YABS: A Domain-Specific Language for Pervasive Computing based on Stigmergy

Peter Barron Vinny Cahill

Distributed Systems Group,
School of Computer Science and Statistics,
Trinity College, Dublin 2,
Ireland.
Peter.Barron, Vinny.Cahill@cs.tcd.ie

Abstract

This paper presents YABS, a novel domain-specific language for defining entity behavior in pervasive computing environments. The programming model of YABS is inspired by nature and, in particular, the observations made by the French biologist Grassé on how social insects coordinate their actions using indirect communication via the environment, a phenomenon that has become known as stigmergy. Following this approach yields a simple yet expressive language that abstracts the complexities of dealing with the variety of underlying technologies typical of pervasive computing environments and that facilitates the incremental construction and improvement of solutions while providing high-level constructs for defining the behavior of entities and their coordination. We show how YABS has been used to program a number of pervasive computing applications both deployed and simulated.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Specialized application languages

General Terms Design, Languages.

Keywords Pervasive Computing, Stigmergy.

1. Introduction

The physical integration of computers into the real world is one of the main challenges of pervasive computing. Kindberg et al. [25] argue that to address such a challenge requires the provision of high-level abstractions that allow components to sense and interact with the physical environment without the difficulties of dealing with low-level devices such as sensors and actuators. In this paper we present one such abstraction in the form of a domain-specific language, called YABS. YABS abstracts the complexities of dealing with the underlying technologies to provide a mechanism for defining the behavior of autonomous entities in a pervasive computing environment and their coordination.

The inspiration for the approach stems from the observations made by the French biologist Grassé on how social insects coordinate their actions using indirect communication via the environment. Holland et al. [22] showed that this phenomenon, which is known as *stigmergy* [20], provides a mechanism that allows the environment to structure itself through the activities of entities within the environment. The state of the environment and the current distribution of the entities within it, determines how the environment and the entities will change in the future. The result is a highly decentralised mechanism for coordinating the behavior of entities that is both robust and extensible.

YABS provides high-level abstractions designed to allow developers harness the same coordination mechanisms used by social insects to develop pervasive computing environments. The approach is encapsulated in a framework that is designed to both support and complement the use of stigmergy, allowing for the incremental construction and improvement of solutions and aiding the ad-hoc composition of pervasive computing environments. It employs a distributed event-based architecture organised in a peer-to-peer fashion so that individual entities are decoupled from each other. YABS is used as a meta-level language to define the behavior of individual entities and, implicitly, to specify how they are to coordinate their behavior within the environment. A base language is used to define the sensors and actuators used in the framework. In this paper we focus on the design of YABS. A more detailed description of the framework, which is called Cocoa, and the low-level abstractions that it provides can be found in [3, 4].

As a domain-specific language YABS provides a simple yet expressive language for defining and coordinating entity behavior that would otherwise be more difficult to express with a general purpose language. Specifying such a language ensures that the right level of abstraction can be found between the underlying technologies and the high-level abstractions sought for the physical integration of computers into the real world. The approach also assists in the rapid development and the reuse of components.

The paper is organised as follows: in section 2, we outline the domain of pervasive computing, describing the challenges of developing such environments and how using stigmergy can aid in their development. The YABS language is described in section 3. Section 4 provides details of a number of pervasive computing applications, both deployed and simulated, that have been developed with YABS. Section 5 looks at some of the related work in the field. Finally, section 6 presents conclusions and future work.

2. A Stigmergic Approach

In this section we outline some of the challenges of developing future pervasive computing environments and consider how the use of stigmergy can aid in their construction.

GPCE'06 October 22–26, 2006, Portland, Oregon, USA.

2.1 Pervasive Computing

Technology for pervasive computing is reaching a point where it is becoming possible to convert many everyday environments into interactive spaces. For example, in education it is used to support students attending lectures [1], in offices to assist workers in meetings [24] or in group collaboration sessions [36]. It has also been used in scientific laboratories to support the work of scientists [21], and in the home to ensure the efficient usage of resources [27]. In addition, pervasive computing has also been used to support the elderly in the home [28].

Typically, these types of interactive spaces have been designed from the ground up to support the anticipated needs of their users and to evaluate the technology deployed in the space. The environments are usually preinstalled and maintained over the period in which they are in use. However, as Edwards et al. [17] point out, it is unrealistic to expect all pervasive computing environments to be constructed in this manner. They believe that physical spaces are more likely to evolve accidentally into pervasive computing environments as technology is incorporated into the space. Kindberg et al. express a similar view in [25] where they argue that pervasive computing systems will tend to form accidentally over the medium-to-long term. A recent study [37] also draws the same conclusions.

This would suggest that pervasive computing environments need to be assembled in a more ad-hoc fashion than has previously been the case. Current approaches to pervasive computing system design do not readily apply themselves to this form of development. They appear to be more conceptually centralised approaches that focus their efforts around coordinating the resources of specific geographical locations. For instance, in the Stanford Interactive Workspaces project [24] all interactions for the iRoom environment are mediated through the iROS system, which is responsible for managing the resources at that particular location. It does not allow the ad-hoc interaction and coordination of components at locations other than those that have been predetermined and where the system has already been installed. While this method may work well for developing pervasive computing environments from the ground up it would be less appropriate to composing environments in the more ad-hoc manner anticipated by Edwards et al. [17].

In this form of development pervasive computing environments emerge from spaces through the migration and accumulation of technology at a particular location. There is no master plan that guides the development or any expert overlooking the construction of the environment. It evolves through ordinary people moving and integrating new technology into a space. Where an environment emerges depends totally on how the occupants arrange the technology. Unlike, for instance, Aura [18] or Gaia [33] the installation of a pervasive computing system into a physical location is not a prerequisite for the environment to form or to operate at those locations. For these types of environments devices and applications need to be able to spontaneously interact at any time and at any place without having to mediate their behavior through a central authority at each specific location.

In allowing environments evolve in an ad-hoc fashion it makes it possible for them to emerge at hotspots of activity where users require and want to use them and so give an illusion that they are always available. Kindberg et al. [25] observe that through the inclusion of new technology or the rearrangement of what is already there that new usage models can be adopted by the occupants. The environment continuously changes and adapts to how those in the space use it and rearrange the technology. It is not bounded by the same constraints normally imposed on a system specifically designed to operate at a particular location; it changes as users move technology into or out off the space. Changes may occur slowly over a period of time or at a much faster rate depending on how the space is being used.

The system software for managing such an environment needs to take a different approach than has otherwise been deemed necessary. In these types of environments the components - devices, physical artifacts, software components, and services - that comprise the system must be organised in a highly decentralised manner. Unlike many other systems [18, 24, 33] there should be no central component in the environment to manage access to resources or coordinate how different components in the environment interact with each other or with those using the environment. The components are in fact the system and as such have to be able to spontaneously interact with each other to coordinate their behavior in a distributed manner. The environment can be though of as a collection of interacting components that through their ad-hoc interactions can form a pervasive computing environment capable of providing services for those occupying the environment.

In order to support the ad-hoc composition of pervasive computing environments in this manner we have identified a number of requirements necessary for development.

R1: Support the physical integration of components into the environment. It is important to abstract the complexities of dealing with the real world to ensure that components can easily be integrated into the physical environment.

R2: Support the autonomy of components. Each component should be an independent entity with the ability to move through the environment unrestricted. It does not depend on other components to operate and is responsible for managing and coordinating its own behavior within the environment.

R3: Support spontaneous interoperability between arbitrary components. It has be to assumed, due to the accidental manner in which these environments form and the heterogeneous nature of the components that comprise them, that components will have little prior knowledge of the other parts of the environment with which they will interact. Consequently, it is important that components be able to discover and spontaneously interact with each other.

R4: Support the decentralised coordination of component behavior. To ensure that a large collection of autonomous components can form a coherent environment it is necessary to be able to coordinate their behavior. As a centralised approach is not feasible it is necessary to provide a decentralised mechanism for coordinating the behavior of components within the environment.

R5: Provide a scalable solution. With the expected large number of components it can be assumed that, as these environments grow, the intensity of interactions will increase with the number of components and users inhabiting the space. Thus, scalability is of particular concern for these types of systems.

R6: Ensure the robust behavior of the system. In pervasive computing failure is considered to be a norm and not an exception. It is therefore necessary for a system to be able to absorb the underlying changes to ensure the environment can behave in a robust fashion.

R7: Support incremental construction and improvement of solutions. In developing the environments described above it has to be expected that they will evolve incrementally over a period of time. It cannot be assumed that an environment of this nature can be developed or installed in one go. It is therefore necessary to allow the incremental construction and improvement of solutions.

R8: Mobility. With the expected migration of technology and the anticipated movement of users it has to be assumed that there will be a high degree of mobility.

R9: Adaptability. It must be assumed that the mobility of users and technology will lead to a situation where the environment is

continuously changing. To overcome this situation it is necessary for components to adapt their behavior to use whatever is available in the immediate environment.

2.2 Stigmergy

In 1959, the French biologist, Grassé observed that social insects could coordinate their actions through the environment without having to directly communicate with each other. They do this using a phenomenon known as *stigmergy* [20]. He also noticed that the local interactions between insects resulted in the emergence of colony-wide behavior. Holland et al. [22] showed that stigmergy provides a mechanism that allows the environment to structure itself through the activities of the entities within the environment. The state of the environment, and the current distribution of entities within it, determines how the environment and the entities will change in the future. This approach provides a robust, self-organising environment, which allows entities to coordinate their behavior in a highly decentralised manner. It is important to stress that individual entities have no particular problem solving knowledge, and that coordinated behavior emerges due to the actions of the society. It also worth noting that while no direct communication is used between individual entities, communication is still maintained through the medium of the environment.

The trail-laying and trail-following used by many species of ants [7] when foraging for food is a classic example of the use of stigmergy in nature. Ants deposit pheromones on their way back from a food source. Foraging ants follow such trails. The process has been shown to be self-organising [13] and capable of optimizing on the shortest path to the food source [19]. The nest building of social wasps [38] is another example of stigmergy used in nature. Nests are built up from wood fibers and plant hairs and cemented together with salivary secretions. These are then moulded by the wasp to form the different parts of the nest. Wasps coordinate the construction of a nest by each individual observing the local structure of the nest and deciding where to build the next part of the nest. Another example is the corpse gathering behavior seen in some species of ants. Worker ants pick up corpses in the nest and drop them in locations of higher corpse concentrations to form piles of corpses in a process which acts to clean the nest.

The potential of social insects has not gone unnoticed. Several research initiatives have looked to harness the coordination mechanisms used by these types of natural systems to develop techniques and algorithms for solving a range of computer-related problems. The ant foraging behavior has inspired a problem-solving technique called ant colony optimization (ACO) [16]. It has been applied to the traveling salesman problem [15], routing in communication networks [11], and vehicle routing [10]. The concept of stigmergy has also had a significant influence on the area of behavior-based robotics, where Beckers et al. believe that the "fit between stigmergy and behavior-based robotics is excellent" [6]. Brueckner et al. [9] has also used stigmergy in agent-based systems to coordinate the actions of agents to find global patterns across spatially distributed real-time data sources. Mamei and Zambonelli [26] have also relied on the concept of stigmergy to coordinate collections of interacting agents in an interactive environment.

2.3 Using Stigmergy in a Pervasive Computing Environment

The mechanisms used to organise these types of systems and the collective behavior that emerges from them is also an appealing construct for pervasive computing.

The idea of simple insects, with little memory or ability to exhibit any real intelligence, maps well to pervasive computing where devices with limited resources are spread across the environment. The large number of devices expected to be deployed into our society matches the scale at which these colonies of social insects work. The constant interaction between components of a pervasive computing environment also ties in neatly with how social insects interact with each other. A colony of social insects are in many ways very similar to a pervasive computing system where large collections of interacting entities roam across the environment.

By applying the principles of stigmergy to such a large collection of interacting components it should be possible to harness the same mechanisms of coordinating large collections of interacting entities as social insects utilise, and in so doing, provide a predominately decentralised method of organising and controlling groups of autonomous components in a pervasive computing environment. This is achieved by components moving through the environment and using local interactions, mediated via the environment, to coordinate their actions with other parts of the system. As with the social insects the components of a pervasive computing system modify their local environment to influence the subsequent behavior of other components.

It should also be noted that in a stigmergic system the environment acts as a shared medium through which entities communicate. Each entity manipulates the local environment in a way that is recognisable to other entities in the surrounding area. The alterations performed by the entities are universally understood by all entities involved making it possible for them to spontaneously interact with each with little or no prior knowledge of the other entities. Used in pervasive computing the environment should also provide a common interoperation model capable of allowing components to interact in a spontaneous manner. The environment acts as a common shared service to all components making it possible to seamlessly integrate any arbitrary component into the interactive environment. It allows for the impromptu interoperability that Edwards et al. [17] advocates is necessary for the successful operation of a pervasive computing environment.

Another advantage of using techniques based on stigmergy in pervasive computing is that it allows a system to harness the same robust behavior as that seen in colonies of social insects [6]. This is partly due to the indirect communication that allows the decoupling of components within the system. Applied to pervasive computing it leads to fewer dependencies between components making the overall system less fragile and more stable to disturbances in the environment. Such an approach also provides a very flexible approach to adapting to a changing environment. This can seen in other projects that have used stigmergy, such as the adaptive routing protocols developed by Caro et al. [11], or in particular stigmergic models [7, 8] based on task allocation or the division of labor observed in some species of social insects.

A very evident characteristic of biological systems using stigmergy is the scale at which these organisms work. A swarm of raiding army ants (Eciton burchelli) may contain up to 200,000 workers [7]. A key to their ability to scale is that all interactions in a stigmergic process are mediated through the local environment. By using this fact in pervasive computing it should be possible to obtain a system that scales. In these cases entities are only interested in observing the state of the environment local to them as it is only this part of the environment that influences their behavior. Applying the same process to pervasive computing would severely reduce interactions with distant locations, therefore, allowing the system to scale more gracefully. Both Satyanarayanan et al. [34] and Kindberg et al. [25] have identified the usefulness of applying such an approach.

It can also be argued that the autonomous nature of individuals allows such systems to be totally extensible, in that, new entities can always be added and updated when necessary. This is possible due to the loose coupling associated with entities of stigmergic systems and their ability to adapt to a changing environment. Applied to pervasive computing the autonomous nature of individual com-

ponents and the loose coupling between them ensures a pervasive computing system can always grow and decay with the addition of new components and the upgrade or removal of old ones. Harnessing these properties makes it possible to develop components separately and for them to be installed into the environment when ready, hence making it feasible to construct a pervasive computing environment incrementally over a period of time.

It would appear in principle that the concept of stigmergy can be used to address the majority of the requirements stated in section 2.1. R1 - physical integration - is the only requirement that cannot be directly satisfied via the use of stigmergy. This is addressed in section 3 by providing a high-level programming abstraction for defining entity behavior. The next section presents a model based on the principle of stigmergy that can be used to develop pervasive computing environments.

2.4 A Stigmergic Model for Pervasive Computing

In modeling a system based on stigmergy there are three things that need to be determined, the first is the environment that collections of interacting entities will use to coordinate their behavior, secondly, are the entities that will use the environment, and thirdly, the means for the individual entities to sense this environment, determine how they react to it, and manipulate it.

In this case we propose to use the general principles of stigmergy to create a model for pervasive computing where context information from environmental sensors provides the common environment for the indirect communication between entities. The social insects observed by Grassé [20] are represented as entities in the model. An entity can represent a person, place, or object as defined by Dey [14]. Entities roam across the environment and act on it by changing their behavior to modify the local environment. The changes in the environment are subsequently reflected in the context information derived from the environmental sensors. Coordinated behavior arises from entities observing their local environment and reacting to the resulting context information according to some rules.

2.4.1 The Local Environment

Context information derived from local sensors is used to describe the situation of each entity. The context of an entity e_i in the environment at time t is represented by $C_{e_i}(t)$. Figure 1(a) represents the context of every entity in the pervasive computing environment at a particular time, i.e., the global context $C_G(t)$ of the environment and is defined by equation 1, where $E(t)$ is the set of all entities that exist at time t.

$$C_G(t) = \{C_{e_i}(t) : e_i \epsilon E(t)\} \tag{1}$$

Crucially however all the information contained in $C_G(t)$ is not required by each individual entity, as the behavior of an entity is only dictated by the context of its local environment. Figure 1(b) illustrates a subset of the context information relevant to an entity. It represents the local environment and defines the entity's *contextual view* $C_{V_{e_n}}(t)$, as defined in equation 2. It holds all context information in $C_G(t)$ that is relevant to the situation of entity e_n at time t. An entity's context $C_{e_i}(t)$ is included in entity e_n's contextual view if the entity is within a certain proximity of e_n. The notion of proximity is used to define what is local to the entity. This is captured in equation 2 where the function $L(e_i, e_n)$ is used to determine proximity and returns $true$ if entity e_i is within the required proximity of entity e_n.

$$C_{V_{e_n}}(t) = \{C_{e_i}(t) : C_{e_i}(t) \epsilon C_G(t) \wedge L(e_i, e_n) = true\} \tag{2}$$

2.4.2 Defining Entity Behavior

The behavioral set B, shown in figure 1(c) and defined in equation 3, represents a finite set of behaviors that entities can use to change their local environment. For example, a light can either turn itself on or off, or a jukebox can play music, pause, or stop playing. The behavioral set defines how entities can change their behavior to modify the environment.

$$B = \{b : b \text{ is a behavior of an entity}\} \tag{3}$$

2.4.3 Reacting to the Local Environment

The last stage manages how each individual entity adapts its behavior to reflect changes in the local environment. Equation 5 defines the function M for mapping $C_{V_{e_n}}(t)$ onto $\mathcal{P}(B)$[1]. $C_{V_e}(t)$, defined in equation 4, represents the collection of all contextual views. The function maps the entity's context information from the local environment onto a behavior, thus initiating a stigmergic response to the environment. For example, if set B is defined by figure 1(c) and $C_{V_{e_2}}(t)$ is the contextual view of e_2 at time t then function M could possibly map $C_{V_{e_2}}(t)$ onto $\mathcal{P}(B)$ as follows $M(C_{V_{e_2}}(t)) = \{b_3\}$

$$C_{V_e}(t) = \{C_{V_{e_i}}(t) : e_i \epsilon E(t)\} \tag{4}$$

$$M : C_{V_e} \rightarrow \mathcal{P}(B) \tag{5}$$

The proximity function L, the behavioral set B, and the M function provide three primitives that define how individual entities behave in response to changes in the local context state of the environment. Over time system-level behaviors may emerge as different entities change their behavior in response to the changing state of their local environment. In order to define the three primitives - L, B, and M - in the framework that we have designed, a domain-specific language, called YABS is provided.

3. YABS - Defining Entity Behavior

YABS is a meta-level language, in that, it defines how individual entities in a pervasive computing environment coordinate their behavior. A base language, Java in this case, is used to define the sensors and actuators used in the Cocoa framework [3, 4]. The foundations of the language are built upon the stigmergic model, in particular the three primitives defined in section 2.4 - L, B, and M - form the basis of the language. Together, they define how an entity is to behave and how it is to coordinate its activities within a pervasive computing environment.

3.1 Overview

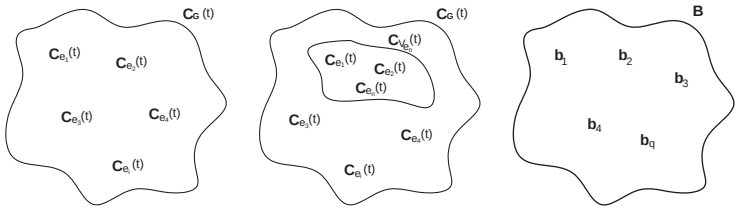
The language uses an interpreter that takes a script containing a description of the desired behaviors and translates them into intermediate objects that the framework uses to represent the behavior of individual entities. The behaviors described in the script characterise how a particular type of entity behaves in an environment and can be reused for all entities of that type.

```
desklight extends light{...}
```

For instance, the code shown above defines an entity of type desklight. Any entity that can be categorised as a desklight may use the behaviors described in the script to regulate how it behaves. The script defines these behaviors in terms of the three primitives - L, B, and M - outlined in section 2.4, as described in more detail in the coming sections.

There is also a requirement to be able to tailor the behaviors defined in a script for different sub-types of entity. For example, a

[1] The power set of behavioral set B.



(a) Global context (b) Contextual view (c) Behavioral set

Figure 1. Using Stigmergy in an Pervasive Computing Environment

desklight can be categorised as a light but may behave in a subtly different manner. To manage these aspects the language allows a script to inherit from another script. This allows the script to inherit behaviors and adjust them to meet the requirements for that sub-type of entity. In the example above, the *desklight* inherits from *light*. The approach helps to promote the reuse of code but also aids in rapid development and incremental construction of pervasive computing environments, in that, it is possible to reuse existing functionality and extend it. The semantics of inheritance are described in later sections.

The hierarchical structure formed by the inheritance relationships are influenced by the presence of four predefined scripts - entity, object, person, place - which enforce a structure on the hierarchy. The latter three - *object*, *person*, and *place* - represent the entities in the stigmergic model with *entity* being the base script for these. These scripts define functionality common to these categories of entity.

The following sections describe the structures used to define entity behavior, in so doing, focuses on how the three primitives - L, B and M - are defined in a script and on the semantics used for inheritance from a script.

3.1.1 Proximity Function

In the language, L, the proximity function can be defined as either a radius, a polygon, or a symbolic area around an entity. Any context emanating from this region will be inserted into the entity's contextual view as described in section 2.4. An example proximity function specification can be seen in the sample code below where the proximity is set to be a 5 meter radius around the current entity.

```
proximity(5)
```

In the next example pairs of coordinates are used to define a polygon: the unit of measurement is meters, and the reference point for the polygon is the position of the entity.

```
proximity(-5,-5,-10,5,-10,20,10,20,10,5,5,-5)
```

The code below shows the use of symbolic proximity, where a predefined area can be used to specify the proximity around an entity. This type of proximity is useful when there is a strong definable boundary, such as a room or building.

```
proximity("F32")
```

Using this type of proximity function helps filter out interference from entities which are nearby, but are not relevant to the current situation, i.e. are outside the boundary. Typically, a symbolic proximity is mapped to an absolute location, or relative location which is specified beforehand (in the base language).

The proximity function, L, must be defined for each entity so that it can determine the scope of its local environment. This is typically achieved in the language by declaring a proximity function in each script to specify L for entities of that type. However, it may also be inherited from a base script if not defined in the extended script. A sub script can also, if the current definition of L in the base script is not compatible, redefine the proximity function.

3.1.2 Behavioral Set

B, the *behavioral set*, defines the set of possible behaviors that can be performed by the entity. The actual implementation of a behavior is not provided in the script but in Java following a particular API defined by the framework [3, 4]. The script specifies behaviors that a particular type of entity can perform by declaring a behavior with a corresponding reference to the implementation of that behavior. For example, in the sample code below behaviors *on* and *off* are declared with reference to the classes that implement the behavior for the entity. In this case, it is the behaviors for turning a light on and off. When the behavior is invoked it executes the Java code that defines the specific behavior that allows the entity to manipulate the environment.

```
behavior on = "ie.tcd.cs.lighton"
behavior off = "ie.tcd.cs.lightoff"
```

The declaration of a behavior for an entity can also be inherited from a base script. For instance, the desklight could inherit the *on* and *off* behaviors from light. It could then use them along with the behaviors it declares to define how desklight entities are to behave. It is also possible, if a particular inherited behavior does not suit, to associate the behavior with a different implementation. This may be required if a sub-type of an entity uses different actuators to manipulate the environment. An example of how this can be achieved is shown in the sample code below where the *on* behavior is defined to use a different implementation for some script.

```
on = "ie.tcd.cs.desklighton"
```

3.1.3 M Function

The primary function of the language is to identify the set of contextual stimuli that influence an entity and to map them onto behaviors that allow entities to modify the local environment. In the stigmergic model, defined in section 2.4, the M function provides the means of mapping an entity's local environment onto B the behavioral set. To use the M function in the language it is first necessary to identify the parts of the environment that act as stimuli to the entity. This is achieved by defining a set of predicates specifying the context information that is of interest to the entity. These are true when matched by information in the entity's current contextual view and can be used to determine the entity's behavior. An example of one such predicate can be seen in the sample code below.

```
context bobperson
bobperson.person = "Bob"
bobperson.location = "Bob House, F32"
bobperson.activity = any
bobperson.time = "lunch time"
bobperson.job = "teacher"
bobperson.music = "rock"
```

In this example the context called *bobperson* is declared. The keyword *person* defines the predicate *bobperson* as identifying a person with the name of *Bob*. It is also possible to identify a *place* or an *object* and by using the *any* operator to specify any person, any object, or any place. The *location* keyword indicates a

position or area that is of interest to the predicate. It is possible to use GPS coordinates, relative coordinates, or symbolic information such as the *"Bob House, F32"*, as used in this example. The *activity* keyword defines what the target entity is doing. This could be a person walking to work, a desklight turned on, or a printer printing. In this example the predicate is interested in Bob doing any activity. The *time* keyword indicates a period, or point in time. This can be specified as an absolute time such as *"Thu Mar 18 21:58:36 GMT 2004"*, or symbolic time such as *"lunch time"*. It must be noted that while symbolic context information can be used the vocabulary needs to be agreed upon beforehand to the extent that symbolic information is matched exactly by the framework.

YABS uses Dey's [14] concept of primary and secondary context information. Primary context information being the identity, location, activity and time of the entity, while secondary context information describes any other information which helps define an entity's situation. In the script secondary context information is declared by specifying any key/value pairing. In the coding sample above the *bobperson* predicate specifies two such pieces of context information. The first describing what job Bob does and the second specifying what music he likes to listen to.

Once the required context predicates have been declared it is necessary to map the entity's contextual view on the behavior set by identifying the stimuli in the local environment that effect the entity's behavior and determining how the entity should modify its behavior in response. How this is achieved in the language can be seen below.

```
map[bobperson, darkroom]onto{
  on()
}
```

In this case, the mapping is accomplished when the context predicates *bobperson* and *darkroom* are found to be matched in the entity's current contextual view. This would indicate that Bob is in a place called *"Bob Institute, F32"* with little light. On obtaining a match for this predicate the behavior can then be triggered for the entity, which in this case is the *on* behavior for a light. The general structure of the mapping statement allows the developer to specify one or more context predicates that must all hold in the entity's current contextual view for the mapping to be successful.

The declaration of context predicates and the definition of mappings for an entity can also be inherited from base scripts. For instance, a script could inherit context predicates *bobperson*, and *darkroom* and use them along with other context predicates it has declared to define mappings. It is also possible, if a particular inherited predicate does not suit, to redefine a predicate. An example of how this can be achieved is shown below, where the location context on the *bobperson* is changed from *"Bob Institute, F32"* to *"Bob Institute, F35"*.

```
bobperson.location ="Bob House, F35"
```

It should be noted that reassigning values of inherited context predicates also effects how the mappings defined in the base scripts are performed. While this is a desirable attribute which allows a script to modify how mappings are triggered in the base scripts care needs to taken to avoid unwanted behaviors.

Mappings are also inherited from base scripts in the same way as behaviors and context predicates and can be used by the extended script to dictate how the entity is to behave along with the other mappings defined in the script. Tailoring how the inherited mappings operate is achieved either by modifying the values of the predicates or by overwriting the mappings to change the behaviors that are mapped. The sample code shown below provides an example of how to overwrite a mapping to change the behaviors that are triggered. In this case, the mapping shown in the previous example is overwritten to change the behavior it triggers from *on* to *halfon*. This is specified by using the same context predicates in the mapping statement as in the inherited mapping.

```
map[bobperson, darkroom]onto{
  halfon()
}
```

3.2 Mapping

The previous sections outlined the basic structure of YABS and have demonstrated how to trigger a behavior for an entity on encountering specific stimuli described by fragments of context information. Since the recognition of what is happening at one particular instance in time is often not sufficient to capture the broader sense of what has occurred, YABS also provides a more expressive means of performing the mappings that can also take into account what has been observed beforehand. Influenced by the work of Allen [2] and that of Pinhanez et al.'s interval scripts [31] the section looks at another method that models the relationships between intervals of time to capture these observations and define entity behavior.

An interval is a length of time marked off by two distinct points in time representing the start and end of the interval. In [2], Allen introduced a model that made it possible to describe the relationship between two intervals of time. He showed that there are 13 possible such relationships, as summarised in figure 2.

Given any two intervals of time it is possible to use one of the relationships illustrated in figure 2 to describe how they are related. For instance, in taking a story such as the one below:

> John was not in the room when I touched the switch to turn on the light.

it is possible to use Allen's interval temporal logic to describe the above story as:

```
S overlap or meet L
S is before, meet, is imeet, or ibefore R
```

where S is the time of touching the switch, L is the time the light was on, and R is the time that John was in the room.

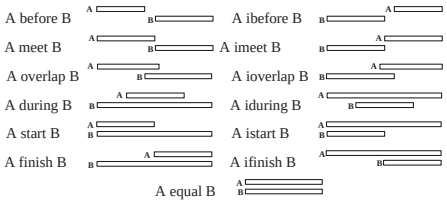


Figure 2. Interval relationships

The importance of Allen's work stems from its ability to provide a means of describing the relationships between intervals without having to explicitly mention the interval duration or specifying the relationships between the interval's extremities. These characteristics are of value when it comes to capturing the broader sense of what is happening in an environment. It is used by the language to describe the temporal relationships between observations so that when the described relationship is satisfied the mapping can be triggered to modify the behavior of the entity. The method is especially useful when you also consider the imprecise nature of the environments in which the entities are anticipated to operate.

Thus, YABS uses the primitive relationships defined by Allen to describe temporal relationships between intervals of time. Entity behavior is then triggered on observing the intervals in the correct temporal sequence. The context predicates described in section 3.1.3 are used to define the duration of the interval. The start of an interval is determined when the context predicate becomes true, and the end is denoted on it becoming invalid. The interval is

deemed active between these two distinct points in time. The script specifies the relationships between intervals by defining a sequence of context predicates. Once the intervals have occurred, as indicated by the script, the appropriate behavior is triggered.

For the purpose of illustration an example is used to explain in more detail the use of interval temporal logic. The sample code shown below demonstrates the use of intervals in a mapping statement.

```
map[contextA, contextB][contextB]onto{...}
```

It uses the context predicates, *contextA* and *contextB*, to describe two different intervals of time. The relationship between the intervals can be defined as *contextA start contextB*, as per Allen's interval temporal logic. The square brackets demarcate the start and end of the intervals, and the sequence defines the relationship between them.

In determining whether a mapping has been triggered the framework investigates each subsequent contextual view to determine what intervals are active. An interval is deemed active when the context predicate is found to hold true in the entity's current contextual view. The interval becomes inactive when the predicate can no longer be found to be true. When the intervals are found to have been active in the correct temporal sequence, as described in the script, the behavior is triggered. In the example above, the relationship is satisfied when both *contextA* and *contextB* have been active in the same period with *contextB* remaining active for a period after *contextA* becomes inactive at which point the mapping can be triggered.

It is also feasible to use the other 12 relationships defined by Allen in the mapping statement. For instance, in the example below *contextA is before contextB*. The symbol *[ ]* indicates that no interval is active for this period of time.

```
map[contextA][][contextB]onto{...}
```

The use of Allen's interval temporal logic provides YABS with a more expressive means of performing mappings that takes into account what has occurred beforehand and not just what has occurred at a single point in time. While it does increase the complexity of the script, the increased expressiveness gained by using Allen's temporal logic outweighs the additional difficulty in defining the behavior of entities. It should also be noted that the mapping statements described in this section use the same semantics for inheritance as those described in section 3.1.3.

3.3 Passing Context Information

The parameters for the behaviors are constructed from the context information which has been defined beforehand within predicates declared in the script, or from context information that is held in the entity's current contextual view. The sample code below demonstrates how parameters are passed in behaviors.

```
context peter
peter.music="folk"
...
play(peter.music)
```

The context information *peter.music* is passed to the *play* behavior when invoked. For context information to be passed the behavior must first be implemented to take a parameter. The context information passed must also match what is expected by the behavior otherwise the invocation will fail. It is, of course, also possible to pass multiple parameters if required.

3.4 Using Embedded Functions

To access information held in the entity's contextual view the language uses embedded functions that allow the data in the contextual view to be analysed and context information deduced without having to directly manipulate the view. The implementations of the embedded functions are not provided by YABS but in the base language. To use a particular embedded function it must first be declared in the script or within one of the base scripts of the script. An example of how an embedded function is declared can be seen below. In this example the embedded function *majority* is declared with a reference to the Java classes that implement the function.

```
efunction majority = "ie.tcd.cs.dsg.cocoa.Majority"
```

The sample code shown below illustrates how such an embedded function is used in a script.

```
context someperson
someperson.location = "Bob Institute, F32"
someperson.music=any
...
play(majority(someperson))
```

The *majority* function, in this case, allows a developer to determine the value of a piece of context information that is most prevalent in the entity's current contextual view. In this example the context *someperson* acts as a filter for the *majority* function excluding any entities in the current contextual view that are not in the location "*Bob Institute, F32*" or do not have any music preference. From the remaining context the *majority* function is used to determine the majority value of the secondary context *music* which is then passed the *play* behavior.

A number of embedded functions are currently supported. These include the *majority* function mentioned above, and also the *minimum*, *random*, *minority*, and *average* embedded functions. These functions are declared in the *entity* script so that they may be accessed by all scripts. Developers can still extend the functionality of the language by either implementing their own embedded functions or changing the behavior of an embedded function by selecting a different implementation of that function as can be seen in the code below.

```
majority = "ie.tcd.cs.dsg.cocoa.MajorityNewImpl"
```

The *maximum* and *minimum* functions allow the developer to retrieve the maximum or minimum values for context information in the entity's current contextual view. The *random* function randomly selects a piece of the context information in the entity's current contextual view. The *minority* function provides opposite functionality to that of the majority function. The *average* function determines the average value for a particular context.

Currently the embedded functions are restricted to being used to derive the parameters to be passed to behaviors. It should also be noted that an embedded function can be called within another if required.

3.5 Redefining L

While the proximity function, L, is usually set for the lifetime of an entity it can, if the circumstances require it, be modified during an entity's lifetime. Modification of the proximity function is, for example, sometimes required by entities when moving from one environment to another. For example, when a PDA moves from a busy street to an office it may redefine L to take into account its new environment. The sample code provides one such example of the proximity function being redefined. Typically, redefining the proximity function is done within the mapping statement as the example illustrates.

```
map[contextB][contextA,contextB]onto{
   displayPicture()
   proximity(8) //8 meter radius around entity.
}
```

4. Developing Entities with YABS

To demonstrate how the high-level abstractions provided by YABS can be use to define and coordinate entity behavior in a pervasive computing environment a prototype implementation [3] of the language was used in combination with the Cocoa framework to develop a series of application scenarios.

4.1 Westland Row

Westland Row is a street located near our laboratory. The street is about 250 meters long, and accommodates a number of cafes, news agents, shops, bars, and a train station. It is a busy street, with commuters, shoppers, cars, and buses using it on a daily basis. A wireless ad-hoc network has been deployed on Westland Row, with a number of nodes placed along the street. The nodes form a sparse population of wireless network nodes and can be configured to create a variety of network models. The network is part of a project [5] investigating the use ad-hoc networks in urban areas. Westland Row provides both a challenging, and an interesting testing ground for developing pervasive computing applications and for determining the effectiveness of YABS to develop and support the incremental construction and improvement of solutions in an urban environment.

Over a period of time we developed and deployed a small number of entities along the street. The following is a description of the current society:

Shop is used to represent the different shops and cafes along the street. It was one of the first entities to be deployed. The current implementation is quite simple, having no defined behavior. It would of course be possible to define a greater range of behavior for this type of entity but for now it serves only as a source of context information concerning the presence of a shop at some location. The shop entities ran on embedded PCs (using PC-104 technology) placed along the street.

Shopping assistant provides information about the different shops on the street. The implementation is based on the Firefox browser which ran on a laptop. The browser was used to display information provided by the shops. One behavior has been implemented for the shopping assistant entity, called *display*, it opens a web page on the Firefox browser. The behavior is triggered when someone is nearby and when information is available to display. The script defining this behavior is shown below.

```
shoppingassistant extends firefox{
  proximity(100) //Set L to radius of 100 meters.
 //Declaring context predicates.
  context SomePerson
  SomePerson.person = any
  context SomeShop
  SomePlace.place = any
  SomePlace.deals = any
  //Define mapping.
  map[SomePerson, SomeShop]onto{
    display(SomeShop.deals)
  }
}
```

Jukebox, as the name might suggest, is an mp3 player. The script, shown below, defines the behavior of the jukebox entity. The *play* behavior is triggered when a person is near the jukebox. The genre of music played depends on what the majority of people prefer to listen to. This is determined by observing the context information from punter entities, in particular the musical preferences of the entities. The *stop* behavior is triggered when there is no one in the vicinity of the jukebox entity. The jukebox entity runs on a laptop using the xmms multimedia player.

```
jukebox extends object{
  proximity(10) //Set L to radius of 10 meters
  //Declare behaviors for
  behavior play = "ie.tcd.PlayBehavior"
  behavior stop ="ie.tcd.StopBehavior"
  //Declaring context predicates.
  context SomePerson
  SomePerson.person = any
  SomePerson.music = any
  context JukeBoxPlay
  JukeBoxPlay.object = this.object
  JukeBoxPlay.activity = "play"
  context JukeBoxStop
  JukeBoxStop.object = this.object
  JukeBoxStop.activity = "stop"
  //Defining mappings.
  map [JukeBoxStop] [JukeBoxStop, SomePerson]onto{
    play(majority(SomePerson))
  }
  map[JukeBoxPlay, SomePerson] [JukeBoxPlay]onto{
    stop()
  }
}
```

Punter presents a person on the street, whether they are shopping, having coffee, or commuting to work. While the script for this entity is quite simple, due to the requirements of the scenario, it is still necessary to represent the pedestrian to allow other entities absorb their context.

In all we deployed two punters entities, five shop's representing some of the shops and cafes on the street, two shopping assistant entities associated with each person, and a jukebox located in one of the cafes halfway down the street. The shop entities along Westland Row remained passive to changes in their environment, which was as expected due to their limited implementation. The shopping assistant, sometimes carried around by people, would display information about the shops as they walk by. The jukebox entity, with its collection of music, would tailor the selection played depending on the users in it's vicinity.

While the scenario is simple, it serves to demonstrate that the high-level abstractions provided by YABS made it possible to add new entities, and remove or upgrade old ones from Westland Row without adversely effecting the rest of the street. This is primarily due to YABS supporting the autonomy and loose coupling between entities which allowed entities to be developed separately and to be installed into the environment when ready. These properties facilitated the incremental construction and improvement of solutions over the lifetime of the environment. Entities appearing and disappearing effect the behavior of other entities but not their correct operation. Moreover, no preinstallation of any software support is required. It should also be noted that the abstractions provided manage to separate the underlying low-level technology from the compositional side of developing pervasive computing applications. The clear separation allows developers using YABS to concentrate on implementing the behavior of individual entities. The approach aids the rapid development of environments and encourages the reuse of the underlying components - sensor and actuators - and of the scripts through the reuse and ability to extend the defined behavior.

4.2 Street Lights

The street-light application scenario demonstrates how system behavior can emerge from the local interactions of entities. The application is based on a set of street lights that you might find along the side of a road. The lights coordinate their activity, through observing their local environment, to ensure the street is sufficiently well lit for pedestrians to walk safely along the path. By default the lights are off to save energy but turn on in the presence of users, or half on when the street light beside is fully on. The effect is to have the light follow the person along the street. Figure 3 provides an illustration of the scenario.

The implementation of the street-light scenario requires the development of two entities; one to represent the street light, and another to present a pedestrian walking along the street.

Punter entity represents the pedestrian walking along the street. The implementation of the entity is based on the punter entities used in the previous scenario.

Streetlight entity represents the street lights along the walkway. The script defining the behavior of the street-light entity can be seen below. The *on* behavior is triggered when a person enters the local environment of the street-light entity. The *halfon* behavior is triggered when other street lights in the vicinity of the entity are *on*.

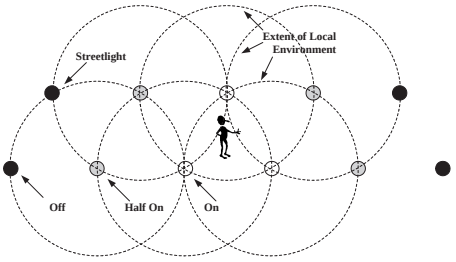


Figure 3. An illustration of the streetlight scenario.

The *off* behavior of the entity is triggered when the person leaves the local environment when no other street lights in the vicinity are on.

```
streetlight extends object{
  proximity(200)    //Set L to radius of 200 meters.
  //Declaring behaviors for streetlight entity.
  behavior on = "ie.tcd.StreetLightOnBehavior"
  behavior halfon = "ie.tcd.StreetLightHalfOnBehavior"
  behavior off = "ie.tcd.StreetLightOffBehavior"
  //Declaring context predicates
  context Walker
  Walker.person = any
  context SomeLightOn
  SomeLightOn.activity = "On"
  context LightOn
  LightOn.object = this.object
  LightOn.activity = "On"
  context LightOff
  LightOff.object = this.object
  LightOff.activity = "Off"
  context LightHalfOn
  LightHalfOn.object = this.object
  LightHalfOn.activity = "HalfOn"
  //Defining mappings for streetlight entity.
  map[LightOff][LightOff,SomeLightOn]onto{
    halfon()
  }
  map[LightOff][LightOff,Walker]onto{
    on()
  }
  map[LightHalfOn,SomeLightOn][LightHalfOn]onto{
    off()
  }
  map[LightHalfOn][LightHalfOn,Walker] onto {
    on()
  }
  map[LightOn,Walker,SomeLightOn][LightOn,SomeLightOn]onto{
    halfon()
  }
  map[LightOn, Walker][LightOn]onto{
    off()
  }
}
```

This scenario was deployed in a simulated environment with a single punter entity and ten street light entities placed along the street as illustrated in figure 3. While the script for the street-light entity is quite simile it still provides an expressive approach to defining its behavior that allows a meaningful environment to form. The scenario illustrates how emergent behavior, a bubble of light appearing to follow a pedestrian, arises from the individual actions of a collection of simple entities. Moreover, the scenario shows reuse of behavior, for the punter entity, in a different execution environment where its location is provided by a location simulator rather than the GPS sensor used in the Westland Row deployment.

5. Related Work

Domain-specific languages have already been used in pervasive computing to both define the behavior of components and to specify how those components are composed to build pervasive computing environments. Their use in these roles has helped pervasive computing systems to abstract the complexities of the underlying system, and aided the rapid development of applications for these types of environments.

The TEA [35] project have designed a language that defines the behavior of small devices such as mobile phones. Actions can be performed when entering a context, when leaving a context, and while in a certain context. In [32], Pinhanez has defined an interval scripting language that can define the temporal behavior of components. Based on PNF-networks [29] Pinhanez et al. have used the interval scripting language to create story-based interactive environments such as It/I [30]. The interval script describes the temporal relationships between different states of the environment, defining when to stop and to start other actions. Pinhanez's PNF-networks are based on Allen [2] temporal intervals.

RCSM [39] have define a language to control the behavior of components. RCSM is an object-based framework that uses an IDL-based language called CA-IDL to generate context-sensitive objects. These objects run on a customised ORB that supports the communication and context-awareness between objects. Developers use the CA-IDL language to define context variables that are used in temporal expressions within the script to trigger either local or remote method invocations on objects in the pervasive computing environment.

Gaia [33] uses high-level languages in a different manner to the above, in that, it uses scripts to compose the components of a pervasive computing system into applications that can be used by those in the environment. They use scripts, based on the interpreted language Lua [23], to describe how to combine the various components in the environment to form an application. In [12], SPREAD have taken a different approach where they have defined a programming abstraction based on the physical environment. The resulting programming model allows applications to be driven by the spatial orientation of objects in the environment.

The language defined in this paper applies some of the techniques used in the above projects to define entity behavior and to facilitate the emergence of pervasive computing environments from collections of autonomous entities. In particular, the language uses similar methods to RCSM [39] in declaring context and triggering actions but extends the approach to include the full range of temporal relationships used by Pinhanez [32] and defined by Allen [2]. However, the approach also includes a number of novel techniques for extending entity behavior and for facilitating the incremental development of pervasive computing environments. The approach also includes methods for tailoring the behavior of entities by the passing of context to the actions.

6. Conclusions and Future Work

This paper has introduced a novel domain-specific language, called YABS, for defining entity behavior in pervasive computing environments. Based on the stigmergic model outlined in section 3 the language provides high-level programming abstractions that combines expressiveness and simplicity with the ability to abstract the complexities of dealing with the underlying technologies. By focusing on defining entity behavior the language allows developers to concentrate their efforts on characterising the behavior of a pervasive computing environment rather than system development while also aiding the incremental construction and improvement of solutions over the lifetime of the environment.

Further work is still required to prove the approach. A major concern at the present time is the manner in which context information is defined. For one entity to understand another entity's situation it must understand the meaning of the context information that it provides. Currently YABS provides the concept of primary

context information, which is well understood and defined, and secondary context information which consists of key/value pairs that are open to interpretation. To tackle the problem we are looking at using alternative approaches to defining context information based on common ontologies.

References

[1] Gregory D. Abowd. Classroom 2000: An experiment with the instrumentation of a living educational environment. *IBM Systems Journal*, 38(4):508–530, October 1999.

[2] James F. Allen. Time and time again: the many ways to represent time. *International Journal of Intelligent Systems*, 6:341–355, 1991.

[3] P. Barron. *Using Stigmergy to Build Pervasive Computing Environments*. PhD thesis, Computer Science Dept, Trinity College Dublin, 2005.

[4] P. Barron and V. Cahill. Using stigmergy to co-ordinate pervasive computing environments. In *Sixth IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'04)*, pages 62–71, December 2004.

[5] P. Barron, S. Weber, S. Clarke, and V. Cahill. Experiences deploying an ad-hoc network in an urban environment. In *IEEE ICPS Workshop on Multi-hop Ad hoc Networks: from theory to reality*, 2005.

[6] R. Beckers, O.E. Holland, and J.L. Deneubourg. From location actions to global tasks: stigmergy and collective robotics. In *Artificial Life IV*, pages 181–189, 1994.

[7] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence From Natural to Artificial Systems*. Oxford University Press, 1999.

[8] E. Bonabeau, A. Sobkowski, G. Theraulaz, and J. Deneubourg. Adaptive task allocation inspired by a model of division of labor in social insects. In *Biocomputing and emergent computation: Proceedings of BCEC97*, pages 36–45. World Scientific Press, 1997.

[9] Sven A. Brueckner and H. Van Dyke Parunak. Swarming agents for distributed pattern detection and classification. In *AAMAS*, 2002.

[10] B. Bullnheimer, R.F. Hartl, and C. Strauss. Applying the ant system to the vehicle routing problem. In *2nd Metaheuristics International Conference (MIC-97)*, Antipolis, France, 1997.

[11] Gianni Di Caro and Marco Dorigo. Antnet: Distributed stigmergetic control for communications networks. *Artificial Intelligence Research*, 9:317–365, 1998.

[12] P. Couderc and M. Banatre. Ambient computing applications: an experience with the spread approach. In *36th Annual Hawaii International Conference on System Sciences (HICSS'03)*, 2003.

[13] J.-L. Deneubourg, S. Aron, S. Goss, and J.-M Pasteels. The self-organizing exploratory pattern of argentine ant. *Journal of Insect Behavior*, 3:159–168, 1990.

[14] A. Dey and G. Abowd. Towards a better understanding of context and context-awareness. In *Workshop on The What, Who, Where, When, and How of Context-Awareness, as part of the 2000 Conference on Human Factors in Computing Systems (CHI 2000)*, 2000.

[15] M. Dorigo, V. Maniezzo, and A. Colorni. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics-Part B*, 26(1):29–41, 1996.

[16] Marco Dorigo, Gianni Di Caro, and Luca M. Gambardella. Ant algorithms for discrete optmization. *Artificial List*, 5(2):137–172, 1999.

[17] W. Keith Edwards and Rebecca E. Grinter. At home with ubiquitous computing: Seven challenges. In *3rd international conference on Ubiquitous Computing*, pages 256–272, Atlanta, Georgia, USA, 2001. Springer-Verlag.

[18] David Garlan, Dan Siewiorek, Asim Smailagic, and Peter Steenkiste. Project aura: Toward distraction-free pervasive computing. *IEEE Pervasive Computing*, 1(2), 2002.

[19] S. Goss, S. Aron, J. L. Deneubourg, and J. M Pasteels. Self-organized shortcuts in the argentine ant. *Naturwissenschaften*, 76:579–581, 1989.

[20] P.-P. Grassé. Le reconstruction du nid et les coordinations inter-individuelles chez bellicositermes natalensis et cubitermes sp. la theorie de la stigmergie: essai d'interpretation du comportement des termites constructeurs. *Insectes Sociaux*, 6:41–81, 1959.

[21] Robert Grimm, Janet Davis, Eric Lemar, Adam MacBeth, Steven Swanson, Thomas Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, and David Wetherall. System support for pervasive applications. *ACM Transactions of Computing Systems*, 22(4):421–486, November 2004.

[22] Owen Holland and Chris Melhuish. Stigmergy, self-organization, and sorting in collective robotics. *Artif. Life*, 5(2):173–202, 1999.

[23] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. Lua-an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[24] Brad Johanson, Armando Fox, and Terry Winograd. The interactive workspaces project: Experiences with ubiquitous computing rooms. *IEEE Pervasive Computing Magazine*, 1(2), April-June 2002.

[25] Tim Kindberg and Armando Fox. System software for ubiquitous computing. *IEEE Pervasive Computing*, 1(1), 2002.

[26] Marco Mamei and Franco Zambonelli. Programming stigmergic coordination with the tota middleware. In *fourth international joint conference on Autonomous agents and multiagent systems*, pages 415 – 422, July 25 - 29 2005.

[27] M. Mozer. The neural network house: An environment that adapts to its inhabitants. In *AAAI Spring Symposium on Intelligent Environments*, pages 110–114, 1998.

[28] E. Mynatt, I. Essa, and W. Rogers. Increasing the oppurtunities for aging-in-place. In *ACM Conference on Universal Usability*, 2000.

[29] C. Pinhanez and A. Bobick. Fast constraint propagation on specialized allen networks and its application to action recognition and control. MIT Tech Report 456, MIT, January 1998.

[30] C. Pinhanez and A. Bobick. It/i: A theater play featuring an autonomous computer graphics character. In *ACM Multimedia'98 Workshop on Technologies for Interactive Movies*, January 1998.

[31] C. Pinhanez, K. Mase, and A. Bobick. Interval scritps: a design paradigm for story-based interactive systems. In *CHI'97*, March 1997.

[32] Claudio Santos Pinhanez. *Representation and Recognition of Action in Interactive Spaces*. PhD thesis, MIT, June 1999.

[33] Manuel Roman, Christopher K. Hess, Anand Ranganathan Renato Cerqueira, Roy H. Campbell, and Klara Nahrstedt. Gaia: A middleware infrastructure to enable active spaces. *IEEE Pervasive Computing*, 1(4):74–83, Oct-Dec 2002.

[34] M. Satyanarayanan. Pervasive computing: vision and challenges. *IEEE Personal Communications*, 8(4):10 –17, Aug 2001.

[35] Albrecht Schmidt, Kofi Asante Aidoo, Antti Takaluoma, Urpo Tuomela, Kristof Van Laerhoven, and Walter Van de Velde. Advanced interaction in context. In *1th International Symposium on Handheld and Ubiquitous Computing (HUC99)*, pages 89–101. Springer, 1999.

[36] N.A. Streitz, J. Geißler, and T. Holmer. Roomware for cooperative buildings: Integrated design of architectural spaces and information spaces. In *CoBuild98*, February 1998.

[37] M. Stringer, G. Fitzpatrick, and E Harris. Lessons for the future: Experiences with the installation and use of today's domestic sensor and technologies. In *4th International Conference on Pervasive Computing*, 2006.

[38] Guy Theraulaz and Eric Bonabeau. A brief history of stigmergy. *Artificial Life*, 5(2):97–116, 1999.

[39] Stephen S. Yau, Fariaz Karim, Yu Wang, Bin Wang, and Sandeep K.S. Gupta. Reconfigurable context-sensitive middleware for pervasive computing. *IEEE Pervasive Computing*, 1(3):33–40, 2002.