, (1), refer to the numbered rectangles in the Figure 3.

the agent system interprets the actor code and starts its execution. When an actor sends a message, such as the return value, to the uQuery Engine, the messaging interface signals the meta-actor waiting on messages from this actor and delivers the message.

4.1 uQueries and Tasks

Execution of uQueries, tasks and steps is controlled by corresponding meta-actors. Meta-actors are also implemented as actors, executing concurrently in separate threads, communicating and synchronizing through asynchronous message passing. The API of meta-actors comprises `start()`, `body()`, and `stop()` methods. `body()` specifies the core function of the meta-actor, in fours steps:

- Checking preconditions for executability,
- Initializing data structures,
- Executing the behavior,
- Sending an asynchronous message to the parent to notify the execution result.

The behavior execution step is specific to each kind of meta-actor. *uQuery meta-actors* start with launching a meta-actor for their main task. The *main task meta-actor* then continues the execution by launching *step meta-actors*. A step meta-actor can only execute in two cases: (1) it is associated to a *root step*, *i.e.*, a step whose construct requires no arguments, or (2) all its effective arguments are available. The execution strategy associated with each step is in charge of guiding this process.

uQuery and task meta-actors are designed as *Composite Meta-actors*, which provides functionality for launching a set of child meta-actors, here respectively tasks and steps, and controlling their execution. If the construct of a step points hierarchically to tasks and subordinate tasks, then, the corresponding step meta-actor creates also hierarchically *subordinate task meta-actors*. The above execution algorithm is applied recursively to the latter.

For example, a uQuery meta-actor is created and launched for executing Liz's query. This meta-actor launches a task meta-actor for the main task, which in turn launches a step meta-actor for its unique root step, *i.e.*, `For a Period Do`. The other step of the main task is not executable since it needs an argument. The `For a Period Do` meta-actor starts a timer to control the number of iterations. Each iteration consists of launching a subordinate task meta-actor for executing the same task, `T1`. The execution result of these meta-actors is collected and constitutes the result of the execution of the `For a Period Do` step. At this point, the main task meta-actor passes this collection to the `Plot Histogram` meta-actor and resumes it.

In the case of `T1`, there is also a unique root step, which is `Do <T2>`. The execution of the associated `T1` meta-actor consists of creating a subordinate `T2` task meta-actor. The `T1` meta-actor sleeps then on a semaphore, waiting for `T2` meta-actor to return a Vehicle or signals an error.

4.2 Low-level Primitives

By low-level primitives we mean services implemented by the library available at the operational level (ActorNet in our case). Such services provide access to platform-specific sensing and data processing functionality. Executing steps that instantiate primitives requires dynamically generating and deploying actor code to the WSN.

The actor template (see Figure 2) is filled by the meta-actors as the steps of the graph data structure for each task are traversed. Each step meta-actor which has its preconditions satisfied and has been signaled, directly or indirectly, by a returning step of the main task, can generate the code for its associated actor, in parallel with other meta-actors.

The resulting actors, in text form, are deployed to the WSN by the meta-actors, through the *Actor Deployment Interface* (ADI). The ADI is a multithreaded server providing socket connections over the Internet for concurrently and remotely deploying and executing actors. The id of its generating meta-actor is included in the actor code, enabling the actor to communicate with its associated meta-actor through the *Actor Messaging Interface*. The meta-actor can sleep until it receives the actor's output object through the messaging interface, after the actor finishes its computation. Arguments and return values are marshaled and unmarshaled using the information kept by the Execution Strategy. In case a primitive cannot successfully complete its computation, a failure is signaled to the relevant meta-actor, which may trigger garbage collection for the task and its associated actors and meta-actors

For example, the `T2` meta-actor concurrently generates and deploys on the ActorNet platform mobile actor code for the three `Detect Beam Event` steps shown is Figure 3. As Beam Event objects are received, the `Extract Edge` meta-actors may be signaled concurrently, and so on.

4.3 Complex uQueries

Now let us consider a query which requires distributed coordination between different sensors. Such a query cannot be executed as a single mobile agent. For example, Pablo wants to take a photo with a camera *only* when a nearby break beam sensor is triggered. The task that represents this query has two steps: detecting a Pulse object and then taking a Photo.

For executing such queries in a coordinated manner, we use the *argument* relation in our meta-model. Whenever an argument is encountered in the query specification, the knowledge level automatically creates a dependency relation between the meta-actors for these two steps. Meta-actors with dependencies check whether they have all the arguments necessary to generate and execute their associated actors. In our example, the meta-actor for capturing a Photo will be made dependent on the `Detect Beam Event` meta-actor. It will be deployed and executed only when a Pulse object is returned by the latter.

We can now consider a scenario where uncoordinated concurrent queries are entered into the system by two different users, Liz and Pablo. Pablo is interested in taking photos of Vehicles when they speed through break beam sen-

sors. We assume that Liz's query remains the same as in the previous section. Each query is represented independently, *i.e.*, there is no structural relationship between query representations. In this case, the execution for each query is triggered concurrently at the knowledge level, according to the algorithm described above. This execution procedure can be subject to optimization, as discussed below.

5 Optimization

The uQuery Engine design presents several opportunities for optimization. First, in the case of concurrent queries, data computed in one query can be reused to satisfy requirements of another. This mechanism is based on exploiting static and dynamic metadata (see Section 2.1).

For example, a Vehicle object produced by one of the queries described above contains dynamic metadata such as a timestamp and the detecting beam sensor's id. When processing a query with a step requiring a Vehicle object as argument, *e.g.*, `Compute Histogram Entry` in task `T1` above, the already-computed Vehicle may be substituted instead of executing the `Identify Vehicle` subtask, assuming the dynamic metadata of the object matches the constraints within the static metadata of the query. Such matching algorithms can be implemented using an inference engine, such as *NéOpus* [Pachet and Perrot, 1994], which provides rule-based reasoning in object-oriented applications. The benefit of this process is to save significant communication, sensing and computation resources in the WSN, where these resources are scarce, since redundant actors are not generated, deployed and executed at the operational level.

Secondly, our query representation allows computing a *computation history* for every business object by looking recursively at the constructs and objects used in its computation. The computation history allows access to the metadata associated with each element of the history. As a result, more complex matching algorithms can be implemented. For example, a query can compare the conditions under which the available business object has been computed, and whether they match the requirements of the current query. Incidentally, computation histories are also useful for *auditability*, that is identifying precisely all computation steps and their contexts for a given event in the system. The same feature allows satisfying *non-repudiability*, *i.e.*, by computing who has initiated which action.

As another optimization opportunity, we consider the *functional specification* of queries, where the user only specifies the desired functionality without explicitly listing all implementation details, such as the identifiers or location coordinates of sensors. The query processing engine can infer the requirements for these resources from the static metadata associated with the query and the contracts of the primitives it invokes. We can then fill in specific sensors and other parameters matching these criteria. In order to find suitable sensing resources meeting these constraints, we extend ActorNet with the *Network Directory Service* (NDS), which performs network monitoring and provides an up-to-date listing of sensor and computing resources available in the network. The NDS is searchable by sensor type, location and other attributes, listed as name-value pairs. Thus the query engine can find appropriate sensors for a query by looking at attributes such as location and orientation. This simplifies the query creation process for the user and provides opportunities for optimizing query execution, by enabling the uQuery Engine to pick the best resources (*e.g.*, closest, least congested, or having the most remaining battery power) satisfying the query specification.

Finally, at the operational level, the mobility of actors allows for load balancing and resource-aware task allocation, even as new actors are added. In the parking garage example, we can choose, based on the current communication traffic and available computing power, whether to move the raw break beam sensor data to a PC for processing, or to move the vehicle detection code, as a system of mobile agents, to the sensor where the data is generated. Also, by exposing available resources (sensors, actuators, data stores, *etc.*) to the knowledge level, compatible resources may be substituted for one another at runtime to simplify scheduling or reduce congestion.

6 Related Work

A survey of solutions currently proposed in the literature reveals a variety of approaches to macroprogramming WSNs: a spreadsheet approach [Woo *et al*., 2006], *EnviroSuite* [Luo *et al*., 2006], a market-based approach [Mainland *et al*., 2004], and *Semantic Streams* [Whitehouse *et al.*, 2005]. Although many of these approaches are quite powerful, none of them provide the language abstractions required for dynamic macroprogramming by end-users as outlined above.

For example, the spreadsheet approach uses an Excel spreadsheet to represent the layout of nodes and insert their functionality in the spreadsheet; queries are resolved by a logic program that generates a composition of services, where a service is a .Net component. The approach satisfies the requirement of usability by non-programmers. However, it is not sufficiently general: it enforces a particular naming grid-based scheme and does not allow for the definition of arbitrary groups of nodes and operations over such groups.

EnviroSuite proposes environmentally immersive programming, an object-based programming model in which individual objects represent physical elements in the external environment. In both EnviroSuite and ActorNet, actors or objects must be created explicitly by programmers to provide a service. Behavioral specifications are not in terms of groups of actors. Protocols to support operations over groups of objects and protocols to implement such specifications may not be re-used.

Traditionally, WSN application development involved fairly static programming languages, operating systems and reprogramming services, for efficiency reasons. For example in TinyOS, the sensor network application components are written in nesC and compiled together with the operating system code and middleware services into a single application image, which can be uploaded to the sensor nodes using the Deluge protocol [Hui and Culler, 2004] prior to program

execution. This approach proves successful in achieving its stated goal of highly efficient utilization of sparse computing resources. Unfortunately, it is ill-suited for an open system comprising a dynamic set of diverse, transient tasks that is the expected workload in ambient systems. If we take Deluge as the deployment method for our target systems (where queries from multiple users, all specified at runtime, need to be transformed into executable code, uploaded on the deployed sensor network and executed), this results in unnecessarily transmitting a large volume of code which has not changed (OS components, routing protocol, *etc.*) along with the newly-generated application code.

7 Conclusion

Ambient Intelligence technologies will enable novel applications and new work practices in many fields. AmI will provide for the integration of real-time data into business processes, enabling real-time decision making and business process definition and modification. Such dynamicity will facilitate responding to situations more efficiently, with a higher degree of quality and end-user satisfaction.

In this paper, we explained how dynamic macroprogramming by end-users can be achieved for ambient systems such as WSNs. The two-level approach to architecting the query engine allows separating query representation and reasoning concerns from those of their effective execution on diverse runtime platforms through model-to-code transformation. The knowledge level comprises the representation of a domain ontology and a service repository, together with the uQuery composition machinery, and implements the rules that govern uQuery transformation, coordination and optimization. Our system allows for effective end-user query specification and automated execution. The query representation meta-model, designed with extendibility and reusability as primary considerations, allows uQuery engine programmers to add specific constructs via classical object-oriented techniques. Business logic primitives are separated from the core of the mobile agent system, facilitating addition of new domain-specific primitives to the system.

Representations of queries are transformed to platform-specific code for ActorNet, dynamically deployed and executed. Using ActorNet as the query execution environment provides dynamicity of macroprogramming, while enabling load balancing and other optimizations to take place.

The presented meta-level architecture for implementing uQuery Engines is prototyped by reusing and extending our previous work on ActorNet and its implementation on Mica2 motes [Kwon et al., 2006], and Dart implemented as an object-oriented framework in different dialects of Smalltalk [Razavi et al., 2006]. This prototype is called *Ambiance Platform* and uses a new implementation of Dart in *Squeak* (http://www.squeak.org/). We further use the *Seaside* framework (www.seaside.st) for the dynamic Web-enabled uQueries programming interface of Ambiance.

A number of important issues remain to be explored. These include dynamic and seamless integration of sensor and actuator nodes, sensitivity to privacy concerns and trustworthiness, and analyzing coherence and integrity of uQueries. For example, the computation history discussed in Section 5 allows security enforcement through dynamic decision making about whether to execute the current step or not. Furthermore, significant optimization may be possible, for instance by integrating learning mechanisms into query specification, transformation and execution.

Acknowledgments

This work is partially funded by the University of Luxembourg, in the framework of the *Åmbiance* project (R1F105K04) and NSF under grant CNS 05-09321, by ONR under DoD MURI award N0014-02-1-0715 to UIUC. The authors would also like to acknowledge the valuable collaboration of F. Arbab, A. Cardon, N. Bouraqadi, P. Bouvry, Ch. Dony, V. Ginot, R. Kumar, R. Johnson, M. Malvetti, S. Nadjm-Tehrani, D. Riehle, S. Sundresh, and J. Yoder.

References

[Agha, 1986] Agha, G.: "Actors: a Model of Concurrent Computation in Distributed Systems". MIT Press, 1986.

[Hui and Culler, 2004] Hui, J.W. and Culler, D.: "The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale." In SenSys'04, pp. 81-94, 2004.

[IST, 2003] IST Advisory Group. Ambient intelligence: from vision to reality, September 2003.

[Kwon *et al*., 2006] Kwon, Y., Sundresh, S., Mechitov, K., Agha, G.: "ActorNet: An Actor Platform for Wireless Sensor Networks." In AAMAS, 2006.

[Luo *et al*., 2006] Luo, L., Abdelzaher, F., He, T., Stankovic, J.A.: "EnviroSuite: An Environmentally Immersive Programming Framework for Sensor Networks." ACM Transaction on Embedded Computing Systems, 2006.

[Mainland *et al*., 2004] Mainland, G., Kang, L., Lahaie, S., Parkes, D.C., and Welsh, M.: "Using Virtual Markets to Program Global Behavior in Sensor Networks." 11th ACM SIGOPS European Workshop, Belgium, 2004.

[Pachet and Perrot, 1994] Pachet F, Perrot JF.: "Rule firing with meta-rules." In SEKE'94. pp. 322–29, 1994.

[Razavi et al., 2006a] Razavi, R., Mechitov, K., Sundresh, S., Agha, G., Perrot, J.-F.: Ambiance: Adaptive Object Model-based Platform for Macroprogramming Sensor Networks. Poster session extended abstract. OOPSLA 2006 Companion. Portland, Oregon, USA (2006).

[Razavi *et al*., 2006b] Razavi, R., Perrot, J.-F., Johnson, R.: Dart: A Meta-Level Object-Oriented Framework for Task-Specific, Artifact-Driven Behavior Modeling. In Proceedings of DSM'06, Portland, Oregon, USA. ISBN 951-39-2631-1, pp 43-55, 2006.

[Whitehouse *et al*., 2006] Whitehouse, K., Zhao, F. and Liu, J.: "Semantic Streams: A Framework for Composable Semantic Interpretation of Sensor Data." In EWSN'06, Switzerland, pp. 5-20, 2006.

[Woo *et al*., 2006] Woo, A., Seth, S., Olson, T., Liu J., and Zhao, F.: "A Spreadsheet Approach to Programming and Managing Sensor Networks." In IPSN SPOTS, 2006.