Chapter 9
Achieving Co-Operation and Developing Smart Behavior in Collections of Context-Aware Artifacts

Christos Goumopoulos, Achilles Kameas

Abstract One of the most exciting and important recent developments in ubiquitous computing (UbiComp) is to make everyday appliances, devices, and objects context aware. A context-aware artifact uses sensors to perceive the context of humans or other artifacts and to respond sensibly to it. Adding context awareness to artifacts can increase their usability and enable new interactions and user experiences. The aim of the research and development work discussed here is to examine at how artifact collections (or ambient ecologies, a metaphor introduced for modelling UbiComp applications) can be made to work together, and provide functionality that exceeds the sum of their parts. The underlying hypothesis is that even if an individual artifact has limited functionality, it can harness more advanced behaviour when grouped with others. The realization of this hypothesis is possible by providing appropriate abstractions and a new affordance (composeability) that objects acquire. Specifically our contribution is firstly to discuss the conceptual abstractions and formal definitions used to model such artifact collections, which are inherent to ubiquitous computing, and secondly to discuss engineering guidelines for building ubiquitous computing applications based on well-known design principles and methods of analysis. It is argued then that the process where people configure and use complex collections of interacting artifacts can be viewed as having much in common with the process where system builders design software systems out of components. The design space consists, in this view, of a multitude of artifacts, which people (re)combine in dynamic, ad-hoc ways. Artifacts are treated as reusable "components" of a dynamically changing physical/digital environment, which involves people. In a nutshell, in this work we have attempted to define ambient ecologies, specify design patterns and programming principles, and develop infrastructure and tools to support ambient ecology designers, developers and end-users.

Keywords ubiquitous computing; ambient ecologies; formal model; context-aware artifacts; design patterns; ontology; composeability; middleware; intelligent systems

Computer Technology Institute, Distributed Ambient Information Systems Group, Patras, Hellas

K. Delaney, *Ambient Intelligence with Microsystems*,

1 Introduction

An important characteristic of Ubiquitous Computing (UbiComp) environments is the merging of physical and digital space (i.e. tangible objects and physical environments are acquiring a digital representation). As the computer disappears in the environments surrounding our activities, the objects therein become augmented with Information and Communication Technology (ICT) components (i.e. sensors, actuators, processor, memory, wireless communication modules) and can receive, store, process and transmit information; in the following, we shall use the term "artifacts" for this type of augmented objects.

Individually, artifacts may have a small range of capabilities but together can exhibit a much broader range of behaviours. Consequently, the true potential of all of these disappearing computers is realised once they are interconnected in digital space to form combinations of artifacts or services to accomplish the goal of its user(s). Because such units can be re-configured, or recombined either by people or another supervisory authority their collective behaviour is neither static nor random and collections of artifacts can evolve to produce new behaviours. Smart behavior, then, either at individual or collective levels, is possible because of the artifacts' abilities to perceive and interpret their environment (peer artifacts being themselves a part of an artifact's environment). Artifacts that is, everyday appliances, devices, and objects become context aware [1].

The aim of the research and development work to be presented in this chapter is to examine at how artifact collections (or ambient ecologies, a term to be introduced in Section 2), can be made to work together, and provide functionalities that exceed the sum of their parts. Specifically our contribution is firstly to discuss the conceptual abstractions and formal definitions used to model such artifact collections, which are inherent to ubiquitous computing and secondly to discuss engineering guidelines for building UbiComp applications based on well-known design principles and methods of analysis.

The realization of our main hypothesis (even if an individual artifact has limited functionality, it can cause the emergence of more advanced behaviour when grouped with others) is possible by providing appropriate abstractions and a new affordance (composeability) that the objects acquire. Composeability can give rise to new collective functionality as a result of a dynamically changing number of well-defined interactions among artifacts. Composeability is perceived by users through presentation - via the object's digital self - of the object's connectable capabilities, and thus providing users the possibility to achieve connections and compose applications of two or more objects. In implementation terms, this is achieved via a communication unit that artifacts possess and the provision of semantic descriptions of their services.

It is argued then that the process where people configure and use complex collections of interacting artifacts can be viewed as having much in common with the process where system builders design software systems from components. The design space consists, in this case, of a multitude of artifacts that people

(re)combine in dynamic, ad-hoc ways. Artifacts are treated as reusable "components" of a dynamically changing physical/digital environment, which involves people. Naturally, the idea of building UbiComp applications out of components is possible only in the context of a supporting component framework which acts as a UbiComp middleware.

1.1 A Motivating Scenario

In order to investigate the previously discussed research direction we define an illustrative scenario that we have started to develop. In the following sections we highlight the parts of this scenario that strongly suggest the use of a component-based approach and explain why composeability is a key factor in this vision.

It's 7:30 in the morning. Catherine, an administrative officer in Brussels, is still sleeping when the alarm clock rings to her wake up. In the meantime the venetian blinds in the bedroom open automatically as do the blinds in the kitchen and the mp3 player turns on playing a random song on her favourite music list. As she leaves her bed the coffee-machine and the toaster start automatically and the bath heater is activated. After taking her morning shower, while she prepares her breakfast, she decides that she would like to change her usual menu for that day. She recalls in the screen of the refrigerator the recipe inventory and she selects through the touch screen interface her favourite Chinese recipe. This new selection initiates a check in the supplies inventory, which indicates that several ingredients are missing. Those ingredients and possibly other that are close to be exhausted are ordered automatically in the web order-service of the nearest supermarket, while SMS (Short Message Service) is used to inform the housekeeper so that she collects the shopping on her way to the house. Catherine is ready to leave her home, when her smart plant alerts her that it needs water. She receives proper notification, which depends upon her location context; when she is inside the house she will get the message through the nearest displaying object that has been set up for that purpose. When she is outside the house she will get an SMS. She then waters the plant and when she leaves her house a taxi is waiting for her outside in the street. The taxi was called for her a few minutes earlier through a conversation between her smart calendar and the appointment Web service of the taxi center. Later in the afternoon Catherine decides to use his smart office to read a book. Objects like books, chair, desk, and lights are instrumented with appropriate sensors that can capture the intent of the user and cooperate to provide an appropriate service, e.g., turning on the desk-lamp. While she is reading her book Catherine receives a notification. It is her friend Amanda inviting her to go for a walk down the near park. An awareness visualization object (for example, a personal item like an electronic bunny moving its ears) near her desk is used as a means for their informal social communication.

1.2 Outline

The remainder of this work is organized as follows. In Section 2 the concept of ambient ecologies is introduced, describing a space populated by connected devices and services that are interrelated with each other, the environment and the people. Aspects of programming ambient ecologies inspired by the component software engineering paradigm are discussed. Section 3 provides in a formal manner the basic elements of our conceptual model for programming ambient ecologies, specifying terms such as artifact, ambient ecology, state, transition and behavior modeling. In the following section the Gadgetware Architectural Style (GAS) is briefly presented as the consistent conceptual and technical referent among artifact designers and application designers. Section 5 puts the conceptual framework in perspective and refines the theoretical work into an application-engineering paradigm. An example is used to demonstrate the features of our definitions. Section 6 discusses the supporting framework for achieving co-operation and developing smart behavior in collections of context-aware artifacts. The framework provides a runtime environment to build applications from artifact components, tools and programming principles in the form of a design pattern to support application designers and developers. Related work is presented in Section 7. Finally we conclude with a discussion on the approach presented and lessons learned, as well as on issues and research problems that may arise regarding the adoption of such assembled systems.

2 The Emergence of Ambient Ecologies

Thanks to developments in the field of electronic hardware, in miniaturization and cost reduction, it is possible nowadays to populate everyday environments (e.g., home, office, car, etc.) with "smart" devices for controlling and automating various tasks in our daily lives.

At the dawn of the ubiquitous computing era, an even larger number of everyday objects will become computationally enabled, while micro/nano sensors will be embedded in most engineered artifacts, from the clothes we wear to the roads we drive on. All of these devices will be networked using wireless technologies evolved from Bluetooth [2], Zigbee [3] or IEEE 802.11 [4] for short range connectivity. Furthermore, the omnipresence of the Internet via phone lines, wireless channels and power lines facilitates ubiquitous networks of smart devices that will significantly change the way we interact with (information) appliances and can open enormous possibilities for innovative applications, as advocated by interaction design experts [5, 6].

The Merriam-Webster OnLine dictionary defines the word *ecology* as *the interrelationship of organisms and their environments* and the word *ambient* as *existing in the surrounding area*. We use the **ambient ecology** metaphor to conceptualize a space populated by connected devices and services that are interrelated with each

other, the environment and the people, supporting the users' everyday activities in a meaningful way. Everyday appliances, devices, and context aware artifacts are part of ambient ecologies. A context-aware artifact uses sensors to perceive humans or other artifacts and to respond sensibly. Adding context awareness to artifacts can increase their usability and enable new user interaction and experiences. Given this fundamental capability single artifacts have the opportunity to participate in artifact-based service orchestration ranging from simple co-operation to developing smart behavior.

Integrating such systems will not only enable ambient ecologies, but the ecology can also partially drive members' interactions. For example, given a collocated set of grocery items, an application might search for recipes and display them on a kitchen screen; once the user confirms the recipe, various appliances could be preset according to the cooking instructions. So, the same ecology enables actions on appliances and artifacts based on the contexts of appliances, artifacts, and users. Through the ecology, appliances and artifacts become aware of each other. In general, the context information that the system uses in reasoning can concern a particular device, appliance, or user, or a collection of such entities. In turn, an entity might be aware only of its own context or that of a specific group of entities. Furthermore, an entity can respond individually to its perceived context or to the group's, or a higher level application might coordinate a response among various devices.

In this work we address the need for a high level of abstraction to describe how context-aware artifacts could work together and to manage their interaction for building ambient ecologies. The presented approach is based on an ontological model of components where artifacts represent everyday objects; hence (a) their services are affected by their physical properties, (b) their context of operation is defined by the existence/availability of objects and (c) their collective functionality is emerging from a set of interactions among them.

Regarding the evolution of ambient ecologies at the level of artifacts we may benefit from borrowing the notion of agents' properties [7]. The concept of intelligent agent refer to a software entity endowed with specific properties such as persistence, context awareness, proactivity, continuity of operation and interactivity with one or more users or other similar entities in a shared environment. These properties require reasoning capabilities and control mechanisms for ensuring agent autonomy.

2.1 Programming ambient ecologies

Since distributed and concurrent systems have become the norm, some researchers are putting forward theoretical models that portray computing as primarily a process of interaction. A challenge related with the programming of ambient ecologies is to establish a common language so that artifacts can actually interact with each other and function in a collaborative manner. The main reason for these semantic interoperability difficulties is the heterogeneity of devices and the large variety of their

embedding context. Devices take part in several activities of our daily lives including environmental controls, lighting, alarm systems and security, telecommunication, cooking, cleaning and entertainment.

There exist a vast number of potential scenarios for integrating such devices. It is not possible to foresee all possible applications and equip devices with functionality that enables collaboration with every other device a customer would like to integrate. Consequently, there is the need for customization mechanisms that can be used for integrating different artifacts into a common process exemplified within the ambient ecology. Such customization mechanisms can be seen as the "programming language" for ambient ecologies. Primary requirements for such a programming language are ease of use and rapid deployment. Effective programming mechanisms for ambient ecologies require innovative paradigms that lift programming to a level of abstraction that is similar to plugging in a new stereo or TV set.

We propose a model which provides a convenient abstraction for the development of small to medium sized ubiquitous computing applications. These systems are powerful enough to support the everyday activities of people (such as home control, shopping entertainment etc); thus, we expect that most user-developed systems will fall into these categories. When more complex systems must be developed (i.e. involving over a dozen interacting artifacts or more than one user), the direct management of interactions becomes difficult, as several issues will now become important and demand the user's attention.

These include how goals and tasks can be distributed over artifacts, how can the distributed control be coordinated in order to insure that the overall system requirements are addressed, how the system can be configured with minimum user intervention, etc. Although in principle such issues can be addressed via direct manipulation, the cognitive load imposed on the user and the extended learning curve may affect the adoption and utilization of the system. We attempt to address this problem by developing end-user tools which would provide abstractions of the applications and support semantically rich interaction. For example, an agent that could learn how users act in their environment could receive user requirements and propose sets of connections to realize desired behaviours.

2.2 The Enabling Paradigm – Component Software

Component-based software systems are assembled from a number of pre-existing software modules called "software components". Thus, software components should be made to be (re)usable in many different application contexts in the construction of software. Using the component paradigm has various benefits: it increases the degree of abstraction during programming, provides proven (error-free) solutions for certain aspects of the application domain, increases productivity, and facilitates maintenance and evolution of software systems. Component-based

software development has become an important part of modern software engineering methods [8]. For example, lightweight components (i.e., fairly small in size) have become part of modern programming languages (e.g., the Swing library within Java).

Our approach uses the principles of software component technology as an enabling paradigm for describing the process where by people configure and use complex collections of interacting artifacts [9].

According to this paradigm a *component* in the UbiComp domain is an artifact, physical or digital, which is independently built and delivered as an autonomous functional unit. It offers interfaces by which it can be connected with other components to compose a larger system, without compromising its shape or functionality. The above definition emphasizes the fact that a component provides functionality in terms of services via well-defined interfaces by sending messages to, and receiving messages from, other components and performing its computation in response to the receipt of triggering events. It also emphasizes the black-box nature of components, which represents the encapsulation of its implementation details.

An *interface* is a description of a set of operations related to the external specification of a component. An interface consists of the artifact properties and capabilities, a set of operations that a component needs to access in its surrounding environment (required interface) and a set of operations that the surrounding environment can access on the given component (provided interface). An *operation* is a unit of functionality implemented by a component, which may map to a method, a function or a procedure.

Although our approach for composing UbiComp applications builds on the foundations of established software development approaches such as object oriented design and component frameworks, it extends these concepts by exposing them to the end-user to be used and configured in dynamic and ad hoc ways. In contrast to the majority of component-based models that have focused on software components with an emphasis on supporting the programmer, our component model embraces a heterogeneous collection of artifacts in a way that is easily comprehensible to end-users. To achieve this, composition tends to be as simple as possible, although some reduction in the expressiveness follows. The analogy with software components upon which the notion of artifact components relies, leads naturally to a visualization of component-based ubiquitous applications as a network of boxes communicating with each other via connecting wires.

The component based architectural abstraction is common in several engineering disciplines (i.e. software, buildings etc). Due to the properties of the digital self of artifacts, users can conceptualize their tasks in a variety of ways, such as stimulus-desired response, rules, sequences and constraints between entities, etc. Consequently, there will always be an initial gap between their intentions and the resulting functionality of an artifact composition, which they will have to bridge based on the experience they will develop after a trial-and-error process.

3 A Conceptual Model for Programming Ambient Ecologies

In practical terms, conceptual modeling is at the core of systems analysis and design. One category of approaches towards development of the theoretical foundations of conceptual modeling draws on ontology. Our approach is based on the so-called Bunge-Wand-Weber (BWW) ontology that describes a set of models in order to model information systems [10]. The BWW ontology is based on the scientific and dialectical-materialist ontology developed by Mario Bunge [11, 12]. Basic constructs of the BWW ontology, which have been used as a starting point for our work include:

- *Thing*: "The world is made of things that have properties".
- *Composite Thing*: "A composite thing may be made up of other things (composite or primitive)".
- *Conceivable State*: "The set of all states that the thing may ever assume".
- *Transformation of a Thing*: "A mapping from a domain comprising states to a co-domain comprising states".
- *Property*: "We know about things in the world via their properties".
- *Mutual Property*: "A property that is meaningful only in the context of two or more things".
- *System*: "A set of things will be called a system, if, for any partitioning of the set, interactions exist among things in any two subsets".

In the following section, we elaborate on those concepts taking into consideration the requirements of the UbiComp application domain. We extend the concept of Thing that of eEntity and the concept of Composite Thing to that of Ambient Ecology. An artifact is defined as a special case of eEntity. The dynamic behavior of artifacts is modeled with statecharts which incorporate states and events. New concepts are introduced like the *plug* and *synapse* in order to provide detailed representation of the interaction among artifacts.

3.1 Basic Elements

Our model defines the logical elements necessary to support a variety of applications in smart spaces. Its basic definitions are given below. A graphical representation of the concepts and the relations between them is given as a UML class diagram in Fig. 9.1.

eEntity: An eEntity is the programmatic bearer of an entity (i.e. a person, place, object, another biological being or a composition of them). An eEntity constitutes the basic component of an Ambient Ecology. 'e' stands here for extrovert. Extroversion is a central dimension of human personality, but in our case the term is borrowed to denote the (acquired through technology) competence of an entity to interact with other entities in an augmented way for the purpose of meaningfully

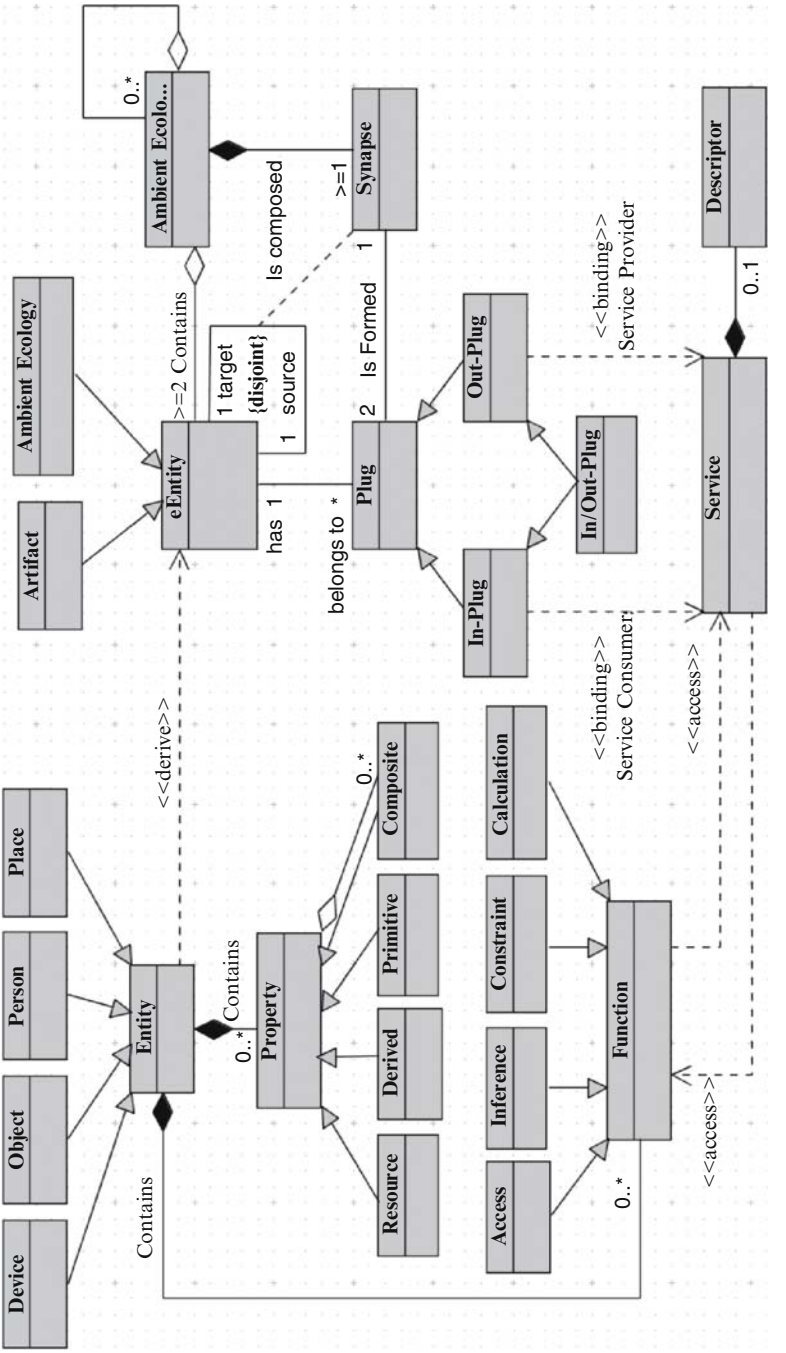

Fig. 9.1 UML model of the ambient ecology concept

supporting the users' everyday activities. This interaction is mainly related to either the provision or consumption of context and services between the participating entities. A coffee maker, for instance, publishes its service to boil coffee, while context for a person may denote her activity and location. An augmented interaction between the coffee maker and the person is the activation of the coffee machine when the person wakes in the morning. For this to happen we will probably need a bed instrumented with pressure sensors (an artifact) and a reasoning function for the persons' process of waking, which may not be trivial to describe. to the entity itself; relational, which relate the entity to other entities; and behavioral, which determine possible changes to the values of structural and relational properties.

Artifacts: An artifact is a tangible object - biological elements like plants and animals are also possible here, see [13] which bears digitally expressed properties. Usually, it is an object or device augmented with sensors, actuators, processing, networking, or a computational device that already has embedded some of the required hardware components. Software applications running on computational devices are also considered to be artifacts. Examples of artifacts include furniture, clothes, air conditioners, coffee makers, a software digital clock, a software music player, a plant, etc.

Services: Services are resources capable of performing tasks that form a coherent functionality from the point of view of provider entities and requester entities. Services communicate only through their exposed interfaces. Services are self-contained, can be discovered and are accessible through signatures. Any functionality expressed by a service descriptor (a signature and accessor interface that describes what the service offers, requires and how it can be accessed) is available within the service itself.

Ambient Ecology: Two or more eEntities can be combined in an eEntity synthesis. Such syntheses are the programmatic bearers of Ambient Ecologies and can be regarded as service compositions; their realization can be assisted by end-user tools. Since the same eEntity may participate in many Ambient Ecologies the whole-part relationship is not exclusive. In the UML class diagram (see Figure 9-1) this is implied by using the aggregation symbol (hollow diamond) instead of the composition symbol (filled diamond). Ambient Ecologies are synthesizable since an Ambient Ecology is an eEntity itself and can participate in another Ecology.

Properties: Entities have properties, which collectively represent their physical characteristics, capabilities and services. A property is modeled as a function that either evaluates an entity's state variable into a single value or triggers a reaction, typically involving an actuator. Some properties (i.e. physical characteristics, unique identifier) are entity-specific, while others (i.e. services) are not. For example, attributes like color/shape/weight represent properties that all physical objects possess. The 'light' service may be offered by different objects. A property of an entity composition is called an *emergent* property. All of the entity's properties are encapsulated in a *property schema* which can be sent on request to other entities, or tools (e.g. during an entity discovery).

Functional Schemas: An entity is modeled in terms of a functional schema: $F = \{f_1, f_2 \ldots f_n\}$, 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, through 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 entity at a given time represent the state of the entity. For an entity E, the set $P(E) = \{(p_1, p_2 \ldots p_n) | p_i = f_i(t)\}$ represents the state space of the entity. 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.

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 entitys' functional context (as it is propagated through the synapses of the entity).

Plugs: Plugs represent the interface of an entity. An interface consists of a set of operations that an entity needs to access in its surrounding environment and a set of operations that the surrounding environment can access on the given entity. Thus, plugs are characterized by their direction and data type. Plugs may be output (O) where they manifest their corresponding property (e.g. as a provided service), input (I) where they associate their property with data from other artifacts (e.g. as service consumers), 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.). From the user's perspective, plugs make visible the entities' properties, capabilities and services to people and to other entities.

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

3.2 Formal Definitions

To define formally the artifacts and the ambient ecology constructs we first introduce three auxiliary concepts: the domain D is a set which does not include the empty element; P is an arbitrary non-infinite set called the set of properties or property schema - each element p of which is associated with a subset D denoted $\tau(p)$ called the type of p; τ is actually a function that defines the set of all elements of D that can be values of a property. The domain D might include values from any primitive data type such as integers, strings, enumerations, or semantically rich ones such as light, sound and image.

3.2.1 Artifact

An artifact is a 4-tuple A of the form (*P*, *F*, *IP*, *OP*) where:

- *P* is the artifacts' property schema
- *F* is the artifacts' functional schema
- *IP* is a set of properties (ip_1, ip_2, …, ip_n) for some integer $n \geq 0$ that are imported from other artifacts (corresponding to input plugs);
- *OP* is a set of properties (op_1, op_2, …, op_m) for some integer $m \geq 0$ that are exported to other artifacts (corresponding to output plugs).

The role of artifacts in an ambient ecology can be seen as analogous to that of primitive components in a component-based system. In that sense they provide services implemented using any formalism or language. Plugs (input and output) provide the interface through which the artifact interacts with other artifacts. The functionality of an artifact is implemented through its functional schema F. In general an artifact produces data on its *OP* set in response to the arrival of data at its *IP* set. There are two special cases of artifacts:

- a source artifact is one that has an empty *IP* set;
- a sink artifact is one that has an empty *OP* set.

A source artifact from the point of view of the application in which it is embedded generates data. For example, an eClock generates an alarm event to be consumed by other artifacts. On the other hand, a sink artifact receives its input data from its input plugs but produces no data. For example, the eBlinds artifact receives the awake event from the eClock and opens the blinds without producing any new data.

3.2.2 Ambient ecology as a composite artifact

Ambient ecologies are synthesizable since an ambient ecology is an entity itself and can participate in another ecology. Then we can formally define an ambient ecology as a 5-tuple Σ of the form (*C*, *E*, *S*, *IP*, *OP*). Let σ be a composite artifact then:

- *C* is the set of constituent artifacts (see previous section for artifact definition) not including Σ at time t. It follows that the composition of σ at time t is:

$$\Theta(\sigma, t) = \{x \mid x \in C\}$$

- *E* is the surrounding environment, the set of entities that do not belong to *C* but interact with artifacts that belong to *C* at time t. It follows that the surrounding environment of σ at time t is:

$$\prod(\sigma, t) = \{x \mid x \notin \Theta(\sigma, t) \wedge \exists y \in \Theta(\sigma, t) \wedge \exists \delta(x, y) \in S\}$$

  where $\delta(x, y)$ denotes a synapse existence between *x* and *y*.

- *S* is a set of synapses that is a set of pairs of the form (*source*, *target*) such that if δ is a synapse, then:

- *source*(δ) is either an input plug of Σ or an output plug of an element of C;
- *target*(δ) is a set of properties of Σ not containing source(δ);
- For each target ρ of δ, τ (*source*(δ)) $\subseteq \tau(\rho)$.
- It follows that the interconnection structure of σ at time t is:

$$\Delta(\sigma,t) = \{\delta(x,y) \mid x,y \in \Theta(\sigma,t)\} \cup \{\delta(x,y) \mid x \in \Theta(\sigma,t) \wedge y \in \prod(\sigma,t)\}$$

- *IP* is a set (possibly empty) of distinct properties that are imported from the surrounding environment (corresponding to input plugs);
- *OP* is a set (possibly empty) of distinct properties, called emergent properties, that are exported to the surrounding environment (corresponding to output plugs);

  Auxiliary to the above we define the following items:
- The property schema of Σ is defined as the set:

$$IP \cup OP \cup \{p \mid \exists x \in C \wedge p \in P(x)\}$$

where $P(x)$ is the property schema of constituent artifact x.

$$\forall x, y \in C, \text{the sets } IP, OP, P(x) \text{ and } P(y) \text{ are pairwise disjoint.}$$

A composite artifact is a set of interconnected artifacts through synapses. A synapse associates an output plug of one artifact (the source of the synapse) with the input plugs of one or more other artifacts (the targets of the synapses). A synapse reflects the flow of data from source to targets. Each target should be able to accept any value it receives from the source, so its type must be a subset of the type of the source. Synapses cause the interaction among artifacts and the coupling of their execution. When a property of a source artifact changes, the new value is propagated via the synapse to the target artifact. The initial change of value caused by a state transition of the source artifact, causes eventually a state transition of the target artifact and thus their execution is coupled.

3.2.3 States, Transitions and Behavior Modeling

A state over a property schema P is a function ϕ: $P \rightarrow D$ such that $\phi(p) \in \tau(p)\ \forall\ p \in P$. A state is an assignment of values to all properties. The dynamics of an artifact are described in terms of its changes of states. When an artifact α undergoes a state change the value of at least one of its properties will alter. A change of state constitutes an event. Thus an event may be defined as an ordered pair $\langle \kappa, \kappa' \rangle$where κ, κ' are states in the state space of α.

If α is an artifact (it can be a composite one) and κ is a state, then an execution of α from κ_1 is a sequence of the form $\kappa_1 \rightarrow \kappa_2 \rightarrow \kappa_3 \rightarrow \ldots \rightarrow \kappa_n$. For each i ≥ 1 three kinds of transitions are identified:

1. κ_i is a propagation of κ_{i-1};
2. otherwise κ_i is a derivation of κ_{i-1};
3. otherwise κ_i is an evaluation of κ_{i-1}

The propagation is the simplest transition as it simply copies values that have been generated by an artifact along the synapses from the artifact's output plug to the other artifacts. These values may arrive at input plugs of some artifacts, which can trigger accordingly an evaluation of the artifact's function(s).

The derivation is a composite transition, which incorporates the propagation and evaluation of a relational property at the synapse level. The derivation associates logically (using logical operators) the properties that are found at the end-points of the synapse essentially deriving a new relational property, which serves as an input plug to subsequent evaluation.

The evaluation transition refers to a situation where the input plugs of an artifact have been defined through propagation or derivation transitions and the function(s) of the artifact can be executed so that the results are passed to its output plugs.

Based on the above discussion it emerges that a natural way to model the behavior of artifacts and the behavior of ambient ecologies viewed as assemblies of artifacts is to use statechart diagrams. Statecharts are a familiar technique to describe the behavior of a system. They describe all of the possible states that a particular object can have and how the object's state changes as a result of events that reach the object. In principle, a statechart is a finite-state machine with a visual appearance that is greatly enhanced by introducing a specialized graphical notation. Statecharts allow nesting of states (hierarchical statecharts). The expressive power of statecharts is enhanced by using Object Constraint Language (OCL) for conditional triggering of communication events. Statecharts play a central role in object-oriented software engineering methodologies (e.g., Unified Process) and is one of the diagrams supported by the UML standard [14]. The UML style is based on David Harel's statechart notation [15]. Statecharts represent states by using rounded rectangles. Input and output control ports are attached to states, representing the states' entry and exit points, respectively. Transitions between states are represented by arrows linking control ports of states. Statecharts may contain ports not attached to any state. These control ports refer to the entry/exit points of superstates. The states of a statechart define the states of the artifact and the links between the states define the events of an artifact.

4 Gas Architectural Style

The ways that we can use an ordinary object are a direct consequence of the anticipated uses that object designers "embed" into the object's physical properties. This association is in fact bi-directional: objects have been designed to be suitable for certain tasks, but it is also their physical properties that constrain the tasks people use them for. According to Norman [16] affordances "refer to the perceived and actual properties of the thing, primarily those fundamental properties that determine just how the thing could possibly be used".

Due to their "digital self", artifacts can now publicize their abilities in digital space. These include properties (what the object is), capabilities (what the object

can do) and services (what the object can offer to others). At the same time, they acquire extra capabilities, which during the formation of UbiComp applications (ambient ecologies), can be combined with the capabilities of other artifacts or adapted to the context of operation. Thus, artifacts offer two new affordances to their users:

- *Composeability*: artifacts can be used as building blocks of larger and more complex systems.
- *Changeability*: artifacts that possess or have access to digital storage can change or adapt their functionality. For example, an artifact can aggregate service information into its repository on behalf of artifacts that are less equipped facilitating in that way a service discovery process.

Both these affordances are a result of the ability to produce descriptions of properties, abilities and services, which carry information about the artifact in digital space. This ability improves object/service independence, as an artifact that acts as a service consumer may seek a service producer based only on a service description. For example, consider the analogy of someone wanting to drive a nail and asking not for the hammer, but for any object that could offer a hammering service (could be a large flat stone). In order to be consistent with the physical world, functional autonomy of UbiComp objects must also be preserved; thus, they must be capable to function without any dependencies from other objects or infrastructure. As a consequence artifacts are characterized by the following basic principles:

- *Self-representation*: the digital representation of artifact's physical properties is in tight association to its tangible self.
- *Functional autonomy*: artifacts function independently of the existence of other artifacts.

We have designed *GAS* (the *Gadgetware Architectural Style*), as a conceptual and technological framework for describing and manipulating UbiComp applications [9]. It consists of a set of architecture descriptions (syntactic domain) and a set of guidelines for their interpretation (semantic domain). GAS extends component-based architectures to the realm of tangible objects and combines a software architectural style with guidelines on how to physically design and manipulate artifacts.

For the end-user, this model can serve as a high level task interface; for the developer, it can serve as a domain model and a methodology. In both cases, it can be used as a communication medium, which people can understand, and by using it they can manipulate the "invisible computers" within their environment.

GAS defines a vocabulary of entities and functions (e.g. plugs, synapses etc.), a set of configuration rules (for interactively establishing associations between artifacts), and a technical infrastructure (the GAS middleware). Parts of GAS lie with the artifact manufacturers in the form of design guidelines and APIs, with people-composers in the form of configuration rules and constraints for composing artifact societies and with the collaboration logic of artifacts in the form of communication protocol semantics and algorithms.

5 Application Engineering Paradigm

To achieve the desired collective functionality, based on the GAS architectural style, one forms synapses by associating compatible plugs, thus composing applications using entities as components. Two levels of plug compatibility exist: Direction and data type compatibility. According to direction compatibility output or I/O plugs can only be connected to input or I/O plugs. According to Data type compatibility, plugs must have the same data type to be connected via a synapse. However, this is a restriction that can be bypassed using value mapping in a synapse. No other limitation exists in making a synapse. Although this may result in the fact that meaningless synapses are allowed, it has the advantage of letting the user create associations and cause the emergence of new behaviors that the artifact manufacturer may never consider. Meaningless synapses can also be seen as having much in common with runtime errors in a program, where the program may be compiled correctly but it does not manifest the behavior desired by the programmer.

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 (sensors/actuators), plug/synapse interoperability and a semantic service discovery protocol.

5.1 Synapse-Based Programming

The introduction of synapse-based programming has been driven mainly by the introduction of the previously discussed enabling paradigm, component software. Traditional software programs have followed the procedure call paradigm, where the procedure is the central abstraction called by a client to accomplish a specific service. Programming in this paradigm requires that the client has intimate knowledge about the procedures (services) provided by the server. However, this kind of knowledge is not possible in an ambient ecology because it is based on artifacts that may come from different vendors and were separately developed. That is why ambient ecology programming requires a new programming paradigm, which we have called synapse-based programming.

In synapse-based programming, synapses between artifacts are not implicitly defined by procedure calls but are explicitly programmed. Synapses represent the glue that binds together interfaces of different artifacts. The basis for synapse-based programming is typically, the so-called, Observer design pattern [18]. The Observer pattern defines a one-to-many dependency between a *subject* object and any number of *observer* objects so that when the *subject* object changes state, all of its *observer* objects are notified and updated automatically. This kind of interaction is also known as *publish/subscribe*. The subject is the publisher of notifications. It sends out these notifications without having to know who its observers are.

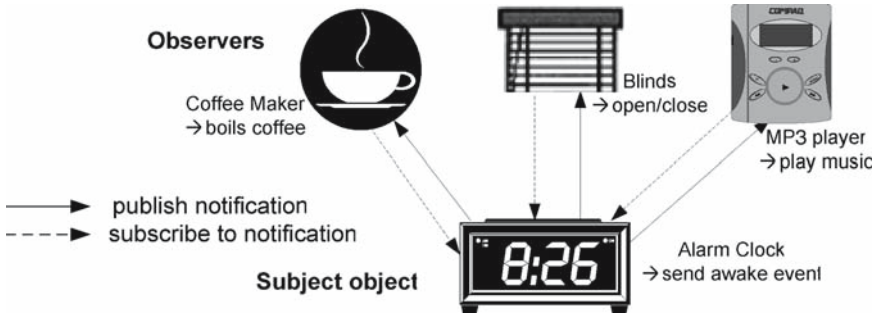

Fig. 9.2 Publish/subscribe model for implementing synapses

The strength of this event-based interaction style lies in the full decoupling in time, space and synchronization between publishers and subscribers [19]. Thus the relationship between subject and observer can be established at run time and this gives a lot more programming flexibility.

In a UbiComp space (see for example the scenario outlined in Section 1.1), the Observer pattern can be applied as in the following diagram (see Fig. 9.2). The Coffee Maker, Blinds, and MP3 player are the *observer* objects. The Alarm Clock is the *subject* object. The Alarm Clock object notifies its observers whenever an awake event occurs to initiate the appropriate service.

The observer pattern works like a subscription mechanism that handles callbacks upon the occurrence of events. Artifacts interested in an event that could occur in another artifact can register a callback procedure with this artifact. This procedure is called every time the event of interest occurs. The typical interfaces of software components have to be tailored for synapse-based programming — they have to provide subscription functions for all internal events that might be of interest to external artifacts. This part of the interface is often called the *outgoing interface* (associated with output plugs) of an artifact, as opposed to its *incoming interface* (associated with input plugs) that consists of all callable service procedures.

5.2 An Example

The following example refers to the motivating scenario discussed earlier in Section 1.1. Fig. 9.3 depicts the internal structure of a composite artifact with the constituent artifacts, their properties and the established synapses. The composition uses two source artifacts (eBook, eChair), one sink artifact (eDeskLamp) and one simple artifact (eDesk). The interconnection is accomplished with three synapses between properties of the constituent artifacts. For example, the *ReadingActivity* property associated with a eDesk artifact depends on the input properties defined as

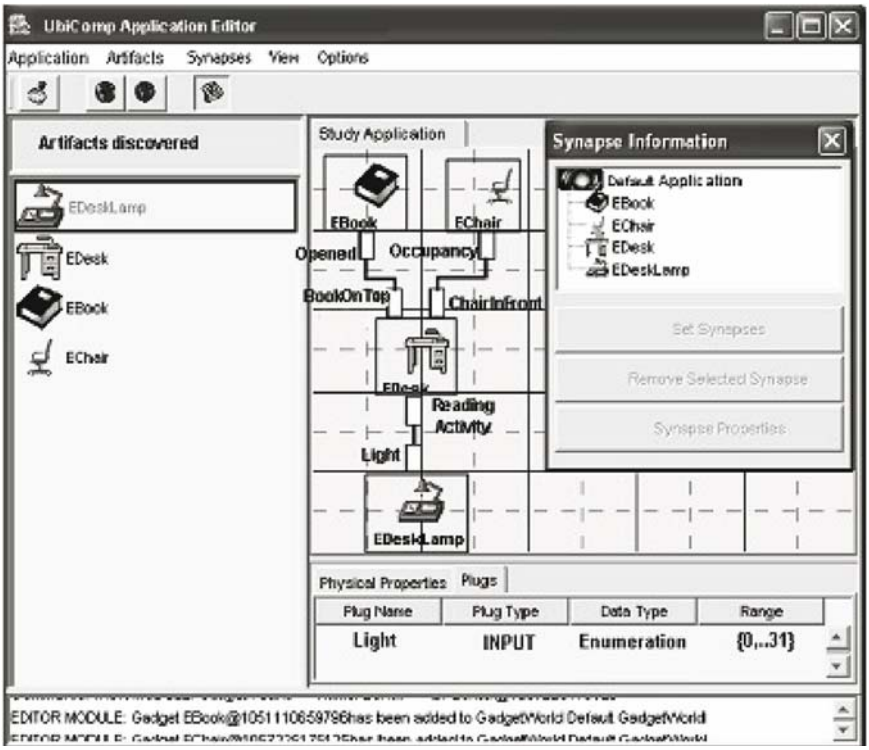

Fig. 9.3 An artifact composition implementing a UbiComp application

BookOnTop and *ChairInFront*; the later have been derived as relational properties between eDesk and the pair of eBook and eChair artifacts, respectively (see Figure 9-3).

This example illustrates the definition of a simple UbiComp application that we may call the eStudy application. The scenario that is implemented is as follows: when the particular chair is near the desk and someone is sitting on it and a book is on the desk and the book is open then we may infer that a reading activity is taking place and we adjust the lamp intensity according to the luminosity on the book surface. The properties and plugs of these artifacts are manifested to a user via the UbiComp Application editor tool [20], an end-user tool that acts as the mediator between the plug/synapse conceptual model and the actual system. Using this tool the user can combine the most appropriate plugs into functioning synapses as shown in Fig. 9.3.

Fig. 9.4 depicts the statechart diagram modelling the behavior of the participating artifacts in the eStudy. States and transitions for each artifact are shown as well as the use of superstates for modelling the behavior of the ambient ecology as a whole. Note that the modelling of the behavior of the artifact/ambient ecology helps us to decide upon the distribution of properties to artifacts and the establishment of

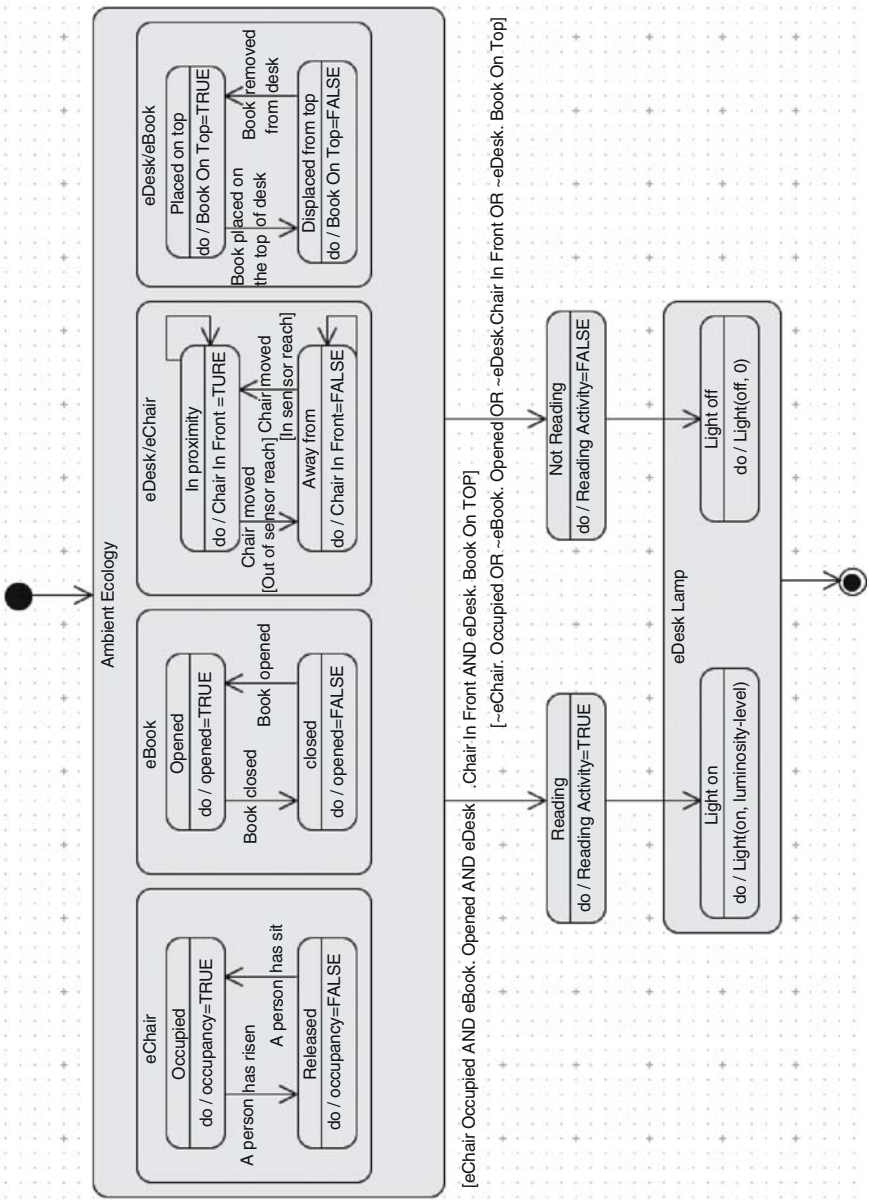

Fig. 9.4 Statechart diagram modelling the behavior of eStudy participating artifacts

synapses. For example, the states that refer to a relational property, like the *ChairInFront* property, identify the end-point plugs of a synapse.

An example of an execution scenario for the above application may have the following sequence of states (the latest defined property is given as underlined):

κ0: {*eChair.Occupancy* = *TRUE*; all other properties undefined}
Propagation applies for the *Occupancy* property
κ1: {eChair.Occupancy = TRUE; *eBook.Opened* = *TRUE*; all other properties undefined}
Propagation applies for the *Opened* property
κ2: {eChair.Occupancy = TRUE; eBook.Opened =TRUE; *eDesk.ChairInFront* = *TRUE*; all other properties undefined}
Derivation applies for the *ChairInFront*, property based on the propagated *Occupancy* property
κ3: {eChair.Occupancy = TRUE; eBook.Opened =TRUE; eDesk.ChairInFront = TRUE; *eDesk.BookOnTop* = *TRUE*; all other properties undefined}
Derivation applies for the *BookOnTop* property based on the propagated *Opened* property
κ4: {eChair.Occupancy = TRUE; eBook.Opened = TRUE; eDesk.ChairInFront = TRUE; eDesk.BookOnTop = TRUE; *eDesk.ReadingActivity* = *TRUE*;}
Evaluation applies for the *ReadingActivity* property based on a simple rule-based formula.
κ5: {eChair.Occupancy = TRUE; eBook.Opened = TRUE; eDesk.ChairInFront = TRUE; eDesk.BookOnTop = TRUE; eDesk.ReadingActivity = TRUE; *eDesk-Lamp.Light* = *On*};
Derivation applies for the *Light* property based on the propagated *Reading-Activity* property.

Although the above example is rather simple, it does demonstrate many of the features of our definitions. From the example, we see that composite artifacts provide an abstraction mechanism for dealing with the complexity of a component-based application. In a sense a composite artifact realises the notion of a "program", that is we can build a UbiComp application by constructing a composite artifact.

6 The Supporting Framework

6.1 GAS-OS Middleware

To implement and test the concepts presented in the previous sections we have introduced the GAS-OS middleware, which provides UbiComp application designers and developers with a runtime environment to build applications from artifact components. We assume that a process for turning an object into artifact has been followed [17]. Broadly it will consist of two phases: a) embedding the hardware

modules into the object and b) installing the software modules that will determine its functionality.

The outline of the GAS-OS architecture is shown in Fig. 9.5 (adopted from [21], where it is presented in more detail). The GAS-OS kernel is designed to support *accepting* and *dispatching* of messages, managing local hardware resources (sensors/actuators), and implementing the plug/synapse interaction mechanism. The kernel is also capable of managing service and artifact discovery messages in order to facilitate the formation of the proper synapses.

The GAS-OS kernel encompasses a P2P Communication Module, a Process Manager, a State Variable Manager, and a Property Evaluator module which are briefly explained in Table 9.1.

Extending the functionality of the GAS-OS kernel can be achieved through plug-ins, which can be easily incorporated into an artifact running GAS-OS, via the 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 [22]. In that way, high-level descriptions of services and resources are possible independent of the context of a specific application, facilitating the exchange of information between heterogeneous artifacts as well as the discovery of services.

GAS-OS can be considered as a component framework, which determines the interfaces that components may have and the rules governing their composition. GAS-OS manages resources shared by artifacts and provides the underlying mechanisms that enable communication (interaction) between artifacts. For example, the proposed concept supports encapsulation of the internal structure of an artifact and provides the means for composition of an application, without having to access any of the code that implements the interface. Thus, our approach provides a clear separation between computational and compositional aspects of an application, leaving

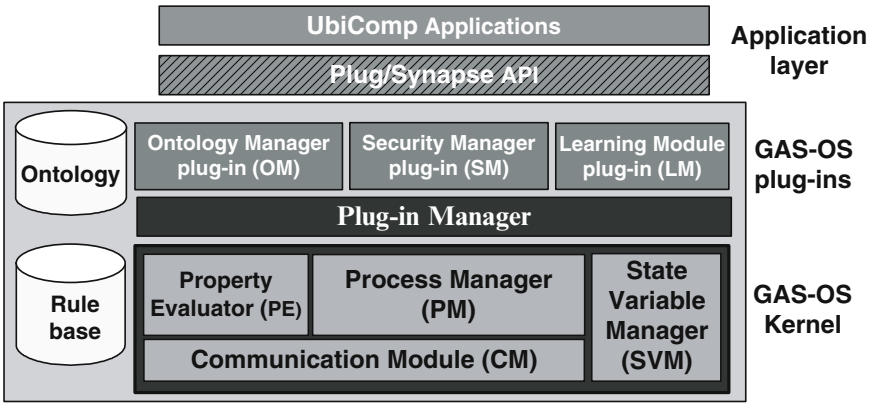

Fig. 9.5 GAS-OS modular architecture

Table 9.1 Modules in the GAS-OS Kernel

Module	Explanation
Communication Module (CM)	The P2P Communication Module is responsible for application-level communication between the various GAS-OS nodes.
Process Manager (PM)	The Process Manager is the coordinator module of GAS-OS 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 from other artifacts to improve communication performance when service discovery is required.
State Variable Manager (SVM)	The State Variable Manager handles the runtime storage of the artifacts' state variable values, reflecting both the hardware environment (sensors/actuators) at each particular moment (primitive properties), and properties that are evaluated based on sensory data and P2P communicated data (composite properties).
Property Evaluator (PE)	The Property Evaluator is responsible for the evaluation of the artifacts' composite properties according to its Functional Schema. In its typical form the Property Evaluator is based on a set of rules that govern artifact transition from one state to another. The rule management can be separated from the evaluation logic by using a high-level rule language and a translator that translates high-level rule specifications to XML, which can be exploited then by the evaluation logic.

the second task to ordinary people, while the first can be undertaken by experienced designers or engineers.

The benefit of this approach is that, to a large extent, the systems design is already done, because the domain and system concepts are specified in the generic architecture; all people have to do is realize specific instances of the system. Composition achieves adaptability and evolution: a component-based application can be reconfigured with low cost to meet new requirements. The possibility to reuse devices for numerous purposes - not all accounted for during their design - provides opportunities for emergent uses of ubiquitous devices, where this emergence results from actual use.

6.2 ECA rule modeling pattern

Event-Condition-Action (ECA) rules have been used to describe the behavior of active databases [23]. An active database is a database system that carries out prescribed actions in response to a generated event inside or outside of the database. An ECA rule consists of the following three parts:

- Event (E): occurring event
- Condition (C): conditions for executing actions
- Action (A): operations to be carried out

An ECA rule modeling pattern is employed to support autonomous interaction between artifacts that are represented as components in a UbiComp environment. The rules are embedded in the artifacts, which invoke appropriate services in the environment when the rules are triggered by some internal or external event. Following this design pattern, the applications hold the logic that specifies the conditions under which actions are to be triggered. The conditions are specified in terms of correlation of events. Events are specified up front and types of events are defined in the ontology. The Process Manager (PM) subscribes to events (specified in applications logic) and the Property Evaluator (PE) generates events based on data supplied by the State Variable Manager (SVM) and notifies the Process Manager when the subscribed events occur. When the conditions hold, the Process Manager performs the specified actions, which could consist of, for example, sending messages through the P2P Communication Module (CM) and/or request an external service (e.g., toggling irrigation, calling a Web service, etc.).

Consider, as an example, the smart plant application discussed in Section 1.1, which enables interactions similar to communication between plants and people. The main artifact is the ePlant. The ePlant decides whether it needs water or not using its sensors readings (e.g. thermistors and soil moisture probe) and the appropriate application logic incorporated in it. A second artifact is a set of keys that is "aware" as to whether it is in the house or not. If we assume that the user always carries her keys when leaving home then the keys can give us information about whether the user is at home or not. User presence at home can be determined by using the Crossbow MICA2Dot mote [24] placed in the user's key-fold. When the user is at home, any signal from the mote can be detected by a base station and interpreted as presence. Fig. 9.6 depicts the flow of information between the middleware components applying the ECA pattern. The ECA rule defined for the ePlant artifact in the above application is:

- E: PlantDryEvent
- C_1: location = HOME A_1:SendNotifyRequest(DRY_PLANT)
- C_2: location != HOME A_2:SendSMSRequest(DRY_PLANT)

The Location Plug actor in Figure 9-6 represents the user location context supplied by the key artifact. The application requires interaction with a couple of artifacts that will respond to the requests produced by the ePlant artifact, such as a notification device (e.g. TV, MP3 player) and a mobile phone for sending/receiving SMS messages corresponding to the DRY_PLANT code.

By employing an ECA rule modeling pattern we can program applications easily and intuitively through a visual programming rule-editing tool. We can modify the application logic dynamically since the application logic is described as a set of ECA rules and each rule is stored independently in an artifact.

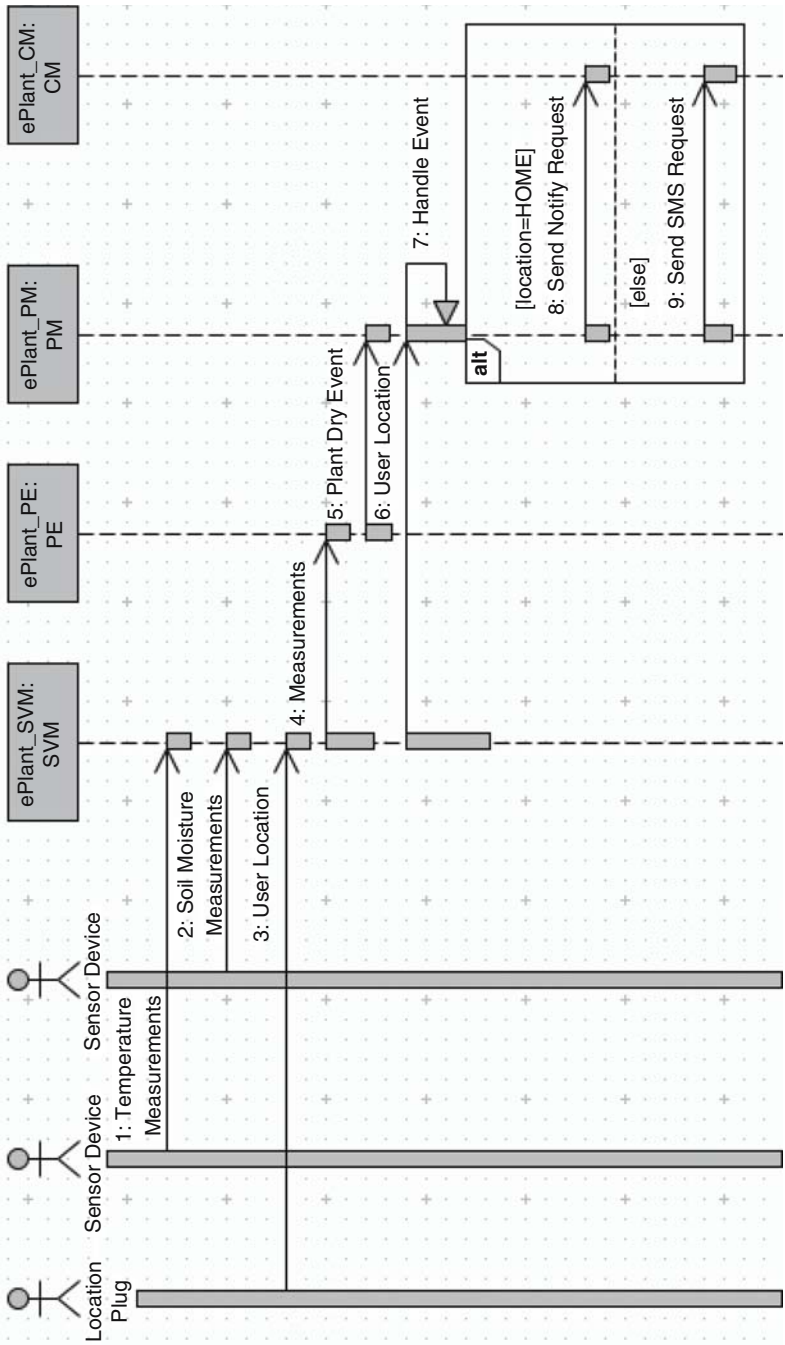

Fig. 9.6 Interaction sequence in the smart plant application

6.3 Tools

A toolbox complements this framework and facilitates management and monitoring of artifacts, as well as other identified eEntities, which when collectively operating, define UbiComp applications. The following tools have been implemented:

- The *Interaction Editor*, which administers the flexible configuration and reconfiguration of UbiComp applications by graphically managing the composition of artifacts into ambient ecologies, the interactions between them, in the form of logical communication channels and the initiation of the applications (see Figure 9-3);
- The *Supervisor Logic and Data Acquisition Tool* (SLADA), which can be used to view knowledge represented into the Ontology, monitor artifact/ecology parameters and manage dynamically the rules taking part in the decision-making process in co-operation with the rule editor;
- The *Rule Editor*, which provides a Graphical Design Interface for managing rules, based on a user friendly node connection model. The advantage of this approach is that rules will be changed dynamically without disturbing the operation of the rest of the system and this can be done in a high-level manner.

In Fig. 9.7 we show as an example the design of the *NotifyUserThroughNabaztag* rule for the wish-for-walk awareness application defined as part of our motivating scenario (see Section 1.1). The rule consists of two conditions combined with an AND gate. The first condition checks the 'wish-for-walk' incoming awareness state. The second condition checks whether the user to be notified is in the living room (this state is inferred by an artifact - an instrumented couch). The rule, as designed, states that when the conditions are met that the user will be presented with the awareness information through an artifact, called *Nabaztag*, as this object will be probably in his/her field of vision.

Using a rule editor for defining application business rules emphasizes system flexibility and run-time adaptability. In that sense, our system architecture can be regarded as a reflective architecture that can be adapted dynamically to new requirements. The decision-making rules can be configured by users external to the execution of the system. End-users may change the rules without writing new code. This can reduce the time-to-production of new ideas and applications to a few minutes. Therefore, the power to customize the system is placed in the hands of those who have the knowledge/need to do it effectively.

6.4 Implementation

The prototype of GAS-OS has been implemented in J2ME (Java 2 Micro Edition) CLDC[1] (Connected Limited Device Configuration), which is a very low-footprint

[1] java.sun.com/products/cldc

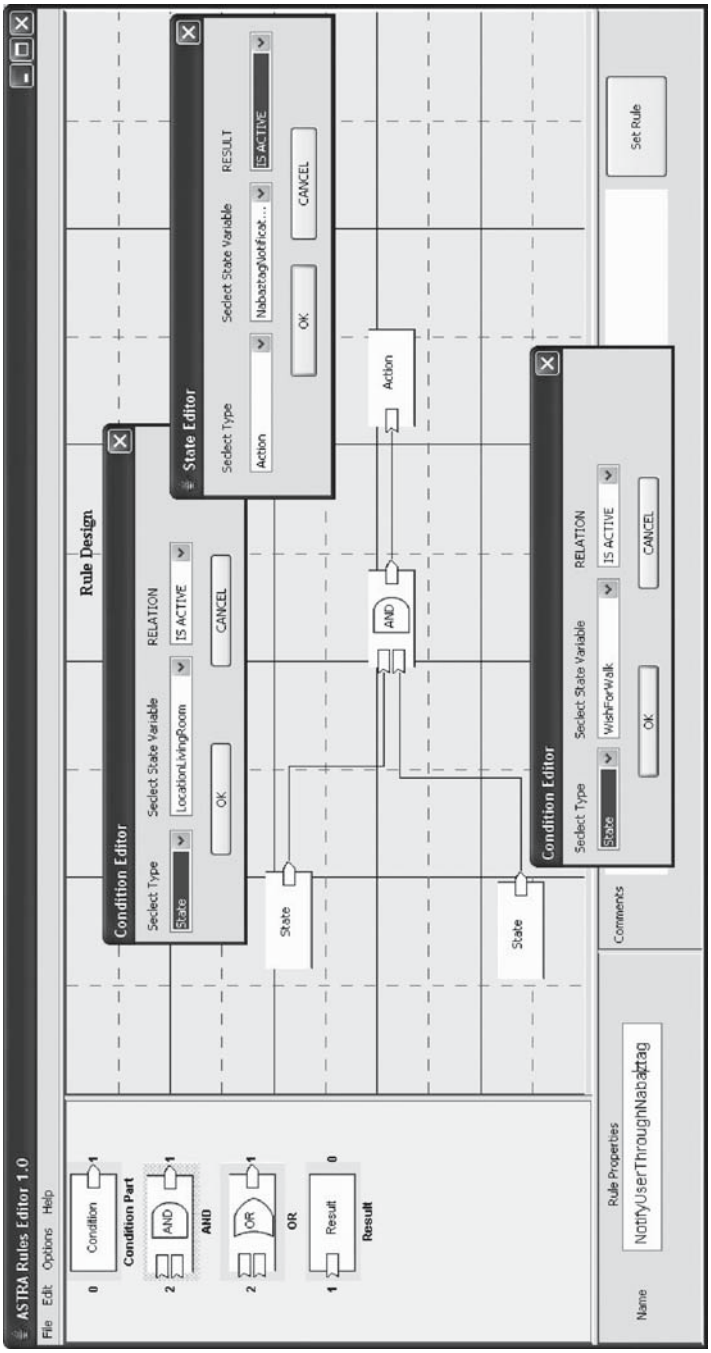

Fig. 9.7 Designing the 'NotifyUserThroughNabaztag' rule for the wish-for-walk awareness application

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. The use of Java as the platform for the middleware decouples GAS-OS from typical operations like memory management, networking, and so forth. Furthermore, it facilitates deployment on a wide range of devices from mobile phones and PDAs to specialized Java processors.

Up to now, GAS-OS 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 artifacts communicate using wired/wireless Ethernet, overlaid with TCP/IP and UPnP (Universal Plug and Play) middleware programmed in Java. The inference engine of the Property Evaluator is similar to a simple Prolog interpreter that operates on rules and facts and uses backward-chaining with depth-first search as its inference algorithm.

We have implemented 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. There are three actors involved in the eRDP:

1. the *Resource Consumer* (RC), an artifact that has need for a resource, possibly with specific attributes and initiating for that purpose a resource discovery process,
2. the *Resource Provider* (RP): an artifact that provides a resource and also advertises the location and attributes of the resource to the Resource Directory, provided that there is one,
3. the *Resource Directory* (RD): an artifact that aggregates resource information into a repository on behalf of artifacts that are less equipped.

The *Resource Directory* (RD) is an optional component of the discovery protocol and its aim is to improve the performance of the protocol. In the absence of an RD, the *Resource Consumers* (RC) and *Resource Providers* (RP) implement all of the functions of the RD with multicast/broadcast messages, with the optional and undeterministic use of resource cache within each artifact. When one or more RDs are present (see Fig. 9.8), the protocol is more efficient, as an RC or RP uses unicast messages to the RDs.

[2] www.embedded-web.com/

[3] www.imsys.se/documentation/manuals/snap_spec.pdf

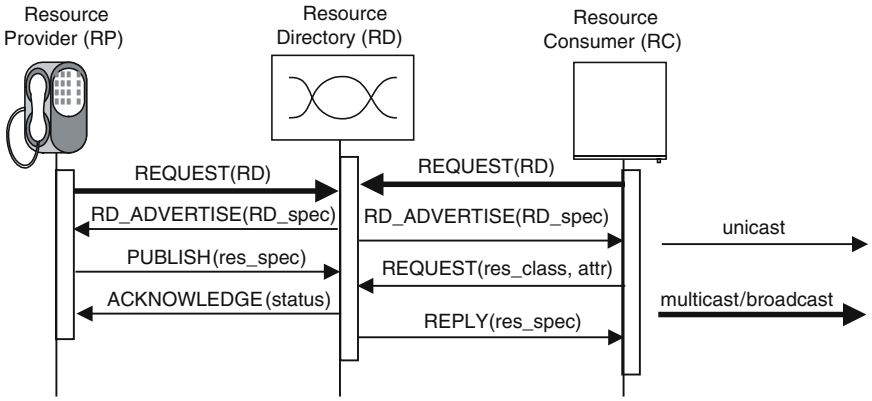

Fig. 9.8 eRDP with a RD facility

```
<res_spec>
<res name> eDeskLamp </ res name>
<res class> light </res class >
<res id> eRDP:PLUG:CTI-eDLamp-ONOFF_PLUG </res id >
<res location> 150.140.30.5 </res location>
<res data> <attrName="power" type="bool" value="false"
<attrName="luminocity" type="integer", value="10"
</res data>
<res timestamp> 4758693030 </res timestamp>
<res expiry> Never </res expiry>
</res_spec>
```

Fig. 9.9 XML description of eDeskLamp resource specification

This service discovery 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. kXML is used for parsing XML messages.

If we assume, for example, that the synapse between the eDesk and eDeskLamp artifacts is broken. When this happens, the system will attempt to find a new artifact having a plug that provides the service classified as "light". The eDesk system software is responsible for initiating this process by sending a message for service discovery to 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 eDeskLanp resource specification is shown in Fig. 9.9.

When the system software of an artifact receives a service discovery message it queries its local service repository in order to find if this artifact has a plug that provides the service "light". When the answer is positive it returns, as a reply, the description of this service as a resource specification. If such a service is not provided by the artifact itself, the repository is checked in order to find if another artifact, with which the queried artifact has previously collaborated, provides such a service. If this is not the case, the query message for the service discovery may pass to another artifact.

7 Related Work

Service composition in ubiquitous computing environments has been investigated mainly by automatic or user-assisted composition of semantically annotated web services [25–27]. Two other important cases in this area are the approach developed by Fujitsu Laboratories and the University of Maryland called *Task Computing* [28] and the approach developed by Xerox PARC called *Recombinant Computing* [29]. In the former case the functionality of the environment is exposed as semantic web services, which the user can in turn discover and arbitrarily compose. In the latter case, a model is used where each computational entity in the network is treated as a separate component. Central to this approach is the notion that users must be the final arbitrator of the semantics of the entities they are interacting with because applications could not have *a priori* knowledge of all of the devices they may encounter.

There are systems that permit users to aggregate and compose networked devices for particular tasks [30]. However, those devices are not context aware, acting more as service providers; e.g., web services usually in the UPnP (universal plug and play) style. Approaches to modelling and programming such devices for the home have been investigated, where devices have been modelled as collections of objects [31], as web services [32], and as agents [33]. However, there has been little work on specifying, at a high level of abstraction, how such devices would work together at the application level.

A palpable assembly as a dynamic combination of devices and services with a programmatic representation that includes both a component and a connector has been proposed [34]. This is similar to our conceptual model for programming ambient ecologies; however, our approach offers a complete framework consisting of an architectural style, a programming model, a supporting middleware and a toolset as opposed to an architectural prototype in the case of the palpable assembly concept.

Other research efforts are emphasizing the design of ubiquitous computing architectures. In the Disappearing Computer initiative, the project "Smart-Its" [35] aimed to develop small devices, which, when attached to objects, enable their

association based on the concept of "context proximity". The objects are usually everyday devices (e.g. cups, tables, chairs etc) equipped with various sensors as well as a wireless communication module, such as RF or Bluetooth. The goal is to add smartness to real-world objects in a post-hoc fashion by attaching small, unobtrusive computing devices to them. While a single *Smart-It* is able to perceive context information from its integrated sensors, a federation of ad hoc, connected *Smart-Its* can gain collective awareness by sharing this information. However, the "augmentation" of physical objects is not related in any way with their "nature", thus the objects tend to become just physical containers for the computational modules they host.

8 Conclusions and Discussion

The ultimate goal of ambient ecologies is to serve people; this undoubtedly entails interaction with, and control by, users – dealing with errors, customizing settings, etc. On the other hand, much of its management (e.g., configuration, handling of faults and adaptation to context) will be done autonomously and people will not be aware of it. An ambient ecology may involve large – even enormous – populations of entities that deploy themselves flexibly and responsibly in a working environment. An entity may be a hardware device, a software agent or an infrastructure server; for some purposes it may be a human; it may also be an aggregation of smaller entities.

The advent of ambient ecologies will change the way we conduct our everyday activities by gradually introducing artifacts that are able to perform local computation, to collaborate with each other and to interact in an adaptive way with the user. A research agenda is needed that will facilitate a user-centered evolution of this new (Ambient Intelligence) environment by defining the conceptual framework and developing an integrated component platform, tools and design methods for people involvement. Research is also required to span across all layers ranging from infrastructure to applications. More specifically, the following multidisciplinary efforts are required to:

- Develop an open framework for conceptualizing the ecologies of devices and services. This framework may consist of a set of concepts implemented as an ontology, a description of capabilities implemented as basic and higher level behaviors and a novel interaction metaphor implemented as a language.
- Research on adaptation mechanisms aimed at understanding how the properties of self-configuration, self-optimization, self-maintenance and robustness arise from or, depend upon the behaviors, goals and self-* properties of individual artifacts, the interactions between them and the context of the application.
- Conceptualize heterogeneity by developing and testing theories of ontology alignment to achieve task-based semantic integration of heterogeneous devices and services.
- Understand the structure and behavior of ambient ecologies and design adaptable and evolvable ecology architectures

- Develop the necessary components and services, including APIs, to interface with existing hardware modules and communication protocols, ontology based knowledge representation and decision-making mechanisms, learning mechanisms, purpose based border negotiation and privacy enforcement and composeable interaction components.
- Develop test-bed applications to demonstrate the capabilities of ambient ecologies.

Research must address these issues from three different perspectives:

1. The theoretical perspective, which focuses upon concepts and models that capture the behavior of ambient ecologies at varying levels of abstraction.
2. The engineering perspective, which focuses upon the architectural challenges posed by the heterogeneous and dynamic nature of their synthesis.
3. The experience perspective, which focuses upon how people might share a world with artifact ecologies.

As is the case with every new technology, the major issue that research and development efforts must address is that of adoption. People are usually reluctant to give up on the habits and procedures they feel comfortable with unless the reward is high. UbiComp systems have great potential, but the application that will pave the way for their adoption has not been engineered yet. Things are made worse by the fear of privacy infringement that is developing among people as they become aware of the ability of novel artifacts to record, process and transmit huge volumes of information, much of it beyond the direct perception or control of people.

Ambient ecologies are complex systems; their global behavior results from local interactions between small collections of artifacts having some kind of property or task-based proximity. Thus, the evolution of ecology behavior, or structure, cannot be programmed. It seems that people will have to learn to co-exist with complex artifact ecologies, which they will only influence, but not be able to command. Consequently, the major goal of our research is to build systems that are at the same time pro-active and understandable, transparent and adaptable, robust and evolvable; thus, they enable people to balance on the thin line between asking and acting.

Although much work still needs to be done, in this work we have attempted to define ambient ecologies, to specify design patterns and programming principles, and to develop the infrastructure to provide a paradigm of application engineering and the tools to support ambient ecology designers, developers and end-users.

Acknowledgements Part of the research described in this chapter was conducted in the EU Funded e-Gadgets (IST-25240) and ASTRA (IST-29266) projects; the authors wish to thank their fellow researchers in those consortiums.

References

1. Loke S. W., 2006, Context-aware artifacts: two development approaches, *IEEE Pervasive Computing*, **5**(2):48–53.

2. Bluetooth, 2008, *The official Bluetooth Website*, Information available at *http://www.bluetooth.com/*, accessed February 2008.
3. IEEE 802.15.4, 2003, IEEE Standard for Wireless Medium Access Control (MAC) and Physical Layer (PHY), Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs), IEEE Computer Society.
4. IEEE 802.11, 1997, IEEE Standard for Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specification, IEEE Computer Society.
5. Norman, D., 1999, *The Invisible Computer*, MIT Press.
6. Bergman, E., 2000, *Information Appliances and Beyond*, Morgan Kaufmann Publishers.
7. Wooldridge, M., and Jennings, N.R, 1995, Intelligent agents: Theory and Practice, *Knowledge Eng. Rev*. **10**(2):115–152.
8. Szyperski C., 1998, *Component Software, Beyond Object-Oriented Programming*, ACM Press, Addison-Wesley, NJ.
9. Kameas, A., et al., 2003, An architecture that treats everyday objects as communicating tangible components, in: Proceedings of the first IEEE International Conference on Pervasive Computing and Communications (PerCom03), IEEE CS Press, pp. 115–122.
10. Wand, Y., and Weber, R., 1990, An ontological model of an information system, *IEEE Transactions on Software Engineering*, **16**(11):1282–1292.
11. Bunge, M., 1977, *Treatise on Basic Philosophy: Volume 3: Ontology I: The Furniture of the World*, Reidel, Dordrecht.
12. Bunge, M., 1979, *Treatise on Basic Philosophy: Volume 3: Ontology II: A World of Systems*, Reidel, Dordrecht.
13. Goumopoulos, C., Christopoulou, E., Drossos, N., and Kameas, A., 2004, The PLANTS system: enabling mixed societies of communicating plants and artefacts, in: Proceedings of the 2nd European Symposium on Ambient Intelligence (EUSAI 2004), Springer LNCS 3295, pp. 184–195.
14. Fowler, M., and Scott, K., 1999, *UML Distilled Second Edition, A Brief Guide to the Standard Object Modeling Language*, Addison Wesley.
15. Harel, D., 1987, Statecharts: a visual formalism for complex systems, *Science of Computer Programming*, **8**(3):231–274.
16. Norman, D. A., 1988, *The Psychology of Everyday Things*, Basic books, New York.
17. Kameas, A., Mavrommati, I., and Markopoulos, P., 2005, Computing in tangible: using artifacts as components of Ambient Intelligence Environments, in: *Ambient Intelligence*, Riva, G. Vatalaro, F. Davide, F. and Alcañiz M. (eds.), IOS Press, pp. 121–142.
18. Gamma, E., Helm, R., Johnson, R., and Vlissides, J., 1995, *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading Mass., Addison Wesley.
19. Eugster, P., Felber, P., Guerraouli, R., and Kermarrec, A., 2003, The many faces of publish/subscribe, *ACM Computing Surveys*, **35**(2):114–131
20. Mavrommati, I., Kameas, A., and Markopoulos, P., 2004, An editing tool that manages device associations in an in-home environment, *Personal and Ubiquitous Computing*, Springer-Verlag, **8**(3–4):255–263.
21. Drossos, N., Goumopoulos, C., and Kameas, A., 2007, A conceptual model and the supporting middleware for composing ubiquitous computing applications, *Journal of Ubiquitous Computing and Intelligence*, American Scientific Publishers(ASP), **1**(2):174–186.
22. Christopoulou, E., and Kameas, A., 2005, GAS Ontology: an ontology for collaboration among ubiquitous computing devices, *International Journal of Human-Computer Studies*, **62**(5):664–685.
23. Paton, N. W., and Diaz, O., 1999 Active Database Systems, *ACM Computing Surveys*, **31**(1):63–103.
24. CrossBow Mica2Dot Data Sheet, Wireless Microsensor, Document Part Number: 6020-0043-04 Rev A, *http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/6020-0043-04_A_MICA2DOT.pdf*, accessed February 2008.

25. Higel S., O'Donnell T., and Wade V., 2003, Towards a natural interface to adaptive service composition, in: Proceedings of the 1st International Symposium on Information and Communication Technologies, ACM Series, pp. 169–174
26. Ben Mokhtar, S., Georgantas, N., and Issarny, V., 2005, Ad hoc composition of user tasks in pervasive computing environments, in: Proceedings of the 4th Workshop on Software Composition (SC 2005), Springer LNCS 3628, pp. 31–46.
27. Charif, Y., and Sabouret, N., 2006, An Overview of Semantic Web Services Composition Approaches, *Electronic Notes in Theoretical Computer Science*, **146**(1):33–41.
28. Masuoka, R., Labrou, Y., Parsia, B., and Sirin, E. 2003, Ontology-enabled pervasive computing applications, *IEEE Intelligent Systems*, **18**(5):68–72.
29. Edwards, W.K., Newman, M.W., and Sedivy J.Z., 2001, The Case for Recombinant Computing, Technical Report CSL-01-1, Xerox Palo Alto Research Center, Palo Alto, CA.
30. Kumar, R., Poladian, V., Greenberg, I., Messer, A., and Milojicic, D., 2003, Selecting devices for aggregation, in: Proceedings of the IEEE Workshop on Mobile Computing Services and Applications, IEEE CS Press, pp. 150–159.
31. Jahnke, J. H., d'Entremont, M., and Stier, J., 2002, Facilitating the programming of the smart home, *IEEE Wireless Communications*, **9**(6):70–76.
32. Matsuura, K., Hara, T., Watanabe, A., and Nakajima T., 2003, A new architecture for home computing, in: Proceedings of the IEEE Workshop on Software Technologies for Future Embedded Systems, IEEE CS Press, pp. 71–74.
33. Ramparany, F., Boissier, O., and Brouchoud, H., 2003, Cooperating autonomous smart devices, in: Proceedings of the Smart Objects Conference (sOc'2003), pp. 182–185.
34. Ingstrup, M., and Hansen, K. M., 2005, Palpable assemblies: dynamic service composition for ubiquitous computing, in: Proceedings of the Seventeenth International Conference on Software Engineering and Knowledge Engineering (SEKE'2005), edited by William C. Chu et al., pp. 632–638.
35. Holmquist, L. E., Mattern, F., Schiele, B., Alahuhta, P., Beigl, M., and Gellersen, H.-W., 2001, Smart-its friends: A technique for users to easily establish connections between smart artifacts, in: Proceedings of the 3rd International Conference on Ubiquitous Computing (UBICOMP 2001), Springer-Verlag LNCS 2201, pp. 116–122.