International Journal of Multimedia and Ubiquitous Engineering
Vol. 4, No. 3, July, 2009

Smart Objects as Components of UbiComp Applications

Christos Goumopoulos[1] and Achilles Kameas[1,2]

[1]Research Academic Computer Technology Institute, DAISy group, 26500 Rion, Patras, Hellas
[2]Hellenic Open University, 23 Sahtouri Str., 26222, Patras, Hellas
{goumop, kameas}@cti.gr[1], kameas@eap.gr[2]

Abstract

This paper presents a component-oriented programming model and middleware support services for building ubiquitous computing (UbiComp) applications that are composed out of smart objects on demand. Applications are realized as graphs representing smart objects and their services binding. Services are provided through high-level abstractions called plugs with semantically rich interfaces that allow them to be discovered and invoked dynamically. Our middleware supports the integration of heterogeneous smart objects by implementing a high level interaction model suited to the end-user and providing dynamic discovery, synthesis and binding of services. In this way we can deploy UbiComp applications that adapt to the dynamics of an Ambient Intelligence (AmI) environment. A smart home application that employees everyday augmented objects is used to illustrate the approach. We give implementation details with an emphasis to the compositional aspects and provide a scalability analysis for the service discovery process.

Keywords: *Smart Object, UbiComp Application*

1. Introduction

One of the major technological trends is to embed sensing, communication, computation and actuation in physical artifacts leading to the creation of smart objects. Smart objects will be an important building block to bridge the gap between the physical and digital world by providing information about aspects of their physical environment. While systems of smart objects will need to build on emerging technologies such as RFID and wireless sensor networks, the envisioned ubiquity of smart objects raises important questions about the digital representation of physical artifacts, their cooperation paradigms, integration into backend infrastructures and the applications that will benefit and influence their design and development.

Up to now, the ways that an everyday object could be used and the tasks it could participate in have usually been determined by its shape. Smart objects overcome this limitation by producing descriptions of their properties, abilities and services in the digital space, thus becoming able to improve their functionality by participating in compositions, learning from usage, becoming adaptive and context aware. This ability improves object independence, as a smart object that acts as a service consumer may seek a service producer based on a service and not object description. The benefit of this compositional approach is adaptability and evolution: a component-based application can be reconfigured with low cost to meet new requirements. The

possibility to reuse devices for several purposes - not all accounted for during their design - opens possibilities for emergent uses of ubiquitous devices, whereby the emergence results from actual use.

In this paper we present a component-oriented programming model and a middleware for building UbiComp applications that are composed out of smart objects. Contrary to the majority of component-based models that have focused on software components with an emphasis to support the programmer our component model embraces a heterogeneous collection of artifacts in a way that is comprehensible even by end-users. By supporting the encapsulation of the internal structure of an artifact the proposed model provides the means for composition of an application, thus can be considered as a programming model, without having to access any code that implements the interface. To achieve this, composition tends to be as simple as possible and is assisted by visual editing tools. Following this programming model, the application programmer is presented with a layer of abstraction in which an artifact is represented as a graph node and its services as input/output plugs i.e., high-level abstractions with semantically rich interfaces that allow them to be discovered and invoked dynamically. Consequently, applications and tasks are realized as graphs representing smart objects and their services binding. Furthermore, our model is pervasive enabling the application to adapt to the dynamics of AmI environments by employing middleware support services.

Building UbiComp applications out of components is possible only in the context of a supporting component framework that acts as a middleware. Our middleware can be considered as a component framework that determines the interfaces that components share and the rules governing their composition. The middleware manages resources shared by artifacts and provides the underlying mechanisms that enable communication among them. Furthermore, our middleware is suited to the dynamics of AmI spaces because it supports the on demand integration of heterogeneous smart objects for the realization of an abstract task or the adaptation of user defined compositions of services by providing dynamic discovery, synthesis and binding of the required services.

The remainder of the paper is organized as follows. Section 2 introduces the principle of compose ability which enables the composition of UbiComp applications by combining the services offered by smart objects. The basic concepts of our programming model are outlined and the smart object structure encapsulated in the model is highlighted. In Section 3 we discuss how the concepts of the model are applied to compose a smart home application. Section 4 presents our middleware support services for building applications out of smart objects on demand. We give implementation details with an emphasis to the compositional aspects and in Section 5 provide a scalability analysis for the service discovery process. Related approaches and work are presented in Section 6. Section 7 concludes this paper by presenting final statements and future work.

2. Compose ability based on smart objects

An important decision to be made when designing smart objects is whether they will be able to function without any infrastructure support (i.e. they are simply tagged objects) and whether they advertise their physical properties. The decision can lead to the development on one hand of lightweight tagged objects that rely on a centralized server support or on the other of resource-rich and autonomous artifacts. We propose a paradigm that contains conceptual abstractions and a middleware that has to run on every object, in order to

treat objects as components of distributed applications composed of UbiComp services [1]. To support this approach, we have adapted basic concepts of component models.

A component in the UbiComp application domain is an artifact that can be independently developed and delivered as a unit, and that offers interfaces by which it can be connected, unchanged with other components to compose a larger system. In this regard, the process where UbiComp applications are composed out of complex collections of interacting artifacts may be viewed as having much in common with the process where system builders design and implement software systems out of components [2].

2.1. A model for component-based UbiComp applications

The basic concepts encapsulated in our model are summarized below.

Artifacts: An artifact is a tangible smart object which bears digitally expressed properties; usually it is an object or device augmented with sensors, actuators, processing and networking unit or a computational device that already has embedded some of the required hardware components. Software applications running on computational devices are also excessively considered to be artifacts. Figure 1 illustrates a few smart object prototypes that are used in our laboratory. The hardware and sensor modules as well as the wiring embedded in the corresponding objects are also visible.

(a) **(b)** **(c)**

Figure 1. Smart object prototypes: (a) a smart desk with weight and proximity sensors; (b) a smart chair with pressure sensors; and (c) a smart book with bending and luminosity sensors

Artifact compositions: As UbiComp technology matures, an increasing number of smart objects will be "virtual", in the sense that they will be composed from distributed objects on a service description basis. As every object acquires a digital representation based on its capabilities and properties and the services it offers or requests, then a smart object may attempt to locate other modules based on a service-based description. Two or more artifacts (simple or composite) can be combined in an artifact composition. Such compositions are the tangible bearers of UbiComp applications and are regarded as service compositions**;** their realization can be assisted by end-user tools.

Properties: Artifacts have properties, which collectively represent their physical characteristics, capabilities and services. A property is modeled as a function that either evaluates an artifact's state variable into a single value or triggers a reaction, typically involving an actuator. Some properties (i.e. physical characteristics, unique identifier) are artifact-specific, while others (i.e. services) may be not. For example, attributes like *color/shape/weight* represent properties that all physical objects possess. The service *light* may be offered by different objects. A property of an artifact composition is called

an *emergent* property. All of the artifacts properties are encapsulated in a *property schema* which can be send on request to other artifacts, or tools (e.g. during an artifact discovery).

Functional Schemas: An artifact is modeled in terms of a functional schema: $F = \{f_1, f_2 \dots f_n\}$, where each function f_i gives the value of an observed property *i* in time *t*. Functions in a functional schema can be as simple or complex is required to define the property. They may range from single sensor readings to rule-based formulas involving multiple properties, to first-order logic so that we can quantify over sets of artifacts and their properties.

State: The values for all property functions of an artifact at a given time are the state of the artifact. For an artifact *A*, the set $P(A) = \{(p_1, p_2 \dots p_n) | p_i = f_i(t)\}$ represents the state space of the artifact. Each member of the state vector represents a *state variable*. The concept of state is useful for reasoning about how things may change. Restrictions on the value domain of a state variable are then possible and can be defined as part of the application specification.

Transformation: A transformation is a transition from one state to another. A transformation happens either as a result of an internal event (i.e. a change in the state of a sensor) or after a change in the artifact's functional context (as it is propagated through the synapses of the artifact).

Plugs: Plugs represent the interface of an artifact. An interface consists of a set of operations that an artifact needs to access in its surrounding environment and a set of operations that the surrounding environment can access on the given artifact. Thus, plugs are characterized by their direction and data type. Plugs may be output (O) in case they manifest their corresponding property, input (I) in case they associate their property with data from other artifacts, or I/O when both happens. Plugs also have a certain data type, which can be either a semantically primitive one (e.g., integer, boolean, etc.), or a semantically rich one (e.g., image, sound etc.).

Synapses: Synapses are associations between two compatible plugs. In practice, synapses relate the functional schemas of two different artifacts. When a property of a source artifact changes, the new value is propagated through the synapse to the target artifact. The initial change of value caused by a state transition of the source artifact causes finally a state transition to the target artifact. In that way, synapses are a realization of the functional context of the artifact.

Constraints: A constraint is a restrictive property relating one or more artifacts. Constraints are used to specify precisely the application behaviour. We have identified two types of constraints: *atomic constraints* and *compositional constraints*.

The atomic constraints involve a single artifact and have the following form:

$A.p_i$ *relop* **c**, where:

- $A.p_i$ is a property of the artifact *A*;
- *relop* $\in \{==, \neq, \geq, >, \leq, <\}$;
- **c** is a constant value.

For example, an atomic constraint may specify that the light service must be provided by an artifact with at least 50 Lux luminosity.

The compositional constraints involve a set of artifacts and one or more of their properties and have the following two forms:

$A.p_i$ *relop* $B.p_j$, where:

- $A.p_i$ is a property of the artifact *A* and $B.p_j$ is a property of the artifact *B*;
- *relop* is defined as above;

F(*expression*) **over** *S relop* **c** [**where** *condition*], where:

- *F* ∈ {**sum**, **avg**, **min**, **max**, **count**, **all**, **any**} with the aggregation functions having the same semantics as in the SQL query language;
- *expression* is an arithmetic expression of artifact properties;
- *S* denotes a set of artifacts;
- [**where** *condition*] is an optional part that specifies whether an artifact in *S* contributes to the evaluation or not depending on the logical expression specified by the *condition*.

For example, to specify that the display service must be provided by the artifact with the minimum distance from a smart desk and that distance should be at most two meters we may give the following compositional constraint:

```
min(abs(x.distance-eDesk.distance)) over {all artifacts x with display service} ≤ 2
```

A formal description of the model presented above is discussed by the authors in [3].

2.2 Smart object structure

In a previous work we have described a methodology for creating smart objects, which are functionally autonomous, extrovert and composeable [4]. In this section we briefly describe the smart object structure which is encapsulated by the proposed model. A set of communicating smart objects form a distributed system whose nodes are permeated by the architecture illustrated in Figure 2. At the heart of the architecture lies the middleware software layer that supports the deployment of UbiComp applications by managing dynamically the logical communication channels (synapses) between the nodes of the distributed system. The I/O unit and connectivity layers administer the communication intricacies (e.g., commercial of the shelf sensor device communication protocols, etc.) in terms of the sensory and interaction communication views respectively of the system component.

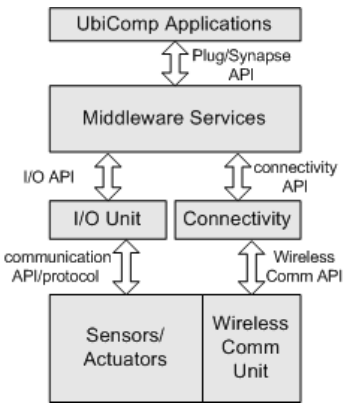

Figure 2. Smart object modular structure.

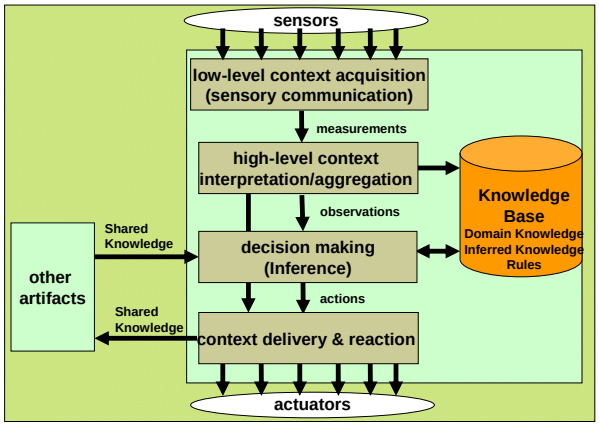

Figure 3. Smart object context management process.

Sensors together with the control circuitry are responsible for transforming measurements from the object's environment to observations (e.g. proximity, pressure, luminosity, etc.) in a digital form. Digital data are communicated to the computational unit using appropriate hardware interfaces (e.g., RS232, USB, Bluetooth, IrDA etc.).

The computational unit runs the required middleware software and based on the data received from sensors and actuators and possibly data received from other smart objects via the networking unit, performs a number of actions which can be manifested either to the environment via the object's actuators or to other objects via the networking module. Finally, the networking unit is a wired or wireless unit (e.g. 802.11) which facilitates the communication with other smart objects and end-user tools.

At a high level, the process performed by a smart object can be viewed as a context management process. We model this process as a measurement-reasoning-actuation control cycle (Figure 3). This logical view of an artifact's operation is comparable to a generic agent architecture [5]. Smart objects model their state on the world on the basis of their self representation (domain knowledge), observations of the world through sensors and sharing of knowledge with other smart objects. The Knowledge Base contains the domain knowledge of an artifact and dynamic knowledge about its situation in the environment. This knowledge is structured into facts and into rules. An inference component processes the knowledge of an artifact as well as knowledge provided by other artifacts either to infer further knowledge and/or to infer actions for the artifact to take in the environment.

3. An example application

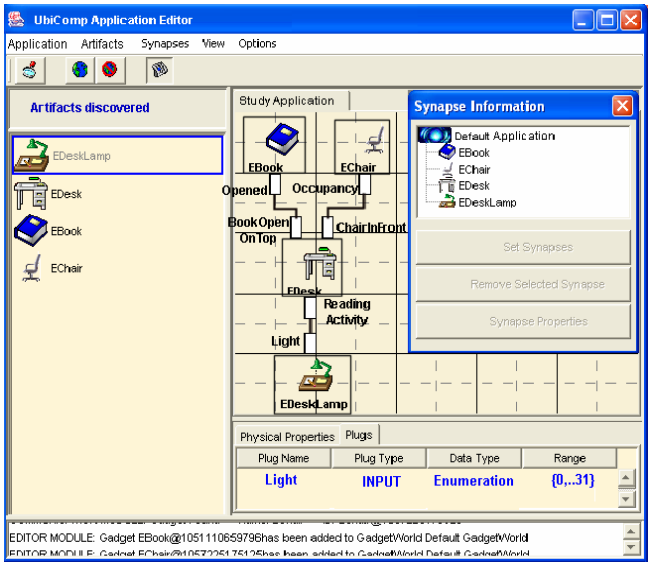

Figure 4. Combined artifacts in the UbiComp application editor.

The concepts presented in the previous section can be better illustrated with an example application. Pat is a 27-year old single woman, who lives in a small apartment near the city centre and studies Spanish literature. A few days ago she had given herself a very unusual present: a few furniture pieces and other devices that would turn her apartment into a smart one! The package included an *eDesk* (it could sense objects on top, proximity of a chair), an *eChair* (it could tell whether someone was sitting on it), a couple of *eLamps* (one could remotely turn them on and off), and some *eBook* tags (they could be attached to a book, tell whether a book is open or closed). Pat had asked the store employee to pre-configure some of

the artifacts, so that she could create a smart studying corner in her living room. Her idea was simple: when she sat on the chair and she would draw it near the desk and then open a book on it, then the study lamp would be switched on automatically. If she would close the book or stand up, then the light would go off.

The behavior requested by Pat requires the combined operation of the following set of artifacts: *eDesk*, *eChair*, *eDeskLamp* and *eBook*. The properties and plugs of these artifacts are shown in Table 1 and are manifested to Pat via the UbiComp Application editor tool, an end-user tool that acts as the mediator between the plug/synapse conceptual model and the actual system [6]. Using this tool Pat can combine the most appropriate plugs into functioning synapses as shown in Figure 4.

In the case of the synapse between *eDesk.ReadingActivity* and *eDeskLamp.Light* plugs, a data type compatibility issue arises. To make the synapse work, Pat can use the UbiComp Editor to define mappings that will make the two plugs collaborate.

Table 1. The properties, plugs and functional schemas of each artifact participating in the eStudy application example.

Artifact	**Properties**	**Plugs**	**Functional Schemas**
eChair	– Sensing chair occupancy (C_1) – Transmitting object type (C_2)	*Occupancy*: {OUT | Boolean}	*eChair.*C_1 ← read(pressure-sensor) *eChair.C2* is an attribute *Occupancy* ← {*eChair.*C_1, *eChair.C2*}
eBook	– Sensing open/ close (C_1) – Transmitting object type (C_2)	*Opened*: {OUT | Boolean}	*eBook.C1* ← read(bend-sensor) *eBook.C2* is an attribute *Opened* ← {*eBook.*C_1, *eBook.C2*}
eDesk	– Sensing objects on top (C_1) – Sensing proximity of objects (C_2)	*BookOpenOnTop*: {IN | Boolean} *ChairInFront*: {IN | Boolean} *ReadingActivity*: {OUT | Boolean}	*eDesk.C1* ← read(RFID-sensor) *eDesk.C2* ← read(proximity-sensor) **IF** *eDesk.C1* == *eBook.C2* **AND** *eBook.C1* == **TRUE** **THEN** *BookOpenOnTop* ← **TRUE** **ELSE** *BookOpenOnTop* ← **FALSE** **IF** *eDesk.C2* ==**TRUE AND** *eChair.*C_1 ==**TRUE** **THEN** *ChairInFront* ← **TRUE** **ESLE** *ChairInFront* ← **FALSE** **IF** *BookOpenOnTop* ==**TRUE AND** *ChairInFront* ==**TRUE** **THEN** *ReadingActivity* ← **TRUE** **ELSE** *ReadingActivity* ← **FALSE**
eDeskLamp	Light service ($S_L()$)	*Light*: {IN | Enumeration}	**IF** *eDesk.ReadingActivity* **THEN** S_L(ON) **ELSE** S_L(OFF)

The definition of the functional schemas of the artifacts, that is the internal logic that governs the behavior of each artifact either when its state changes or when a synapse is activated are predefined by the artifact developer. Rules that require identification of the remote artifact, can be specified using the property schema information which is available in the representation of each of the two artifacts that participate in a synapse.

The *eBook*, *eChair* and *eDesk* comprise an artifact composition whose emergent property is manifested via the *ReadingActivity* plug. This plug allows the connection of this composition to other artifacts or compositions. Any artifact composition can be edited to extend the functionality of the application. For example, consider that Pat also buys an *eClock* and wants to use it as a 2 hour reading notification. The *eClock* owns an alarm plug that when activated, via a synapse, counts the configurable number of hours and then rings the alarm. To implement her idea, what Pat has to do is to use the UbiComp Application editor to create a synapse between the *ReadingActivity* plug of the *eDesk* and the alarm plug of the *eClock* and specify the number of hours in the Properties dialog box of the *eClock*.

4. Middleware support

The idea of building UbiComp applications out of components is possible only in the context of a supporting component framework that acts as a middleware. The kernel of such a middleware is designed to support basic functionality such as accepting and dispatching messages, managing local hardware resources, affording the plug/synapse interoperability and providing a semantic service discovery protocol as will be explained in the following.

4.1. Kernel

To implement and test the concepts presented in the previous sections, we have developed a middleware layer that provides a uniform abstraction of artifact services and capabilities and shields the application programmer from the complexities of the underlying data communications and sensor/actuator access components of each distributed node. The middleware supports the features of our programming model and provides UbiComp application designers and developers with a runtime environment to build applications out of artifact components.

The outline of the middleware architecture is shown in Figure 5. The kernel is designed to have a low memory footprint and supports only accepting and dispatching messages, managing local hardware resources (sensors/actuators), performing context state evaluation, and implementing the plug/synapse interaction model. The kernel is also capable of managing service and artifact discovery and binding in order to facilitate the formation of the proper synapses in a dynamic way. Extending the functionality of the kernel can be achieved through plug-ins, which can be incorporated via a Plug-in Manager. Using ontologies, for example, and the Ontology Manager plug-in, all artifacts can use a commonly understood vocabulary of services and capabilities, in order to mask heterogeneity in context understanding and real-world models.

The *Process Manager* (*PM*) is the coordinator module and the main function of this module is to monitor and execute the reaction rules defined by the supported applications. These rules define how and when the infrastructure should react to changes in the environment. Furthermore, it is responsible for handling the memory resources of an artifact and caching information of other artifacts to improve communication performance when service discovery is required.

The *State Variable Manager* (*SVM*) handles the runtime storage of artifact's state variable values, reflecting both the hardware environment (sensors/actuators) at each particular moment and properties that are evaluated based on sensory data and P2P communicated data.

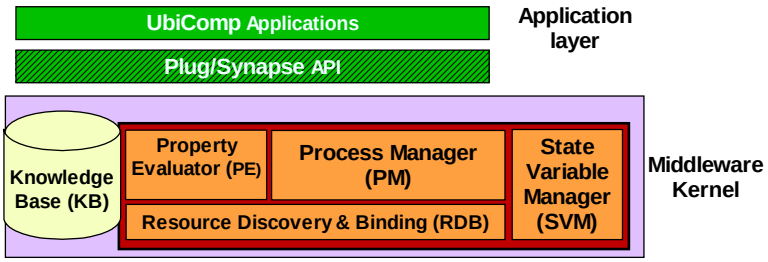

Figure 5. Middleware kernel architecture.

The *Property Evaluator* (*PE*) is responsible for the evaluation of artifact's properties according to its functional schema. In its typical form the PE is based on a set of rules stored in the artifact's *Knowledge Base* (*KB*) and an *Inference Engine* (*IE*) that govern artifact transition from one state to another. The IE sub-module processes the facts of an artifact as well as facts provided by other artifacts to infer further knowledge and to infer actions for the artifact to take. Note that the rules stored in an artifacts' rule base may only contain parameters, states and structural properties that are defined into the artifacts' private KB. For the initialisation of the context management process apart from the rules a set of initial facts are necessary. For example an initial fact may define the existence of an artifact by denoting its parameters, states and reactions that can participate in its rules and their initial values. In order to create such an initial fact the PE uses knowledge stored in the artifacts' KB.

The IE supports the decision-making process and it is based on a simple Prolog interpreter that uses backward-chaining with depth-first search as inference algorithm. Compromises in terms of expressiveness and generality were necessary to facilitate implementation on a micro-controller platform. For example, backtracking is only possible over local predicates. In order to initialise its process execution, the IE needs the artifact initial facts and the rules stored in the KB. The IE is informed for all the changes of parameters values measured by artifacts sensors. When it is informed for such a change it runs all the rules of the rule base. If a rule is activated, this module informs the PM for the activation of this rule and for the knowledge that is inferred. The artifacts state and reaction is determined from this inferred knowledge.

The KB describes also the services that artifacts provide so that to support the service discovery mechanism. The PM provides methods that query the KB for the services that an artifact offers as well as for artifacts that provide specific services. The *Resource Discovery and Binding* module gets from the PM the necessary knowledge stored in an artifact KB relevant to its services, in order to implement the service discovery mechanism. Finally the PM using this mechanism and a service classification can identify artifacts that offer similar semantically services and propose objects that can replace damaged ones. Therefore, it supports the deployment of adaptive and fault-tolerant UbiComp applications.

4.2. Resource Discovery and Binding (RDB)

Service and resource discovery will play an integral role in the realization of smart systems. Since mobile devices are very resource limited, in order to reduce the significance of these limitations they should be able to discover and use the resources of surrounding devices.

As another example, a service discovery mechanism is needed so that if a synapse is broken, e.g., because of an artifact's failure, another artifact that offers a similar semantically service could be found. In the example application scenario discussed previously if the synapse between *eDesk* and *eDeskLamp* is broken, because of a failure at the *eDeskLamp*, a new artifact having a property that provides the service "light" could be found. Thus for a UbiComp environment this mechanism must be enhanced to provide a semantic service discovery. This means that it must be possible to discover all the relevant services.

Since for the UbiComp applications a semantic service discovery mechanism is useful and the replacement of artifacts depends on the services that artifacts offer, a service classification is necessary. In order to define such a service classification we first identified some services that various artifacts may offer; some results of this work are presented in Table 2. From these results it is clear that the services offered by artifacts depend on artifacts' physical characteristics and their sensors/actuators.

Table 2. Services offered by artifacts

Artifact	Offered services
eLamp	switch on/off, light, heat
eDeskLamp	switch on/off, light, heat
eBook	open/close, number of pages, current page
eDrawer	contains objects yes/no, number of objects, open/close, locked/unlocked
eMoodCube	current position
eMobilePhone	send SMS, send email, make phone call, get calendar, get contacts
eMusicPlayer	sound, sound volume, kind of music, play/pause/stop, next/previous track
eCarpet	object on it yes/no, objects' position, pressure, weight, frequency

Next we had to decide how we should classify the services. The classification proposals that we elaborated are the following: by object category, by human senses and based on the signals that artifacts' sensors/actuators can perceive/transmit.

4.2.1. eRDP protocol

We have defined a lightweight Resource Discovery Protocol for eEntities (eRDP) where the term resource is used as a generalization of the term service. eRDP is a protocol for advertisement and location of network/device resources with a semantic description. We have followed the architecture of the Service Location Protocol [7] and defined three actors in the eRDP : (i) *Resource Consumer* (*RC*), an artifact that has a need for a resource, possibly with specific attributes, initiating for that purpose a resource discovery process, (ii) *Resource Provider* (*RP*): an artifact that provides a resource, also advertises the location and attributes of the resource to the *Resource Directory*, provided that there is one, and (iii) *Resource Directory* (*RD*): an artifact that aggregates resource information into a repository on behalf of artifacts that are less equipped. The RD is an optional component of the discovery protocol and its aim is to improve its performance. In the absence of an RD, RCs and RPs implement all RD's functions with multicast/broadcast messages with the optional and undeterministic use of resource cache within each artifact. When one or more RDs are present (Figure 6), the protocol is more efficient, as an RC or RP uses unicast messages to the RDs.

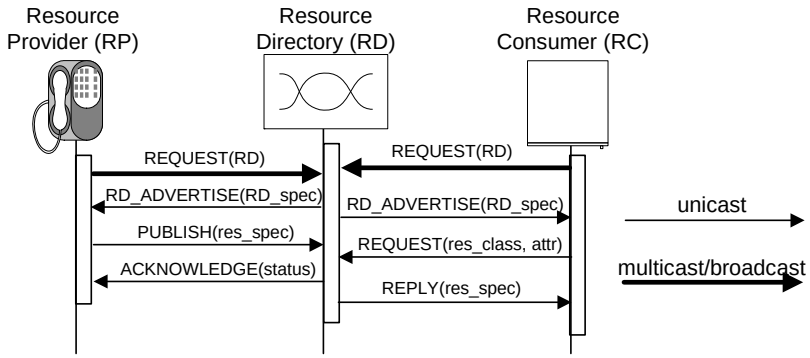

Figure 6. eRDP with a RD facility.

The protocol makes use of typed messages codified in XML. Each message contains a header part that corresponds to common control information including local IP address, message sequence number, message acknowledgement number, destination IP address(es) and message type identification. The prototype was written in Java using J2ME CLDC platform. kXML is used for parsing XML messages.

One of the KB uses is to describe the services that the artifacts provide and assist the service discovery mechanism. In order to support this functionality the middleware kernel provides methods that query the KB for the services that a plug provides as well as for the plug that provide a specific service. Therefore, the middleware provides to the calling process the necessary knowledge stored in artifact KB relevant to the artifact services, in order to support the service discovery mechanism. Similarly the middleware can answer queries for plugs compatibility and artifacts replaceability.

Suppose now that in the scenario discussed in the previous section the synapse between *eDesk* and *eDeskLamp* is broken. When this happens, the system will attempt to find a new artifact having a plug that provides the service “light”. The *eDesk* system software is responsible to initiate this process by sending a message for service discovery to the other artifacts (RD may be present or not) that participate in the same application or are present in the surrounding environment. This type of message is predefined and contains the type of the requested service and the service’s attributes. A description of the eLamp resource is shown in Figure 7.

The options related to the communications module that are defined are the multicast address and port used for the discovery of artifacts, and the TCP port on which this artifact should accept P2P connections. Then attributes of the local artifcat are defined, like its name, and the properties. In the XML description the physical properties of the artifact are defined as name-type-value triples, e.g. the eLamp from which the configuration file was taken can give maximum 50Lux luminosity.

When the system software of an artifact receives a service discovery message, forwards it to the PM of the middleware kernel. Assume that the artifact eLamp is available and that this is the first artifact that gets the message for service discovery. The eLamp PM first queries the KB of eLamp in order to find if this artifact has a plug that provides the service “light”.

If we assume that the eLamp has the plug “LampLight” that provides the service light, the PM will send to the system software a message with the description of this service. If such a service is not provided by the eLamp, the PM queries the eLamp KB

in order to find if another artifact, with which the eLamp has previously collaborated, provides such a service. In case of a positive answer it returns as a reply the description of this service. If the queried artifact, in our example the eLamp, has no information about an artifact that provides the requested service, the control is sent back to system software, which is responsible to send the query message for the service discovery to another artifact.

```
<resSpec>
    <resName> eLamp </resName>
    <resClass> light </resClass>
    <resId> ELamp@1066821815562 </resId>
    <commConfig>
         <TCP_PORT>6500</TCP_PORT>
         <MULTICAST_PORT>6501</MULTICAST_PORT>
         <MULTICAST_ADDR>225.225.225.225</MULTICAST_ADDR>
    </commConfig >
    <resData>
         <attrName=“power” type=”bool” value=”false”
         <attrName=”luminocity” type=”integer”, value=”50”
    </resData>
    <resTimestamp> 4758693030 </resTimestamp>
    <resExpiry> Never </resExpiry>
</resSpec>
```

Figure 7. XML description of eLamp resource

4.2.2 Dynamic Binding

For a UbiComp application the knowledge of the exact location of the artifacts and services that are required to realize the application may be quickly become obsolete. For instance, the dynamic nature of AmI environments can cause problems during application execution such as network or service unavailability. Dynamic binding is used to address the above issues. Given an abstract task specification dynamic binding tries to locate, synthesize and bind the services that are required to implement the task. Services are provided through plugs abstractions with semantically rich interfaces. Dynamic binding is performed by the Binder Algorithm as outlined in Figure 8.

The binding algorithm receives as input an abstract task specification (which may be read from a planning module, a user profile etc.), which describes a process to achieve a task without reference to actual artifacts, but to semantically rich descriptions of the services required in each step. The algorithm classifies these descriptions into *Plugs* (i.e. service providers/consumers), *Synapses* (between plugs) and *Constraints* (on the Synapses). Then, for each Synapse, it tries to map the specified input and output onto specific Plugs offered by a specific set of artifacts. When such a mapping is achieved, the corresponding artifacts (which possess the source and target Plugs) are informed about the established Synapse, so that they can start communicating without further mediation. The established Synapse is recorded in the abstract task specification. In the end, this will be transformed into a concrete task specification, listing the actual artifacts that realize the task and their interaction. An abstract task specification is a 4-tuple *T* of the form (*sourcePlugs*, *targetPlugs*, *Synapses, Constraints*) such that:

- *sourcePlugs* is a sequence of distinct properties and/or services;
- *targetPlugs* is a sequence of distinct properties and/or services;

- *Synapses* is a set of pairs of the form (*origin, destinations*) such that if σ is a synapse, then:
  - *origin(σ)* is a source plug of *T;*
  - *destinations(σ)* is a set of properties of *T* not containing *origin(σ)*.
- *Constraints* are either atomic or compositional as defined in Section 2.1

```
Algorithm Binder
Input: Abstract task specification T
      parse T into sourcePlugs, targetPlugs, Synapses, Constraints
      forall σ ∈ Synapses
            send origin(σ) to eRDP
            send destinations(σ) to eRDP
            receive eRDP Reply
            if corresponding artifacts satisfy semantic criteria and
                  constraints then cache artifact info
            endif
            create a Synapse message
            send Synapse message to service provider artifact to bind
            send Synapse message to service consumer artifact to bind
            record Synapse in Concrete task specification
      endfor
end
```

Figure 8. Binder algorithm

```
<PLUG>
     <NAME> Book State Plug </NAME>
     <DESCRIPTION> Outputs boolean values denoting its open/closed
     state. </DESCRIPTION>
     <INTERFACE> #OPENED# </INTERFACE>
     <TYPE> OUTPUT </TYPE>
     <DATATYPE> BOOLEAN </DATATYPE>
</PLUG>
<PLUG>
     <NAME> Book Luminocity Plug </NAME>
     <DESCRIPTION> Outputs integer values to describe the luminocity
     level on its surface. '1'=INADEQUATE / '2'=COMFORTABLE / '3'=TOO
     MUCH </DESCRIPTION>
     <INTERFACE> #LUMINOCITY_PLUG# </INTERFACE>
     <TYPE> OUTPUT </TYPE>
     <DATATYPE> INTEGER </DATATYPE>
</PLUG>
```

Figure 9. XML description of eBook plugs

It is implied that the implementation of a task specification can be represented as a graph of connected artifacts. A synapse associates a source plug of one artifact, the origin of the synapse, with the target plug of one other artifact, the destination of the synapse. For example, Figure 4 shows the internal structure of the task specification

that corresponds to the eStudy application example in the context of the editing tool. Figure 9 gives an example of plug descriptions in XML.

Synapse establishment can be also performed at runtime in the context of the Binder algorithm. As an example, let's consider the synapsing process among the 'ReadingActivity' plug of the eDesk and the 'Light' plug of the eDeskLamp. Figure 10 shows the sequence of messages for the synapse establishment. Synapse request occurs after the Binder algorithm has discovered the artifacts that provide the required services. The eDesk sends a "Synapse Request" message to the eDeskLamp. The message contains information concerning the eDesk and its ReadingActivity plug as well as the name of the Light plug. When the eDeskLamp receives the message it first checks the plug compatibility of the ReadingActivity and Light plugs. In the example the Reading plug is output and the Light plug is input, so the direction compatibility test is passed. Data type incompatibility does not halt the synapsing process, however it needs to be dealt via the use of mappings. Following, an instance of the Light plug is created in the eDeskLamp and a positive response is sent back to the eDesk. The instance of the Light plug is notified for changes by its remote counterpart plug, created by the eDesk, and this interaction serves as an publish/subscribe channel. In case of a negative plug compatibility test, a negative response message is sent to the eDesk, while no instance of the ReadingActivity plug is created.

After connection has been established, the two plugs are capable of exchanging data. Output plugs (ReadingActivity) use specific objects to encapsulate the plug data to send, while input plugs (Light) use specific event-based mechanisms to become aware of incoming plug data. When the value of the shared object of the ReadingActivity plug changes the instance of the Light plug in the eDeskLamp is notified and a synapse activation message is sent to the eDeskLamp. The eDeskLamp receives the message and changes the shared object of its ReadingActivity plug instance. This, in turn, notifies the target Light plug, which reacts as specified.

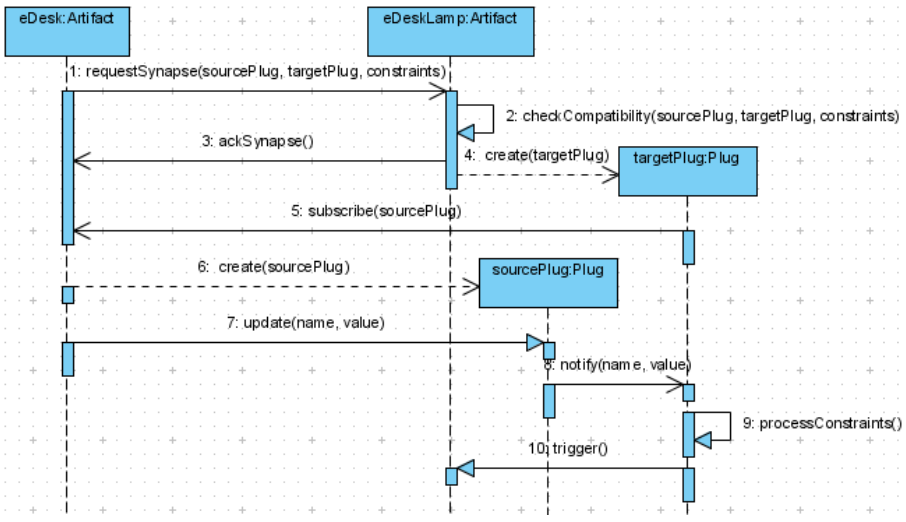

Figure 10. Synapse establishment sequence diagram.

Finally, in Figure 11 we give a few code snapshots on the implementation of the *update()* and *processConstraints()* methods shown in the above sequence diagram.

```
public Synapse(String sArtifactID, String sPlugName, String
tArtifactID, String tPlugName) {
    …
  }

public boolean update(String name, String value) {
  if (isInferenceInstalled) {
    StateChangedEvt e = new StateChangedEvt(name, value);
    informInferenceEngine(e);
    return true;
  } else {
    return false;
  }
}
```

```
public boolean processConstraints(Ruleset constraints)  {
…
    IRuleManager ruleManager =
    RuleManager.getRuleManager();
    int rulesSize = constraints.getAttributes().length;
    for (int i=0;i<rulesSize;i++) {
        Element ruleElement = ruleset.getAttribute(i);
        ruleManager.setRules(ruleElement);
    }
    try {
    inferenceEngineEntity.startInferenceEngine();
    } catch (Exception e) {
        System.out.println("Could not start Engine");
    }
….
}
```

Figure 11. Supporting code snapshots.

4.3. Implementation

The middleware prototype has been implemented in J2ME (Java 2 Micro Edition) CLDC[1] (Connected Limited Device Configuration), which is a very low-footprint Java runtime environment. The proliferation of end-systems, as well as typical computers capable of executing Java, make Java a suitable underlying layer providing a uniform abstraction for our middleware. Furthermore, it facilitates deployment on a wide range of devices from mobile phones and PDAs to specialized Java processors.

Up to now, our middleware has been tested in laptops, IPAQs, in the EJC (Embedded Java Controller) board[2] and on a SNAP board[3]. Both EJC and SNAP boards are network-ready, Java-powered plug and play computing platforms designed for use in embedded computing applications. The EJC system is based on a 32-bit ARM720T processor running at 74 MHz and has up to 64Mb SDDRAM. The SNAP device has a Cjip microprocessor developed by Imsys which has been designed for networked, Java-based control. It runs at 80 MHz and has 8 Mb SDDRAM. The main purpose of programming our middleware to run on these types of boards was to demonstrate that the system was able to run on small embedded-internet devices.

The code size of the current implementation of the middleware kernel is approximately 200 KB. Measuring the memory footprint is crucial in order to indicate that middleware can be executed on resource constraint devices. We measured the memory footprint of the middleware kernel running upon the Sun Personal Java on a Compaq IPAQ PDA reference system. The dynamic memory allocated depends on the number of plugs provided by an artifact as well as the number of synapses it

[1] java.sun.com/products/cldc
[2] www.embedded-web.com/
[3] www.imsys.se/documentation/manuals/snap_spec.pdf

participates. Using the JProbe Memory Profiler tool we measured that upon starting a smart object needs approximately 24 KB of memory for the initialization of the kernel components. Each plug costs 512 bytes in memory. An extra memory of 416 bytes is required for every plug participating in a synapse. Each synapse finally costs 856 bytes. The following formula shows memory footprint for an artifact with *P* plugs, when *X* plugs take part in *Y* synapses: Mem = 24KB + *P**512 + *X**416 + *Y**856; For example, an artifact with three plugs and one synapse per plug, like eDesk in our case, requires approximately 29 KB.

Using code instrumentation, we measured the average time for making a synapse and for communicating in our example application (Table 3). These measurements include the overhead of the IEEE 802.11b protocol and the discovery phase as specified previously, while messages exchanged vary from a few bytes to 1 KB. Synapse times refer to the amount of time needed from the point the user initiates the application up to the time this synapse is completed. We note that after synapses are established communication between objects is fast, with average time of a few hundreds of milliseconds, satisfying our requirement for real time response.

Table 3. Example application set-up overhead

	1st Synapse	2nd Synapse	3rd Synapse	Data Exchanged
Time (ms)	1832	1612	1622	400

5. Scalability

The consumption of bandwidth or the amount of traffic that a service/resource discovery protocol generates is a feature that is extremely important when devices are mobile and wireless like the networked everyday objects. Protocols that use fewer messages are more desirable because wireless bandwidth is a scant resource as well as mobile device power. We analyze the usage of bandwidth made by eRDP when a RD is present and when a RD is not present with an aim to identify the conditions under which the one case should be more appropriate than the other. The analysis process is based on the work presented in [8].

For the bandwidth usage analysis we assume that eRDP messages are encapsulated within network and MAC layer (e.g. 802.11b) messages. Consider the parameters *REQ_SZ*, *REP_SZ*, *PUB_SZ*, *ACK_SZ* and *ADV_SZ*, which represent the size of the corresponding messages (REQUEST, REPLY, PUBLISH, ACK and RD_ADVERTISE) of the eRDP. For the analysis we also assume that *n* is the number of RPs, *m* is the number of RCs. The analysis is divided in three phases: (i) *RD Discovery* (*RDD*), (ii) *Resource Publications* (*RP*) and (ii) *Resource Lookup* (*RL*).

The bandwidth usage (BU), generated by the RDD phase when there is no RD entity is given by the following formula, given that the REQUEST messages are sent a constant of *k* times:

$$RDD\_BU_{withoutRD} = REQ\_SZ \times k \times (n+m) \quad \textbf{(Eq. 1)}$$

The corresponding BU when an RD is involved is given below:

$$RDD\_BU_{withRD} = (REQ\_SZ + ADV\_SZ) \times (n+m) \quad \textbf{(Eq. 2)}$$

The RP phase is necessary when there is at least one RD in the network, otherwise the incurred cost is zero. In the presence of a single RD, the amount of traffic in bytes, generated by this phase is given by the following equation:

$$RP\_BU = n \times (PUB\_SZ + ACK\_SZ) \quad \textbf{(Eq. 3)}$$

In the RL phase we have two kinds of messages, REQUEST and REPLY. In the case of a network without an RD entity, let's assume that the REQUEST messages are sent a constant of λ times and that ω percent of the RPs return a reply. In the absence of an RD, and with a resource request frequency f_{rr}, the amount of traffic in bytes, generated by this phase is given by the following equation:

$$RL\_BU_{withoutRD} = m \times f_{rr} \times (\lambda \times REQ\_SZ + n \times \omega \times REP\_SZ) \quad \textbf{(Eq. 4)}$$

In the case of a network with an RD entity, a unicast REQUEST message is sent to the RD and a unicast REPLY message is sent back to the RC. When more than one publication matches the requested resource the RD is responsible for selecting the most appropriate provider taking also into account routing information should this be available. Thus, the amount of traffic in bytes is given by the following equation:

$$RL\_BU_{withRD} = m \times f_{rr} \times (REQ\_SZ + REP\_SZ) \quad \textbf{(Eq. 5)}$$

Combining (Eq. 1) and (Eq. 4) we get the overall bandwidth consumption in a configuration without an RD.

$$BU_{withoutRD} = REQ\_SZ \times (k \times (n+m) + \lambda \times m \times f_{rr}) + REP\_SZ \times m \times f_{rr} \times n \times \omega \quad \textbf{(Eq. 6)}$$

Combining (Eq. 2), (Eq. 3) and (Eq. 5) we get the overall bandwidth consumption in a configuration with an RD.

$$BU_{withRD} = REQ\_SZ \times (n + m(1 + f_{rr})) + REP\_SZ \times m \times f_{rr} + n \times (PUB\_SZ + ACK\_SZ) + ADV\_SZ \times (n+m) \quad \textbf{(Eq. 7)}$$

A useful assessment of the above analysis is the condition under which the bandwidth usage with an RD is less than the bandwidth usage without an RD, that is the condition such that $BU_{withRD} < BU_{withoutRD}$. Given the relative size of the eRDP message types where REP_SZ=PUB_SZ=ADV_SZ=L, REQ_SZ=0,6L and ACK_SZ=0.3L and if we assume that n=m and k=3 then by combining (Eq. 6) and (Eq. 7) we get the following expression:

$$n > \frac{0.9}{f_{rr} \times \omega} + \frac{1}{\omega} \times (1.6 - 0.6 \times \lambda) \quad \textbf{(Eq. 8)}$$

Giving typical values to the above parameters (ω=0.1, λ=2, f_{rr}=1.5), which we expect to hold for the UbiComp application domain, we get that $n > 10$. Therefore, *the configuration with an RD leads to better bandwidth usage when the number of RCs/RPs is at least 10*. Otherwise the introduction of the RD adds more overhead than a configuration without one.

6. Related work and discussion

Traditional middleware infrastructures like CORBA, Jini, UPNP have been used to address the needs of application developers targeting distributed systems. However, we have found that such systems are either too heavyweight to be applied to mobile hosts, or are not powerful and flexible enough to address the requirements of such systems. Furthermore, most of them are language or system dependent, and on the other hand, they try to provide as much functionality as possible, which leads to very complex and resource consuming systems, unsuitable for small devices.

There are systems that permit users to aggregate and compose networked devices for particular tasks [9]. However, those devices are not context aware but act more as service providers, e.g., Web services usually in the UPnP style. Approaches to modelling and programming such devices for the home have been investigated, where devices have been modeled as collections of objects [10], as Web services [11], and as agents [12]. However, there has been little work on specifying at a high level of abstraction how such devices would work together at the application level taking into account artifacts, which people can combine in dynamic ways.

The approach for the dynamic binding of services used in our middleware is similar to the one used in the OASiS framework, which implements a service-oriented middleware for pervasive ambient-aware sensor networks [13]. OASiS enables a wireless sensor network application to adapt to network failures and environmental changes by employing a dynamic service discovery protocol. However, their approach uses fixed identifiers for the description of services, instead of the semantically rich interfaces supported by the plug abstraction in our case. A flexible composition algorithm for smart device services has been developed as part of the middleware infrastructure developed in the context of the EU-IST project AMIGO [14]. Based on service descriptions, the algorithm builds up relevant service composition in a given situation, using contextual information. The algorithm dynamically selects services using a semantic and contextual matchmaking algorithm and consistently compose them using Constraint Satisfaction Problem solving.

Other research efforts are emphasizing on the design of ubiquitous computing architectures. For example, project “Smart-Its” [15] creates autonomous smart objects by attaching small computational devices to physical objects. However, this “augmentation” is not related in any way with their “nature”, thus the objects ends up to be just physical containers of the computational modules they host. Project 2WEAR [16], considers smart wearable objects whose digital self is not always related to their physical self, as many of the smart objects presented are just computational devices offering digital services (e.g. storage device). Furthermore, the functionality which is feasible without infrastructure support is not clear enough. Ambient Agoras [17] aims at providing situated services, place-relevant information, and feeling of the place to the users, so that they feel at home in the office, by using mobile and embedded information technology. “Ambient Agoras” aims at turning every place into a social marketplace of ideas and information where people can interact and communicate. However, this approach aims at building new smart objects from scratch rather than augmenting physical objects.

The proposed component-oriented programming model and middleware have provided to our research team and others a useful medium for exploring new approaches on merging the physical and digital space in AmI environments. This happened by reusing and extending our framework to new application domains examined in research projects undertaken by our group and associate colleagues. We mention our effort to create digital interfaces to nature, in particular to selected species of plants, enabling the development of synergistic and scalable mixed communities of communicating artifacts and plants by providing each plant with a description of its properties and state to enable a seamless interaction in scenarios ranging from domestic plant care to precision agriculture [18]. Another direction is explored by the authors in the context of awareness systems. Awareness systems are a class of computer mediated communication systems that help individuals or groups build and maintain a peripheral awareness of each other. Such systems promise to address pressing social problems:

elderly living alone, families living apart for large parts of the working week, monitoring the well being of an ill relative, etc. In [19], we present how smart objects in a person's environment can be used to capture and convey awareness information under this person's control.

7. Conclusions and future work

We have presented a component-oriented programming model and middleware support services for building UbiComp applications that are composed out of smart objects on demand. Our middleware supports the integration of heterogeneous smart objects by implementing a high level interaction model suited to the end-user and providing dynamic discovery, synthesis and binding of services that allow the development of UbiComp applications that adapt to the dynamics of an AmI environment.

Different smart object models may use different terms to describe the same concept and may follow different policies to perform the same task. As a consequence, applications and services developed for one system model often cannot be ported in other systems. One solution is to develop and discover mappings and relationships between different ontology-based systems, a process called ontology alignment [20]. The ontology alignment can be described as: given two ontologies each describing a set of discrete entities (which can be classes, properties, rules, predicates, or even formulas), find the correspondences, e.g., equivalence or subsumption, holding between these entities. To make our system available on a larger scale and adaptable to other systems developed for different models we aim to apply ontology alignment in order to find those elements which have the same intended meaning.

Acknowledgement

Part of the research described in this paper was conducted in the ATRACO (ICT-216837) project. The authors would like to thank the anonymous reviewers for their suggestions for improving this paper.

References

[1] A. Kameas, S. Bellis, I. Mavrommati, K. Delaney, M. Colley, and A. Pounds-Cornish, "An architecture that treats everyday objects as communicating tangible components", Proceedings of the first IEEE International Conference on Pervasive Computing and Communications, IEEE CS Press, pp. 115-122, March 23 - 26, 2003.

[2] C. Szyperski, "Component Software, Beyond Object-Oriented Programming". ACM Press, Addison-Wesley NJ, 1998.

[3] C. Goumopoulos, and A. Kameas, "Ambient Ecologies in Smart Homes", The Computer Journal, 2008; doi: 10.1093/comjnl/bxn042.

[4] N. Drossos, and A. Kameas, "Building composeable smart objects", 1st International Workshop on Design and Integration Principles for Smart Objects (DIPSO 2007), Innsbruck, Austria, 2007.

[5] N.R. Jennings, "On agent-based software engineering", Artificial Intelligence, vol. 117, no. 2, pp. 277-296, 2000.

[6] I. Mavrommati, A. Kameas, and P. Markopoulos, "An editing tool that manages the device associations", Personal and Ubiquitous Computing, ACM, Springer-Verlag London Ltd., vol. 8 no. 3-4, pp. 255-263, 2004.

[7] E. Guttman, "Service location protocol: Automatic discovery of IP network services", IEEE Internet Computing, vol. 3, no. 4, pp. 71-80, 1999.

[8] J. Govea, and M. Barbeau, "Comparison of Bandwidth Usage: Service Location Protocol and Jini" Technical Report TR-00-06, School of Computer Science, Carleton University, October 2000.

[9] R. Kumar, V. Poladian, I. Greenberg, A. Messer, and D. Milojicic, "Selecting devices for aggregation", Proceedings of the IEEE Workshop on Mobile Computing Services and Applications, IEEE CS Press, pp. 150–159, 2003.

[10] J. H. Jahnke, M. d'Entremont, and J. Stier, "Facilitating the programming of the smart home", IEEE Wireless Communications, vol. 9, no. 6, pp. 70-76, 2002.

[11] K. Matsuura, T. Hara, A. Watanabe, and T. Nakajima, "A new architecture for home computing", Proceedings of the IEEE Workshop on Software Technologies for Future Embedded Systems, IEEE CS Press, pp. 71-74, 2003.

[12] F. Ramparany, O. Boissier, and H. Brouchoud, "Cooperating autonomous smart devices", Proceedings of the Smart Objects Conference (sOc'2003), pp. 182-185, 2003.

[13] I. Amundson, M. Kushwaha, X. Koutsoukos, S. Neema, and J. Sztipanovits, "OASiS: A Service-Oriented Middleware for Pervasive Ambient-Aware Sensor Networks", Technical Report ISIS-06-706, Institute for Software Integrated Systems, Vanderbilt University, 2006.

[14] M. Vallee, F. Ramparany, and L. Vercouter, "Flexible composition of smart device services", Proceedings of the International Conference on Pervasive Systems and Computing, CSREA Press, pp. 165-171, 2005.

[15] L. E. Holmquist, H.-W. Gellersen, A. Schmidt, M. Stro-hbach, G. Kortuem, S. Antifakos, F. Michahelles, B. Schiele, M. Beigl, and R. Mazé, "Building Intelligent Environments with Smart-Its", IEEE Computer Graphics & Applications, vol. 24, no. 1, pp. 56-64, 2004.

[16] S. Lalis, A. Savidis, A. Karypidis, J. Gutknecht, and C. Stephanides, "Towards Dynamic and Cooperative Multi-Device Personal Computing", in: Streitz, N., Kameas, A., Mavrommati, I. (eds.) The Disappearing Computer, LNCS, Springer, Heidelberg, vol. 4500, pp. 182-204, 2007.

[17] N. A. Streitz, C. Rocker, T. Prante, D. v. Alphen, R. Stenzel, and C. Magerkurth, "Designing Smart Artifacts for Smart Environments", IEEE Computer, vol. 38, no. 3, pp. 41-49, 2005.

[18] C. Goumopoulos, A. Kameas, and B. Oflynn, "Proactive Agriculture: An Integrated Framework for Developing Distributed Hybrid Systems", Proceedings of the 4th International Conference on Ubiquitous Intelligence and Computing (UIC-07), Springer-Verlag, LNCS 4611, pp. 214-224, 2007.

[19] C. Goumopoulos, A. Kameas, E. Berg, and I. Calemis, "A Service-Oriented Platform for Pervasive Awareness Systems", Proceedings of the Service Oriented Architectures in Converging Networked Environments Workshop (SOCNE 2009), Bradford, UK, May 26-29, 2009.

[20] M. Ehrig, "Ontology Alignment - Bridging the Semantic Gap", Springer, New York, NY, US, 2007.