Spatial Policies for Sentient Mobile Applications

David Scott, Alastair Beresford
Laboratory for Communication Engineering,
University of Cambridge Department of Engineering,
William Gates Building, 15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK
Phone: +44 (0)1223 765339, email: {djs55,arb33}@eng.cam.ac.uk

Alan Mycroft
University of Cambridge Computer Laboratory
William Gates Building, 15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK
Phone: +44 (0)1223 334621, email: am@cl.cam.ac.uk

Abstract

Mobile Applications are programs which are able to move themselves between hosts on the network. Sentient Applications are programs which can exploit the existence of pervasive networked sensor devices to observe their environment and react accordingly. We believe that properly designed and constrained Sentient Mobile Applications provide a good foundation for building applications for pervasive computing environments.

The aims of this work are threefold: (i) motivate the use of Sentient Mobile Applications in next-generation pervasive computing environments; (ii) describe the role of policy in building Sentient Mobile Applications; (iii) demonstrate the need for policy to control Sentient Mobile Applications once they have been deployed.

1 Introduction

The goal of pervasive computing is to create systems that *disappear* [22]—systems that fade into the background leaving users free to concentrate on their own tasks rather than explicitly "using the computer". Sentient Computing works towards this goal by adding *perception* to software; applications become more responsive and useful by observing and reacting to their physical environment [12]. The ability to sense the location of people and objects is an important building-block of such sentient systems. One of the most natural ways a program can react to user movement is to move itself around the network. Such Sentient Mobile Applications open up a number of tantalising possibilities, including: (*i*) exploiting resources near to the user's current location (e.g. multimedia hardware, keyboards and mice); (*ii*) supporting the illusion that user applications and their state is omnipresent [10] – allowing a user to use any application from any location; and (*iii*) maximising efficiency by spreading resource-intensive tasks to where resources are underused.

Unfortunately the use of Sentient Mobile Applications also presents several difficult challenges including the following problems: (*i*) how do we write applications so they are not tied closely to the specific sensing technology being used; (*ii*) how do we structure mobile applications to maximise flexibility, analysability, code-reuse; and (*iii*) how do we monitor and constrain mobile code-based applications once they have been let loose?

For the benefits of Sentient Mobile Applications to be realised we must confront these challenges; we require both a simple sensor technology-neutral mechanism for users to specify the behaviour of their own agents (to make them be *good citizens* of the pervasive computing world) whilst allowing people to constrain the behaviour of foreign agents operating in their space: in their office or with their resources.

In previous work [18] we described an abstract mathematical model of the world intended to describe the behaviour of Sentient Mobile Applications. This work is summarised in Section 2.2. We also described a modal-logic based language for creating security assertions about that world, which we summarise in Section 5.1. In this paper we focus on the practical aspects of implementing our system and of the role of policy in both designing and deploying real Sentient Mobile Applications. We present a scheme for expressing per-application mobility policies enabling applications to automatically react to changes in their environment. We describe a mechanism for expressing global security policies enabling users to control the behaviour of applications "in the wild". We show how conflicts between users may occur and we describe a suitable conflict resolution metapolicy for resolving these differences.

The remainder of this paper is structured as follows: Section 2 outlines the some useful background information to this project and Section 3 describes our motivation for applying policy techniques to the problem. Section 4 describes building applications with an abstracted mobility policy while Section 5 motivates the need for a *global* mobility policy mechanism and describes our proposal. A description of related work may be found in Section 6, future directions are discussed in Section 7 and Section 8 concludes.

2 Background

In this section we define the term *Sentient Mobile Application* and describe the spatial model on which we base our policy mechanisms. For completeness we briefly discuss the low-level sensor system we use to keep our spatial model up to date and we describe some middleware which abstracts away the low-level details of different sensor systems behind a common interface, based on our spatial model.

2.1 Sentient Mobile Applications

We start by defining a *Sentient Mobile Application* as a mobile piece of software which is able to observe its environment in both a physical and virtual (or "electronic") sense, adjusting its behaviour accordingly. Changes to the *physical* environment (e.g. when the laptop containing the program is moved to a different room) are observed by network-attached sensors assumed to be permeating the world while changes to the *virtual* environment (for instance when a user runs another program on the same computer) are observed by custom system software running on the host computer itself. In response to these observations, an application may react in a number of different ways including anything from playing a sound through a pair of computer-attached speakers to migrating completely to another host.

In order to be generally useful, such environmentally-aware applications require a sensor- and application domain-neutral model of the world, on which to base their reactions. We have designed a simple model for this purpose which is inspired by the theoretical concept of an *ambient* [2]. An ambient is simply defined as a *bounded* place (anything from a virtual machine to a physical room) where computation happens. In our model we use this concept to combine together data from physical sensors (such as the observation that "the laptop is in the meeting room") as well as data from computers (such as "the music playing agent is running on the laptop") in one single system, described in the following section.

2.2 Describing Spaces

We model the world as a tree of nested *entities*, analogous to ambients in the Ambient Calculus. We divide our entities into *sorts* each representing a different kind of object. We have predefined the following set of sorts, useful for modelling a typical office environment:

room: a physical volume of space corresponding to buildings, floors, corridors, offices, cupboards etc.

person: an autonomous physical entity (e.g. a human or a robot) with the ability to move between **room**s

workstation: an immovable physical object which can host computer processes

laptop: a mobile physical object which can host computer processes

context: a (possibly virtual) machine capable of running mobile code

agent: a piece of mobile code

Note that this list is by no means exhaustive; it is possible to define sorts specific to the application being modelled. For example, it would be sensible to define a sort **aircraft** for an application designed for the aviation industry.

Sorts are used to constrain how entities may nest. The sorts defined above constrain nesting in the following way: **room** entities may nest only in other **room** entities (e.g. a cupboard within an office or an office within a building). Physical objects such as **person**, **laptop** and **workstation** entities may nest within **room** entities. **laptop** entities may nest within (i.e. be carried by) **person** entities. **context** entities may only nest within other **context** entities or computer devices (**laptop** and **workstation** entities). Finally, application code, represented by **agent** entities, may only exist within a **context** (every computer is assumed to have at least one context: that provided by its native operating system).

By way of example, consider a simple environment containing people (named Alice, Bob and Charlie), offices (Bob's office and Charlie's office), computers (a PC and two laptops) and several mobile agents (including one called "music player"). A graphical depiction of the model corresponding to this world at a particular time is displayed in Figure 1. Note the following things about this model:

- Alice is carrying a laptop inside Charlie's office. This laptop is running a pair of mobile applications (in this case both are called "agent").

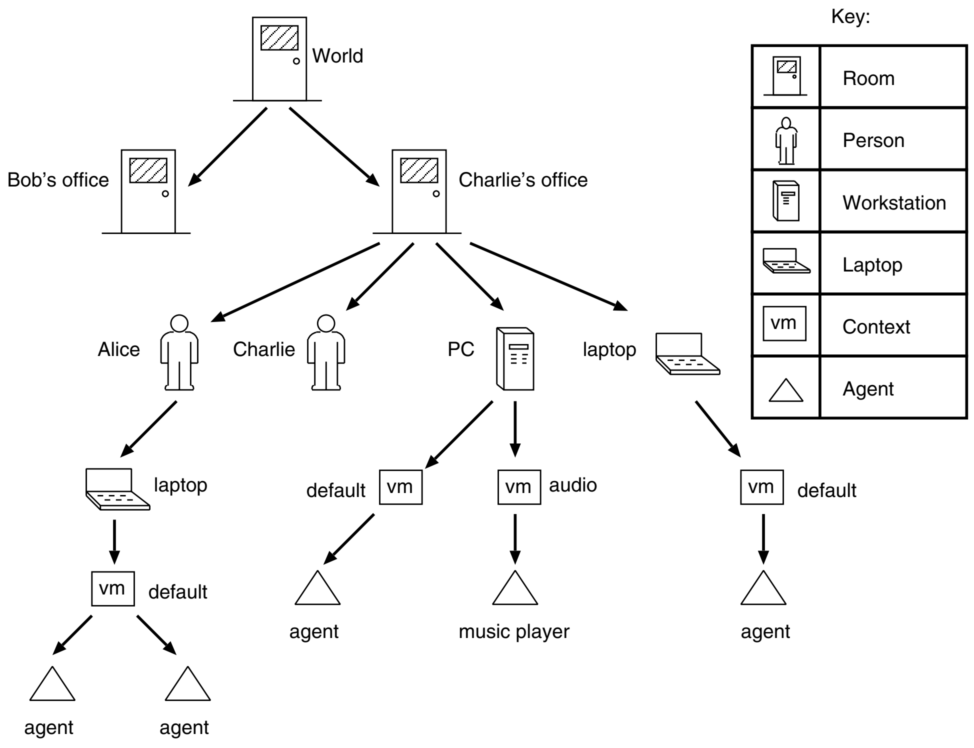

Figure 1. An example World model.

- The PC in Charlie's office has been configured with an extra context called "audio". Charlie has associated this context with the permission to play sound on the computer's associated sound hardware; this permission has been granted to the agent "music player" nested within.

2.3 Dynamically Updating the Spatial Model

To bridge the gap between our abstract spatial model and real-life location systems we have built a distributed Java application, called the World Modelling Service (WMS). In order to be generally useful, the WMS is not specialised to one particular location system but rather is designed to be location-technology agnostic. To support a new location system one must write a small adaptor module which converts the native notions of space and location onto the *entities* supported by the WMS. We currently use two such adaptors: A "Spirit Adaptor" and an "Aglet Adaptor".

The Spirit [1] system is a piece of location-monitoring middleware developed originally at AT&T Laboratories Cambridge. Its primary function is to take a stream of raw location events received from a network of sensors and combine these with a spatial database to produce and disseminate high-level location events concerning people and objects within the lab. Objects to be tracked are equipped with Active-Bat [21] devices — radio-triggered ultrasound-emitting location tags. The clients of the system are location-aware applications, for example the "active map" program which displays a top-down view of an office complete with the positions and orientations of people and equipment, updated in real time.

Spirit has a spatial indexing engine capable of generating events whenever tracked "regions" overlap (e.g. when the space occupied by a person intersects with a zone around a workstation). The first task of the WMS Spirit adaptor is to provide a mapping between these volumes of 3D space and specific entities in the entity hierarchy. An example of such a mapping is shown in Figure 2. The bottom half of the figure shows the output of the *active-map* application monitoring two rooms — "Room 9" and "Room 10". The Spirit adaptor has associated these rooms, two people (userid "kjm25" and "acr31") and a pair of workstations with entities inside the WMS.

Once the mapping is established, the Spirit adaptor listens for high-level events from the Spirit system (such as "user kjm25 has left Room 9") and makes the corresponding changes in the WMS.

The Aglet adaptor[1] allows us to track the locations of Mobile Agents as they migrate from host to host. Combined with the Spirit adaptor, this allows us to simultane-

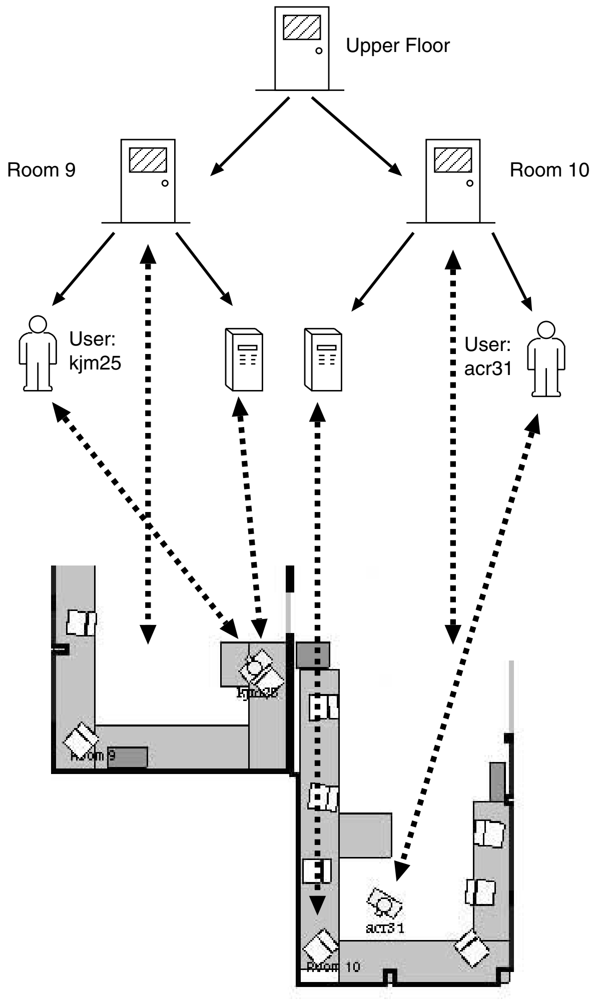

Figure 2. An example of the mapping between Spirit objects (bottom) and WMS entities (top)

[1]The word "Aglet" (coined by IBM) is a cross between the word "Agent" and the word "Applet".

ously monitor the positions of all people, computers and applications in our environment.

3 Overview

Our Sentient Mobile Applications are based on Mobile Agents – pieces of code which have the ability to spontaneously migrate themselves between hosts on the network. Although the ability to migrate at will is very powerful, it is quite difficult to use in practice; it has been observed [14] that the agent migration primitive is similar in spirit to the `go-to` programming statement which has a reputation for allowing confusing programs to be written [7].

3.1 The Role of Policy

In order to reduce confusion to a minimum we propose that *all* mobility aspects of an application should be abstracted out and written as separate policy documents. This has a number of advantages over simply writing the application logic and the mobility policy together. Specifically it enables us to

- increase code reuse through being able to share the same mobility policy amongst several different applications;
- make debugging easier since the code is not tangled up with the main application logic; and
- retrofit mobility on top of existing non-mobile applications without having to extensively modify the existing code base.

We apply policy in the following contexts:

- applications are deployed with an individual mobility policy (referred to as "per-application mobility policy"), described in Section 4;
- users provide global assertion-based policies which apply to arbitrary applications and spatial regions (referred to as "system-level policy"), described in Section 5); and
- a spatially-based Conflict Resolution metapolicy deals with conflicts between policies and is described in Section 5.4.1.

The diagram in Figure 3 shows where these policy pieces sit in relation to each other and how they interact. Each application's individual mobility policy decides when and where to migrate the application in response to observations of the spatial model. These migration requests are sent to the system where they are checked for consistency with the installed system-level policies. If the system-level policies

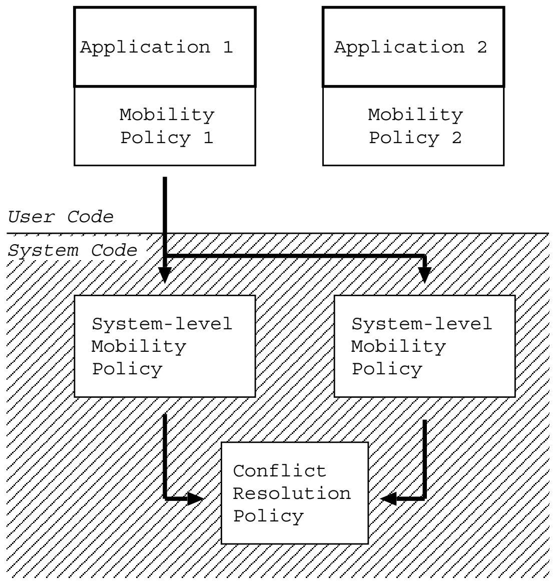
Application 1
Mobility Policy 1
Application 2
Mobility Policy 2
User Code
System Code
System-level Mobility Policy
System-level Mobility Policy
Conflict Resolution Policy

Figure 3. An overview of the system emphasising the role of policy. In the diagram the mobility policy of Application 1 emits a migration request. This request is checked against the system-level policies before being granted.

conflict with each other then the conflict resolution metapolicy makes the final decision on whether to allow the migration request or not.

4 Per-Application Mobility Policies

For maximum flexibility we represent per-application mobility policies directly as classes in the Java programming language. Over time, we hope to build up a comprehensive library of such classes to facilitate the rapid development of future applications. On creation, a mobility policy object is handed two references by the system: one to the WMS which allows the object to observe changes to the spatial model and a second reference to an "Application Manager" object which allows it to dynamically control the behaviour of the application by asking it to stop, start and migrate. The following sections describe this mechanism in more detail by means of a series of simple examples.

4.1 Example: The Follow-Me Policy

The "Follow-Me" policy instructs the application to physically follow a particular user around. Whenever the user leaves a room, the application stops running. Whenever the user enters a new room, the application migrates to a host in the new room and continues where it left off. This mobility policy is useful in a number of contexts, for example

- migrating a user's desktop to a terminal near their current physical location (a process known as "teleporting" [17]) to ensure that user applications are always easily accessible; and
- ensuring that a music playing program is always playing music that the user can physically hear, by migrating to a computer in the same room.

Java-like pseudocode for the Follow-Me policy may be written as follows:

```
public class FollowMe{
  ...
  public FollowMe(WMS spatial_model,
                  Manager manager,
                  Entity me) {
    spatial_model.addCallback(this);
    ...
  }

  Entity findNearbyComputer(Entity x){
    // returns an entity near "x"
    // capable of running the
    // application
  }

  public entityLeft(who, where){
    if (who.equals(me)){
      manager.stop();
    }
  }
  public entityArrived(who, where){
    if (who.equals(me)){
      Entity x=findNearbyComputer(where);
      manager.migrate(x);
    }
  }
}
```

In the constructor the `FollowMe` object registers itself with the WMS. When entities change location in the spatial model the two callback methods `entityLeft` and `entityArrived` are executed on the policy object. Note that in the Follow-Me policy only events relating to the user being tracked are relevant and all other events are silently ignored. When the tracked user leaves his/her current room, the `entityLeft` method calls the `manager.stop` method which requests that the application stop executing. When the tracked user enters another room, the `entityArrived` method attempts to migrate the application to a new computer, located inside the user's new room.

4.2 Example: The When-No-one's-Here Policy

This policy is useful for running long-term non-interactive tasks on otherwise idle computers. Whenever a user enters a room with one of these applications, the application is paused to prevent any possible degradation of the quality of service to the user. Only when the last person leaves the room is it safe for all such background tasks to be restarted.

Java-like pseudocode for this policy may be written as follows:

```
public class WhenNooneHere{
...
  public WhenNooneHere(WMS spatial_model,
                       Manager manager,
                       Entity where) {
  spatial_model.addCallback(this);
  ...
  }

  int countPeopleHere() {
    // return number of people in
    // same room as me
  }

  public entityLeft(who, where){
    if ( (manager.amStopped()) &&
         (countPeopleHere() == 0) )
      manager.start();
  }
  public entityArrived(who, where){
    if ( (manager.amRunning()) &&
         (countPeopleHere() > 0) )
      manager.stop();
  }
}
```

The utility function `countPeopleHere` returns the number of people in the same room as the application. The `entityLeft` and `entityArrived` methods use this information to decide whether or not to let the application execute.

5 System-level policy

As so far described, Sentient Mobile Applications, although potentially very useful, have also some inherent drawbacks. Since applications know about the configuration of both the physical world (i.e. the locations of people and other physical objects) and the "virtual" world of computer resources, a user could deliberately write an application which performs a Denial Of Service (DOS) attack against either people or computers. Such an application could, for example, follow a particular user around and play loud music on nearby speakers to prevent them talking to anyone. Alternatively a malicious application could be designed to use up all computer resources belonging to a particular group of people, to reduce their productivity. Therefore as an unfortunate side-effect of building infrastructure to produce more useful (more "sentient") applications we have additionally created many more opportunities to wreak havoc.[2]

To prevent anarchy, it is essential to provide some way to constrain the activities of mobile applications; we must have some way of imposing system-wide policies to restrict their behaviour. Unless such a mechanism is put in place, it is highly likely that sentient mobile applications will never be widely deployed due to fears over their safety.

To address this need, we propose a second, complementary, spatial policy-based mechanism (called `SpatialP`) to constrain the behaviour of mobile applications. This framework provides users with an easy, comprehensive way to restrict and monitor the activities of applications within a suitably augmented environment. By using location-based policies we hope to exploit structure which users are already familiar with. People are used to security policies governing physical spaces (e.g. "no unauthorised personel are allowed in this area"); we extend this idea seamlessly into the ethereal world of mobile applications. The following is summarised from our FASE'03 paper [18].

5.1 Policies in `SpatialP`

A security policy in `SpatialP` is defined as a 4-tuple

$$\langle \mathit{location}, \mathit{formula}, \mathit{times}, \mathit{onfail} \rangle$$

where *location* is an expression designating a set of specific entities where the assertion given by *formula* should hold. The system will block actions (e.g. agent creation or migration) which would violate the policy. If, with respect to the time period described by *times*, the assertion becomes violated (e.g. by the physical movement of an object) then the system will attempt to execute the command described in the field *onfail*.

We define a path as an ordered sequence of entity names which uniquely specify a single entity using the nesting relation, $\downarrow$. We say that $a \downarrow b$ if b is a child of a, i.e. b is contained within one level of nesting of a. Therefore a path $p = a_1 \ldots a_n$ names an entity if $a_1 \downarrow a_2 \wedge \ldots \wedge a_{n-1} \downarrow a_n$. For example, in the diagram in Figure 1 a sequence of entity names uniquely specifying the entity `music player` could be written

```
[  World, Charlie's office, PC,
   audio, music player          ]
```

The policy field *location* is used to quantify over a set of entities. In our system these sets are described by a form of regular expression. We first define the $\downarrow^*$ operator as the reflexive transitive closure of $\downarrow$ and then write the syntax of location expressions as follows:

element	$\leftarrow$	η	(entity name)
	$\mid$	$\{\eta, \eta\}$	(alternation)
	$\mid$	`*`	(any)
location	$\leftarrow$	*element*	(root)
	$\mid$	*location* / *element*	(direct nesting)
	$\mid$	*location* / . . . / *element*	(transitive nesting)

We define the *matching set* of a *location* l as the set of paths *paths* where $\forall p \in \mathit{paths}$ (with $p = a_1, \ldots, a_n$)

- every step e_1/e_2 in l corresponds with a step $a_1 \downarrow a_2$ where the element e_1 *matches* a_1 and e_2 *matches* a_2;
- every step $e_1/\ldots/e_2$ in l corresponds to a sequence of steps $a_1 \downarrow \ldots \downarrow a_n$ for some n where the element e_1 *matches* a_1 and e_2 *matches* a_n;
- the element $\{\eta_1, \eta_2\}$ *matches* the entity with name η if $\eta_1 = \eta$ or $\eta_2 = \eta$;
- the element `*` *matches* any entity; and
- the trivial path element η *matches* an entity with name η.

Therefore the *location* field allows us to

- concisely name entities whose paths diverge at a point (using alternation) e.g. $a/\{b, c\}$ matches the same entity as a/b and a/c;
- have paths with dislocations (using $/\ldots/$) e.g. $a/\ldots/b$ can match the same entities as a/b as well as $a/c/d/b$; and
- quantify over entities without naming them e.g. $a/$`*` can match the same entities as a/b, a/c and a/d.

The *location* field provides a similar function to that of XPath [4], used for naming elements of XML documents.

The policy field *times* can contain one of two possible types of values: $\mathit{Always}(t)$ and $\mathit{Sometime}(\mathit{from}, \mathit{to}, t)$.

[2]This problem is not unique to our system; it is an unfortunate fact of life that when features are added to a system, they may often be used for "evil" as well as "good".

In both cases the parameter t specifies how much "reaction" time the system has before the policy *onfail* action is executed. The value $Always(t)$ indicates that the policy *formula* should hold for all time during which the system is running. Any violation will trigger the *onfail* action after the reaction t time has elapsed. The value $Sometime(from, to, t)$ states that the *formula* should hold at some point in the time interval between the times *from* and *to*. This is similar to the concept of *obligation* in traditional Role-Based Access Control (RBAC) systems i.e. stating that someone *should* perform some action at some point [6].

The policy field *onfail* specifies an action to take should the policy be violated. The action can of the following types:

- $Log(message)$ causes a message to be written to a log;
- $Kill(location)$ asks the system to terminate agents identified by the path *location*;
- $Freeze(location)$ requests the agents identified by *location* be temporarily suspended; and
- $Create(agent)$ which requests that the system create the agent named by *agent*.

For both the *Kill* and *Freeze* values we adopt the convention that if the location expression has a missing initial element (i.e. it starts with `/` or `/.../`) we automatically replace the first element with the full path to the specific entity the formula is currently being applied to. For example if the policy *location* field is a`/*` and the policy is violated at a/b then the *onfail* expression *Kill* $/c$ is expanded to *Kill* $a/b/c$ i.e. a request to terminate the *only* entity named by $a/b/c$ and not a`/*/`c. This ability to refer to previously matched data in a pattern is also found in other systems using regular expressions, e.g. perl [20].

The policy field *formula* contains an expression written in a simple spatial modal logic similar to the Ambient Logic [3]. The syntax may be written as follows, where η ranges over entity names:

$$\begin{array}{rcll}
\mathit{formula} & \leftarrow & \mathbf{T} & \text{(true)} \\
 & \mid & \neg \mathit{formula} & \text{(negation)} \\
 & \mid & \mathit{formula} \vee \mathit{formula} & \text{(disjunction)} \\
 & \mid & \mathbf{0} & \text{(void)} \\
 & \mid & \eta[\mathit{formula}] & \text{(named location)} \\
 & \mid & \mathit{formula} \mid \mathit{formula} & \text{(composition)} \\
 & \mid & \Diamond f & \text{(somewhere)} \\
 & & & \\
\mathbf{F} & \triangleq & \neg \mathbf{T} & \text{(false)} \\
a \wedge b & \triangleq & \neg(\neg a \vee \neg b) & \text{(conjunction)} \\
\Box a & \triangleq & \neg \Diamond \neg a & \text{(everywhere)}
\end{array}$$

These constructs may be familiar to those versed in modal logics, but we summarise their meaning in the following section.

5.2 Satisfaction

We say that an entity a satisfies the logical formula f (i.e. the formula f holds at a) by writing $a \models f$. Intuitively, we may think of a formula f as *matching* an entity a if $a \models f$. The relation, $\models$ is defined informally as follows:

- $a \models \mathbf{T}$ for any entity a
- $a \models \neg f$ if $a \models f$ does not hold
- $a \models f \vee g$ if either $a \models f$ or $a \models g$ hold
- $a \models \mathbf{0}$ if a is "nothing" i.e. the absence of anything (note that $\eta[\neg \mathbf{0}]$ holds of an entity if the entity consists of the place η and η has at least one entity nested within.)
- $a \models \eta[f]$ if $a \equiv n[M]$ and $\eta = n$ and $M \models f$
- $a \models f \mid g$ if $a \equiv N \mid M$ and $f \models N$ and $g \models M$
- $a \models \Diamond f$ if $\exists b . a \downarrow^* b$ and $b \models f$

For example, the formula $\mathbf{0}$ only matches "nothing" (or "void") i.e. the absence of anything. The formula $f \mid g$ matches a if a can be written as the composition of two expressions N and M such that f matches N and g matches M. The formula $\Diamond f$ matches a if there is an entity b somewhere in the tree rooted at a where b matches f.

5.3 Policy Examples

In this section we briefly demonstrate the kinds of policies expressible in `SpatialP` by means of a simple example, based in the fictional office environment modelled in Figure 1. Imagine the scenario where the user "Alice" has just written a "Follow-me" Sentient Mobile Application (called `music player`) which is programmed to follow Alice, playing music wherever she goes. In order to *monitor* the agent, Alice installs a system-wide spatial policy (note that this policy only passively logs events) like the following:

$$\begin{array}{llcll}
\langle & \mathit{location} & = & \texttt{World}, & \\
 & \mathit{formula} & = & \Diamond(\texttt{Alice}[\mathbf{T}] \mid & \\
 & & & \Diamond\texttt{music player}[\mathbf{T}] \mid \mathbf{T}), & \\
 & \mathit{times} & = & \mathit{Always}(10\ \mathit{seconds}), & \\
 & \mathit{onfail} & = & \mathit{Log} & \rangle
\end{array} \quad (1)$$

The policy asks the system to continuously monitor the spatial formula (given by the *formula* field) against the entity called `World` (which in this case is the root of the entity hierarchy) and to write a message to a log file if the formula is violated for more than 10 seconds. The choice of 10 seconds is a bit arbitrary but should be a reflection of the latency inherent in the system i.e. the delay between Alice moving and the application noticing and following her. The formula may be interpreted as follows:

- The subformula `Alice`[**T**] matches the entity `Alice`, with *any* sub-entities (recall that the formula **T** matches *anything*). In real-world terms, this means "Alice, who may or may not be carrying anything".

- The subformula $\Diamond$`music player`[**T**] matches an entity if the `music player` entity is *somewhere inside*.

- The subformula `Alice`[**T**] | $\Diamond$`music player`[**T**] | **T** matches if the entity `Alice` is present and the `music player` is somewhere inside an entity nearby (note the final | **T** indicates that other entities may *also* be present); and so therefore

- the whole formula $\Diamond$(`Alice`[**T**] | $\Diamond$`music player`[**T**] | **T**) matches an entity if `Alice` is somewhere inside and the `music player` is somewhere inside the space next to her, independent of any other entities being present.

Consider a second user, Bob, who would rather have peace and quiet where he works. To prevent wandering music playing agents disturbing him he writes a rule:

$$\begin{array}{llll} \langle & \mathit{location} & = & \texttt{World/*}, \\ & \mathit{formula} & = & \Box\neg\texttt{Bob}[\mathbf{T}] \vee (\Diamond\texttt{Bob}[\mathbf{T}] \\ & & & \wedge\Box\neg\texttt{audio}[\neg\mathbf{0}]\ ), \\ & \mathit{times} & = & \mathit{Always}(3\ \mathit{seconds}), \\ & \mathit{onfail} & = & \mathit{Freeze}\ \texttt{/.../audio/*} \quad \rangle \end{array} \tag{2}$$

The policy *location* field `World/*` causes the rule to be applied to all children of the entity named `World`, i.e. in the diagram in Figure 1 this corresponds to all the offices, `World/Bob's office` and `World/Charlie's office`. The same formula is applied individually to each of these entities. The formula $\Box\neg$`Bob`[**T**] holds if the entity `Bob` is nowhere inside the office; the formula $\Diamond$`Bob`[**T**] holds if the entity `Bob` is *somewhere* inside the office and the formula $\Box\neg$`audio`[$\neg$**0**] holds if there is not a non-empty `audio` context anywhere within the office. Taken together, the whole *formula* may be read as

> Either Bob is not inside the office concerned (in which case there is no violation) or he is inside the office but there is no sound playing.

If the policy is violated in the office named x then the onfail action is expanded to *Freeze* `World/`x`/.../audio/*` causing audio playing agents inside office x to be frozen.

5.4 Policy Conflict

Clearly there is scope for policies, written by different people, to come into conflict. In the above music player example, Alice's FollowMe policy will come into conflict with Bob's policy if Alice and Bob are in the same room together. We need a conflict resolution algorithm to decide which of them will "win". Should the music be allowed to play or should the peace be kept?

5.4.1 Example: Spatial Conflict Resolution Strategy

We observe that, in a typical office environment, resources tend to be associated with particular people: individuals are associated with their own offices and managers are associated with sets of offices (each associated with individual subordinates). It would be considered inappropriate for one person to enter another person's office and play loud music without their consent; the person entering would be expected to follow the guidelines set down by the office's "owner" while in "their" space. Additionally, if the "boss" banned all music playing in offices then we would expect that rule to take precedence over the desires of everyone else.

Our example conflict resolution strategy follows these principles. We associate entities in our WMS with sets of users. We adopt the convention that policies belonging to a person associated with an entity nearer the root of the hierarchy override those of people further down. So, for example, if we associate the root entity with the "boss" then their wishes will always override everyone elses, while individual users are still able to set local policies controlling their own spaces.

So, in the case of Alice and Bob it depends on *where* the policies come into conflict. Assuming Alice and Bob are both at the same level in a company then if Alice visits Bob in *his* office then *his* policy will override Alice's and there will be silence. On the other hand if Bob visits Alice in *her* office then *her* policy will override his and the music will continue playing. If they meet in neutral territory (e.g. someone else's office) then there is no clear winner. The system will take no action in this case; it is up to the owner of that space to decide which policy should take priority.

6 Related Work

Separately specifying application *concerns* (i.e. properties or areas of interest) and then using some mechanism to compose the fragments back together is the main idea behind recent research into Aspect-Oriented Programming (AOP) [9]. Applying this principle to separate out the mobility aspect from a mobile agent application has been considered before. FarGo [11] aimed to cleanly separate the application logic from its “layout” i.e. the dynamic locations of the code components. It provided a method for dividing code up into modules and then separately specifying how the modules must be located. Lauvset et al [14] further suggest factoring agents into “function”, “mobility” and “management” aspects. Montanari et al [15] propose using *obligation* policies in Ponder [5] to compel agents to move in response to external stimuli. Our system is unique in that it allows users to create policies in terms of both “physical” and “virtual” spaces. We believe that this fusion allows the creation of both useful and easily understandable policies.

Retrofitting mobility onto existing applications has been used before by other systems. The Sprite [8] project allows processes to be transparently migrated to other (idle) machines by transferring state (e.g. virtual memory) across the network. Sprite has the built-in notion that users “own” their workstations and would automatically evict foreign code when the owning user logs in on the computer. This is one of many possible policies which can be written using our system. For example, rather than waiting for the user to complete the log-in process (which might take some time due to a high load on the machine) we can write policies which preemptively evict other processes when the user first arrives in the room.

Jiang and Landay [13] consider risks to privacy in context-aware systems. They base their work on the abstraction of *information spaces*, similar to our *entities*. They envisage a system where documents have associated *privacy tags* which are used to prevent the unwanted leakage of data. They do not specifically consider the use of mobile agents nor do they consider using a modal logic to create security policies.

Steggles et al [19] describe the architecture of Sentient “Follow-Me” mobile applications at AT&T Laboratories, Cambridge. The major difference between this and our work is that they only consider the case of a trusted environment where security is less of an issue and they do not propose building applications out of mobile agents. Unlike our work, they do not attempt to factor out mobility policy as a separate concern.

The LocALE (Location-Aware Lifecycle Environment) framework provides a CORBA-based [16] mechanism to control the lifecycle (i.e. creation and destruction) and location of software objects residing on a network. LocALE defines the notion of a *Location Domain* – a group of machines physically located in the same place. This allows the system to migrate objects to particular physical locations rather than specific hosts. Objects created in LocALE have two additional attributes: (*i*) *Location* specifying in which *Location Domain* the object should be created; and (*ii*) *Constraints* specifying whether the object is allowed to move within its *Location Domain* or remain static. Our work subsumes theirs; we can write policies similar to their *Constraints* but can also express more flexible spatial policies.

7 Discussion and Future Work

There are many opportunities for future work based on this project. One of our goals is to create a way of structuring applications to make the task of building Sentient Mobile Applications easier. Realistically, to fully test our work we must involve more developers, encourage them to use this system and to gather feedback from them.

Although we believe that policies based on space will be easy for users to understand (we base this claim on the observation that people already understand and use spatial policies in the real world e.g. signs on doors saying “this area off-limits to unauthorised personel”) there are still a few things to do which would make the system even easier to use. For example, we need a simple graphical interface to allow users to query the state of the system and to write policies.

We also wish to extend the model of the world by adding extra metadata to our World Modelling Service. It might prove useful to find out information such as when the last location update was registered – for example, an application containing sensitive information might like regular assurance that it is still in a secure location. Such an application would terminate itself if it does not see an update for an long period of time.

Another possibility is to attempt some form of install-time security policy checking. The per-application policy may be viewed as a kind of contract with the system which could be checked for inconsistency with installed global policies. An application which passed the checks would be allowed to run freely whereas one which did not may be refused permission to execute, or be executed in a heavily monitored environment.

Finally we would like to investigate ways to make the system more scalable. Currently the WMS represents the whole of the world as a single tree – fine for a single organisation but not a very good representation for an environment populated by untrusting groups of peers (e.g. on the Internet). It may be possible to draw an analogy between being represented in multiple simultaneous hierarchies and a user talking in multiple “internet chat rooms” at once. A transition (i.e. a migration) might only be allowed if permitted by *all* of the hierarchies concerned. This is a high priority for future work.

8 Conclusion

In this paper we described the usefulness of a new class of applications, dubbed “Sentient Mobile Applications”. These applications are able to perceive their environment, both physically and electronically, and move themselves between hosts on the network in order to best achieve their goals. We argued that, in order for this style of application to become widely adopted, we need mechanisms both to make the task of creating applications easier and also to make it possible to impose constraints on running applications to prevent them from getting out of control. We proposed policy-based solutions to both these problems: a mechanism for abstracting mobility policy from the application which hopefully makes application development easier whilst a modal-logic based global policy language called `SpatialP` provides a way to restrict the behaviour of the application at runtime.

Acknowledgement

This work was supported by the Schiff Foundation and EPSRC. David Scott and Alastair Beresford are sponsored by AT&T Laboratories, Cambridge.

The authors wish to thank Andy Hopper for his support.

References

[1] N. Adly, P. Steggles, and A. Harter. SPIRIT: a Resource Database for Mobile Users, 1997.

[2] L. Cardelli and A. D. Gordon. Mobile Ambients. In M. Nivat, editor, *Proceedings of Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 1378, pages 140–155. Springer-Verlag, Berlin, Germany, 1998.

[3] L. Cardelli and A. D. Gordon. Anytime, Anywhere Modal Logics for Mobile Ambients. In *Principles of Programming Languages (POPL)*, 2000.

[4] W.-W. W. Consortium. XML Path Language (XPath) Specification, November 1999. `http://www.w3.org/TR/xpath/`.

[5] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. Technical Report DoC 2000/1, Imperial College of Science Technology and Medicine, Department of Computing, 180 Queen’s Gate, London. SW7 2BZ, April 2000.

[6] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. In *POLICY*, pages 18–38, 2001.

[7] E. W. Dijkstra. GOTO Considered Harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[8] F. Douglis and J. K. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software - Practice and Experience*, 21(8):757–785, 1991.

[9] T. Elrad, R. E. Filman, and A. Bader. Aspect-Oriented Programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001.

[10] A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster. The Anatomy of a Context-Aware Application. In *Mobile Computing and Networking*, pages 59–68, 1999.

[11] O. Holder, I. Ben-Shaul, and H. Gazit. Dynamic Layout of Distributed Applications in FarGo. In *International Conference on Software Engineering*, pages 163–173, 1999.

[12] A. Hopper. 1999 Sentient Computing. *Phil. Trans. R. Soc. Lond.*, 358(1):2349–2358, 2000.

[13] X. Jiang and J. A. Landay. Modeling Privacy Control in Context-Aware Systems. *IEEE Pervasive Computing magazine*, 2002.

[14] K. J. Lauvset, D. Johansen, and K. Marzullo. Factoring mobile agents. In *In Proceedings of the 9th IEEE Conference and Workshops on Engineering of Computer-Based Systems*, Sweden, April 2002.

[15] R. Montanari and a. C. S. Gianluca Tonti. Programming Agent Mobility. In M. Klusch, S. Ossowski, and O. Shehory, editors, *Cooperative Information Agents VI Proceedings*, pages 287–296, Madrid, Spain, September 2002. Springer-Verlag.

[16] Object Management Group. *Common Object Request Broker Architecture (CORBA/IIOP)*, 1995.

[17] T. Richardson. Teleporting: Mobile X sessions. *The X Resource*, 13(1):133–140, 1995.

[18] D. Scott, A. Beresford, and A. Mycroft. Spatial Security Policies for Mobile Agents in a Sentient Computing Environment. Source URL: `http://www.recoil.org/~djs/papers/unpublished/entity-model/paper.pdf`, October 2002.

[19] P. Steggles, P. Webster, and A. Harter. The Implementation of a Distributed Framework to support ’Follow me’ Applications. Technical report, The Olivetti and Oracle Research Laboratory, 1998.

[20] L. Wall and R. L. Schwartz. *Programming Perl*. O’Reilly & Associates, Inc., Sebastopol, CA, 1992.

[21] A. Ward, A. Jones, and A. Hopper. A New Location Technique for the Active Office, 1997.

[22] M. Weiser. The Computer for the 21st Century. *Scientific American*, 9 1991.