Between Simulator and Prototype: Crossover Architecture for Testing and Demonstrating Cyber-Physical Systems[*]

Tomasz Paczesny, Jarosław Domaszewicz,
Przemysław Konstańczuk, Jacek Milewski, and Aleksander Pruszkowski

Institute of Telecommunications, Warsaw University of Technology
Nowowiejska 15/19, 00-665 Warsaw, Poland
{t.paczesny,domaszew,apruszko}@tele.pw.edu.pl,
{p.konstanczuk,j.milewski}@stud.elka.pw.edu.pl

Abstract. Consider the development of a new middleware targeted at cooperating smart objects. Each smart object should have an embedded node connected to the object's sensors and actuators. Building a prototype of such a middleware is inherently labor-intensive, especially when it comes to crossing the cyber-physical boundary, i.e., node-to-object interfacing. Also, soon one needs to be able to validate the middleware's emerging API. Consequently, two separate "products" are usually developed: a programmer-oriented simulator and an actual, node-based prototype. Both are less than perfect for testing and demonstration purposes, and there is hardly any reuse of work invested in producing them. We propose an architecture that enables intermediate, crossover setups combining elements of the simulator and of the prototype. The key idea is system-wide decoupling of the cyber domain from the physical domain, by means of a dedicated entity. The architecture emphasizes incremental formation of testing and demonstration setups, reusability of elements needed to create them, and flexibility in combining those elements. We validate our architecture with a proof-of-concept infrastructure and a number of experimental setups.

Keywords: pervasive computing, cooperating smart objects, middleware, simulation, demonstration techniques.

1 Introduction

Consider the generic problem of developing a new pervasive computing middleware targeted at networked smart objects (e.g., home objects). The likely goal of the middleware is to simplify making applications based on object cooperation. Each participating object should be equipped with an embedded node, usually

[*] This work has been co-funded by the FP7/ICT Programme of the European Union, under project POBICOS (Platform for Opportunistic Behaviour in Incompletely Specified, Heterogeneous Object Communities), contract Nr. FP7-ICT-STREP-223984.

microcontroller-based, connected to some object-related sensors and actuators and able to wirelessly communicate with nodes embedded in other objects. The middleware layer should reside on top of the system software of each node and expose some middleware-specific facilities to the application layer. Importantly, the objects are likely to differ from one another as to their functionality and available sensors and actuators.

Efforts to develop such a cooperating object middleware may differ as to the specifics of the middleware's design goals, its programming model, and architecture. In each case, however, building a proof-of-concept middleware prototype is an inherently labor-intensive and difficult task. The main reasons for the difficulties is that the system under development is distributed among multiple nodes, the nodes communicate wirelessly (i.e., very unreliably), and the selected node platform, being embedded, is usually quite difficult to program.

Additional reasons that make the development of the prototype labor-intensive have to do with crossing the cyber-physical boundary, i.e., interfacing nodes (the cyber domain) to objects (the physical domain). First, one should do the interfacing to a number of *different* real objects, each with its specific functionality and a set of sensors and actuators. The more different kinds of objects are interfaced to, the better the prototype becomes for testing and demonstration purposes. Each interfacing, however, is a small, non-trivial project of its own. Second, some objects may simply not be available for interfacing, due to being, e.g., too costly or bulky.

A related complication in the overall middleware development process is that until the prototype is built, there is hardly any way to validate the middleware's emerging programming model or API. There is a need for an environment enabling one to experiment with the API, by debugging and running example applications. Without such an environment, "paper-based" programming exercises remain as the only programming model validation option.

As a result of the above, the work typically proceeds in two complementary and separate directions. The first one is to build a programmer-oriented system simulator. Early in the project it is much easier to develop the simulator, a desktop software artifact, than the actual, node-based prototype. On the other hand, the simulator, being centralized, usually does not include any distributed middleware protocols, and, more importantly, it takes care of objects only by software components that simulate them. As such, it is not a very appealing testing and demonstration tool; the results obtained from such a software-only simulator are inevitably perceived as "distant from reality."

The other direction is to develop an actual prototype, based on a selected embedded node platform. The difficulties of that work have been elaborated above. Typically, a working version of the prototype is obtained much later in the project; moreover, only a quite limited number of objects are interfaced to. Thus one ends up with two *separate* "products": a (software-only) simulator and a limited-scale prototype (see Fig. 1). Both of them are much less than perfect when it comes to testing and demonstration, and there is hardly any reuse of work invested in producing them.

In this paper we propose an architectural approach which enables intermediate, crossover setups that combine elements of the simulator and of the prototype. The architecture emphasizes incremental formation of increasingly complex setups, reusability of elements needed to create them, and flexibility in combining those elements. As a result, interesting testing experiments and demonstrations can be had much earlier in the middleware development process, when parts of the actual system are still missing. We validate our architecture with a compliant proof-of-concept infrastructure and a number of experimental setups.

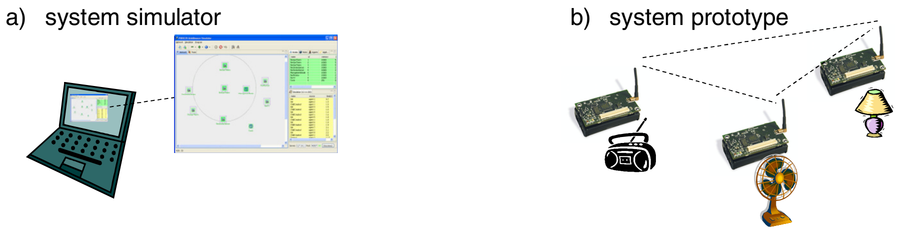

Fig. 1. Two separate, non-synergetic products of a cyber-physical system development

The structure of the paper is as follows. Section 2 presents the crossover architecture. Section 3 focuses on our proof-of-concept implementation of the architecture and related experimental setups. Related work is covered in Section 4, and the paper is concluded in Section 5.

2 Simulator/Prototype Crossover Architectural Approach

We perceive a cyber-physical system as a collection of *nodes* and *objects* (Fig. 2). An embedded node (a cyber artifact) is interfaced to an object (a physical artifact, e.g., a lamp, heater, fan, etc.), by means of sensors and actuators provided by the object. A node executes a program, controls the object's sensors and actuators, and communicates with other nodes. The sensors and actuators, which interact with the surrounding environment, are accessed by nodes through a *sensors/actuators interface*. The sensors/actuators interfaces of all nodes define a boundary between the *cyber domain* and the *physical domain*. Making a clear distinction between these two domains and identifying interfaces between them is essential to our approach.

To avoid misunderstandings, we note the obvious fact that the cyber domain has a physical grounding as well. For example, the nodes consume energy to run and use a physical medium to communicate. In our architecture, however, the physical domain includes objects proper (i.e., without embedded nodes) and the surrounding environment that can be probed or altered with sensors and actuators.

Recall the two canonical products of the middleware development process: the simulator and the prototype. Both can be mapped onto the model presented in Fig. 2. As for the simulator, components simulating nodes belong to the cyber domain, while those simulating objects and the surrounding environment – to the physical domain. For the prototype, nodes constitute the cyber domain, while interfaced objects belong to the physical domain. In both systems, however, the two domains are rather tightly coupled, through *internal* interfaces.

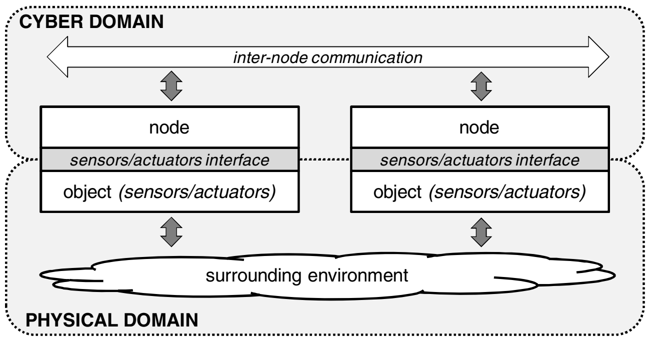
CYBER DOMAIN
inter-node communication
node
sensors/actuators interface
object (sensors/actuators)
node
sensors/actuators interface
object (sensors/actuators)
surrounding environment
PHYSICAL DOMAIN

Fig. 2. A model of a cyber-physical system

The key point of our approach is to decouple the cyber domain from the physical domain, by means of a dedicated, separate entity. Specifically, we postulate that one of the first steps in the middleware development process should be to specify and implement a *Sensors/Actuators Abstraction Infrastructure* (Fig. 3). The infrastructure should fulfill the following requirements:

- It should offer node ports, used to connect to either *simulated or real* nodes. Similarly, it should offer object ports, used to connect to *simulated or real* objects. Both local and remote connections should be allowed.
- It should support a protocol used to communicate with nodes and objects. At the test/demo setup time, the protocol should make it possible (a) to connect a node or object to a port, and (b) to pair a node with its object. At runtime, the protocol should make it possible for paired nodes and objects to exchange sensor and actuator data.
- The sensor and actuator data exchanged between a node and a paired object should be based on a systematic, abstract representation of sensors and actuators that can be encountered in the domain in question (e.g., the home domain).

Having the Sensors/Actuators Abstraction Infrastructure in place, one can develop compatible implementations of nodes and objects. On the cyber side one can have (a) a *multi-node simulator* or (b) a network of real nodes. In the latter case, a *node proxy* is usually needed to connect an actual node to the Sensors/Actuators Abstraction Infrastructure. On the physical side one can have simulated or real objects. Each simulated object may be implemented as a standalone, small application, which we call an *object simulet*. Alternatively, a set of simulated objects may be implemented within a single *environment simulator*, which, besides the objects themselves, models the surrounding environment in which the objects reside. To connect a real object to the infrastructure, one usually needs an *object proxy*.

One can then freely combine node and object implementations to create test/demo setups. Owing to the protocol and the systematic sensors/actuators representation, the cyber part cannot tell a difference between a real and simulated physical part (and vice versa). Many useful combinations are possible; the most typical ones are shown in Fig. 4 and listed below in the order, in which they are most likely to occur in an actual middleware development process. In the figure we do not depict node and object proxies. A feature to be observed is reusability of constituent elements.

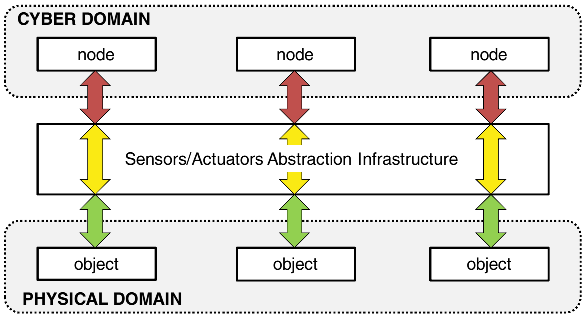

Fig. 3. Domain decoupling with Sensors/Actuators Abstraction Infrastructure

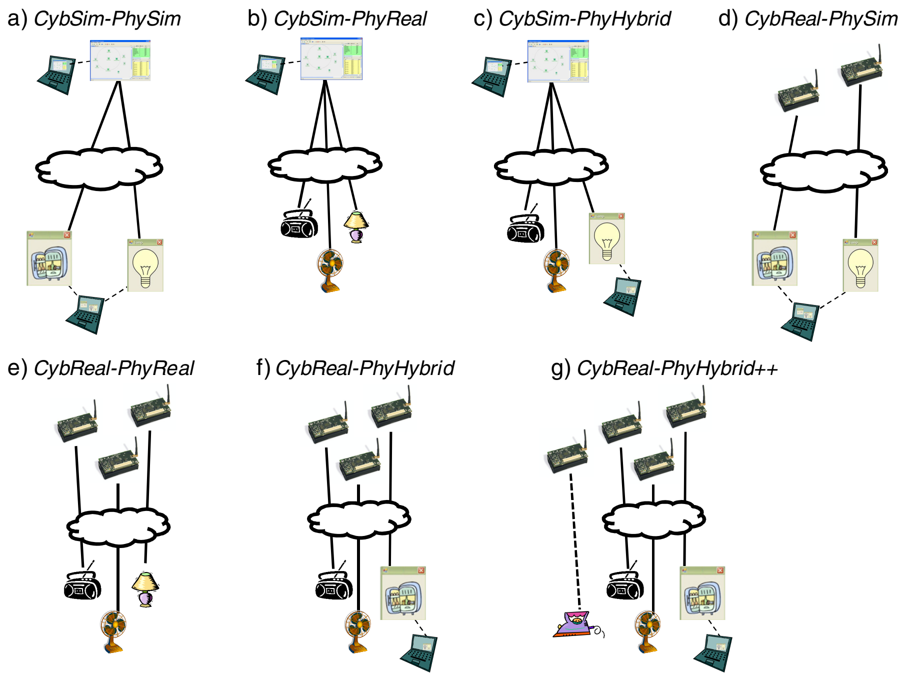

Fig. 4. Typical usage scenarios for our architectural approach

The *CybSim:PhySim* combination (Fig. 4a.) puts together a multi-node simulator and either an environment simulator or a set of object simulets. Note that this combination is functionally equivalent to an integrated simulator mentioned in the introduction, the differing factor being that an explicit decomposition has been introduced by means of the Sensors/Actuators Abstraction Infrastructure.

The *CybSim:PhyReal* combination (Fig. 4b.) differs from the previous one in that real objects replace simulated ones. This way one can test/demo the system in a centralized and thus easily controllable way, while creating a very realistic user experience through real objects. This can be achieved without having an actual middleware implementation or objects actually interfaced to nodes.

The *CybSim:PhyHybrid* combination (Fig. 4c.) originates from the previous two. The multi-node simulator works with *both* real objects and object simulets, thus increasing the number and/or diversity of objects used in a test or demo. Note that elements from the previous two combinations can simply be reused in the present one; no extra implementation effort is required.

The *CybReal:PhySim* combination (Fig. 4d.), puts together real, wirelessly communicating nodes with either an environment simulator or a set of object simulets. This setup allows one to test/demonstrate actual middleware software without complexities related to real object interfacing (or before any real objects are available).

The *CybReal:PhyReal* combination (Fig. 4e.) is functionally equivalent to the usual prototype setup, with nodes and objects being connected through the infrastructure rather than node-internal interfaces. While the indirect connection is suboptimal, it is easier to achieve than real, hardware integration. Moreover, this combination is obtained by reusing elements from the previous combinations.

Merging the two preceding combinations leads to *CybReal:PhyHybrid* (Fig. 4f.), where a mix of real and simulated objects is used with real nodes. This may prove useful when one wants to enhance a setup with objects too bulky, complex, or expensive to actually interface to (e.g., a refrigerator).

Finally, one should mention the *CybReal:PhyHybrid*++ combination (Fig. 4g.) There, real nodes using the Sensors/Actuators Abstraction Infrastructure coexist with real nodes already fully integrated with their objects (as exemplified by the node fully integrated with an iron). Interestingly, that combination features objects of three kinds: (a) fully integrated, (b) connected through the infrastructure, and (c) simulated.

The only significant limitation in combining the above-described artifacts is that, on the cyber side, one cannot mix a multi-node simulator and real nodes, the main reason being that the multi-node simulator does not implement per-node middleware protocols (which is our assumption in this paper).

Note, that to obtain a final, integrated, “traditional” prototype, node and object proxies need to be removed and “traditional” node-to-object interfacing still needs to be performed. Thus the development of the proxy applications is the main overhead of our architectural approach.

3 Proof-of-concept Implementation and Experiments

The main technical choice in our proof-of-concept implementation is the definition of a sensors/actuators representation. We adopt a bi-directional command/event model (Fig. 5), used, e.g., in the nesC programming language. In this model, sensors and actuators are represented by commands that can be executed on them (e.g., `SwitchOn()` executed on an on/off switch actuator) and events they can deliver (e.g., `FireDetected()` invoked by a fire detecting sensor). Consequently, every object can be described with a list of commands and events it supports. Both commands and events are modeled as typical functions, i.e. they may have an argument list and may return a value. We do not make any assumptions about possible commands and events, their semantics, nor how they are processed.

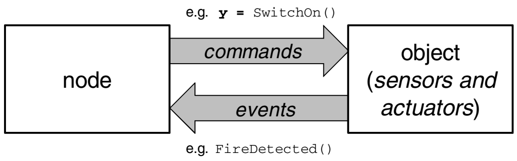

Fig. 5. Sensors/actuators representation

The presented representation is adopted in our *Sensors/Actuators Abstraction Protocol* (SAAP), used by nodes and objects to communicate with a Sensors/Actuators Abstraction Infrastructure (see Fig. 3). The design goals of the protocol were simplicity and platform-independence; these were achieved with seven text-based message types outlined in Table 1.

Table 1. Messages of the SAAP protocol used by nodes and objects to communicate with a Sensors/Actuators Abstraction Infrastructure

Message type	Arguments
CONNECT	{NODE | OBJECT} *client_label resources_list*
DISCONNECT	
EVENT	*request_id request_label parameters*
EVENT_RETURN	*request_id return_value*
COMMAND	*request_id request_label parameters*
COMMAND_RETURN	*request_id return_value*
LINK_STATUS	{ON | OFF}

An example of a SAAP message exchange is illustrated in Fig. 6. CONNECT is used by clients to register themselves with the infrastructure, providing a unique client label and a list of supported commands and events. DISCONNECT removes a client from the infrastructure's registry. Once two clients (a node and an object) are paired by the infrastructure, the LINK_STATUS ON message is sent to them. From that moment, until receiving LINK_STATUS OFF, nodes may issue COMMAND messages (which are forwarded by the infrastructure to the paired object), and objects may issue EVENT messages (which are forwarded to the paired node). Clients receiving such messages may respond with COMMAND_RETURN or EVENT_RETURN, respectively, to pass the returned value. The returned value is linked to a command or event message by means of a request ID (*request_id*).

The Sensors/Actuators Abstraction Infrastructure has been implemented [1] as an application called *Management Server* (MServer), written in JAVA and running on top of TCP/IP. MServer accepts SAAP protocol messages, creating a separate TCP socket for every client. Notably, nodes and objects connecting (locally or remotely) to MServer are not necessarily separate applications or hardware entities. It is perfectly possible for one application (e.g., a multi-node simulator) to represent multiple nodes, as long as there is one connection with MServer per node; the same applies to objects.

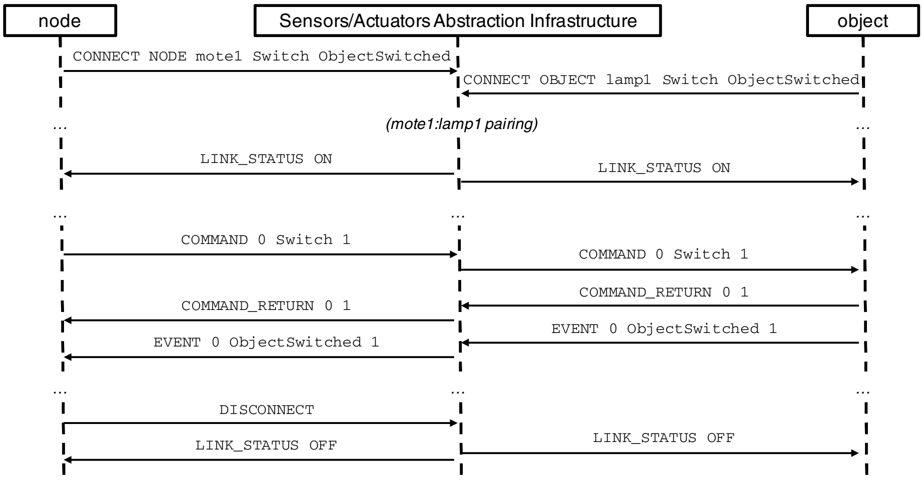

Fig. 6. An example of SAAP message exchange

It is in the responsibility of the user of MServer to set up an experiment by pairing nodes with objects, in such a way that events generated by an object are supported by its node, and commands invoked by a node are supported by its object. The pairing is done through a command console provided by MServer. For the user, clients are distinguished by the labels they supply when connecting to MServer (*client_label* from Table 1.) Auxiliary features of MServer include browsing through connected clients and monitoring SAAP traffic ongoing between them.

For the experiments with SAAP and MServer, we developed a number of compatible clients. The node clients include (a) a simple multi-node simulator representing two nodes, with an integrated application logic (see below), and (b) an iMote2-based [2] real node featuring a cooperating object middleware, called POBICOS [3]. The object clients include (a) a light sensor simulet, (b) a lamp simulet, and (c) a real lamp controlled with a smart plug. For the real node (iMote2) and real object (the lamp), tiny proxy applications are used. The proxies communicate with actual devices over RS232 connections.

The SAAP clients were then easily combined in different setups. All the setups were used to run a simple home automation application, *TwilightSwitch*. The application turns the lamp on when it is dark and switches it off when it is bright. The setups are presented in Fig. 7. The first setup (Fig. 7a) represents *CybSim:PhySim* combination. In the second setup (Fig. 7b), we have replaced the lamp simulet with the real lamp, thus obtaining the *CybSim:PhyHybrid* combination. The third setup (Fig. 7c), realizing the *CybReal:PhyHybrid++* combination, included two iMote2 nodes with a *TwilightSwitch* application running on top of the POBICOS middleware. One of the nodes was directly interfaced to a real brightness sensor, while the other one used the lamp simulet. Finally, in the fourth setup, representing the *CybReal:PhyReal* combination (Fig. 7d), both real brightness sensor and real lamp were used. For all the setups, we observed correct system behavior.

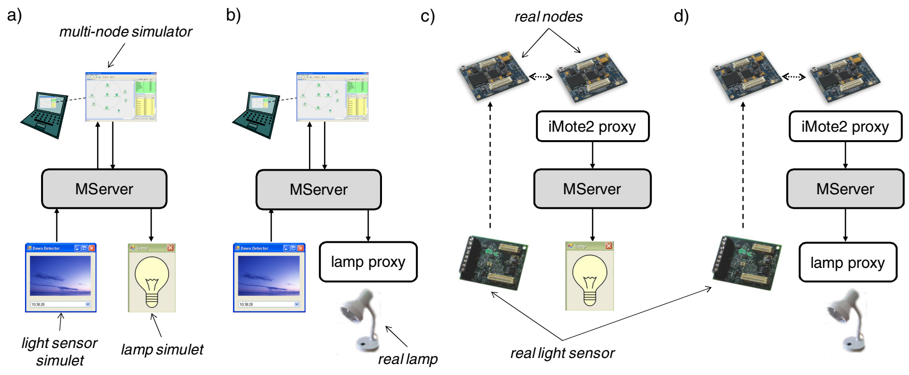

Fig. 7. Proof-of-concept experimental setups

4 Related Work

DiaSim [4] and eHomeSimulator [5] are examples of elaborate pervasive computing simulators. Similarly to our approach, they clearly separate the environment and the application logic, by means of a well-defined API. They allow one to use the same application logic with both simulated and real devices. However, both solutions support only centralized application logic and are not as general as our approach; they work in the context of a specific pervasive computing system and application development methodology.

UbiWise [6] uses a game engine to provide an interactive 3D environment, in which the user can freely move and interact with a variety of devices. Some of the devices are simulated, and some may be real. However, the separation between the cyber and physical domains is not clear, as each device is a self-contained entity that includes both application logic and sensors/actuators.

An approach similar to our object simulets is used in the ATLAS platform [7]. The Sensor/Actuator Emulator [8] provides a set of lightweight applications equipped with a GUI imitating an actual device. Such simulated devices can be then used in ATLAS applications together with real devices. Nevertheless, the approach considers only the specific case of the ATLAS platform, in which the application logic is centralized.

Huebscher and McCann in [9] consider an architecture where data from simulated sensors and actuators is provided to distributed nodes simulated with TOSSIM [10]. This resembles our *CybSim:PhySim* combination, because of a clear interface between the multi-node simulator (TOSSIM) and a sensor/actuator simulator (Context-Aware Simulator). However, no further combinations are considered in [9].

The PoSim simulator [11] of the POBICOS middleware clearly separates a simulated node and its sensor/actuator resources. This makes it possible to have the *CybSim:PhySim* and *CybSim:PhyReal* combinations, which were both exercised in the POBICOS project [12]. However, in that solution, the cyber domain cannot be easily populated with real nodes.

It appears that a number of existing pervasive computing systems and/or their simulators make attempts at supporting combinations of real and simulated elements in crossover setups. However, to the best of our knowledge, no existing work proposes an approach as general and flexible as the one presented in this paper. In particular, the possibility of connecting real nodes with object simulets appears to be a unique feature of our approach.

5 Conclusion

Using the nodes and objects introduced above, we have easily obtained four different crossover setups for running an example application. The experiments confirmed usability of our infrastructure, as well as the possibility to flexibly combine real and simulated elements of the system. The approach should easily scale to bigger and more sophisticated setups. We believe that many existing pervasive computing systems could benefit from adopting our architecture, especially if they have already introduced a clear interface between the cyber and physical domains.

References

1. Konstańczuk, P., Milewski, J.: Home environment modeling for POBICOS platform simulator. BSc thesis, Warsaw University of Technology (2010)
2. Memsic Corporation, `http://www.memsic.com/`
3. STREP/FP7-ICT: Platform for Opportunistic Behaviour in Incompletely Specified, Heterogeneous Object Communities (POBICOS), `http://www.ict-pobicos.eu`
4. Bruneau, J., Jouve, W., Consel, C.: DiaSim: A parameterized simulator for pervasive computing applications. In: Mobile and Ubiquitous Systems: Networking & Services, MobiQuitous (2009)
5. Armac, I., Retkowitz, D.: Simulation of Smart Environments. In: IEEE International Conference on Pervasive Services, pp. 322–331 (2007)
6. Barton, J., Vijayaraghavan, V.: Ubiwise: A Ubiquitous Wireless Infrastructure Simulation Environment. Technical report, HPL-2002-303, HP Labs (2002), `http://www.hpl.hp.com/techreports/2002/HPL-2002-303.pdf`
7. King, J., Bose, R., Yang, H.-I., Pickles, S., Helal, A.: Atlas: A Service-Oriented Sensor Platform: Hardware and Middleware to Enable Programmable Pervasive Spaces. In: Proceedings of 31st IEEE Conference on Local Computer Networks (2006)
8. Sensor/Actuator Emulator for the Atlas Platform, `http://www.icta.ufl.edu/atlas/emulator/AtlasSensorEmulatorProgrammersManualv1.1.pdf`

9. Huebscher, M.C., McCann, J.A.: Simulation model for self-adaptive applications in pervasive computing. In: Proceedings of 15th International Workshop on Database and Expert Systems Applications, pp. 694–698 (2004)
10. Levis, P., Lee, N., Welsh, M., Culler, D.: TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In: Proceedings of the First ACM Conference on Embedded Networked Sensor Systems, SenSys (2003)
11. Georgakoudis, G., Koutsoumbelias, M., Lalis, S., Lampsas, P.: D4.2.3 Final System Simulator. STREP/FP7-ICT POBICOS project deliverable (2011)
12. Palacka, V., Prekop, J., Koyš, J., Chabada, J., Bujna, M.: D5.1.3 Application development report. STREP/FP7-ICT POBICOS project deliverable (2011)