Domaszewicz, J.; Lalis, S.; Paczesny, T.; Pruszkowski, A.; Ala-Louko, M.
„Graspable and Resource-flexible Applications for Pervasive Computing at Home”
IEEE Communications Magazine, vol.51, no.6, pp.160-169, June 2013
doi:10.1109/MCOM.2013.6525610, URL: http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=6525610&isnumber=6525582

Graspable and Resource-Flexible Applications for Pervasive Computing at Home

Jaroslaw Domaszewicz [1], Spyros Lalis [2], Tomasz Paczesny [1],
Aleksander Pruszkowski [1], Mikko Ala-Louko [3]

[1] Warsaw University of Technology, Warsaw, Poland
[2] IRETETH/CERTH and University of Thessaly, Volos, Greece
[3] VTT Technical Research Centre of Finland, Oulu, Finland

ABSTRACT

We envision the home populated with regular objects that allow programmatic access to their sensors and actuators. The objects jointly form a pervasive computing platform, open for third party independently developed applications. A challenge is how to support people in deploying and managing such applications. Today, the user perceives an application as an immaterial artifact, accessed through a screen-based interface of a general-purpose computing device. Contrary to that established paradigm, we propose to reify the pervasive computing application as a simple physical thing, called the “application pill.” The pill can easily be grasped and operated: the user brings the pill home, switches it on, and checks if it works just by glancing at its on/off diode. As the application is destined for many homes, each featuring a different collection of objects, the user should be provided with high-level feedback on how well the application can work in her home. Accordingly, the application pill is also equipped with a simple “functionality level” indicator. The degree to which the application can deliver its functionality on top of an available object collection is captured as a single number and displayed by the pill. We present a concrete proof-of-concept elaboration and implementation of these ideas in a pervasive computing middleware platform targeted at cooperating objects.

I. INTRODUCTION

Regular objects augmented with sensors and actuators are becoming an ever increasing part of the home. Significant potential can be unleashed by transforming a collection of such smart objects into an open platform for applications that can engage the sensing and actuating resources of individual objects in a combined way. Independent developers would then be able to produce pervasive computing applications that people can acquire at any time and run on top of the smart object collection available in their homes.

To realize this vision, one must deal with programming-level issues stemming from the inherent distribution and heterogeneity of such a system. However, especially for the home domain, it is equally important to consider the user-level issues of application deployment and management. Regarding the latter, the current state of things leaves a lot to be desired. To begin with, the user typically has to go through the motions of installing application software on some computer-like device, which is already cumbersome and discouraging for most people. Then the application has to be started, stopped, and monitored via some graphical user interface (GUI) on a computer screen; that is quite disruptive and annoying in the context of everyday activities in the home. The big question is whether a pervasive computing application could take a form that “weaves itself into the fabric of everyday life” [1], instead of being an immaterial software artifact that can be accessed only with a general-purpose computing device.

Furthermore, while the application relies on sensors and actuators to provide its functionality, each home is populated with different objects, which makes it rather unlikely for the application's resource requirements to be fully satisfied in all cases. To enable deployment in different homes, the application should operate in a resource-flexible way, by allowing graceful degradation of its functionality and continuing to work even if some requested sensors and actuators are missing. But while such resource-flexibility is indeed highly desirable, it also gives rise to other issues: how to capture the degree to which the application can deliver its functionality in the home where it is deployed, and how to communicate this information to the user.

Our work tackles the above challenges, making the following contributions. First, we introduce the concept of *reified, graspable* applications. The idea is for the pervasive computing application to take the form of a small physical thing, the "application pill," which is as simple to handle as a flashlight: pick it up, switch it on, and confirm that it works just by glancing at it. Next, to account for resource flexibility, we propose that the user be provided with the application's *functionality level*, a single number that captures how well the application is expected to work on top of objects available in the user's home. The functionality level is displayed by the application pill itself, in the spirit of the familiar signal strength indicator found on mobile phones. Finally, we discuss how the above concepts have been implemented in POBICOS, a pervasive computing platform for cooperating home objects.

This article is organized as follows. First, we give a brief overview of related work. Next, we present our vision of the home as an open platform for resource-flexible pervasive computing applications, and introduce the concept of reified, graspable applications in this context. Then we describe a proof-of-concept implementation of graspable, resource-flexible applications, and the model for capturing their functionality level in a structured, tree-based way. Finally, we conclude the article.

II. RELATED WORK

Our approach is related to the idea of tangible user interfaces [2], where GUIs on computer screens are replaced by objects in the real world, thereby "giving physical form to digital information." Work in this area has focused mainly on the interaction between the user and the application to achieve more natural and efficient forms of input/output, without involving a keyboard, mouse, and screen. For example, in the Urban Planning Workbench [2], the user manually rotates the hands of a clock to set the time of day (and, indirectly, the "position" of the Sun) or moves model buildings, and then observes the computed shadows "cast" by these buildings. However, despite the advances in this field, the application *itself* is still treated as a typical software artifact. We take the concept of tangible interfaces a step further, by applying the very same principle not to input/output, but to the task of application deployment and management. The application is reified as a physically distinct, graspable object (the application pill), which the user can literally bring home, start/stop, and see if it is working, in much the same way as he would do with a flashlight.

Graspable applications are related to information appliances and appliance-centric computing [3]. Like traditional appliances, information appliances are single-function devices with simple dedicated controls. The most minimalistic MP3 players are an excellent example, also in terms of the form factor we envision for the application pill. As noted in [4], easy-to-use information appliances can be a way to introduce ubiquitous computing to the home, without burdening the user with unwelcome system administration chores. The application pill shares these properties of an information appliance, so it could be considered as a new genre of the latter. The main difference is that, unlike an information appliance, the pill lacks the sensor and actuator resources needed to deliver the application functionality; the pill is resourceless in this respect and cannot work in a standalone mode. Instead, the application pill interacts with regular objects found in the home and takes advantage of their sensors and actuators.

To relieve the programmer from having to deal with the unknown (at design time) availability of resources, a lot of work has been done on enabling the end user to produce simple functionality by herself. For example, in [5, 6], guided by the system, the user can browse the devices and services in her home, and form connections between them in order to establish the desired data flow or I/O. However, this requires quite a lot of involvement on the part of the user, who becomes responsible for

composing the applications that will run in her home. In contrast, we wish for pervasive computing applications to be developed by professionals and brought into the home as ready-to-use tangible objects that exploit available sensors and actuators with hardly any configuration. The applications should be able to provide certain (possibly reduced) value, even if some requested sensors and actuators are missing, as argued in [7] for what the authors call "partial installations."

In our proof-of-concept realization of the functionality level, we assume tree-structured applications and a corresponding recursive formula. We share such a tree-based approach with a number of works; indicative examples include [8, 9]. Most closely related to our approach is [8], where the programmer provides code for calculating the so-called satisfaction metric. Contrary to that, our functionality level is derived by the middleware, based on programmer-supplied hints (declarations). It is harder to devise formulas than high-level declarations; also, while code is more expressive, in either case there is inevitable simplification involved in summarizing a complex phenomenon with a scalar. Our approach also bears some resemblance to the monitoring of the non-functional parameters in PCOM [9]. There, when some resources are missing, and the component can no longer fulfill its contractual obligations, it is disqualified. In our approach, in the spirit of resource flexibility, there are no such binary decisions. More important, both [8, 9] focus on adaptive applications, which means that poorly performing component subtrees have to be replaced with better ones, according to what resources are available. In that sense, the application tree is always "complete." The OPPORTUNITY framework [10] follows a similar adaptive approach in order to support flexible context and activity recognition using different sensor configurations. While we do not dismiss adaptation (in fact, our approach might be integrated with tree-based adaptation), we focus on applications that continue to work and offer some (reduced) functionality, even if, due to resource unavailability, the application tree is only partially instantiated.

III. ECOSYSTEM FOR RESOURCE-FLEXIBLE PERVASIVE COMPUTING AT HOME

Our vision of pervasive computing for the domestic environment draws from an ecosystem of three basic stakeholders, each playing a different role in how smart objects and applications enter the home (Fig. 1a). Serving as an open reference for the home domain, the sensor/actuator application programming interface (API) captures relevant sensing and actuation aspects in the form of corresponding primitives.[1] Object manufacturers conceive and produce smart objects, which, in addition to their regular functionality, expose their sensors and actuators through appropriate primitives of this API. On the other hand, application developers pick the primitives needed to implement the sensing and actuation components of the application. The processes of object and application development are decoupled: the object manufacturer does not know which applications will exploit the object's sensing and actuation resources, while applications are written without knowing which objects will provide the required sensors and actuators.

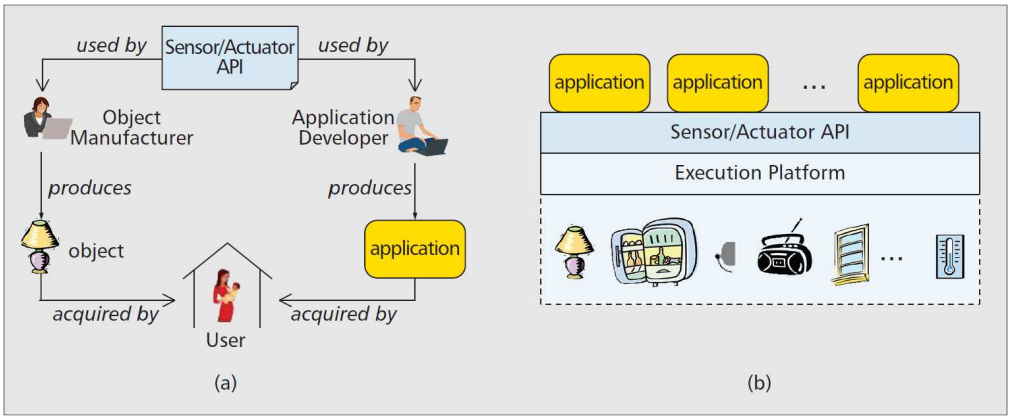

Figure 1. a) Key stakeholders involved in the introduction of smart objects and applications in the home; b) at home, objects with different sensors and actuators jointly form a platform for the execution of applications.

[1] The design of such an API is not relevant to the concepts and mechanisms presented in this article, and thus is not elaborated on here.

The user purchases objects for their regular functionality. At home, objects jointly form a platform for the execution of pervasive computing applications (Fig. 1b). The platform emerges as a *side-effect* of populating the home with objects that are useful for its inhabitants as individual entities, without consideration for the applications that might be deployed in the home; this corresponds to what the authors of [4] refer to as an "accidentally formed domestic computing environment." In a similar vein, the user acquires pervasive computing applications for the functionality they are supposed to offer, but without knowing how well the object collection in his home can actually support these applications. Of course, these applications cannot be life critical as there is no guarantee that they will work to their full extent (or at all). One should rather think of them as a bonus to be enjoyed on top of the regular functionalities of the home objects.

For the programmer, this "unplanned" acquisition of objects and applications amounts to a challenge, as is it is unlikely for any two homes to feature the same object collection. This practically means that the objects (sensors and actuators) of the target platform are unknown at the time when the application is written. Therefore, if the application is to be of value across a wide range of homes, it must be designed to work in a resource-flexible way. In this article, by *resource flexibility* we understand that the application should have the ability to function (instead of quitting) even if it is lacking some sensors and actuators.

However, resource flexibility comes with uncertainty about how well the application will actually work in a home. In fact, it could be that some tasks of the application cannot be performed at all, even if the applications adapt to different sensor and actuator configurations. As a result, the application may be able to deliver its functionality only partially. If so, the application should inform the user about the functionality it *can* deliver, based on the sensors and actuators found in the home where it is deployed. This is crucial for setting the right expectations, and for allowing the user to identify and dismiss applications that cannot function well enough.

IV. REIFYING RESOURCE-FLEXIBLE APPLICATIONS

The deployment and management of pervasive computing applications in the home should be intuitive and relatively effortless for the user. We let ourselves be inspired from the way people interact with simple everyday things. We also take into account the functionality feedback that needs to be provided by resource-flexible applications.

THE CONCEPT OF A GRASPABLE APPLICATION

The prevalent approach is to treat pervasive computing applications as conventional programs: immaterial artifacts that have to be downloaded, installed, started/stopped, and inspected using a computer-like device. This view corresponds to what the author of [2] refers to as "bits" in cyberspace, which are exposed to and manipulated by the user through a GUI that "tied down as it is to the screen, windows, mouse and keyboard, is utterly divorced from the way interaction takes place in the physical world." Contrary to this approach, to use again the metaphors of [2], we believe it is convenient for applications to be "atoms" in the real world so that they can be handled using naturally developed skills and practices for manipulating physical objects.

In the spirit of ubiquitous computing [1], we advocate that the best way for the application to "disappear" is (somewhat paradoxically) to reify it as a regular object that blends into the domestic environment. People should perceive and use applications as they do other simple things in the home, relieved from the alien and complex world of computer systems. As a yardstick, think how easy it is for anyone to operate a flashlight: it only has an on/off switch, and it is trivial to check if it works, just by glancing at it. We wish this to hold for pervasive computing applications as well.

To achieve this goal, we propose the concept of the graspable application: a small physical artifact that embodies a *single* application and features an absolutely minimal interface (one that does not employ any general-purpose user interface elements) for controlling and monitoring its operation. The point is for the user to conceptually identify the application itself with a physical object, which can be grasped and manipulated as easily as the flashlight.

THE APPLICATION PILL: A GRASPABLE APPLICATION FOR RESOURCE-FLEXIBLE COMPUTING

This subsection presents our "canonical" design of a graspable application object, in the context of the ecosystem described earlier. To guide the design, we propose the following requirements:

- (R1) The application object should provide a means for the user to turn the application on/off and to see whether the application is turned on.
- (R2) The application object should offer feedback on the functionality that the resource-flexible application can deliver on top of the objects (sensors and actuators) available in a home. The feedback should be high-level, consciously sacrificing precision for simplicity; this is acceptable, given that the applications we have in mind are non-critical.
- (R3) Optionally, the object can offer "tangible" controls for setting application parameters.
- (R4) Finally, it should carry the application's code and possibly provide a runtime for executing the code (this requirement has no impact on the user interface design of the application object).

Based on these requirements, we envision the graspable application as a small object, roughly the size of a matchbox, with a simple user interface, as shown in Fig. 2. Concerning (R1), the object has a pushbutton to start/stop the application and a diode to show whether the application is running. As to (R2), our approach is to capture the functionality that can actually be provided by the application as a *fraction* of the functionality it would be able to provide if it had at its disposal all the sensors and actuators it requests from the underlying object community; we refer to this as the *functionality level* of the application. The functionality level is a scalar, which can be intuitively displayed via a simple gauge, like the ones used to show the signal strength of wireless devices. Such an indication is meaningful irrespective of the actual functionality of the application in question: if the application can operate at a mere 10 percent of its potential, it is probably not worth running in one's home, whereas if it can deliver 75 percent of its intended functionality, it will most likely be useful. Finally, as to (R3), the application object may feature one or more knobs, each for setting a single parameter (e.g., the setpoint for a temperature control application). Obviously, due to limited real estate, application designers should introduce as few such parameters as possible.

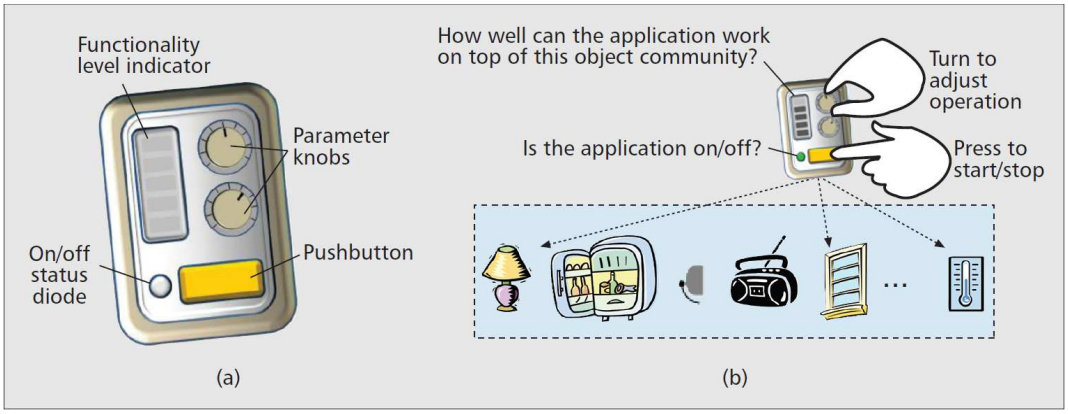

Figure 2. The "canonical" application pill design (objects are not shown to scale): a) the application pill object, featuring a minimal interface to control and monitor the resource-flexible application; b) the application is started/stopped by pressing/depressing the button; the diode shows the on/off status of the application; the knobs are used to set application parameters; the linear indicator shows the functionality level of the application.

To stress that such a graspable application object should be truly minimal (in terms of both size and interface), and to hint that it carries the application's code, we call it the *application pill*. Application pills could be marketed in numerous ways, for example, sold in stores with home appliances and furniture, or bundled with home improvement magazines. A pill could be accompanied by a one-page manual describing what the application will do for the user at different values of the functionality level.

From the user's perspective, the application pill *is* the application. The pill can be casually placed anywhere in the home (Fig. 3). The on/off status and functionality level of the application can be

observed at a glance, and the application parameters can be set by turning the knobs, both without having to resort to a computer screen or another gadget. If the user decides (perhaps after consulting the pill's manual) that the functionality level is too low to be of value to her, she can turn the application off by depressing the button of the pill.

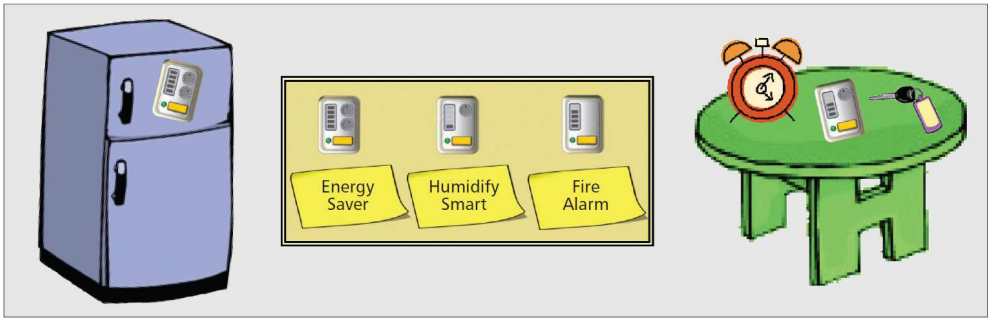

Figure 3. Application pills can be put at various convenient places in the home (objects are not shown to scale).

Of course, one can imagine several variations of our "canonical" application pill design. For instance, the pill could inform the user about changes in the functionality level in some attention-grabbing way (e.g., via sound or vibration). Alternatively, the pill could be designed as an application-dedicated ambient display [11]; one option is for the color of the entire pill to change to reflect the application's functionality level. A concrete example along these lines is provided in the next section, where we discuss our proof-of-concept implementation.

It is important to note that the application pill should not be transformed into yet another attention-hungry computer-like device. If the application has to support complex input/output functions, these should be delegated to other devices with a powerful interface that are likely to be found in the home, such as TVs, computers, and smart phones. More details about how we envision the configuration and input/output of pervasive computing applications to be performed via regular objects can be found in [12].

V. PROOF-OF-CONCEPT: THE POBICOS PLATFORM

In this section, we describe how the concept of graspable and resource-flexible applications is implemented in the FP7 project POBICOS. POBICOS researched system support for opportunistic monitoring and control applications on top of an ad hoc community of cooperating home objects. It delivered a complete platform along the lines of the computing ecosystem described earlier, including a programming model, a middleware prototype, and an application/object development toolchain. POBICOS was demonstrated with a demand-response energy-saving application developed by SAE-Automation (Slovakia), and showcased at Accenture Technology Labs (France) and the Center of Renewable Energy Resources (Greece).

POBICOS applications consist of lightweight mobile components, which are written in C and compiled into a binary for the POBICOS virtual machine (VM) [13]. The middleware provides the VM-based runtime environment and implements the underlying mechanisms for secure object community formation [14], as well as component communication and mobility [15]. The POBICOS middleware runs on Imote2 nodes from Memsic.[2] The implementation on top of TinyOS 2.1 takes 289 kbytes of code memory and 225 kbytes of RAM (without any optimization). Wireless communication between nodes is achieved through a Z430-RF2480 ZigBee modem from TI,[3] attached to the Imote2 via an external board. Simple objects, like motion detectors, lamps, and an alarm, are connected to the Imote2 via the GPIO interface. More complex objects, such as a TV set-top box, a fan, and an air conditioner, are controlled via custom-made RS232 adapters and power plugs.

[2]http://www.memsic.com/support/documentation/wireless-sensor-networks/category/7-datasheets.html?download=139%3Aimote2-multimedia
[3]http://focus.ti.com/docs/toolsw/folders/print/ez430-rf2480.html

THE POBICOS APPLICATION PILL PROTOTYPE

The POBICOS application pill is a spherical object, with a pushbutton on the side and a monochrome diode on the top (Fig. 4a). Internally, the pill contains an Imote2 node (Fig. 4b) that stores the application binary and hosts the POBICOS middleware on top of which the application executes.

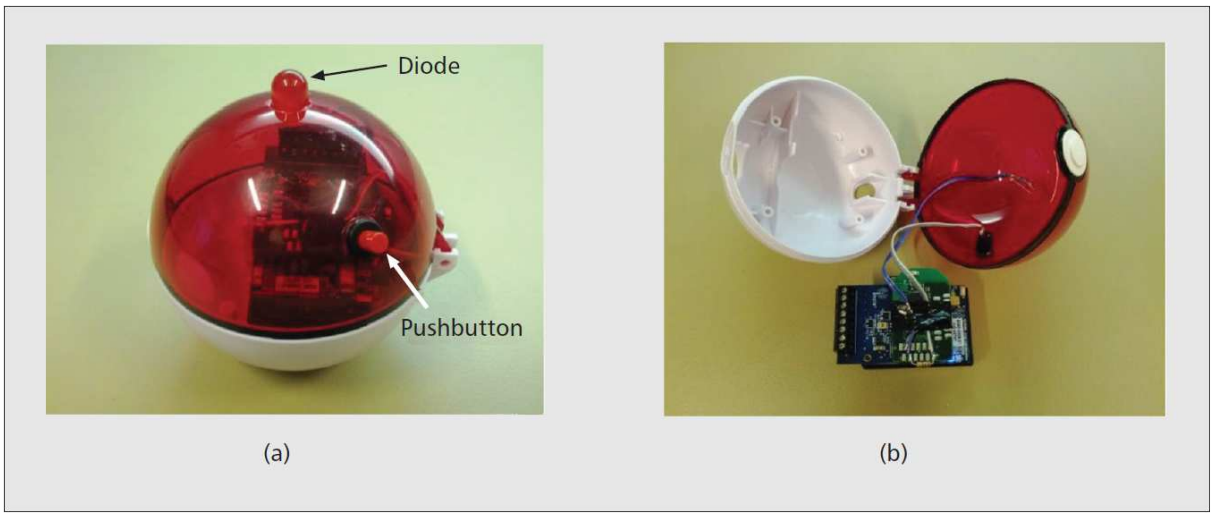

Figure 4. The POBICOS application pill: a) the user view: a pushbutton for starting/stopping the application, and a diode that shows the on/off status as well as the functionality level of the application; b) the internal view: the Imote2 (with the application binary and the POBICOS middleware) wired to the pill's button and diode.

The POBICOS pill is a variation of the "canonical" design presented earlier. The pushbutton plays the same role: the application is started and stopped by pressing and depressing it, respectively. However, the diode serves a dual purpose: to indicate both the application's on/off status and its functionality level (there is no linear indicator for this): if the application is off, the diode is off too; if the application is on, the diode blinks slowly when the functionality level is high, and faster as the functionality level drops (to catch the user's attention); if the application encounters a fatal problem or crashes, the diode remains constantly on. To keep the implementation simple, the pill does not have any parameter knobs; instead, POBICOS applications use default values for their parameters.

RESOURCE-FLEXIBLE APPLICATIONS IN POBICOS

POBICOS applications are structured as a set of cooperating components, called agents. Following the approach of hierarchical control systems [16], agents are organized in a tree. The edges of the tree represent parent-child relationships as well as the communication channels for exchanging messages. Leaf agents interact with the physical environment by acquiring information or effecting change via the sensors and actuators of smart objects; these agents are referred to as *non-generic* because they rely on specific sensing and actuating primitives which are supported only by the objects that feature the corresponding sensors and actuators. Non-leaf agents implement higher-level aggregation, processing, and control tasks, using general-purpose primitives supported by all objects; these agents are called *generic*.

At runtime, the development of the tree is guided by the application logic. It starts with the root agent, which uses the agent creation middleware primitive to create its children. Those, in turn, create their own children, and so on. The process continues until leaf agents are created. Applications can be programmed to change the agent tree at any time during execution. Here, though, we focus on the simpler case where the desired functionality can be implemented using a fixed tree structure.

Each agent runs on a specific object; the agent-to-object mapping is maintained by the middleware. For a generic agent, a creation request results in one new agent instance, placed on an

object irrelevant to the programmer. The non-generic agent uses programmer-selected sensing and actuating primitives, and thus can run only on matching objects: those that support these primitives (have suitable sensors and actuators). Non-generic agent creation requests can be made in two modes: single or multiple. The single mode results in one new instance, placed on a single matching object that is picked by the middleware. The multiple mode results in an instance placed on *every* matching object. If there are no matching objects, no instance is created, and the application tree does not develop into the complete tree envisioned by the programmer. Notably, the POBICOS middleware allows the sharing of sensors, but imposes a priority-based mutual exclusion policy on actuators. As a consequence, the creation of a non-generic actuating agent may also fail because all matching objects are already occupied by other actuating agents.

To give an example, Fig. 5 shows the tree of the *HumidifySmart* application, which prevents wasteful operation of point-of-use humidifiers (portable one-room units). For each room in the house, the root agent (ROOT) creates a generic room humidity manager agent (MGR). In turn, each MGR creates for its room a human presence agent (HPR), a window sensing agent (WIN), and a humidifier control agent (HUM). The HPR agent creates human activity agents (ACTs). WIN, HUM, and ACT are non-generic. Matching objects for WIN and HUM are a window with an open/close sensor and a humidifier that can be controlled programmatically, respectively. A matching object for ACT is any one that can detect a human-generated event (e.g., opening a refrigerator door, flushing a toilet, or switching a lamp on). WIN and HUM are created in the single mode (the application assumes up to one window and one humidifier per room), while ACT is created in the multiple mode (to detect as many human-generated events as possible).

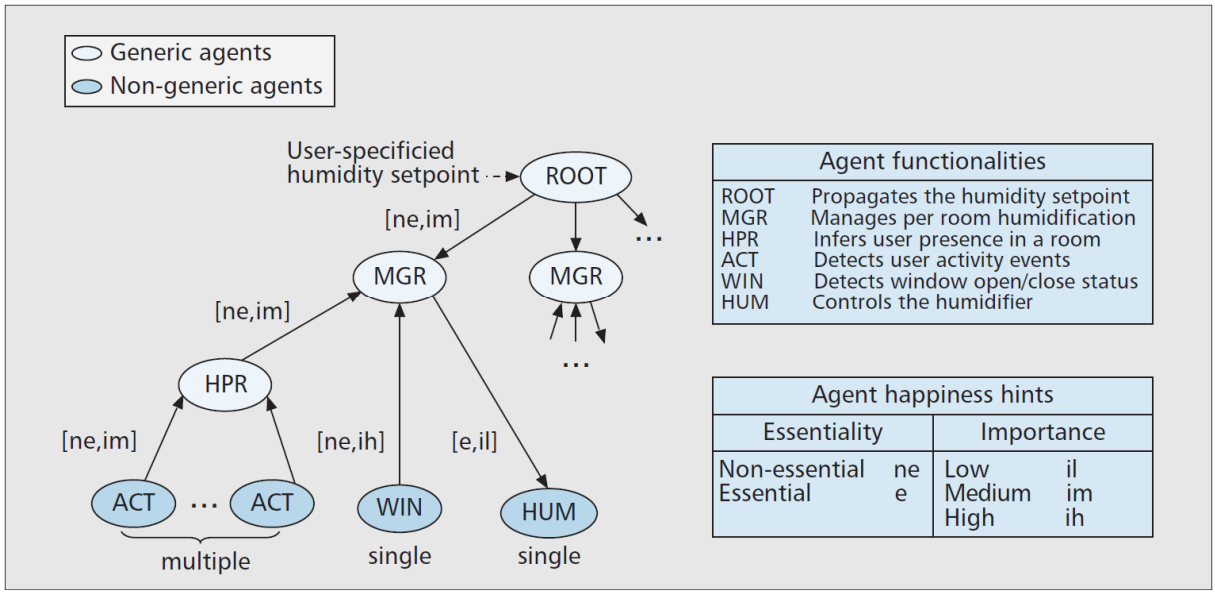

Figure 5. The agent tree of the HumidifySmart application. The direction of the edges reflects the flow of information between agents. The programmer-supplied happiness hints for the children of each agent are shown in brackets.

HumidifySmart works as follows. The HPR agent infers user presence/absence in its room, depending on whether human-generated events are reported by the ACT agents. Based on the presence information from HPR and the status of the window reported by WIN, the MGR agent produces a command for the humidifier. If nobody is in the room or the window is open, the command is to deactivate the humidifier to avoid wasteful operation. Otherwise, the command contains the humidity setpoint, which MGR agents receive from ROOT when the user changes it via a parameter knob on the Humidify Smart application pill. Finally, HUM accepts the command from MGR and controls the humidistat accordingly.

As noted earlier, the agent tree may not develop fully. The root cause for a partial instantiation of the application tree is a lack of objects with sensors and actuators needed by some non-generic agents. More specifically:

- If, for a given non-generic agent, there are no matching objects, no instance will be created, no matter what the creation mode.
- If there are few matching objects, the number of instances created in the multiple mode may be significantly less than expected by the programmer.
- Furthermore, if some non-generic agents are missing, their (generic) parent may not be able to deliver any value, and should quit (in which case, the entire subtree disappears as well). This, in turn, may render the parent of that generic agent useless as well, and so on.

In fact, given that the target collection of objects is unknown at application development time, a lack of some sensor and actuator resources and thus a partial instantiation of the application tree are more likely than not.

The key feature of a resource-flexible application is that it continues to run and deliver useful functionality, even if some required sensor and actuator resources are missing, and the tree is instantiated only partially. For example, HumidifySmart can be programmed to work in a resource-flexible way, as follows. The quality of inferences of the human presence agent, HPR, is high when there are many ACT agents: the more objects can detect human activity, the better. However, even if there are very few ACTs, or just a single one, HPR may still provide some (less accurate) presence information. Only if there are no ACT agents at all should HPR quit. Now consider the room manager, MGR. It can deliver value even if there is no HPR or no WIN agent. If HPR quits but WIN exists, and it is known when the window is open, humidification of outside air is avoided (although an empty room may be unnecessarily humidified). If there is no WIN, but HPR works, humidification is made human-presence-sensitive (although one may wastefully humidify outside air). Even without either HPR or WIN, the MGR agent can deliver some value; ROOT can use such MGRs to forward the user-specified humidity setpoint to all the humidifiers in the home. Only if there is no humidifier in the room (and hence no HUM) is the MGR agent unable to do anything useful and should quit. Finally, ROOT can deliver value (to the user) as long as at least one MGR agent exists.

AGENT HAPPINESS: CAPTURING THE FUNCTIONALITY LEVEL OF RESOURCE-FLEXIBLE APPLICATIONS

Clearly, the availability of sensor and actuator resources needed by the application determines the instantiation of the application tree, and thus affects the functionality that can be provided to the end user. Now, imagine that one could capture, with a single number, the degree to which the application's sensor and actuator requirements are satisfied in a given home. Such a resource satisfaction degree could then be interpreted as an indicator of the functionality that can be expected of the application in that home. To make this interpretation legitimate, individual resources should contribute to the resource satisfaction degree according to how much they contribute to the functionality.

To work out the above idea, we introduce a model based on *needs* and *happiness* (we use some psychological analogies). Both are defined for generic agents only. The happiness level of a generic agent *A*, *h(A)*, captures the availability of underlying sensors and actuators, but is interpreted as an indication of how well *A* is able to serve its "customer," that is, the parent agent or the end user (if *A* is the root); the agent is happy if it can perform well. Accordingly, the functionality level displayed on the application pill is equal to the happiness of the root agent, *h(ROOT)*. Needs, in turn, refer to the agent's children — to perform well, the agent needs its children to be available and perform well themselves. The model ties an agent's happiness to its needs. The happiness model is as follows:

- The agent has a collection of needs, each of which can be satisfied to a degree, from not being satisfied at all to being satisfied perfectly.
- Each need has a level of *importance* (low, medium, or high), which reflects how much a need, if satisfied, positively contributes to the agent's happiness.
- Each need is either *essential* or non-essential; while importance is about positive contribution to the level of happiness, being essential states a prerequisite, which, if missing, renders the agent unhappy (the agent cannot be happy if any of its essential needs is satisfied to a low degree).

- The agent's happiness level is obtained, using a "happiness formula," from the degrees to which individual needs are satisfied, as well as from programmer-supplied declarations, called *happiness hints*, on the importance and essentiality of those needs.

Both happiness levels and degrees of need satisfaction take on values in [0,1] with 0 denoting total unhappiness and the complete lack of need satisfaction, and 1 denoting perfect happiness and perfect need satisfaction. Let *B* be a child of a generic agent *A*. If *B* is generic, the degree d_B to which the respective need of *A* is satisfied equals (recursively) the happiness of *B*, $d_B = h(B)$. If *B* could not do anything useful and no longer exists (has quit), $d_B = 0$. Now assume that *B* is non-generic, that is, it requires a matching object (one with the specific sensors or actuators) to exist. If *B* is created in the single mode, the need is either perfectly satisfied ($d_B = 1$) or completely unsatisfied ($d_B = 0$), depending on whether the instance of *B* has been created. For the multiple mode, the need satisfaction is an increasing function of the number *k* of instances created (the more, the better). For example, one can capture this relationship with $d_B(k) = 1 - 2^{-k/K_{1/2}}$, where $d_B = 0$ if no instance is created ($k = 0$), $d_B = 0.5$ for $k = K_{1/2}$, and d_B approaches 1 as *k* increases further.

Next we cover the so-called additive version of the happiness formula (the full formula is more elaborate). Assume a generic agent *A* has *N* needs, *M* of which are essential. Let $d_1, d_2, \ldots, d_N$ be the degrees to which these needs are satisfied, $p_1, p_2, \ldots, p_N$ be values representing the importance of the needs (say, 1 for low, 2 for medium, and 3 for high), and $i_1, i_2, \ldots, i_M$ be the indices of the essential needs. Then the happiness level of *A* is given by

$$h(A) = \min\left(\sum_{n=1}^{N} w_n d_n, S(d_{i_1}), S(d_{i_2}), \ldots, S(d_{i_M})\right)$$

Above, $w_n = p_n / (p_1 + p_2 + \ldots + p_N)$ is the importance weight of the *n*th need, and $S : [0,1] \rightarrow [0,1]$ is an *essentiality modeling function*, which returns a small value if an essential need is poorly satisfied (to drive the happiness level low); otherwise, its value should not affect the happiness level. Currently, we let $S(x) = x$ if $x \leq x_{poor}$ for some x_{poor}, and $S(x) = 1$ otherwise.

A couple of notes about this model are in order. First, the happiness level of any agent is ultimately driven by the existence of non-generic agents (for the multiple mode, by the number of created instances). These exist if and only if there are matching objects, that is, objects which support the sensing and actuating primitives used by the agents (via suitable sensors and actuators). Thus, happiness levels indeed capture sensor and actuator resource availability. Second, the above formulas take into account the partial instantiation of the application tree, as described earlier: a possible lack of a non-generic agent, a low number of non-generic agents resulting from the multiple mode, and the fact that a generic agent will quit if unable to deliver any value. Third, happiness hints represent the programmer's judgment as to how resource availability affects (directly or indirectly) the functionality delivered by agents and the application as a whole. This is why it is justified to interpret happiness levels in terms of functionality. Finally, the provision of happiness hints (with a dedicated middleware primitive) is the only overhead imposed on the programmer. Otherwise, the happiness scheme is transparently implemented by the middleware.

DEPLOYMENT CASE STUDY: HUMIDIFYSMART

Consider again the HumidifySmart application. The happiness hints supplied by the programmer appear next to the edges of the application tree (Fig. 5). As to essentiality, MGR cannot affect humidification without a humidifier, so HUM is essential. Actually, this is the only essential need in HumidifySmart. For example, as explained earlier, MGR can provide some functionality even without WIN (window sensing) or HPR (human presence sensing), so these agents are non-essential. In fact, having non-essential needs is a *prerequisite* for resource-flexibility. As to importance, WIN is highly important to MGR, whereas HPR is of medium importance. This reflects the programmer's judgment that avoidance of humidification while the window is open constitutes the key functionality of the application, more so than the presence-sensitive operation. Interestingly, while HUM is essential, it is of low importance. The reason is that the application is not about humidification as such, but about

making humidification smarter (the functionality achievable with humidifiers alone is very modest). This illustrates the fact that essentiality and importance are "orthogonal."

Based on the programmer's hints and the happiness formula, it is straightforward to calculate the happiness levels of the application's agents for any concrete case of sensor/actuator availability. For example, Fig. 6 shows two homes with respective instantiations of the HumidifySmart application tree. To make the tree instantiations easier to follow, all generic agents are shown to exist, even the totally unhappy ones, i.e., those that are unable to provide any value (in reality, such agents would quit and cease to exist, and their children would disappear as well). Of course, this does not affect the happiness values of their parents since having a totally unhappy child is equivalent to not having that child at all.

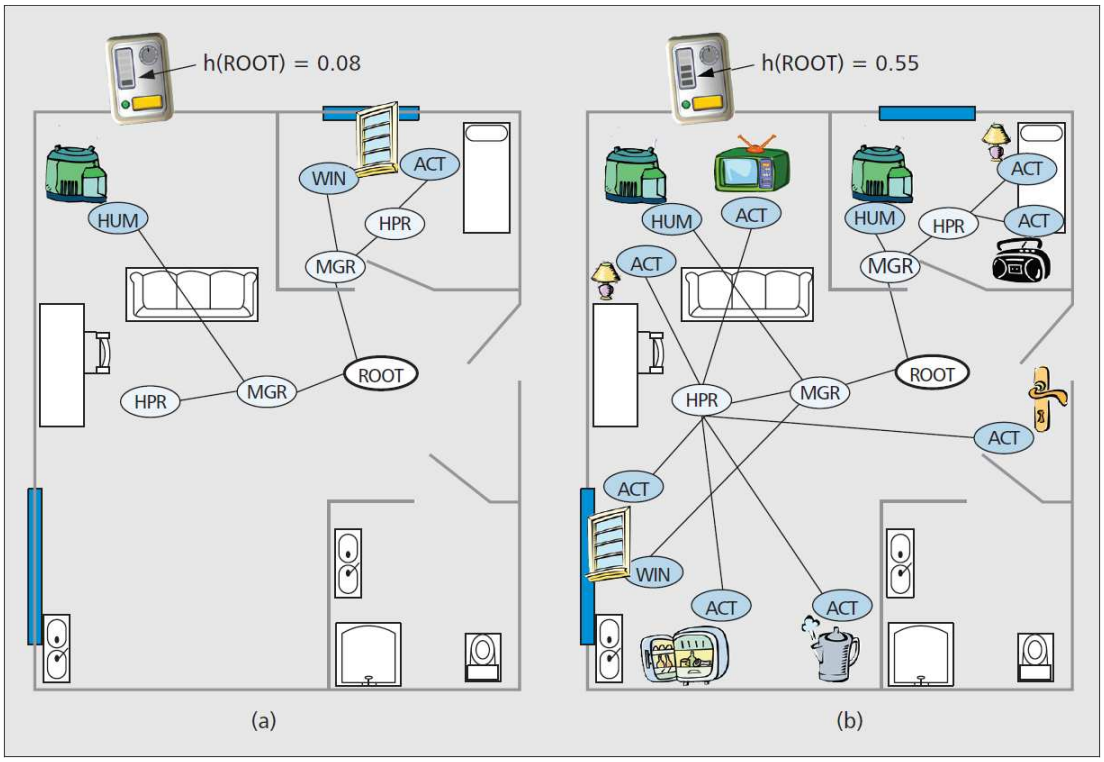

Figure 6. Deployment of HumidifySmart in two different homes (smart objects are colored). Non-generic agents ACT, WIN, and HUM are depicted next to the matching objects on which they were created. The ACT agents are placed on objects that can detect any human-generated event. Generic agents are shown as locationless, as their placement is irrelevant. The pill's indicator shows the functionality level of the application (the happiness level of the ROOT agent).

To calculate happiness, one first needs to observe if and how many non-generic agents exist. Table 1 provides a summary for each home. For instance, the bedroom in Fig. 6a has a sensor-equipped window (hence the instances of WIN and ACT), but not a humidifier (thus no HUM). On the other hand, the living room in Fig. 6b has a humidifier (the HUM agent), a sensor-equipped window (the WIN and ACT agents), and five more objects detecting human activity (the other five instances of ACT).

Table 1. Agent happiness for two indicative deployments of the HumidifySmart application.

Home	Room	**Existing non-generic agents**			**Happiness of generic agents**		
		HUM	**WIN**	**No. of ACTs**	***h*(HPR)**	***h*(MGR)**	***h*(ROOT)**
a)	Bedroom	No	Yes	1	0.13	0.00	0.08
	Living room	Yes	No	0	0.00	0.17	
b)	Bedroom	Yes	No	2	0.24	0.25	0.55
	Living room	Yes	Yes	6	0.56	0.85	

Table 1 also lists the corresponding happiness levels of all generic agents: HPR, MGR, and ROOT (to carry out these calculations, we let $K_{1/2} = 5$; the value of x_{poor} is irrelevant in this case). Note that the MGR agent in the bedroom of Fig. 6a is totally unhappy due to the lack of a humidifier (no HUM), even though WIN and ACT do exist. This is because HUM is essential for MGR. The MGR agent in the living room of Fig. 6a is somewhat happy (0.17) in spite of the fact that there are no WIN and no ACT agents; the (admittedly small) added value this MGR can still deliver is that the humidifier will operate according to the setpoint specified by the user for the entire home. Also, the four cases included in the table show how the happiness of an HPR agent increases with the number of its ACT agents. Finally, the happiness of ROOT is the mean of that of the room managers (MGRs) because all have the same (medium) importance and thus equal weight in the happiness formula. As shown in Fig. 6, the happiness of ROOT is displayed by the application pill as the functionality level.

VI. CONCLUSION

We propose the concept of the reified, graspable application, which can be acquired, placed, controlled, and monitored just like a simple everyday thing. By doing so, we extend the concept of tangible interfaces to the application as such. We use the common flashlight as our reference point for simplicity in the design of graspable applications. Given that such applications have to be resource-flexible, we place special emphasis on offering high-level feedback on the functionality that can be delivered based on the available sensors and actuators, so the user can assess, at a glance, how well the application can work in her home.

It is important to note that the POBICOS proof of concept is just an exemplification of the key idea of graspable and resource-flexible applications. An alternative approach is illustrated by our canonical pill design. Also, it is certainly possible to capture the application functionality level in a different way, as well as to extend the concept to non-tree-structured applications.

Even with our simple application pill prototype, it was very easy to deploy, start/stop, and monitor the functionality level of the demo applications that were developed in the course of POBICOS. Comprehensive experimentation with real users was beyond the scope of the project, but is an important direction for future work.

Finally, one should stress that, in general, the concept of the reified, graspable application is not inherently tied to resource flexibility (and the provisioning of the functionality level). Bundling the two makes good sense for our pervasive computing ecosystem. However, for different environments one could consider graspable applications that are not resource-flexible, as well as resource-flexible applications that are operated in a traditional way, via a general-purpose, computer-like device. Another direction for future work is to explore different uses of the above two concepts, taken separately.

ACKNOWLEDGMENT

This work was funded in part by the 7th Framework Program of the European Community, project POBICOS, contract no. FP7-ICT-223984. We would like to thank Manos Koutsoubelias for implementing the agent happiness scheme in the POBICOS middleware. We would also like to thank Markus Taumberger and Tomasz Tajmajer for contributions to the application pill design.

REFERENCES

[1] M. Weiser, “The Computer for the 21st Century,” *Sci. Amer*., Sept. 1991, pp. 94–104.
[2] H. Ishii, “Tangible Bits: Beyond Pixels,” *Proc. 2nd Int’l. Conf. Tangible and Embedded Interaction*, invited paper, 2008, pp. xv–xxv.
[3] D. A. Norman, “The Invisible Computer: Why Good Products Can Fail, the Personal Computer Is so Complex, and Information Appliances Are the Solution,” 1998,MIT Press,ISBN0-262-64041-4.

[4] W. K. Edwards and R. E. Grinter, "At Home with Ubiquitous Computing: Seven Challenges," *Proc. 3rd Int'l. Conf. Ubiquitous Computing*, 2001, pp. 256–72.
[5] J. Humble *et al.*, "Playing with the Bits: User-Configuration of Ubiquitous Domestic Environments," *Proc. 5th Int'l. Conf. Ubiquitous Computing*, 2003, pp. 256–63.
[6] P. Wisner and D. Kalofonos, "A Framework for End-User Programming of Smart Homes Using Mobile Devices," *Proc. 4th IEEE Consumer Commun. and Networking Conf.*, 2007, pp. 716–21.
[7] C. Beckmann, S. Consolvo, and A. LaMarca, "Some Assembly Required: Supporting End-User Sensor Installation in Domestic Ubiquitous Computing Environments," *Proc. 6th Int'l. Conf. Ubiquitous Computing*, 2004, pp. 107–24.
[8] J. M. Paluska *et al.*, "Structured Decomposition of Adaptive Applications," *Proc. 6th Annual IEEE Conf. Pervasive Computing and Commun.*, 2008, pp. 1–10.
[9] C. Becker *et al.*, "PCOM — A Component System for Pervasive Computing," *Proc. 2nd Annual IEEE Conf. Pervasive Computing and Commun.*, 2004, pp. 67–76.
[10] M. Kurz *et al.*, "The OPPORTUNITY Framework and Data Processing Ecosystem for Opportunistic Activity and Context Recognition," *Int'l. J. Sensors, Wireless Commun.and Control*, Special Issue on Autonomic and Opportunistic Commun., vol. 1, no. 2, 2011, pp. 102–25.
[11] Z. Pousman and J. Stasko, "A Taxonomy of Ambient Information Systems: Four Patterns of Design," *Proc. ACM Working Conf. Advanced Visual Interfaces*, 2006, pp. 67–74.
[12] S. Lalis *et al.*, "Tangible Applications for Regular Objects: An End-User Model for Pervasive Computing at Home," *Proc. 4th Int'l. Conf. Mobile Ubiquitous Computing, Systems, Services and Technologies*, 2010, pp. 385–90.
[13] A. Pruszkowski, T. Paczesny, and J. Domaszewicz, "From C to VM-targeted Executables: Techniques for Heterogeneous Sensor/Actuator Networks," *Proc. 8th IEEE Wksp. Intelligent Solutions in Embedded Systems*, 2010, pp. 61–66.
[14] P. Tarvainen *et al.*, "Towards a Lightweight Security Solution for User-Friendly Management of Distributed Sensor Networks," *Proc. 2nd Conf. Smart Spaces*, 2009, pp. 97–109.
[15] N. Tziritas *et al.*, "Middleware Mechanisms for Agent Mobility in Wireless Sensor and Actuator Networks," *Proc. 3rd Int'l. Conf. Sensor Systems and Software*, 2012, pp. 30–44.
[16] M. D. Mesarovic, "Multilevel Systems and Concepts in Process Control," *Proc. IEEE*, vol. 58, no. 1, 1970, pp. 111–25.

BIOGRAPHIES

JAROSLAW DOMASZEWICZ (domaszew@tele.pw.edu.pl) is an assistant professor at the Institute of Telecommunications, Faculty of Electronics and Information Technology, Warsaw University of Technology (WUT), Poland. His research interests include pervasive and mobile computing, middleware, programming models, and resource modeling. He is also experienced in embedded systems and real-time multiprocessors. He received his Ph.D. from Texas A&M University.

SPYROS LALIS (lalis@inf.uth.gr) is an associate professor at the Computer and Communication Engineering Department, University of Thessaly, Greece, and a research associate at the Institute for Research & Technology Thessaly of the Centre for Research and Technology Hellas. His research interests span programming environments, operating systems, distributed systems, and ubiquitous computing. He received his Ph.D. in computer science from ETH Zurich.

TOMASZ PACZESNY (t.paczesny@funandmobile.com) received his M.S. from WUT in 2008. Until 2012, he was a research assistant at the Institute of Telecommunications of WUT. Currently, he is a software architect at a mobile software development company, FUN and MOBILE Sp. z o.o. His research interest is mainly in mobile, context-aware, and ubiquitous systems and services.

ALEKSANDER PRUSZKOWSKI (apruszko@tele.pw.edu.pl) is a member of technical staff at the Institute of Telecommunications, WUT. His expertise areas are embedded systems, virtual machines for small platforms, intelligent home, and ubiquitous computing. He received his M.S. from WUT.

MIKKO ALA-LOUKO (mikko.ala-louko@vtt.fi) received his M.Sc. degree in electrical engineering from the University of Oulu, Finland, in 2010. He is currently working at VTT Technical Research Centre of Finland as a research scientist. His main research interests include distributed systems, wireless sensor networks, and security of resource-constrained systems.