

Towards a Model-driven Approach to Develop Applications based on Physical Active Objects

Luciano Baresi, Paolo Beretta, Roberto Fraccapani, Carlo Ghezzi, and Filippo Pacifici
Dipartimento di Elettronica e Informazione – Politecnico di Milano
Piazza Leonardo da Vinci, 32 – 20133 Milano (Italy)
bares@elet.polimi.it

Abstract

The increasing diffusion of ubiquitous communication infrastructures and physical active objects —like sensors and smart tags— is motivating the integration of these devices into advanced distributed systems. The novelty of these technologies has imposed a “code and fix” approach; only a few methodologies have been developed to address the integration of heterogeneous technologies to provide the user with sophisticated and flexible abstractions of real world objects.

To this end, the paper proposes the first results of a model-driven approach for the development of applications based on heterogeneous physical active objects. We propose a metamodel and a framework for automatic code generation based on Jini. The approach is exemplified on a simple case study in the domain of advanced logistics.

1 Introduction

In these last years, the increasing diffusion of ubiquitous communication infrastructures and small devices is motivating the development of applications that acquire significant information from real world objects directly. Sensors [6], active and passive RFID tags [3], and GPS receivers —hereafter collectively referred to as *physical active objects* (PAOs)— are increasingly embedded into advanced distributed systems.

Oftentimes, applications mix different PAOs, but the number of available programming models, and the lack of standards for the communication among objects supplied

by different vendors, hamper their integration. A viable development framework should provide the user with homogeneous abstraction layers, where the peculiarities of the different PAOs are hidden. The user should be able to decide programatically how PAOs are associated with real world objects and also how they interact with the application logic.

Although there are many different PAO technologies and software development systems designed for them, just a few are aimed at the integration of different technologies. To this end, the paper proposes a *model-driven* approach towards the adoption of heterogeneous PAO technologies. We solve the problem by abstracting the PAOs into software objects related to the application domain (tanks, containers, boxes, if we consider logistics). We propose a conceptual model to support the design of these systems independently of the technology used for their implementation. We also propose a Jini-based framework for the automatic generation of the application code.

The approach is demonstrated through an example in the domain of advanced logistics. It addresses the problem of tracing containers by means of different PAOs with respect to their state and context. We suppose that containers be traced in two different ways: with GPS/GPRS devices for long-range monitoring, and with RFID tags (with EPC tag ids [17]), for short-range tracking. The RFID tag is used also to store information about the actual contents of the container. The GPS infrastructure is used when the container is transported on the road, but it is switched off when it enters a parking area. At the entrance gate, the RFID tag informs about its actual contents. This is to avoid that “incompatible” containers be

parked in adjacent lots.

The rest of the paper is organized as follows. Section 2 describes the conceptual model proposed for the *model-driven* development of PAO-centric applications. Section 3 presents the supporting framework. Section 4 demonstrates the approach through a case study. Section 5 surveys some related approaches and Section 6 concludes the paper.

2 Layers and models

The main goal of this paper is to gain an understanding of how to design distributed applications where PAOs constitute the edge of the network. Specifically, we wish to provide a systematic approach to support a seamless integration of the physical objects existing in the real world and their counterparts represented in distributed systems, through which the real world is controlled and managed. In these applications, we need to support access to functionality and information provided by PAOs equipping real-world objects (such as the containers in our case study).

The layered architecture sketched in Figure 1 aims at becoming a reference model for this class of applications. The lowest level (*Physical objects level*) is where PAOs are situated. The functionality to access physical-level objects is provided by the next layer up —the *Device-access level*. This level includes the **Operation Managers**, which provide the functions to access the physical objects and generate events when certain relevant phenomena occur in the physical world. Operation managers are provided by the middleware infrastructures we use to interact with PAOs. The *Logical level* is constituted by **Logical Objects**, which provide the logical representation of physical objects. Each logical object represents a specific instance of a physical device. There is a one-to-one mapping between logical and physical objects. The events generated by the operation managers when physical objects perform certain actions low up to the corresponding logical objects, which change their state in a way that reflects what happened in the real world. For example, when an operation manager raises an event because a specific RFID tag is seen by a reader. Likewise, the operations which access the information of a logical object are accomplished by invoking the appropriate operations offered by the relevant operation manager.

The *Application level* is the highest in the hierarchy. **Application Objects** represent abstractions as they are naturally seen by the application. They abstract away from logical objects and hide the existence of different individual logical objects associated with the same application-level entity. A possibly many-to-many relationship exists between application and logical objects, but we may also have objects at both levels with no corresponding entities at the other level.

As an example, an application object may be a container, with an attribute that yields the container’s location. At the application level, we can ignore that the position is obtained by using different sensing devices. If the container is being transported, its position is obtained via GPS antenna. The “location” attribute of the logical object representing the container’s RFID is used if the container is stored in a parking area whose entry and exit gates are equipped with RFID readers. Thus, one application object is associated with two different logical objects. As another example, when a container is loaded on a truck, which is bound to a logical object, it may gather also its localization mechanism. In this case, we would

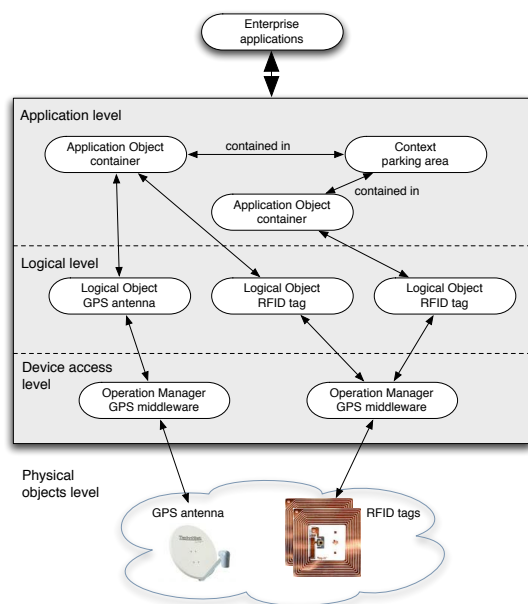


Figure 1: Abstraction levels.

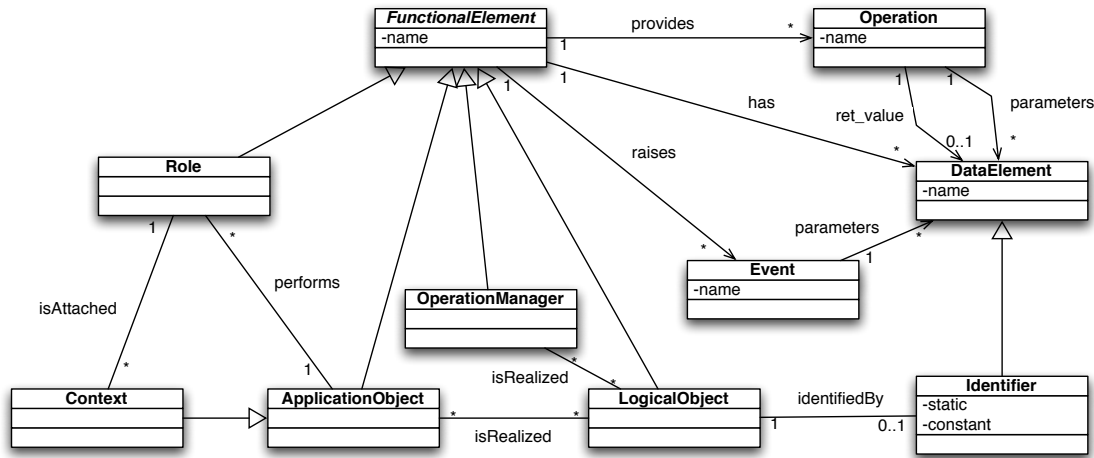


Figure 2: Main elements of our metamodel.

have several application objects bound to a single logical object.

Besides application objects, the application level also contains **Contexts**. A context is a particular kind of application object (as the metamodel presented in the next subsection will show). Moreover, application objects may belong to a context: for example, several containers may be in the same parking area.

Notice that in the layered architecture of Figure 1 there is a flow of events from lower layers to upper layers and a flow of operation invocations from upper layers to lower layers. This is consistent with the C2 architectural style [16].

Creation and destruction of instances are governed by special-purpose rules. Besides the simple manual instantiation of logical/application objects, in many cases it is possible to automatically instantiate a logical object when the specific PAO is seen by an access gateway (for example an RFID logical object can be instantiated when a reader into the system sees that tag for the first time). More complex instantiations can be performed by specifying how logical objects can trigger the instantiation of application objects or how they can manage their lifecycle.

2.1 Proposed metamodel

The metamodel of Figure 2 formalizes the concepts informally introduced above. Its main element is class **FunctionalElement**, which provides operations, raises events, and can keep a state. It is an abstract element, and thus it cannot be instantiated directly, but only through elements that extend it. **Operations** and **Events** have parameters, each one associated with a **DataElement** of a specific data type. The state of **FunctionalElements** is also composed of a set of **DataElements**. Every **FunctionalElement** has methods to manage its state, to internally notify raised events, and to let other elements subscribe to specific types of events and invoke operations provided by the element.

Several elements (with different semantics) extend **Functional Element**: **ApplicationObject**, **LogicalObject**, and **OperationManager**, already introduced in the previous section. **ApplicationObjects** and **Contexts** have methods to attach and detach **LogicalObjects** and **ApplicationObjects**, respectively, and to attach themselves to contexts. In particular, **ApplicationObject** provides methods to attach itself to **LogicalObjects** and **Contexts**, while **Context** provides methods to have **ApplicationObjects** be attached.

These elements correspond to those defined to compose the three abstraction levels of Figure 1. The *Application*

level is composed of **ApplicationObjects** and **Contexts**. They are bound through **Roles**, which represent the role played by the object when in a **Context**. This means that the operations and events associated with a **Role** can be invoked onto the object when it is in a context, and thus the interface supplied by an **ApplicationObject** changes because of the **Context** it is associated with.

The *Logical level* is composed of **LogicalObjects**. An **ApplicationObject** gets some of its data from the **LogicalObjects** it is associated with. The *Device access level* comprises **OperationManagers**. Each **LogicalObject** is associated with one **OperationManager**.

Both **ApplicationObjects** and **LogicalObjects** can be complemented with UML statecharts [4] to specify their behavior and pave the ground to automatic code generation. Events are requests from other elements or notifications from lower abstraction levels, conditions predicate over components' states and actions are method calls.

Each **LogicalObject** is associated with a single statechart, while **ApplicationObjects** can have as many statecharts as the roles they play in different **Contexts**. Currently, we impose that a **Role** only adds new methods and events, and we also impose that an **ApplicationObject** plays one **Role** at a time.

3 Jini framework

The conceptual framework is complemented with a Jini-based framework to support the development of PAO-centric applications. Jini [10] is a service oriented technology that fosters the development of highly dynamic and distributed applications. Elements are loosely coupled and can interact both via method invocations and through events.

The framework (Figure 3) oversees both the code generation and the execution of the distributed system. *Models* are translated automatically into code skeletons by means of the *Code Generator* with the help of the *Eclipse Java Emitter Template framework* (part of the EMF framework [1]). JSP-like *templates* define explicitly the code structure and get the data they need from the model via EMF.

Functional elements are implemented as Jini services. Operations and events are implemented with public methods and through the Jini event based system, respectively.

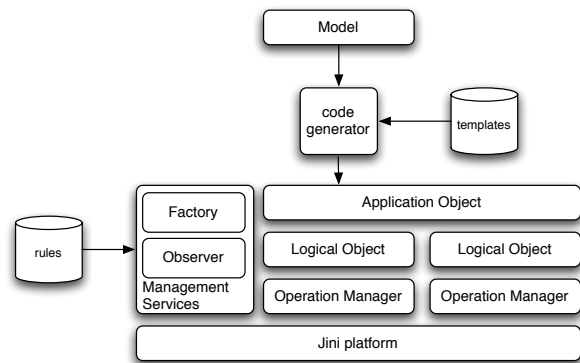


Figure 3: Jini-based framework.

Relationships between model elements are implemented through bindings between Jini services.

The second part of the generation process translates the statecharts associated with logical and application objects to implement their behavior by using the approach presented in [11]. Usually, this part of the generation process needs to be integrated with hand written code since the definition of a complete behavioral model would be too expensive and does not pay off.

Application Objects can adapt their behavior dynamically. They can both change their associated *Logical Objects* and the role they play. The former goal is accomplished by embedding adaptation logic into application objects. The latter adaptation is harder since Java does not provide native mechanisms to modify interfaces at runtime. Role management is implemented by using the *proxy* design pattern (Figure 4). The main functionality of an application object is supplied through an interface. For every role, there exists a dedicated interface that extends the original one. The implementation provides all the interfaces of the different roles, but a proxy for every role only offers a particular interface and forwards the method calls to the implementation. When the object changes role, the old role proxy service is destroyed and the new role service is created.

Management Services are in charge of lifecycle management. Our framework aims at supporting both the manual and automatic instantiation of elements. Special-purpose *rules* feed these services to enforce application-

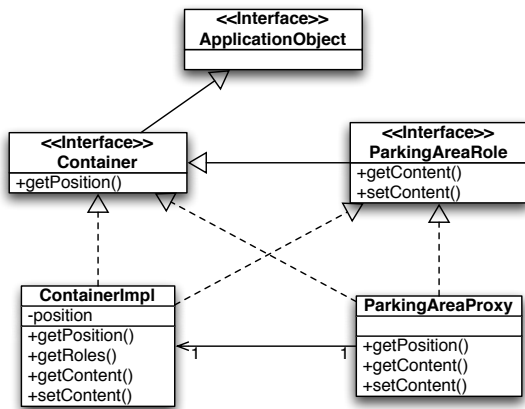


Figure 4: Role management.

specific policies for creating logical and application objects.

The *Factory* is an implementation of the abstract factory pattern. It is in charge of instantiating the services that belong to the application and of initializing the system by setting all required contexts. This component does not deal with the events that come from the physical world for the instantiation of objects. It only creates services either directly or with the help of other factories that are specific to the single functional element.

The *Observer* is used to listen to the physical world, through the *Operation Managers* specific to the different technologies. It is divided into a *Listener* and *Event sources*, with the former that receives the events from the latter. Every event source is developed by the user and interfaces with an *Operation Manager*. The difference between an event source and an operation manager is that the latter represents the interface of the software component that gets the events from the field (for example from a middleware for RFID tags), while the former defines how the application uses the events (for example it can ask the operation manager to listen to a particular category of RFID tags) and raises events that are understood by the listener. When the listener discovers a new physical object, the factory is in charge of instantiating the corresponding service (for example when a tag specified in the model is seen for the first time).

Currently, we only have a limited prototype; the com-

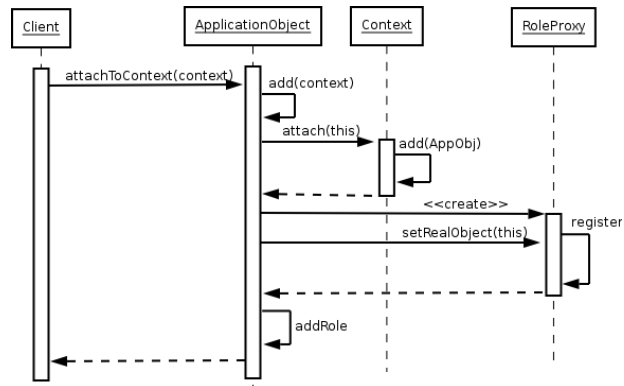


Figure 5: An application object joins a new context.

plete implementation of the lifecycle manager is part of our future work. As example of common management activities, we can dig down into an interesting scenario. If we think of role management, we have to consider the cases in which application objects join a new context or leave it. In the former case, the application object enters a new context and thus assumes a new role (Figure 5). The application object itself executes the method to join the context, the proxy service for the particular role is instantiated, and the reference to the real service is passed to it. The service is then registered and becomes available. In the latter case, the application object calls the method to leave the context, which destroys the lease of the proxy service for that role and the service is not available anymore.

4 Example application

Given the scenario introduced in Section 1, Figure 6 shows the main objects of a prototype implementation that comply with the metamodel of Figure 2. Elements are stereotyped with the name of the classifier they instantiate to improve the readability of the diagram and also to emphasize the links with the underlying metamodel.

Every **Container** is an application object that offers the capability of getting its position in an adaptive way by hiding the underlying technology. The **Container** can belong to two different contexts: **Road** and **ParkingArea**. In the latter case, the **Container** also offers the capabili-

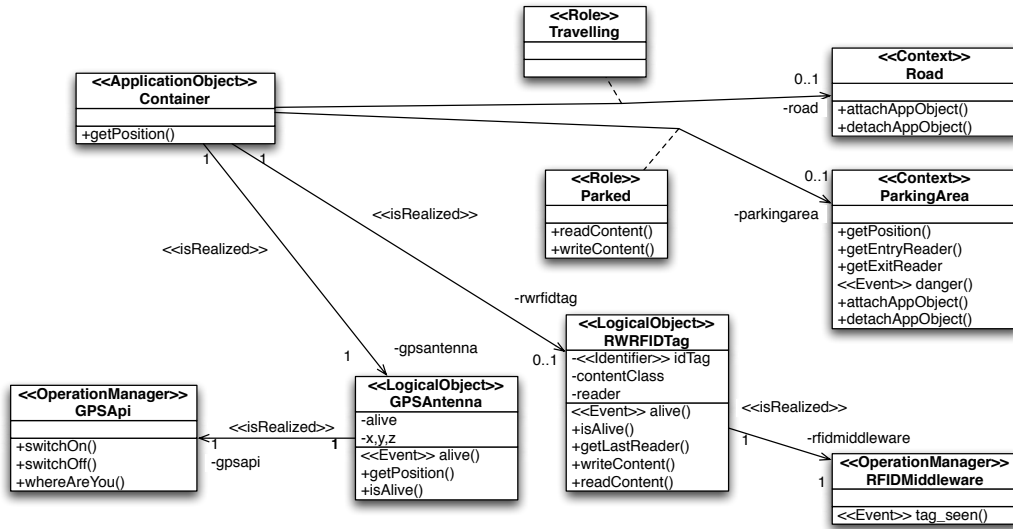


Figure 6: Structural description of the model

ties of setting and getting its content. This is done through RFID technologies that are not available when the container is travelling. A **ParkingArea**, as a whole, also offers the capability of raising alarms when two containers, which contain incompatible goods, are too close. This means that the **ParkingArea** keeps track of the content of each **Container** and uses this information to compute some compatibility analyses. This check is triggered by the event raised by the parking area when a new **Container** asks for joining the context (**ParkingArea**). The context scans its content and checks the new container against all the others for incompatibilities. If the goods in the new container are not compatible with those stored in one of the containers already in the area, the system raises an alarm and identifies the containers that may cause problems.

The application object (**Container**) has bindings with two logical objects: one with **GPSAntenna** and the other with **RWRfidTag**. Both the logical objects know when their PAO must be used and how to compute the position of the application object (even if the GPS gives geospatial coordinates and the RFID tag only gives the identifier of the last reader that intercepted it).

The **Container** adapts its behavior by changing the role

it plays. The statechart of Figure 7 explains how it works. The automaton comprises two parallel states. The first region describes the behavior of the main object; the second region describes the behavior added by role **Parked**. Role **Travelling** does not add behavior and thus it is not considered here.

If we consider the main behavior, event *getPosition*, which corresponds to the call of operation *getPosition*, is handled in two different ways depending on the used logical object. When the **Container** enters the **ParkingArea**, the event gathered from the logical object triggers a change in both performed role (second region) and used logical object (first region).

The second region (i.e., role **Parked**) also checks, whenever a **Container** is **Parked**, if its content is compatible with the contents of the other containers.

5 Related work

The problem of integrating PAOs in distributed systems has gained more and more interest in the last couple of years in the fields of ubiquitous and pervasive computing.

Some approaches focus on the simple integration of a

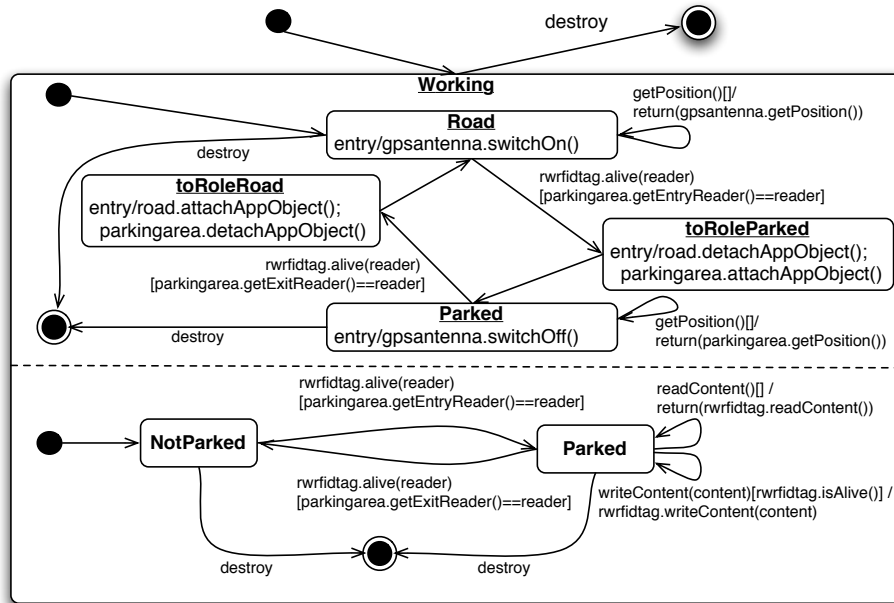


Figure 7: Dynamic description of Application Object

specific technology without providing adaptivity or sophisticated abstractions. For example, the EPC Global Network [17] approach for RFID tags, along with its implementations [9, 12, 2], provides an infrastructure for event-based access of data coming from RFID readers and allows a world-wide sharing of this information.

Other approaches focus on providing high-level abstractions for the events generated by PAOs. For example, [18, 7, 5, 15] provide a framework to associate higher level meaning with events generated by RFID readers and to aggregate them into complex business-level events. These last approaches add abstraction, but do not go beyond a single technology (RFID) and only concentrate on providing “smart” events to the user. Also [8] provides an approach to build high-level descriptions of applications for sensor networks, which are then translated directly into the code for each node.

Similarly to our approach, some proposals define “generic” high level models for ubiquitous systems based on a logical model of the environment. For example, [14] proposes abstractions of physical world objects capable

of executing software and [13] offers a similar approach that concentrates on the discovery of objects available in a virtual space.

6 Conclusions and future work

This work presents the first experiments with a model-driven methodology centered around physical active objects. The methodology is based on abstractions that allow users to conceive their applications by concentrating on their business objects. The links between PAOs and application objects are set by means of dedicated rules.

We present a metamodel, for the structural design of these applications, and a Jini framework for their implementation and deployment. A simple case study shows how the developer can separate the application logic from the issues related to the physical active objects.

There are several research issues we still want to address. We want to extend the metamodel to consider non-functional attributes associated with PAOs. The frame-

work will follow the evolutions of the metamodel, but we also want to add new features to consider the autonomic adaptation of running applications.

References

- [1] Eclipse modeling framework, (emf). <http://www.eclipse.org/emf/>.
- [2] IBM ale. <http://www.alphaworks.ibm.com/tech/alepreview>.
- [3] RFID journal. www.rfidjournal.com.
- [4] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language user guide second edition*. Addison Wesley Longman Publishing Co., Inc., 2005.
- [5] P. De, K. Basu, and S. K. Das. An ubiquitous architectural framework and protocol for object tracking using rfid tags. In Proceedings of *MobiQuitous*, pages 174–182, 2004.
- [6] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In Proceedings of ASPLOS, pages 93–104, 2000.
- [7] Y. Kim, M. Moon, and K. Yeom. A framework for rapid development of RFID applications. In Proceedings of *ICCSA (4)*, pages 226–235, 2006.
- [8] M. Kuorilehto, M. Kohvakka, M. Hännikäinen, and T. D. Hämäläinen. High abstraction level design and implementation framework for wireless sensor networks. In Proceedings of *SAMOS*, pages 384–393, 2005.
- [9] S. Microsystem. *Datasheet Sun Java System RFID Software*, 2005.
- [10] J. Newmarch. *A programmer's guide to Jini technology*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2000.
- [11] I. Niaz and J. Tanaka. An Object-Oriented Approach To Generate Java Code From UML Statecharts. *International Journal of Computer & Information Science*, 6(2), 2005.
- [12] Oracle. *Oracle Sensor Edge Server Administrator's Guide 10g Release 2 (10.1.2)*. Oracle, December 2004.
- [13] A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. H. Campbell, and M. D. Mickunas. Olympus: A high-level programming model for pervasive computing environments. In Proceedings of *PerCom*, pages 7–16, 2005.
- [14] I. Satoh. A location model for pervasive computing environments. In Proceedings of *PerCom*, pages 215–224, 2005.
- [15] A. Suenbuel, J. Schaper, and T. Odenwald. Towards a comprehensive integration and application platform for large-scale sensor networks. In Proceedings of *UCS*, pages 232–244, 2004.
- [16] R. Taylor, N. Medvidovic, K. Anderson, E. Whitehead Jr, and J. Robbins. A component-and message-based architectural style for GUI software. In *Proceedings of ICSE*, pages 295–304, 1995.
- [17] K. Traub, G. Allgair, H. Barthel, L. Burstein, J. Garrett, B. Hogan, B. Rodrigues, S. Sarma, J. Schmidt, C. Schramek, R. Stewart, and K. K. Suen. The EPC-global architecture framework. Technical report, EPC-global, Inc., 2005.
- [18] F. Wang, S. Liu, P. Liu, and Y. Bai. Bridging physical and virtual worlds: Complex event processing for RFID data streams. In Proceedings of *EDBT*, pages 588–607, 2006.