Ambiance: A Mobile Agent Platform for End-User Programmable Ambient Systems

Reza RAZAVI [a,1], Kirill MECHITOV [b], Gul AGHA [b], Jean-François PERROT [c]
[a] *University of Luxembourg, FSTC, LUXEMBOURG*
[b] *University of Illinois at Urbana-Champaign, IL, USA*
[c] *Université Pierre et Marie Curie, LIP6, Paris, FRANCE*

Abstract. We are interested in situations where multiple uncoordinated non-professional programmer end-users want to exploit the Ambient Intelligence (AmI) infrastructure on their own, without calling in embedded systems programmers, in order to support their daily activities. Our goal is allowing them to achieve this objective through on-the-fly creation and execution of high-level programs that we call *uQueries* (for user-defined or ubiquitous queries). The key challenge then is to support the cost-effective and stepwise development of *uQuery engines*---systems for end-user programming and execution of uQueries. We present a meta-level architecture that addresses this issue by leveraging Artificial Intelligence methods that make possible the separation of uQuery representation and reasoning concerns from those of their effective execution through model-to-code transformation. We show that (1) interconnections between ambient devices may be dynamically specified as control flows between high-level descriptions of their primitive functionality, (2) specifications may be elaborated by concurrent, uncoordinated end-users through a Web interface, and (3) they may be automatically distributed and concurrently executed on ambient devices as a system of mobile agents. We have created a prototype of this architecture, the *Ambiance Platform*, which has allowed experimental validation of the approach using an application scenario proposed in the state-of-the-art of relevant research areas. This experience led us to identify important issues to be explored, including dynamic and seamless integration of sensor and actuator nodes into the system. Furthermore, opportunities exist for significant performance and resource use optimization, for instance by integrating learning mechanisms into uQuery specification, transformation and execution.

Keywords. Ambient Intelligence, Artificial Intelligence, Sensor Networks, Macroprogramming, Adaptive Object-Models, Mobile Agents, Actor Systems

Introduction

Ambient Intelligence (AmI) envisions the '*invisible*' incorporation into our surrounding environment and everyday objects of billions of loosely-coupled sensing, actuating,

[1] Corresponding Author: Reza Razavi, FSTC, University of Luxembourg, 6, rue Richard Coudenhove-Kalergi, L-1359 Luxembourg, LUXEMBOURG; E-mail: razavi@acm.org. The work communicated in this chapter has been mostly conducted while the corresponding author was acting as principal investigator on the Åmbiance project funded by the University of Luxembourg (2005-2006).

computing and communicating components as part of an *AmI infrastructure* [1, 2]. The aim is to discover new ways of supporting and improving people's lives and enabling new work practices. See Berger *et al.* [3] for a number of examples.

AmI infrastructure components such as Micro-Electro-Mechanical Systems (MEMS), smart materials, ad hoc smart dust networks, and bio-inspired software are being intensively researched and developed. The *Cyber Assist* project by Nakashima [4] describes a fully developed example. Another key enabling technology is networks of potentially very large numbers of wirelessly connected small, autonomous, distributed, and low-powered computers endowed with limited processing, storage, sensing, actuation, and communication capabilities. Such a system is called a *Wireless Sensor Network* (WSN), and each sensor node a *mote* [5, 6].

The problem is now to write programs for the AmI infrastructure. This is no easy task, given the complexity and dynamicity of the structures and the diversity of potential users and their needs. Our work is concerned with situations where multiple uncoordinated end-users need to exploit the AmI infrastructure on their own, in order to solve problems in everyday life and to support their daily activities in different domains. They should be able to achieve this goal by on-the-fly writing and executing high-level programs that we call *uQueries* (for user-defined or ubiquitous queries/macroprograms), without calling in specialists of embedded systems programming. This is motivated by the diversity of functionalities that end-users expect from the AmI infrastructure, (see Böhlen [7] for a general argument and Richard and Yamada [8] for a typical example), further amplified by the unpredictability of the phenomena being monitored and the potential changes in the ambient computing infrastructure. From the critical standpoint taken by Huuskonen [9] we clearly adopt the "Person-centric" approach. In our view, this environment calls for a number of Artificial Intelligence techniques to be applied, notably knowledge representation and machine learning. We shall also see that multi-agent systems likewise play an important role as observed by Nakashima [4].

We consider an AmI infrastructure that comprises both WSNs and more traditional computing artifacts such as PCs, gateway nodes, and handheld mobile devices (although the issues specific to mobile devices are not dealt with in this chapter). Let us call each hardware component an *ambient node*. This infrastructure is *open* in that both ambient nodes and uQueries may enter and leave the computing environment dynamically. Based on resource availability and optimization criteria, available ambient nodes coordinate and determine their mutual application execution responsibilities at runtime.

As a motivating example, consider a scenario from Microsoft Research [10], where the ambient infrastructure, installed in a parking garage, comprises break beam sensors and security camera nodes. Two ordinary end-users, namely Liz and Pablo, who work independently, desire to use the ambient system for their own purposes. 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. We assume that the deployed system does not include ready-to-use specific functionalities required by typical end-users such as Liz and Pablo. It should therefore be programmed, deployed and executed by the users themselves.

In the following sections we describe an architecture that satisfies the above requirements and provides a ubiquitous and omnipresent interactive Web-enabled environment for programming and executing uQueries, illustrated through application to the above example scenario.

1. Problem Statement

1.1. Inappropriateness and Complexity of Current Programming Techniques

Current system development and deployment techniques do not transfer well to programming ambient systems. New computation models and software development methodologies are required. Satoh observes, for instance, that "Ambient intelligence technologies are expected to combine concepts of intelligent systems, perceptual technologies, and ubiquitous computing*."* [11].

In particular, effectively programming WSNs is difficult due to their typical resource limitations. Moreover, sensor nodes are prone to failure (for example, if they run out of energy), and communication between them is unreliable. Programming such networks requires addressing those limitations. Unfortunately, current methods for WSN programming lead developers to mix high-level concerns such as quality of service requirements, for instance timeliness, reliability, application logic, adaptivity, with low-level concerns of resource management, synchronization, communication, routing, data filtering and aggregation. This makes developing software for WSNs a costly and error-prone endeavor, even for expert programmers (see a simple illustrative case in the next subsection).

Macroprogramming has been proposed as a technique for facilitating programming WSNs. Macroprogramming enables the definition of a given distributed computation as a single global specification that abstracts away low-level details of the distributed implementation. The programming environment first automatically compiles this high-level specification down to the relatively complex low-level operations that are implemented by each sensor node, and then deploys and executes these operations [12]. However, macroprogramming is of interest for specialized embedded systems programmers, not for end-users. On the contrary, as explained above, we are interested in situations where both the users' requirements and the WSN environment may be dynamic.

Thus, the key challenge is to develop *uQuery engines*---systems that support end-user programming and execution of uQueries. In particular, this requires enabling specifications by multiple concurrent and uncoordinated end-users of queries, which may convey a complex logic (comprising control constructs and hierarchical structures). It also requires deploying and executing such specifications in a concurrent and resource-aware manner. This chapter presents a technique to support the cost-effective and stepwise development of uQuery engines.

As we shall see, our approach makes a central use of a two-level multi-agent system, together with a knowledge base about the target application domain. Adaptive learning behavior for the agents remains to be implemented in our system.

1.2. Illustration

Consider a simple WSN application scenario where we want to detect, on demand, an object passing through a break beam sensor. The algorithm is as follows:

1. Wait for a request from the user.
2. Perform the requested action:
    2.1 Execute `detectBeamEvent`() primitive.
    2.2 Keep checking the sensor until a change in status is detected.
3. Send the result of the detection event back to the user.

The user (programmer) is responsible for choosing the right OS and network components, and assembling them along with the specific application logic into an executable program. The code excerpt below presents an implementation of a simple WSN program for the above algorithm that such a programmer could write in the nesC language for the TinyOS sensor platform. Comment blocks denoted by `/* … */` indicate additional code segments not related to application functionality.

```
// Detect break beam event application (code excerpt)
configuration Example {}
implementation {
    // list of application components
    components Main, ExampleM, LedsC, GenericComm, TimerC,
            Photo, CC1000ControlM;
    // statically link all components
    Main.StdControl -> GenericComm;
    Main.StdControl -> TimerC;
    Main.StdControl -> Photo;
    Main.StdControl -> ExampleM;
    ExampleM.SendMsg -> GenericComm.SendMsg[10];
    ExampleM.ReceiveMsg -> GenericComm.ReceiveMsg[10];
    ExampleM.CC1000Control -> CC1000ControlM;
    ExampleM.Timer -> TimerC.Timer[unique("Timer")];
    ExampleM.Leds -> LedsC;
    ExampleM.PADC-> Photo;
}
module ExampleM {
    /* … */
}
implementation {
    TOS_Msg msgbuf;
    uint8_t msglen, sendPending;
    volatile uint8_t ioPending;
    uint16_t ioData;

    /* … */
    // primitive function #20: detect beam event (using photo sensor)
    uint16_t detectBeamEvent();
    // I/O: convert split phase non-blocking I/O to blocking I/O
    uint16_t IO(uint16_t a, uint16_t b) __attribute__((C,spontaneous)) {
            while (ioPending) yield();
            if (a == 20) { call PADC.getData(); ioPending=1; }
            while (ioPending) yield();
            return ioData;
    }
    async event result_t PADC.dataReady(uint16_t data) {
            ioPending=0; ioData=data;
            return SUCCESS;
    }
```

```
    // Communication: receive requests for execution and send results
    void sendPacket(uint8_t *buf, uint8_t n)
                                        __attribute__((C,spontaneous)) {
            memcpy(msgbuf.data, buf, n);
            msglen = n;
            if (call SendMsg.send(TOS_BCAST_ADDR, msglen, &msgbuf)
             == SUCCESS)
                    sendPending = 1;
    }
    uint8_t isSendPending() __attribute__((C,spontaneous)) {
            return sendPending;
    }
    event result_t SendMsg.sendDone(TOS_MsgPtr mp, result_t success) {
            if (!success) call Timer.start(TIMER_ONE_SHOT, 200);
            else {
                    call Leds.redToggle(); sendPending = 0;
            }
            return SUCCESS;
    }

    event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr mp) {
            TOS_Msg m;
            call Leds.greenToggle();
            if ((uint8_t)mp->data[0] == 20) {
                    m.data = deref(detectShadow());
                    sendPacket((uint8_t *)m.data, strlen(m.data));
            }
            return mp;
    }
    event result_t Timer.fired() {
            return call SendMsg.send(TOS_BCAST_ADDR, msglen, &msgbuf);
    }

    /* … */
    // Implementation of detectBeamEvent primitive
    uint16_t detectBeamEvent() {
            int i;
            uint16_t data, avg = 0;
            ledSet(0);
            for (i = 0; i < 10; i++)
                    avg += IO(2, 0);
            avg /= 10;
            while ((data = IO(2, 0)) > avg - 15) yield();
            ledSet(7);
            return list(2, newWord(20), newWord(data));
    }
}
```

The majority of the code is not related to the specific application domain (detecting a break beam event) but to managing resources, communication and low-level control flow in the WSN node. As is explained in the following sections, our approach allows an end-user to write the same program by means of simply creating a uQuery, which contains one *step*: `Detect Beam Event`. The executable code, corresponding to the one illustrated above is generated by model transformation techniques (see Section 4.3, specifically the last paragraph, and Figure 9).

The remainder of the chapter is organized as follows. Section 2 provides an overview of our solution. Sections 3 and 4 describe and illustrate respectively the uQuery representation and execution mechanisms. Section 5 is devoted to our end-user programming interface. Section 6 enumerates some optimization opportunities. Section 7 explores the related work, before concluding in Sections 8 and 9.

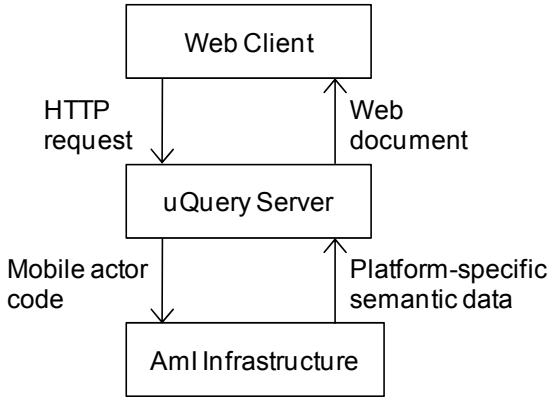

Figure 1. High-level view of the uQuery Engine system architecture

2. Architectural Style of uQuery Engines

The key technical requirements of uQuery engines are dynamic end-user programmability by multiple uncoordinated end-users, and automated deployment and execution. To meet these requirements, we propose the *architectural style* [13] which is illustrated by Figure 1. There are three subsystems as follows. On the top, the *Web Client* subsystem is responsible for providing uncoordinated concurrent end-users with a domain-specific Web programming interface (for Vehicle Tracking, in our example case). At the bottom, the *AmI Infrastructure* subsystem provides an interface to hardware and software platforms of the ambient nodes, in our case, mostly WSN nodes. In the middle, the *uQuery Server* subsystem is responsible for representing uQueries and processing them for execution on the AmI infrastructure.

The Web Client subsystem communicates with the uQuery Server subsystem by standard *HTTP Requests*. This communication serves two main purposes: elaborating the uQuery on the one hand, and executing it on the other hand. Accordingly, the uQuery Server subsystem responds with dynamically generated Web documents, which embody either the current state of the program being built, or its execution results. The uQuery Server subsystem communicates in turn with the AmI Infrastructure subsystem by non-standard mobile actor code bundles in text format (Scheme-like code). It receives semantically meaningful execution results, which belong to the application domain ontology, but in (WSN) platform-specific format. The receiver, uQuery Server subsystem, is responsible for encoding the results into an application-specific format, before embedding them into Web documents.

Standard Web 2.0-compliant browsers are used as Web Clients. The AmI Infrastructure subsystem encompasses an open set of heterogeneous ambient nodes. We impose an important constraint on the software interface of these nodes: they must provide dynamic mobile agent code deployment and execution services. In the case of sensor nodes, such interface is provides by platforms like *ActorNet* [14] and *Agilla* [15]. As for the main component of this architecture, the uQuery Server subsystem, it is designed and implemented as a meta-level object-oriented application as follows.

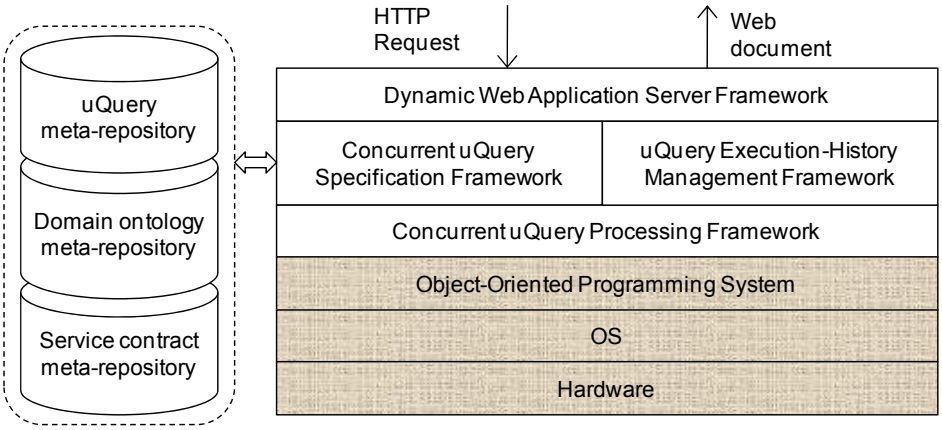

Figure 2. uQuery Server subsystem architecture

As we said, the uQuery Server has a double purpose of specification and execution. Such a situation has already been studied in Software Engineering, notably by Ralph Johnson and his colleagues under the name of adaptive object-models (AOM) [16]. Accordingly, our Server follows the architectural style of AOMs, which defines a family of architectures for dynamically end-user programmable object-oriented software systems. More particularly, this design belongs to the class of *flow-independent* AOMs [17], where the expected behavior (here uQueries) is specified at run-time as a high-level description of the control flow between reusable and sharable computational entities that embody the basic application domain algorithms (for example, for vehicle tracking). In the reminder of this chapter, we call the latter *services*, following the service-oriented computing paradigm, and suggest wrapping and representing the basic functionality provided by ambient nodes also as services.

Our design is illustrated in Figure 2, and encompasses four components and several metadata repositories as follows. On the top, the *Dynamic Web Application Server Framework* component provides the standard functions of a dynamic Web server application, by serving HTTP requests from concurrent clients and by dynamically generating Web documents. We use the Seaside framework (www.seaside.st) for this purpose, which provides a high-level dynamic Web server programming interface by reuse and extension. At the bottom, the *Concurrent uQuery Processing Framework* component controls uQuery processing (transformation, optimization, and deployment). In the middle, there are two components. The *uQuery Execution-History Management Framework* component is responsible for storing and providing an extensible interface to access the executions of uQueries, for analyzing, rendering and visualizing results, and also for optimization purposes. This component is not currently mature enough to be communicated here. The *Concurrent uQuery Specification Framework* component is responsible for uQueries representation.

From the functional perspective, this architectural style keeps track of two types of *static* and *dynamic metadata*: Static metadata corresponds to the knowledge acquired and represented at built-time concerning the business ontology, and the available services (*contracts*). They are read- and write-accessible by the Concurrent uQuery Specification Framework component (see meta-repositories in Figure 2). The dynamic

metadata corresponds to the knowledge acquired at run-time concerning the user-defined uQueries, their executions (domain objects and uQuery execution history), the available service implementations, and ambient nodes and their resources (see *Network Directory Service* in Section 6). These are read- and write-accessible by all components, but primarily by the *Concurrent uQuery Processing Framework* component (see the repositories in Figure 5). Overall, the representation and processing of this metadata is a major role of the architectural style of uQuery engines, specifically its *knowledge level*. The execution of uQueries *per se* is the responsibility of the *operational level*.

The knowledge level representation and processing role is shared between the two Concurrent uQuery Programming and Processing Framework components. The processing component provides additionally a hook to the operational level execution role, which is accomplished by AmI infrastructure node platforms (here: ActorNet). The following sections provide more details on the design of these two major components of the uQuery engine architectural style.

3. Concurrent uQuery Specification Framework

We now describe the design of the Concurrent uQuery Specification Framework component for representing uQueries in our system. The processing component will be explained in the Section 4, and the end-user programming interface in Section 5.

In [17] we describe *Dart*, a unique (to the best of our knowledge) method for dynamically specifying control flow in flow-independent AOM architectures that satisfies the requirements of support for dynamicity, end-users accessibility, as well as extendibility and reusability by programmers, in particular for plugging specific execution strategies (such as dynamic multi-agent distributed execution, as it is the case here). We therefore suggest reusing Dart for representing uQueries.

Figure 3 illustrates the core design of Dart using the UML notation. A *uQuery* is represented as a *graph*, which aggregates *tasks* where each task is a set of interrelated steps. A *step* reifies a service request by bridging a *construct* with a *domain object* that results from its execution. A construct specifies the computation method for that object. The most common type of construct is the *primitive construct*, which reifies a low-level primitive service request. By low-level primitive we mean a method or a library function, which implements low-level logic, such as mathematical algorithms, and platform-specific sensing and data processing functionality. A step referring to a primitive construct is called a *primitive step* (or *leaf* step). Each construct refers to (or *instantiates*) a *contract* which holds metadata 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. In order to execute a construct, we need to make platform-specific decisions, which are delegated by each contract to the associated *execution strategy*. The *service repository* for a uQuery engine holds these contracts. The *business ontology* holds the descriptions of *domain concepts*, together with their relationships and constraints. Steps hold instances of domain concepts, domain objects, which conform to the ontology. Result and argument specifications refer to domain concepts that define their types. Finally, to each task may be associated one or several *organizations*, which determines the visual representation of its steps. Different representations, such as list, grid and freeform, are possible since order is irrelevant when executing Dart graphs (see the next Section).

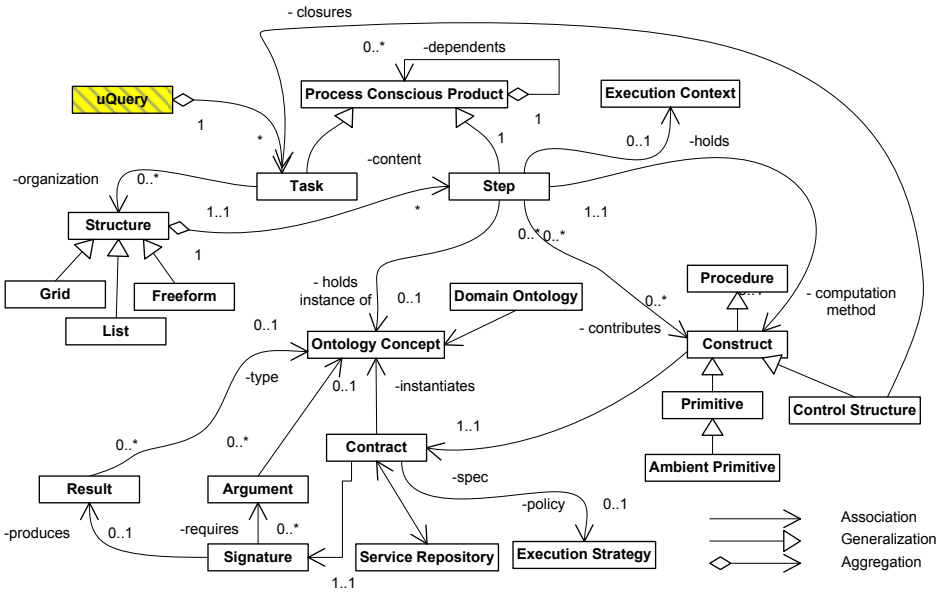

Figure 3. Detailed design of the Concurrent uQuery Specification Framework component in UML

We assume that both the business ontology and the service repository for the business domain are provided. Actually, both may be implemented as part of a single domain ontology written in OWL (with the help of the Protégé editor), and accessed (with the Pellet reasoner) by means of queries written in SPARQL. We further assume that the service repository is comprehensive enough to allow all uQueries of interest to be explicitly represented as a composition of its elements. Of course, such a repository would greatly benefit from the advanced techniques for context representation proposed by Dapoigny and Barlatier [18].

For illustration, 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.,* low-level nesC code for the Mica2 sensor platform, accessible through an ActorNet interface). 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 uQuery 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 4. We denote the return value of tasks by an upward arrow, and the "contributes" relation of Figure 3 by a dashed arrow. The entire box represents a uQuery. The boxes 2, 3 and 4 represent tasks. The rest of the boxes represent steps. We skip describing the details of the ontology and services required, which are outlined in the figure.

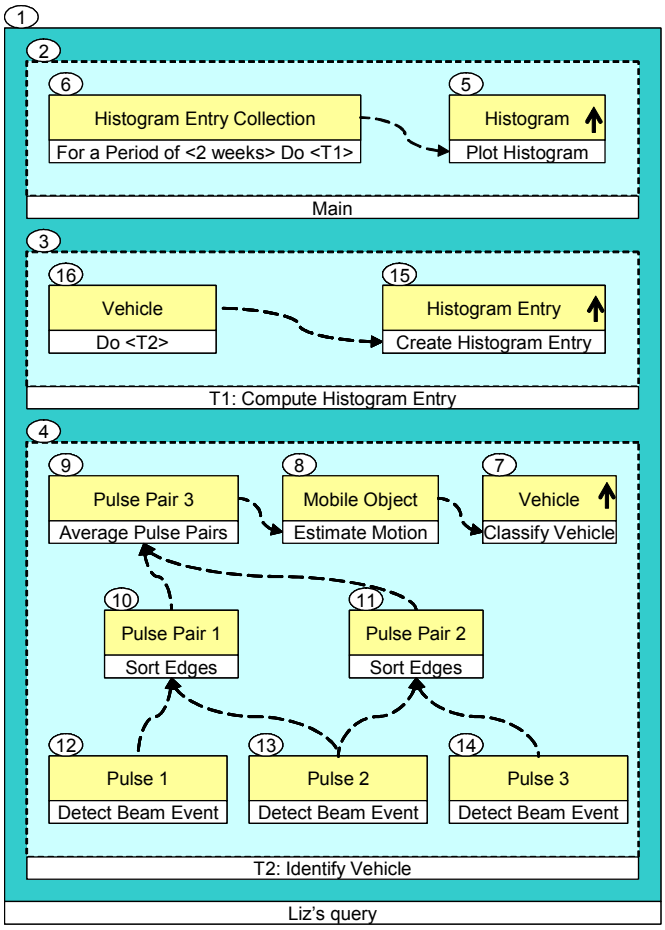

Figure 4. Representation of Liz's uQuery by three nested and subordinate tasks, and several steps[2]

The corresponding uQuery, called `Liz's uQuery` (1),[3] 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 end-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

[2] Numbers in ovals are used for annotation, and are not a part of the representation.

[3] In this paragraph and the 5th paragraph of Section 5, numbers between parentheses, *e.g.*, (1), refer to the numbered rectangles in respectively Figure 4 and Figure 12.

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. Section 5 describes and illustrates how uQueries like this may be specified using our Web interface.

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

- *Primitive calls*: designed as steps that call a primitive operation 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`, and thereby 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`* (illustrated by the directed arrow from Control Construct to Task in Figure 3), which keeps track of a collection of subordinate tasks whose number and semantics depend on the desired functionality, based on the metadata 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.

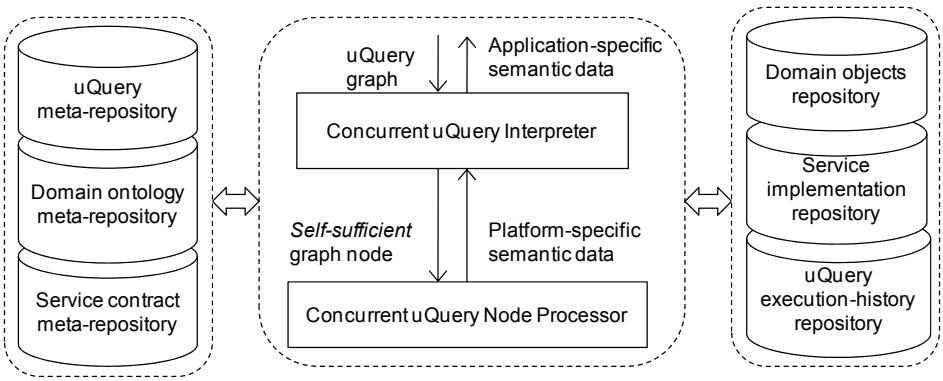

Figure 5. Concurrent uQuery Processing Framework component architecture

4. Concurrent uQueries Processing Framework

Now, we explain how uQuery graphs are concurrently executed by describing the design of the Concurrent uQuery Processing Framework component as illustrated in Figure 5.

We propose an interpretation mechanism coupled with a truly dynamic deployment of independent code segments, which may interact and migrate, running on distributed sensor nodes. Specifically, in alignment with experimental systems such as *Actalk* [19], this component is designed as a framework for concurrent execution of uQueries' object-based representations. Specifically, we use mobile actors as the execution model, which ensures feasibility and improves scalability by splitting computations on appropriate nodes and merging them according to resource availability and control flow. We separate the *actor* at the operational level, which is a thread responsible for executing primitive steps, from the *meta-actor* at the knowledge level, which is a thread that represent control flow through the computation, controls the execution on a single uQuery graph node, and carries state. Moreover, our architecture allows meta-level objects to modify their behavior in more fundamental ways through *introspection* and *intercession*, for example, if the meta-actors are endowed with learning mechanisms.

In our current example, a vehicle identification uQuery is decomposed in terms of a set of meta-actors. These meta-actors control the concurrent and distributed collection of data by actors dynamically deployed on beam and camera sensors nodes to process and analyze the data. Note that the analysis may trigger further data collection, or reuse previously collected data.

Subsections 4.1, 4.2, and 4.3 describe the design of three subcomponents that implement these decisions.

4.1. Concurrent uQuery Interpreter

The *Concurrent uQuery Interpreter* subcomponent is responsible for concurrent uQuery graph traversal, meta-actor creation, encoding execution results into application-specific formats, and storing execution results and uQuery executions. It reads uQuery graphs from the corresponding repository and updates related repositories. In particular, uQuery enactments history and domain objects repositories are respectively updated with uQuey executions and application-specific semantic data (domain objects). The execution results are received in a platform-specific format, and are encoded into application-specific formats using correspondence tables. In general, all domain objects that result from the execution of steps in a uQuery are stored. The final result of the execution of a uQuery corresponds to the result of the execution of the step in the main task that is tagged as the return value.

The graph traversal algorithm is associated with meta-actor type hierarchy illustrated in Figure 6 using the UML notation. The algorithm associates a meta-actor to each graph node. Dart node types present in uQuery graphs, *i.e.*, uQuery, Task and Step are respectively mapped to *uQuery Meta-Actor*, *Task Meta-Actor*, and *Step Meta-Actor* types. Each meta-actor is uniquely responsible for processing its corresponding node. The process is initiated by creating a uQuery meta-actor, which recursively takes charge of the creation of the other meta-actors. The underlying algorithm is described in the next subsection. Each meta-actor is concurrently linked to the whole system only by knowing the address (called "actor name" in Actor terminology) of its *parent*

meta-actor, which is the meta-actor that has created it. Each meta-actor keeps also track of all its child meta-actors. The uQuery meta-actor has access to the whole system of meta-actors, which is structured as a distributed connected graph (since uQuery graphs are connected by construction). The parent of all uQuery meta-actors is the uQuery engine, which is itself designed as a meta-actor type (see Figure 6).

The order is irrelevant in creating meta-actors. Concurrent execution of meta-actors is allowed by construction, since the execution of Dart graph nodes is only restricted by their data dependencies. In other words, those nodes that do not have direct or indirect data dependencies, or have their required data available, may execute in parallel at any time. The interpretation process for a uQuery reaches its end when all relevant meta-actors have returned from execution (possibly with an error).

The occurrence of an error in one node may result in garbage collecting the whole system of corresponding meta-actors and stopping the execution, or spawning a new meta-actor for another try, for example using different resources. Algorithms for non-intrusive, scalable distributed mobile active object garbage collection, proposed by Wei-Jen Wang and Carlos A. Varela [20], may be considered here.

The next Subsection describes the design of meta-actors and their processing algorithm *per se*.

4.2. Concurrent uQuery Node Processor

The *Concurrent uQuery Node Processor* subcomponent is responsible for processing a single graph node and returning its result, when applicable. Results are provided in a platform-specific format. Autonomous and concurrent processing of nodes is possible since Dart graph nodes are *self-sufficient*. Self-sufficiency refers to the fact that each individual service request embedded in a graph node is provided, by design, thanks to pointers to the service contract and its argument steps, with the required knowledge about the arguments, resources, context and method required for its execution. Meta-actors are designed as active objects [19], executing concurrently in separate threads, communicating and synchronizing through asynchronous message passing, specifically for fetching the required data. They arrive to the end of their lifecycle when the execution is terminated (either successfully or by an exception).

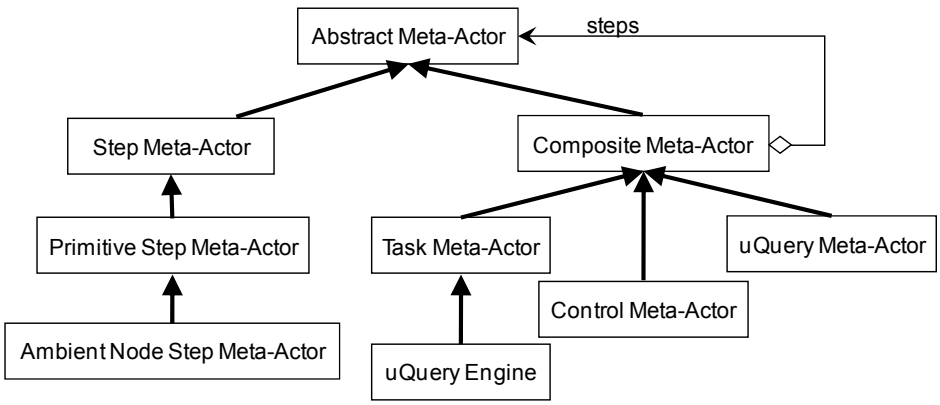

Figure 6. Detailed design of the Concurrent uQuery Processing Framework component in UML

The interface of meta-actors comprises `start()`, `body()`, and `stop()` methods. The `body()` method specifies the core function of meta-actors, which contains four steps: (1) checking preconditions for executability, (2) initializing data structures, (3) executing the behavior, and (4) sending an asynchronous message to the parent to notify it of the execution result. The behavior execution step (No. 3) is specific to each kind of meta-actor. As we have explained in [21], it may comprise a set of sophisticated actions, including implementation matching, resource matching, decision making and optimization, online scheduling and task allocation, marshaling, and deployment. Specifically, meta-actors screen the runtime environment, allocate resources, spawn other meta-actors and actors, and control the lifecycle of those actors. Such a lifecycle, described in more detail in the next subsection, involves activation and registration, request management, application logic, and result dissemination.

Dart graph nodes are also *minimally-constrained*. Minimally-constrained refers to delaying as long as possible placing constraints necessary to execute a specific instance of the service, in other words, the service instance does not refer to information that can be computed or supplied to it at run-time. This characteristic, together with the openness, resource-limitedness and error-proneness attributes of the AmI infrastructure have motivated that sophisticated processing architecture.

At the beginning of their execution, *uQuery* (root) *meta-actors* launch a meta-actor for their main task. The *main task meta-actor* 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 also creates 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 uQuery. 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 the `T2` meta-actor to return a Vehicle or signal an error.

The domain objects that are expected as uQuery enactment result are basically obtained from the execution of meta-actors in charge of primitive step nodes. The role of the other meta-actors is essentially to control the execution flow. Primitive step meta-actors may proceed in two ways for computing their results: interpretation or compilation. Interpretation is used when the primitive step embodies a local function call. In that case, the execution of that function is invoked locally, by computing and

passing the required arguments. On the fly code generation, deployment and execution is used in the more complex case of *ambient node primitive steps*, which embody a sensing or actuating service request that must be executed on an ambient node. This process is implemented by a specialization of the current subcomponent and is described in the next Subsection.

4.3. Ambient Node Primitive Processor

The *Ambient Node Primitive Processor* subcomponent is responsible for executing ambient node primitive steps. It collects raw sensor data, transforms them into domain objects encoded into platform-specific representations, and routes them to the uQuery processor subcomponent.

This requires access to a platform interface that provides (1) deploying and executing dynamically generated agent code, (2) dynamically discovering and providing access to all sensors in the WSN, and (3) implementing the elements of the service repository. To the best of our knowledge, the only existing system that fully satisfies our operational level requirements is ActorNet [14]. ActorNet agents are based on the actor model of computation [22]. The ActorNet runtime, illustrated in Figure 7, consists of a lightweight 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.

Figure 8 illustrates our approach for integrating a platform such as ActorNet. First, the actor template in Figure 9 is filled by the *Code Generator* module. Each *Ambient Node Primitive Meta-actor* having its preconditions satisfied and being 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 generated code pieces are deployed to the actor platform 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.

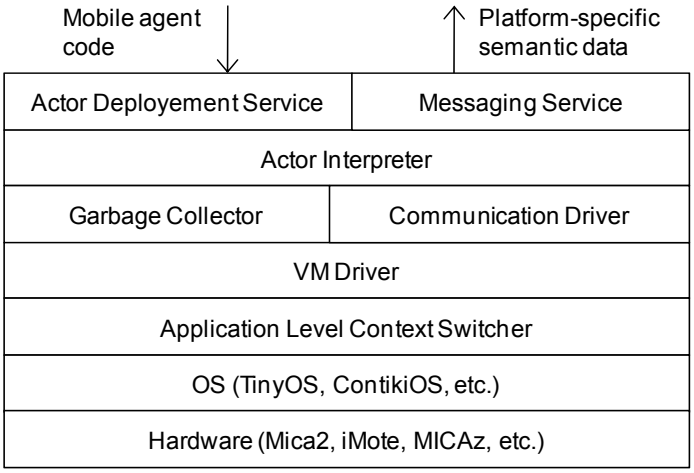

Figure 7. The architecture of ActorNet, a major module of the Ambient Node Primitive Processor subcomponent

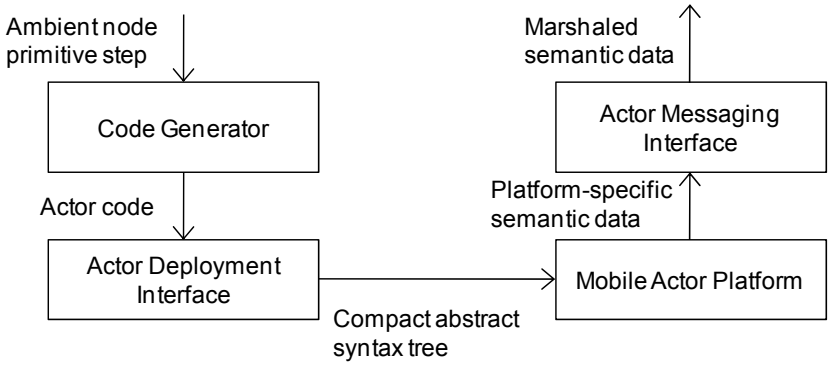

Figure 8. Interactions between modules involved in executing sensor primitive steps

Once deployed, the agent is parsed and transformed from a text string to a more compact abstract syntax tree representation, more suitable to sending across low-bandwidth wireless networks. The id of its generating meta-actor is embedded 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.

The code generator fills the template illustrated in Figure 9 in four locations: meta-actor id on lines 5 and 11, application logic and list of arguments on line 14, and execution location and uQuery engine server address on lines 5 and 12, respectively. In Figure 9, semicolons ';' indicate comments. Characters in *italics* indicate values filled in by the code generator, as an example, for the case of the three `Detect Beam Event` steps in the `T2` meta-actor shown in Figure 4. As Beam Event objects are received as results of these executions, the `Extract Edge` meta-actors may be signaled concurrently, and so on.

Note that this ActorNet code segment embodies a much more compact, higher-level specification of functionality similar to that of the NesC code in Section 1.2. One actor is launched, and it calls the Detect Beam Event primitive (primitive function with index 1 in line 12 Figure 9), which blocks until an event is detected. Scheme functions are used for communication, migration and control flow, while low-level resource management is left up to the ActorNet interpreter and the underlying services.

4.4. Complex uQueries

Now let us consider a uQuery which requires distributed coordination between different sensors. Such a uQuery 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 uQuery has two steps: detecting a Pulse object (generated by the beam sensor) and then taking a Photo with the camera.

```
1.(  (lambda (migrate)        ; start of the actor function
2.
3.   (seq                     ; sequentially execute the following:
4.
5.     (migrate 200 111)      ; migrate to the destination node
6.                            ;   (destination id 200 and meta-actor id
7.                            ;   111 are given as parameters)
8.
9.     (par                   ; create a new actor to reduce code size
10.
11.      (extmsg 111          ; send result back to the meta-actor…
12.        (migrate 100       ; after migrating back to the source node
13.
14.          (prim 1 nil))))) ; execute primitive function with index 1
15.                           ; and with an empty list of arguments
16.                           ; Note: the primitive is evaluated first,
17.                           ; then migration, then sending the message
18.
19.  (lambda (adrs val)       ; implementation of the migrate function
20.    (callcc (lambda (cc)   ; using "call with current continuation"
21.      (send (list adrs cc (list quote val)))))))
```

Figure 9. ActorNet mobile agent dynamically created by instantiating and filling a generic agent template

For executing such uQueries in a coordinated manner, we use the *argument* relation in our meta-model (see Figure 3). Whenever an argument is encountered in the uQuery 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 uQueries 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 sensors. We assume that Liz's uQuery remains the same as in the previous section. Each uQuery is represented independently, *i.e.*, there is no structural relationship between uQuery representations. In this case, the execution for each uQuery 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. End-User Programming Web Interface

Implementing end-user programming languages that satisfy both technical and cognitive requirements is a challenging task. This may explain why Pane and Myers observe that, when new programming languages are designed, the findings of Psychology of Programming (PoP) and related fields such as the *Cognitive Dimensions Framework* [23] do not seem to have much impact [24].

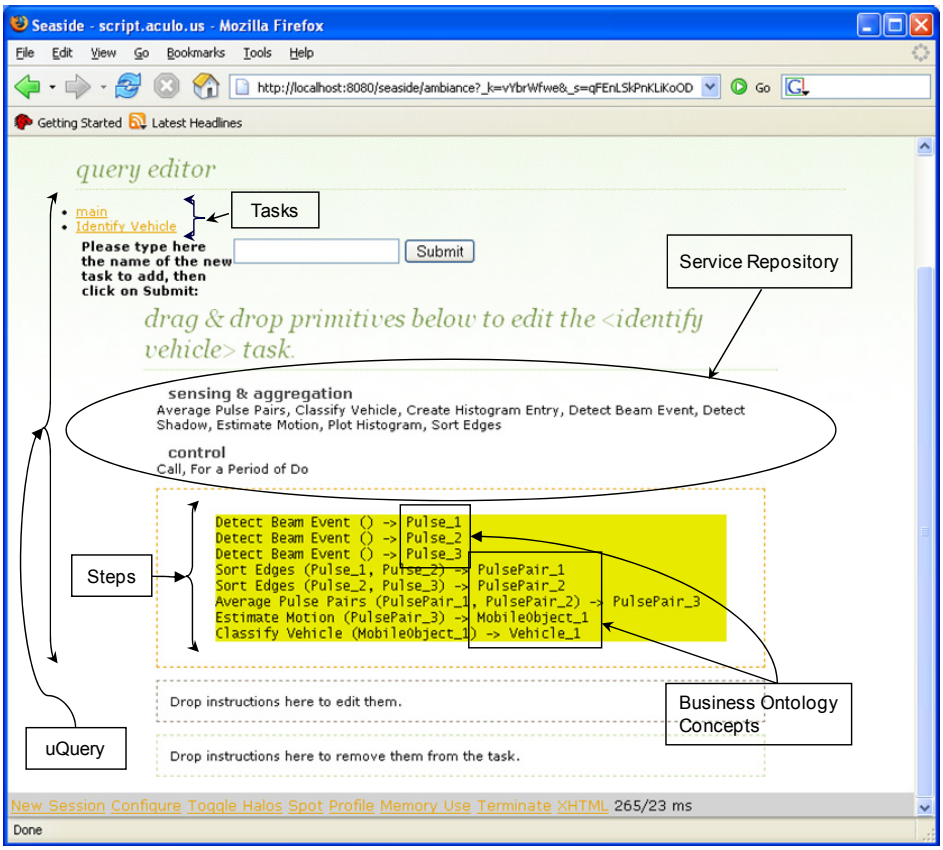

Figure 10. A snapshot of the Web interface of our uQuery engine prototype

Dart is, on the contrary, designed taking these attributes as primary considerations. We have chosen the spreadsheet as a basis for designing the end-user programming features of Dart, since it is the most widely used end-user programming metaphor [25]. The two main underlying assumptions are that (1) our end-users belong to the large category of *domain experts,* who are described by Bonnie A. Nardi as "having the detailed task knowledge necessary for creating the knowledge-rich applications they want and the motivation to get their work done quickly, to a high standard of accuracy and completeness" [25], and (2) complex behavior can be modeled by (a) associating operations to cells that compute objects, and (b) relating together those cells as "operation result holders" that provide the argument to new operations.

This is how programming takes place through Dart-based end-user programming interfaces. As illustrated by Figure 10, in the Web-enabled end-user programming interface that we have prototyped for the case of uncoordinated end-users Liz and Pablo, a uQuery is specified by direct manipulation techniques as a collection of tasks, where each task comprises a set of steps. Each task *frame* corresponds to a sheet in the spreadsheet paradigm. Each step plays the role of an "operation result holder", and is defined by specifying a call to a domain-related service, available from the service repository. This corresponds to the cell-operation association in the spreadsheet

paradigm. It should be noted, however, that the interface in Figure 10 does not match the traditional spreadsheet interfaces, which use a grid layout. Nevertheless, the programming method remains consistent with the spreadsheet paradigm sketched out above. As explained in Sections 3 and 4, steps may be graphically organized using different visual representations, such as list (Figure 10), grid, or even freeform (like in Figure 4), without any semantic impact.

Step specifications may refer to the expected results of previous calls, in terms of ontology concepts. The data flow in such cases is specified by dragging the step's visual representation and dropping it into the *instruction editing area*, situated at the bottom of the programming interface (see Figure 10). That area is then expanded with a set of fill-in widgets, one per required argument (see Figure 11). The end-user may then select from a menu the name of the appropriate step, and validate the modification. For example, the two arguments of the first `Sort Edge` step (fourth line in the list of steps in Figure 10) are set to Pulse_1 and Pulse_2, which refer to the names assigned automatically to the results of steps in the first two lines of the same task. The same method is used for modifying data flows. A more sophisticated interface may allow specifying data flow by direct drag-and-drops (without using the editing area).

Figure 10 illustrates more specifically the specification of `T2: Identify Vehicle`, which is a subordinate task and part of the `Liz's uQuery`, described and illustrated in Section 3 and Figure 4. The other tasks and steps of that uQuery may be defined in the same way. Pablo may define its uQuery by reusing `T2: Identify Vehicle` (assuming it is already defined by Liz), and creating two new tasks (see Figure 12). A `main` task (2), and another subordinate task, called `T3: Photograph Speedy Vehicle` (3). The latter nests a call to `T2: Identify Vehicle` (5). Each time a vehicle is identified by (5), it is passed to a primitive step called `Photograph If Speedy` (6), which takes a photo of the vehicle if it is speedy. This photo is then sent back to the caller `main` task, as the result of the execution of (3), which stores it in a collection and also sends it to Pablo (for example by email). This procedure repeats forever due to the definition of the unique step of the `main` task. The same uQuery may be defined in alternative ways, depending on how the service repository is designed. For example, instead of `Photograph If Speedy` (6), the uQuery engine may propose primitives like `Check If Speedy` and `Capture Camera` (described also in Section 3), which would allow a more fine-grained definition of (6). Instead of screenshots, we have chosen the "manual" notation of uQueries for illustrating Pablo's uQuery in Figure 12, since it is more compact.

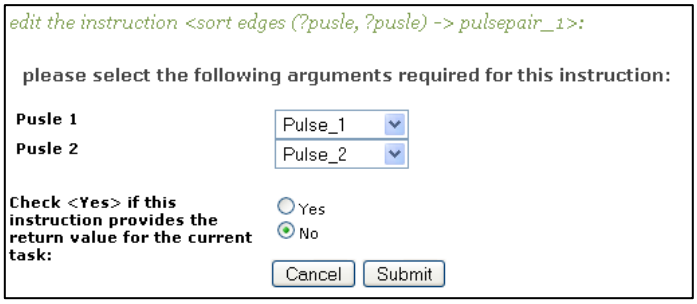

Figure 11. Example of data-flow editing by the Web interface of our uQuery engine prototype

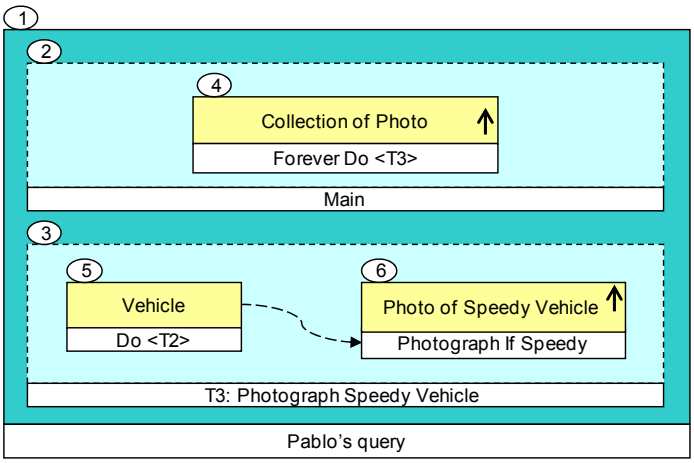

Figure 12. Representation of Pablo's uQuery by three tasks, one reused (`T2: Identify Vehicle`)

It is worthy to note that Dart extends the classic features of spreadsheets by allowing the users to integrate a set of domain and task-specific built-in primitives with relative ease, to associate any type of business object to the spreadsheet cells, and to use that object as an argument for defining new formulas (service calls), to incorporate user-defined functions comparable to those described in [26], and to take advantage of a rich set of general-purpose and domain-related control structures.

Dart is designed as an abstract object-oriented framework. It was initially implemented in *Cincom VisualWorks Smalltalk* [27], and used to refactor and improve the end-user programming language of an ecological simulation system. In this case, the end-users are experts in ecology, whose job is to model ecosystems (individual and social behavior of animals, plants, *etc.*) and to observe and study the evolution of their behavior through simulation and statistics [28]. Our new implementation of Dart is based upon *Squeak 2.8* (www.squeak.org) and *Seaside 2.7* systems.

6. Optimization

The uQuery engine design presents several opportunities for optimization as follows.

Knowledge level optimizations: Our uQuery representation allows composing 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 (see Figure 3). As a result, more complex matching algorithms can be implemented. For example, a uQuery can compare the conditions under which the available business object has been computed, and whether they match the requirements of the current uQuery. 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,* by computing who has initiated which action.

Operational level optimizations: We consider also 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 uQuery processing engine can infer the requirements for these resources from the static metadata associated with the uQuery and the contracts of the primitives it invokes. We can then fill in specific sensors and other parameters matching these criteria. 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 uQuery engine can find appropriate sensors for a uQuery by looking at attributes such as location and orientation. This simplifies the uQuery creation process for the user and provides opportunities for optimizing uQuery 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 uQuery specification.

Further, 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.

Concurrent uQuery optimizations: Data computed in one uQuery can be reused to satisfy requirements of another. This mechanism is based on exploiting static and dynamic metadata maintained by the uQuery engine (see Section 2). For example, a Vehicle object produced by one of the queries described above is associated with dynamic metadata such as a timestamp and the detecting beam sensor's id (see *Execution Context* abstraction in Figure 3). When processing a uQuery 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 uQuery. Such matching algorithms can be implemented using an inference engine, such as *NéOpus* [29], which provides rule-based reasoning in object-oriented applications. This process eliminates generating, deploying and executing redundant actors at the operational level, with the benefit of saving significant communication, sensing and computation resources in the WSN, where these resources are scarce.

7. Related Work

A survey of solutions currently proposed in the literature reveals a variety of approaches to macroprogramming WSNs: a spreadsheet approach [10], *EnviroSuite* [6], a market-based approach [12], and *Semantic Streams* [5]. 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 [30] 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.

Other comparable approaches include the Tenet architecture for tiered sensor networks [31], Sensor Webs [32], and Sensor Grids [33]. A major difference from our architecture is that we don't attribute *a priori* 'master' and 'mote' roles to the architecture components, masters being traditional artifacts having the responsibility to control the behavior of motes. In addition, none of these architectures provide the combination of an expressive and easy-to-use end-user programming for uQuery specification (Dart) with a Turing-complete mobile agent language for deployment and execution (ActorNet). Further, network flooding techniques are in general used for dynamic code deployment, instead of fine-grained code deployment available in a mobile agent platform like ActorNet.

Finally, P. Levis *et al.* [34, 35] observe that a fairly complicated action, such as transmitting a message over the radio, could be represented as a single bytecode instruction provided by an application-specific instruction set, and provides a framework for implementing high-level application-specific virtual machines on motes and for disseminating bytecode. Dart behaves, in some manner, as a framework for developing application-specific instruction sets, and thereby allows developing uQuery engines by reuse. Its coupling with a mobile agent language as explained in this chapter provides it with a powerful execution facility on motes, which is more expressive that a virtual machine such as Maté.

8. 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 everyday life activities, enabling real-time decision making and workflow 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 chapter, we explained how dynamic uQuery programming by end-users can be achieved for ambient systems comprising WSNs and traditional computing artifacts such as PCs, gateway nodes and handheld mobile devices, by extending the architectural style of Adaptive Object-Models. The resulting two-level approach to architecting uQuery engines allows separating uQuery 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 approach allows for effective end-user uQuery specification and automated execution. The uQuery 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 uQuery execution environment provides the dynamicity of macroprogramming, while enabling load balancing and other optimizations to take place. We thereby combine both expressiveness of a Turing-complete language with the simplicity of a domain-related language.

The presented meta-level and mobile-agent architecture for implementing uQuery engines is prototyped by reusing and extending our previous work on ActorNet and its implementation on Mica2 motes [14], and Dart implemented as an object-oriented framework in different dialects of Smalltalk [27, 17]. This prototype is called *Ambiance Platform* and employs a new implementation of Dart in *Squeak*. We have further used the *Seaside* framework for developing the Web-enabled uQuery programming interface of Ambiance Platform.

9. Perspectives

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 coherence and integrity analysis of uQueries. For example, the computation history discussed in Section 6 allows security enforcement through dynamic decision making about whether to execute the current step or not. Significant optimization may also be possible, for instance by integrating learning mechanisms into uQuery specification, transformation and execution. A first report of our ongoing work on these topics and more particularly on dynamic global resource management appears in [21]. Our longer term plans comprise topics such as a more robust and scalable networking subsystem for disseminating mobile code, and reliably delivering responses, in particular in presence of disconnections caused by mobility. For reliability and fault-tolerance, we would like to explore the integration of techniques for exception handling in presence of asynchronous active objects as proposed by Ch. Dony *et al.* [36].

Furthermore, ambient nodes and more particularly wireless sensors, actuators and Radio Frequency Identification (RFID) tags, allow developing digitally augmented real-life products with better value for both the manufacturers and customers. The distinguishing feature of these *ambient products* is their aptitude to smoothly incorporate themselves into the larger context of physically augmented business processes, by providing real-time "field" knowledge all along their lifecycle. Smart features of *ambient processes* such as personalization, predication, and anticipation, allow delivering unprecedented services to end-users and thereby increasing significantly competitive advantages as well as customer comfort, safety, security and satisfaction. Another application perspective of this work consists of developing networks of collaborating ambient products to enable applications such as predictive maintenance, reverse logistics, and ambient traceability (full lifecycle product and process identification, authentication, location and security).

Finally, we would like to explore conditions under which domain experts could be replaced by monitoring agents endowed with the same expertise. We plan considering morphology-based techniques elaborated by Campagne & Cardon [37] to simulate emotions in a robot, where so-called *analysis agents* are expected to control a population of subordinate (and massively multiple) *aspectual agents*. These analysis agents are supposed to possess cognitive capacities and to modify the behavior of aspectual agents. For this purpose, the structure of aspectual agents is designed to be changeable at runtime. Their behavior is "governed by some sort of augmented transition network (ATN) that can be parameterized by a set of values" [37]. This is precisely where our proposed architecture would be employed.

Acknowledgments

This work is partially funded by the University of Luxembourg, in the framework of the *Âmbiance* 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 comments and collaboration of J.-C. Augusto, F. Arbab, A. Cardon, N. Bouraqadi, P. Bouvry, Ch. Dony, B. Foote, the late and deeply regretted V. Ginot, R. Karmani, Ph. Krief, R. Johnson, M. Malvetti, S. Nadjm-Tehrani, D. Riehle, D. Shapiro, S. Sundresh, and J. Yoder.

References

[1] IST Advisory Group. "Ambient Intelligence: from vision to reality - For participation in society & business," September 2003.

[2] Ducatel, K., Bogdanowicz, M., Scapolo, F., Leijten, J. and Burgelman, J-C. (eds.). "ISTAG Scenarios for Ambient Intelligence in 2010," *IPTS-ISTAG, European Commission*, 2001.

[3] Berger, M., Fuchs, F. and Pirker, M. "Ambient Intelligence – From Personal Assistance to Intelligent Megacities," this volume, IOS Press, 2007.

[4] Nakashima, H. "Cyber Assist Project for Ambient Intelligence," Section 3.5.1 of this volume, IOS Press, 2007.

[5] Whitehouse, K., Zhao, F. and Liu, J. "Semantic Streams: A Framework for Composable Semantic Interpretation of Sensor Data," *Third European Conference on Wireless Sensor Networks*, Zurich, Switzerland, pp. 5-20, February 2006.

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

[7] Böhlen, M. “Help From Strangers - Media Arts In Ambient Intelligence Research,” this volume, IOS Press, 2007.
[8] Richard, N. and Yamada, S. “Two Issues for an Ambient Reminding System: Context-Awareness and User Feedback,” this volume, IOS Press, 2007.
[9] Huuskonen, P. “Run to the Hills! Ubiquitous Computing Meltdown,” this volume, IOS Press, 2007.
[10] Woo, A., Seth, S., Olson, T., Liu J. and Zhao, F. “A Spreadsheet Approach to Programming and Managing Sensor Networks,” *Fifth International Conference on Information Processing in Sensor Networks Track on Sensor Platform, Tools and Design Methods for Networked Embedded Systems* (IPSN SPOTS), Nashville, Tennessee, USA, April 2006.
[11] Satoh, I. “*Mobile Agents for Ambient Intelligence*,” Ishida, T., Gasser, L., and Nakashima, H. (eds.), Massively Multi-Agent Systems I, Lecture Notes in Artificial Intelligence, Springer Verlag, vol. 3446, pp. 187-201, 2005.
[12] Mainland, G., Kang, L., Lahaie, S., Parkes, D.C., and Welsh, M. “Using Virtual Markets to Program Global Behavior in Sensor Networks,” *Eleventh ACM SIGOPS European Workshop*, Belgium, 2004.
[13] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R. and Stafford, J. “Documenting Software Architectures: Views and Beyond,” *Addison-Wesley Professional*, ISBN: 0201703726, September 2002.
[14] Kwon, Y., Sundresh, S., Mechitov, K. and Agha, G. “ActorNet: An Actor Platform for Wireless Sensor Networks,” *Fifth International Joint Conference on Autonomous Agents and Multiagent Systems* (AAMAS 2006) , Future University, Hakodate, Japan, May 2006.
[15] Fok, C.L., Roman, G.-C. and Lu, C. “Mobile Agent Middleware for Sensor Networks: An Application Case Study,” *Fourth International Conference on Information Processing in Sensor Networks* (IPSN'05), Los Angeles, California, pp. 382-387, April 2005.
[16] Yoder, J. W. and Johnson, R. E. “The Adaptive Object-Model Architectural Style,” *Third IEEE/IFIP Conference on Software Architecture* (WICSA3), pp. 3-27, Montréal, Canada, August 2002.
[17] Razavi, R., Perrot, J.-F. and Johnson, R. E. “Dart: A Meta-Level Object-Oriented Framework for Task-Specific, Artifact-Driven Behavior Modeling,” *Sixth OOPSLA Workshop on Domain-Specific Modeling* (DSM’06), Gray, J., Tol-vanen, J.-P., Sprinkle, J. (eds.), Computer Science and Information System Reports, Technical Reports, TR-37, University of Jyväskylä, Finland, pp. 43-55, October 2006.
[18] Dapoigny R. and Barlatier, P. “Towards a Context Theory for Context-aware systems,” this volume, IOS Press, 2007.
[19] Briot, J.-P. “Actalk: A framework for object-oriented concurrent programming - design and experience,” *In Bahsoun,J.-P., Baba,T., Briot, J.-P., and Yonezawa, A. (eds.), Object-Oriented Parallel and Distributed Programming*, Hermès Science Publications, Paris, France, pages 209-231, 2000.
[20] Wang W.-J. and Varela, C. A. “Distributed Garbage Collection for Mobile Actor Systems: The Pseudo Root Approach,” *First International Conference on Grid and Pervasive Computing* (GPC 2006), Lecture Notes in Computer Science, Springer, vol. 3947, pp. 360-372, Taichung, Taiwan, May 2006.
[21] Mechitov, K., Razavi, R. and Agha, G. “Architecture Design Principles to Support Adaptive Service Orchestration in WSN Applications,” *ACM SIGBED Review*, vol. 4, no. 3, 2007.
[22] Agha, G. “Actors: a Model of Concurrent Computation in Distributed Systems,” *MIT Press*, 1986.
[23] Green, T. R. G. and Petre, M. “Usability analysis of visual programming environments: a ‘cognitive dimensions’ framework,” *Journal of Visual Languages and Computing* 7, 131-174, 1996.
[24] Pane, J.F., Myers, B.A. “The Influence of the Psychology of Programming on a Language Design: Project Status Report,” *Proceedings of the 12th Annual Meeting of the Psychology of Programmers Interest Group*, A.F. Blackwell and E. Bilotta (eds.), Italy: Edizioni Memoria, p. 193-205, 2000.
[25] Nardi, B.A. “A Small Matter of Programming: Perspectives on End User Computing,” *MIT Press*, 1993.
[26] Peyton-Jones, S., Blackwell, A. and Burnett, M. “A User-Centred Approach to Functions in Excel,” *International Conference on Functional Programming*, ACM, Uppsala, Sweden, p. 165-176, 2003.
[27] Razavi, R. “Tools for Adaptive Object-Models – Adaptation, Refactoring and Reflection,” (in French: “Outils pour les Langages d'Experts – Adaptation, Refactoring et Réflexivité”) *Université Pierre et Marie Curie*, Department of Computer Science Technical Report (based on doctoral dissertation), vol. LIP6 2002/014, 285 pages, Paris, France, November 2001.
[28] Ginot, V., Le Page, C., and Souissi, S. “A multi-agents architecture to enhance end-user individual-based modeling,” *Ecological Modeling* 157, pp. 23-41, 2002.
[29] Pachet, F., and Perrot, J.-F. “Rule firing with metarules,” *Sixth International Conference on Software Engineering and Knowledge Engineering* (SEKE’94), pp. 322–29, 1994.
[30] Hui, J.W. and Culler, D. “The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale,” *Second ACM Conference on Embedded Networked Sensor Systems* (SenSys’04), pp. 81-94, Baltimore, Maryland, November 2004.

[31] Gnawali, O., Greenstein, B., Jang, K.-Y., Joki, A., Paek, J., Vieira, M., Estrin, D., Govindan, R. and Kohler, E. "The TENET Architecture for Tiered Sensor Networks," *Fourth ACM Conference on Embedded Networked Sensor Systems* (SenSys'06), Boulder, Colorado, November 2006.

[32] Delin, K. A., Jackson, S. P., Johnson, D. W., Burleigh, S. C., Woodrow, R. R., McAuley, J. M., Dohm, J. M., Ip, F., Ferré, T. P.A., Rucker, D. F. and Baker, V. R. "Environmental Studies with the Sensor Web: Principles and Practice," Sensors 5, pp. 103-117, 2005.

[33] Lim, H. B., Teo, Y. M., Mukherjee, P., Lam, V. T., Wong, W. F. and See, S. "Sensor Grid: Integration of Wireless Sensor Networks and the Grid," *Local Computer Networks*, 2005.

[34] Levis, P. and Culler, D. "Maté: A tiny virtual machine for sensor networks," *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOSX), pp. 85-95, October 2002.

[35] Levis, P., Gay, D. and Culler, D. "Active sensor networks," *Second Symposium on Networked Systems Design & Implementation* (NSDI '05), Boston, MA, USA, May 2005.

[36] Dony, Ch., Urtado, Ch. and Vauttier, S. "Exception Handling and Asynchronous Active Objects: Issues and Proposal," Advanced Topics in Exception Handling Techniques, *Lecture Notes in Computer Science*, Springer, vol. 4119, pp. 81-100, September 2006.

[37] Campagne, J. C. and Cardon, A. "Artificial emotions for robots using massive multi-agent systems," *Social Intelligence Design International Conference* (SID2003), London, 2003.