Spatial Security Policies for Mobile Agents in a Sentient Computing Environment

David Scott[1], Alastair Beresford[1], and Alan Mycroft[2]

[1] Laboratory for Communications Engineering, University of Cambridge
[2] Computer Laboratory, University of Cambridge
William Gates Building, 15 JJ Thompson Avenue, Cambridge CB3 0FD, UK
{djs55,arb33}@eng.cam.ac.uk, am@cl.cam.ac.uk

Abstract. A Sentient Computing environment is one in which the system is able to perceive the state of the physical world and use this information to customise its behaviour. Mobile agents are a promising new programming methodology for building distributed applications with many advantages over traditional client-server designs. We believe that properly controlled mobile agents provide a good foundation on which to build Sentient applications.
The aims of this work are threefold: (*i*) to provide a simple location-based mechanism for the creation of security policies to control mobile agents; (*ii*) to simplify the task of producing applications for a pervasive computing environment through the constrained use of mobile agents; and (*iii*) to demonstrate the applicability of recent theoretical work using *ambients* to model mobility.

1 Introduction

The goal of pervasive computing is to create systems that *disappear* [18]—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 [7]. 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 (e.g. consider a desktop "teleportation" program which moves a user's screen, mouse and keyboard to the computer nearest their current physical location). Mobile code opens 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—allowing a user to use any application from any location; and (*iii*) maximising efficiency by spreading resource-intensive tasks to where resources are under-used.

Unfortunately the use of mobile code has severe security implications [14]. Aiming to deploy mobile agent-based applications on the global Internet, the research community has concentrated on solving two difficult problems: (*i*) preventing malicious mobile code from gaining unauthorised access to resources controlled by the hosting machine; and (*ii*) stopping a malicious virtual machine learning secret information such as cryptographic keys by disassembling mobile agents. There is a third critical problem which is less well-studied: user control. How do we allow users to control the activities of mobile agents in an intuitive way, providing just enough security whilst still reaping the benefits of mobile code? How can users trust agents, written by other people, to respect their wishes when they enter their space and use their resources?

Clearly users need an understandable mechanism to constrain the behaviour of agents. They need a way to control what happens with things that matter to them in *their* world i.e. their data and their computers. Without this ability users will never fully trust mobile code technology and will never allow it to be used in practice.

To address this need, we propose a framework for creating spatial (i.e. location-based) security policies for mobile agents. This framework provides users with an easy to comprehend way to restrict and monitor the activities of agents within a Sentient Computing 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 personnel are allowed in this area"); we extend this idea seamlessly into the ethereal world of mobile agents.

We believe that our framework will (*i*) promote the development of mobile-code based Sentient Computing applications; (*ii*) allow users fine-grained control over where agents are allowed to run, bypassing problems associated with situations where the agent does not trust the machine or vice-versa; and (*iii*) provide a demonstration of the applicability of recent theoretical work in the research community modelling mobility.

The remainder of this paper is structured as follows: Sect. 2 describes the design of our framework in detail. Examples of possible policies are found in Sect. 3. Related work may be found in Sect. 4, and Sect. 5 concludes.

2 Spatial Policy Framework

In this section we describe (*i*) how we model the world incorporating both physical objects and mobile agents; (*ii*) our language for expressing mobility security policies; and (*iii*) our proposal for an arbitration scheme to reconcile conflicting policies. Implementation details are omitted for brevity; they may be found elsewhere [15].

2.1 Overview

Users write security policies to influence both their own agents and any objects in which they have an interest (e.g. the computers in their office). Examples of policies might be "this agent should never leave my office" or "never let any agent enter this zone". The system continuously monitors the locations of both physical objects and mobile agents, keeping track of which policies are being violated. Since the system has no physical presence, it cannot act to block the movements of physical objects. Therefore the system cannot *guarantee* that policies will never be violated; instead policies are associated with an *action*, a command to be executed if and when a policy is violated (e.g. a command to kill the offending agent).

In contrast to physical objects, the system has full control over the life-cycle of mobile agents. Agents requesting permission to migrate between hosts will have their requests blocked if the movement would violate a policy. Agents which are *physically* moved (typically by being carried on a laptop by a user) in violation of a policy may find themselves being suspended or killed.

Policies written by different people may conflict with each other. The system implements an arbitration scheme which exploits the natural structure of the spatial model to resolve these conflicts when they arise.

2.2 Modelling the World

We model the world as a tree of nested *entities*, analogous to *ambients* in the Ambient Calculus [2]. We begin our description by defining the following set of terms:

entity name: label used to name entities, equivalent to an entity minus any contents. Examples include the names of physical places, computers and mobile agents. By convention we use η to range over all entity names and a to range over mobile agent entity names.

entity: description of a particular location (given by an entity name) *along with* its contents. Note that, in a similar fashion to the Ambient Calculus, we are not restricted to describing only physical places but can represent any bounded region where activity happens. For example an office containing people may be described as an entity, as can a virtual machine containing mobile code. By convention we say that e ranges over entities.

path: sequence of entity names describing a route through the entity hierarchy naming a specific entity. Paths are written in the form $\eta_1 \downarrow \ldots \downarrow \eta_n$ and are described further in Sect. 2.4.

path expressions: regular expression-like facility to efficiently name a set of entities. Path expressions are described further in Sect. 2.4.

We divide our entity names into *sorts* each representing a different kind of object. The exact sorts used in any deployed system will depend on the kinds of things being modelled (e.g. an aviation-based system may introduce the sort "aircraft"). Here we restrict ourselves to use the following sorts:

room: a physical volume of space corresponding to buildings, offices etc.
person: an autonomous physical entity able 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 virtual machine capable of running mobile code
agent: a piece of mobile code

We write $e \lhd s$ to mean entity e is of sort s. The formula $\mathit{SortContainable}(s_1, s_2)$ holds when entities of sort s_2 may be nested inside entities of sort s_1. This formula is defined graphically in Fig. 1. Intuitively, physical objects may nest in the obvious way (e.g. a **workstation** may nest inside a **room**). For convenience we define a relation $\mathit{Containable}(e_1, e_2)$ which indicates that entity e_2 is permitted to nest inside e_1. These relations are related as follows:

$$\mathit{Containable}(e_1, e_2) \Leftrightarrow e_1 \lhd s_1 \wedge e_2 \lhd s_2 \wedge \mathit{SortContainable}(s_1, s_2)$$

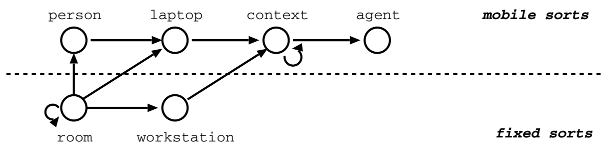


Fig. 1. The relation *SortContainable*

We define a partial function, $privs : entity \rightharpoonup permission\ set$, which is only defined on **context** entities and gives the set of permissions which are granted automatically to any **agent** nested inside. Examples of permissions include "can_play_sound" and "can_record_sound". This association between entities and sets of permissions allows us to use our spatial security policies to control more than simply the *location* of mobile agents; by creating appropriate **context**s we can control access to arbitrary resources. By convention, every computer (**workstation** or **laptop**) has at least one default **context** with an empty permission set.

A state of our world model may be written down with the following syntax (where η ranges over a set of entity names):

$$\begin{array}{lcll} entity & \leftarrow & entity \mid entity & \text{(siblings)} \\ entity & \leftarrow & \eta[entity] & \text{(nesting in a place } \eta) \\ entity & \leftarrow & \mathbf{0} & \text{(void)} \\ entity & \leftarrow & !\eta & \text{(entity factory)} \end{array}$$

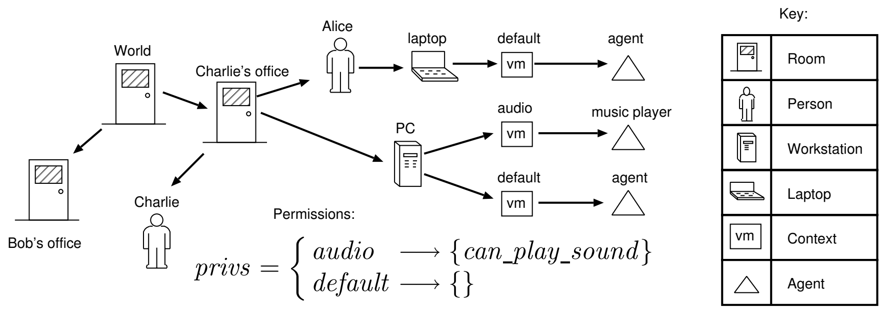
By convention we consider an entity factory $!\eta$ to be a special kind of entity which can create other entities, i.e. the factory $!\eta$ can spawn the entity $\eta[\mathbf{0}]$, an empty entity ready for population. Note that every mobile agent which wishes to be created must be associated with at least one of these factory entities.

The entity $e = \eta[e_1 \mid \ldots \mid e_n]$ is well-sorted if $Containable(e, e_1) \wedge \ldots \wedge Containable(e, e_n)$. The case $n = 0$ corresponds to the entity being empty i.e. $\eta[\mathbf{0}]$. Observe that this syntax is similar to the subset of the ambient calculus which has no active processes and which describes only the structure of space, like that used in the semistructured data format described in [1].

As is conventional in mobility theory we next define a congruence relation, $\equiv$, under which entities are equal up to simple rearrangements of parts. In addition to reflexivity, symmetry, transitivity and context ($X \equiv Y \implies \eta[X] \equiv \eta[Y]$) this relation (often referred to as a structural congruence relation) is also commutative ($X \mid Y \equiv Y \mid X$), associative ($X \mid (Y \mid Z) \equiv (X \mid Y) \mid Z$) and "ignores zeros" ($X \mid \mathbf{0} \equiv X$).

2.3 Example

Consider a simple environment containing people, computers and several mobile agents. A graphical depiction of the model corresponding to this world at a particular time is displayed as follows:



$$privs = \begin{cases} audio \longrightarrow \{can\_play\_sound\} \\ default \longrightarrow \{\} \end{cases}$$

There are various things to note about this configuration:

1. Alice is carrying a laptop inside Charlie's office. This laptop is currently running some mobile agent code. Note that such agents have potentially entered the room without having to migrate to a different host.
2. The PC in Charlie's office has been configured with an additional **context**, called `audio`. This **context** has been associated with a permission, *can play sound*, allowing agents to play sounds on a set of attached speakers. Therefore the **agent**, `music player` is able to play music in the office.

Although we have presented entity names as flat identifiers, they are likely to be more complicated in practice. For example they could contain secret data or be protected by a digital signature – possible benefits of such schemes include: (*i*) policies could be applied to whole classes of agents (e.g. all those signed by a particular key); or (*ii*) only people possessing the secret data would be able to successfully name an agent in a policy. For simplicity in the rest of this paper we will continue to use simple english names (like `music player`) for mobile agents.

2.4 Paths and Path Expressions

We uniquely name a single specific entity by providing a path from the root 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. A path to an entity will therefore have the form $\eta_1 \downarrow \eta_2 \downarrow \ldots \downarrow \eta_n$. For example, in the diagram in Sect. 2.3 an expression for the location of the entity `music player` would be `World` $\downarrow$ `Charlie's office` $\downarrow$ `PC` $\downarrow$ `audio` $\downarrow$ `music player`.

Path expressions, similar to regular expressions, are used to quantify over a set of paths. We first define the $\downarrow^*$ operator as the reflexive transitive closure of $\downarrow$ and then write the syntax of location expressions as follows:

$$\begin{array}{rcll} element & \leftarrow & \eta & \text{(entity name)} \\ & | & \{\eta, \eta\} & \text{(alternation)} \\ & | & * & \text{(any)} \\ \\ expression & \leftarrow & element & \text{(root)} \\ & | & expression\texttt{/}element & \text{(direct nesting)} \\ & | & expression\texttt{/...}\texttt{/}element & \text{(transitive nesting)} \end{array}$$

We define the *matching set* of a path expression exp as the set of paths $paths$ where $\forall p \in paths$ (with $p = p_1 \downarrow \ldots \downarrow p_n$)

- every step e_1/e_2 in exp corresponds with a step $p_1 \downarrow p_2$ in p where the element e_1 *matches* p_1 and e_2 *matches* p_2;
- every step $e_1/\ldots/e_2$ in exp corresponds to a sequence of steps $p_1 \downarrow \ldots \downarrow p_n$ for some n where the element e_1 *matches* p_1 and e_2 *matches* p_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 name; and
- the trivial path element η *matches* an entity with name η.

Path expressions provide a similar function to that of XPath [4], used for naming elements of XML documents.

2.5 Updating the Model

The model is updated dynamically to reflect the real-time configuration of the environment. Location sensors register changes in the physical configuration of the world (e.g. the movements of objects) which are then reflected by changes in the model. In addition, mobile agents may be programmatically created, frozen, killed or migrated, constrained only by the installed security policies. We define legal updates to the world by a labelled transition relation, $\xrightarrow{\gamma}$. We use labels to represent the side-effects of transitions, in particular the emission ($emit(\gamma)$) and reception ($receive(\gamma)$) of an agent during migration. The absence of a label on a transition indicates the lack of side-effects. A valid transition must have no labels at the top level – labels must always be matched and cancelled by the rule (migrate) described below. For brevity we write $a \leftrightarrow b$ if the transition is reversible i.e. if both $a \rightarrow b$ and $b \rightarrow a$ are legal transitions. The runtime system (described in a companion paper [15]) ensures that every event that occurs is represented by a legal transition.

For entities X, Y, Z and entity names a, b, c where $a \lhd person$, $b \lhd room$ and $c \lhd laptop$ we define the following rules:

$$\begin{array}{lcll} a[X] \mid b[Y] & \leftrightarrow & b[a[X] \mid Y] & \text{(walk in/out)} \\ a[X] \mid c[Y] & \leftrightarrow & a[c[Y] \mid X] & \text{(pick up/put down)} \end{array}$$

In plain terms these rules describe how a person may freely walk into and out of rooms and pick up or drop any portable physical objects (represented by entities of sort **laptop**).

For simplicity everything that can happen to a mobile agent (i.e. being created, frozen, defrosted, killed or migrated) is considered as a sequence of primitive operations of the following two types: (*i*) leaving a particular context; (*ii*) entering a particular context. For example an agent creation is considered a single event – the new agent entering its initial context. Killing an agent is a single leaving event. An agent migration from a to b is considered a sequence of two events: (*i*) leaving the source context a; and (*ii*) entering the destination context b. Freezing an agent is considered as a migration into a special context called $frozen$ and defrosting is a migration out again.

To represent the installed security policies, we assume a pair of infix predicates, *can_enter* and *can_leave*, defined later in Sect. 2.8, which for a given agent d and context e behave as follows:

d *can_leave* e	holds if the policies allow d to leave the context e
d *can_enter* e	holds if the policies allow d to enter the context e

For entities $d \lhd agent$, and $e \lhd context$ we write the rule:

$$e[!d \mid X] \quad \rightarrow \quad e[d[\mathbf{0}] \mid !d \mid X] \quad \textit{iff } d \textit{ can\_enter } e \quad \text{(agent created)}$$

This rule asserts that agents may be created in those places containing an appropriate agent factory (represented by $!d$) provided the new agent is allowed to enter the surrounding context. Similarly, agent destruction is only permitted if the agent is allowed to leave the containing context, as described by this rule:

$$e[d \mid X] \quad \rightarrow \quad e[X] \quad \textit{iff } d \textit{ can\_leave } e \quad \text{(agent killed)}$$

Agents are frozen by moving them into specially created *frozen* contexts, created dynamically. These contexts are associated with no permissions i.e. $privs(\textit{frozen})$ is $\{\}$. Consider the example in Sect. 2.3 – if the `music player` agent were frozen then it would lose the ability to play sound. The acts of freezing and defrosting are described by the following rules:

$$\begin{array}{llll} e[d \mid X] & \rightarrow \; e[\textit{frozen}[d] \mid X] & \textit{iff } d \textit{ can\_leave } e & \text{(agent frozen)} \\ e[\textit{frozen}[d] \mid X] & \rightarrow \; e[d \mid X] & \textit{iff } d \textit{ can\_enter } e & \text{(agent defrosted)} \end{array}$$

Note that, to be frozen, an agent must be allowed by the security policies to leave its current context. There is no guarantee the agent will ever be unfrozen again; unfreezing may only occur if the agent has permission to reenter the original context.

Agent migration between contexts is handled by the following rules:

$$\begin{array}{llll} e[d \mid X] \; \xrightarrow{emit(d)} & e[X] & \textit{iff } d \textit{ can\_leave } e & \text{(agent leaves)} \\ e[X] \; \xrightarrow{receive(d)} & e[d \mid X] & \textit{iff } d \textit{ can\_enter } e & \text{(agent enters)} \end{array}$$

$$\frac{X \xrightarrow{emit(\gamma)} Y \quad Y \xrightarrow{receive(\gamma)} Z}{X \rightarrow Z} \quad \text{(migrate)}$$

Note that the act of migration is a compound operation where the side-effect $emit(\gamma)$ must be matched by a corresponding side-effect $receive(\gamma)$. Therefore migration may only happen if the policies allow both the leaving step and the arriving step; it is impossible for the agent to get stuck somewhere in between. It is important to emphasise that only the results of toplevel transitions are visible to applications – applications cannot see any intermediate states of the model. We get away with this because our work so far has focused on a trusted "intranet"-style environment where complications due to unreliable network communication and partial failure are minimised.

Agent migration could be represented differently if we allowed agents to simply climb the entity hierarchy and then walk down again – the approach taken in the Ambient Calculus. This would allow us to simplify our rules by removing the labels on our transition relation. However, allowing an agent to move anywhere in the hierarchy could lead to violations of the sorting rules (described in Sect. 2.2). Additionally there is a subtle semantic difference with respect to the security policies: by using the "teleporting" approach described here, only the configurations at the start (the leaving step) and at the end (the arriving step) are relevant. If the agent were to have to walk from one place to another then the migration could potentially be blocked by a policy attached to an entity somewhere in the middle.

To complete our description of how the model can be updated we have the following rules where X' and Y' are entities:

$$!\eta \quad \rightarrow \quad !\eta|\eta[\mathbf{0}] \quad \textit{iff}\ \eta[\mathbf{0}] \not\triangleleft \mathit{agent} \quad \text{(non-agent entity created)}$$

$$\frac{X \xrightarrow{\gamma} Y}{\eta[X] \xrightarrow{\gamma} \eta[Y]}\ \text{(nested update)} \qquad \frac{X \xrightarrow{\gamma} Y}{X \mid Z \xrightarrow{\gamma} Y \mid Z}\ \text{(parallel update)}$$

$$\frac{X' \equiv X, X \xrightarrow{\gamma} Y, Y \equiv Y'}{X' \xrightarrow{\gamma} Y'}\ \text{(update} \equiv\text{)}$$

Informally the first rule states that non-agent entities may be created in entity factories (note that agent entities may only be created if allowed by the security policies, using the rule (agent created) described earlier). The other three rules state that transitions may occur anywhere in the entity nesting hierarchy, in parallel with arbitrary other entities up to structural congruence. Note that the labels on the transitions are preserved but must eventually be cancelled further up the tree (by the rule (migrate)).

2.6 Expressing Policies

A security policy is defined as a 4-tuple $\langle \mathit{location}, \mathit{formula}, \mathit{times}, \mathit{onfail} \rangle$ where $\mathit{location}$ is a path expression (see Sect. 2.4) designating a set of specific entities where the assertion given by $\mathit{formula}$ should hold. 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.

The policy field times can contain one of two possible types of values: $\mathit{Always}(t)$ and $\mathit{Sometime}(\mathit{from}, \mathit{to}, t)$. In both cases the parameter t specifies how much "reaction" time the system has before the policy onfail action is executed. The value $\mathit{Always}(t)$ indicates that the assertion $\mathit{formula}$ should hold for all time during which the system is running. The value $\mathit{Sometime}(\mathit{from}, \mathit{to}, t)$ states that $\mathit{formula}$ should hold[1] at some point in the time interval between the times from and to.

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

[1] This is similar to the concept of *obligation* in traditional Role-Based Access Control (RBAC) systems i.e. it states that someone *should* perform some action during some time interval.

- $Log(message)$ causes a message to be written to a log;
- $Kill(pathexpr)$ asks the system to terminate agents identified by the path expression $pathexpr$;
- $Freeze(pathexpr)$ requests agents named by $pathexpr$ be frozen; and
- $Create(path)$ requests the agent factory named by $path$ create an agent.

For both the $Kill$ and $Freeze$ values we adopt the convention that if the path expression has a missing initial element (i.e. it starts with `/` or `/../`) we automatically prepend 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 $a \downarrow b$ then the $onfail$ expression $Kill$ /c is expanded to $Kill$ a/b/c i.e. a request to terminate *only* the entity named by $a \downarrow b \downarrow c$ and not any other element (e.g. $a \downarrow d \downarrow c$). This ability to refer to previously matched data in a pattern is also found in other systems using regular expressions, e.g. perl [17].

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

$$\begin{array}{lcll}
formula & \leftarrow & \mathbf{T} & \text{(true)} \\
 & | & \neg formula & \text{(negation)} \\
 & | & formula \vee formula & \text{(disjunction)} \\
 & | & \mathbf{0} & \text{(void)} \\
 & | & \eta[formula] & \text{(named entity)} \\
 & | & !\eta & \text{(named agent factory)} \\
 & | & formula \mid formula & \text{(composition)} \\
 & | & \Diamond e & \text{(somewhere modality)}
\end{array}$$

F (false), $a \wedge b$ and $\Box a$ (everywhere modality) may be written using the core syntax as $\neg\mathbf{T}$, $\neg(\neg a \vee \neg v)$ and $(\neg\Diamond\neg a)$ respectively. These constructs may be familiar to those versed in modal logics, but we summarise their meaning in the following section.

2.7 Satisfaction

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

- $e \models \mathbf{T}$ for any entity e
- $e \models \neg f$ if $e \models f$ does not hold
- $e \models f \vee g$ if either $e \models f$ or $e \models g$
- $e \models \mathbf{0}$ if e is "nothing"
- $e \models !\eta$ if $e \equiv !\eta$
- $e \models \eta[f]$ if $e \equiv n[M]$ and $\eta = n$ and $M \models f$
- $e \models f \mid g$ if $e \equiv N \mid M$, $f \models N$ and $g \models M$
- $e \models \Diamond f$ if $\exists e'.e \downarrow^* e'$ and $e' \models f$

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

2.8 Reasoning about Policies

If we allow individual users to write their own security policies then we must also provide a mechanism to resolve policy conflicts when they arise. Conflicts between rules in our system are similar to those found in Active Databases [5]. Many mechanisms have been proposed, ranging from a simple numeric priority schemes to more complex algorithms comparing rules based on their generality [8] (e.g. the more general rule holds except when the less general does not or v.v.). There is no single best strategy that works perfectly in all circumstances. Our main goal is to make the system be intuitive enough for ordinary users to understand. Security policies in our system are based on a spatial modal logic therefore we also use a spatial mechanism for arbitrating between conflicting policies.

Recall that we model the state of the world as a nested tree of entities (see Sect. 2.2). We observe that within a real life enterprise people too are often arranged into a hierarchy, with the boss at the top, managers in the middle and normal employees at the leaf nodes. In such an organisation, a manager would be able to set a policy which would override those of subordinates but which could itself be overridden by the boss. These two hierarchies, one describing the world and one describing the people, can be linked together by associating entities with a set of people ("owners" or "administrators") via a function

$$owners : entity \rightarrow person\ set$$

such that for an entity e we have $owners(e) = \{person_1, \ldots, person_k\}$ where $person_1, \ldots, person_k$ are the direct "owners" of e. In a typical configuration, the boss would "own" the root entity while normal employees would "own" their individual offices. Our scheme for arbitrating between conflicting policies may be informally described as:

> For a proposed change to entity e, policies instituted by a user $u'' \in owners(e'')$ override those policies instituted by a user $u' \in owners(e')$ where $e'' \downarrow^* e'$ and $e' \downarrow^* e$ as long as $e'' \neq e'$ and $u'' \neq u'$.

Recall from Sect. 2.5 that the installed security policies may be represented by a pair of predicates, *can_leave* and *can_enter* which, given an agent and a context hold precisely when an agent is allowed to leave or enter the context respectively. Both of these predicates are computed in the following way: For a proposed change in the configuration at context c (e.g. an entity wishes to leave c) we first compute the set of users who "own" any of the entities on the path $p_1 \downarrow \ldots \downarrow p_n$ from the "root" entity p_1 which designates c

$$users = \bigcup_{k=1}^{n} owners(p_1 \downarrow \ldots \downarrow p_k)$$

Each user $u \in users$ is allocated a single vote on the proposed change. Note that this effectively means that although users may write policies about entities they do not "own" these policies will be easily overridden by other users who *do* "own" these entities. A user u votes for the proposed change if the number of their policies which are in violation decreases, votes against if the number in violation increases and abstains otherwise. We define a function $vote(user)$ as follows:

$$vote(user) = \begin{cases} -1 & \text{if } user \text{ votes against the proposal} \\ 0 & \text{if } user \text{ abstains} \\ +1 & \text{if } user \text{ votes for the proposal} \end{cases}$$

We then compute the value of

$$overall\ vote = \sum_{i=1}^{n} \sum_{o \in owners(p_1 \downarrow \ldots \downarrow p_i)} prio(i) vote(o)$$

where $p_1 \downarrow \ldots \downarrow p_i$ refers to the ith entity on the path $p = p_1 \downarrow \ldots \downarrow p_n$ and the function $prio(i)$ gives the priority of owners of this entity. One possible priority function is given by $prio(i) = x^{-i}$ where x is a tunable vote weighting factor. The parameter x determines how many people who "own" an entity p_n are needed in order to equal the vote of a single person who "owns" a "more important" entity p_{n-1}. If $x > max_i\,(|owners(p_i)|)$ then it is impossible for the owner of a more important entity to be overridden by a group of people who own a less important entity. The system will allow the proposed change if $overall\ vote \geq 0$ and veto it otherwise.

3 Policy Examples

In this section we demonstrate the kinds of policies which are expressible in our system by means of a series of examples set in a typical shared workplace environment. A snapshot of the world configuration is presented in Sect. 2.3. The top-level entity is named `World` and contains child entities `Bob's office` and `Charlie's office` representing the offices of users named Bob and Charlie respectively. We assume that ordinary employees by default "own" the entities corresponding to their offices and for the sake of an interesting example we further assume that Bob is the boss and also "owns" the top-level entity, `World`.

A user, called Alice, writes and deploys a "follow-me" music playing mobile agent which follows her around, playing music where she goes. She is worried about the agent running amok and so writes the following policy to enable the system to monitor the agent:

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

"for all time, wherever in the `World` I am, an agent called `music player` should be in the same space as me. If this is not true for more than 10 seconds, log the error"

Remember that $e \models f \mid g$ holds whenever f and g are children of e and that **T** matches anything, including **0**, the absence of anything. In the formula above the third **T** means that the formula will hold irrespective of whatever else is in the same space as Alice.

The consequences of this policy are summarised as follows:

1. When the `music player` attempts to migrate, the system prevents the agent from *leaving* the same room as the user. Note it does not directly force the agent to move properly, it just stops it from moving inappropriately.
2. Upon observing Alice move to a new room the system assumes the agent is broken if it has not followed her within 10 seconds. The system will log the error for Alice to use in debugging her errant agent.
3. If Alice moves to a room which already has a `music player` agent the system will not complain even if Alice's agent fails to follow her.

Consider a second user, Bob, who is Alice and Charlie's boss. Bob prefers peace and quiet where he works. To prevent wandering music playing agents disturbing him he writes a rule:

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

"if ever I'm in an office with a music playing agent, freeze the agent if it has not left within 3 seconds"

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 Sect. 2.3 this corresponds to all the offices, `World` $\downarrow$ `Bob's office` and `World` $\downarrow$ `Charlie's office`. The same formula is applied individually to each of these entities. The formula $\Box\neg\texttt{Bob}[\mathbf{T}]$ holds if the entity `Bob` is nowhere inside the office; the formula $\Diamond\texttt{Bob}[\mathbf{T}]$ holds if the entity `Bob` is *somewhere* inside the office and the formula $\Box\neg\texttt{audio}[\neg\mathbf{0}]$ 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.

The consequences of this policy are summarised as follows:

1. If a `music player` agent attempts to migrate inside the same office as Bob the request will be denied, assuming that his policy is not overridden by anyone more senior in the company.
2. If a `music player` agent running on a laptop or PDA is physically moved inside his office by someone else, that agent will be frozen.

Now consider what will happen when Alice enters Bob's office. Clearly the two policies 1 and 2 now conflict. Alice's mobile agent will attempt to migrate inside Bob's office so the system will apply the conflict resolution rules described in Sect. 2.8. Assuming the system knows that Bob "owns" the entity named `World` (since he is the boss) his policy will override those belonging to Alice and the migration request will be blocked.

Imagine a third user, Charlie, with more malicious intent. This user attempts to lure hapless agents into his domain and then trap them there forever. He decides to go after Alice's music playing agent and writes the following:

$$\begin{array}{llll} \langle & \mathit{location} & = & \texttt{World/Charlie's office}, \\ & \mathit{formula} & = & \Diamond(\texttt{music player}[\mathbf{T}]), \\ & \mathit{times} & = & \mathit{Always}(0\ \mathit{seconds}), \\ & \mathit{onfail} & = & \mathit{Log} \qquad\qquad\qquad\qquad \rangle \end{array} \tag{3}$$

"for all time the `music player` agent should remain inside my office."

Consider what happens when Alice is enticed into Charlie's office for a coffee and biscuit. Initially Alice's `music player`'s request to migrate into Charlie's office is accepted since it does not violate any policy (in fact it causes rule 3 to no longer be in violation – an improvement!) When Alice leaves the office the music player attempts to follow her. Charlie's and Alice's rules are now in direct conflict. Unfortunately for Alice since Charlie "owns" his office his policies take priority and therefore the agent's request to leave is denied. What can Alice do? The only solution for Alice in this situation is to appeal to a higher authority – in this case Bob – someone whose policies are ranked higher than Charlie's. Bob may write a policy to evice Alice's agent, overriding Charlie's wishes.

4 Related Work

There have been many proposed mobile agent systems, e.g. TACOMA [12] (Tromsø And COrnell Moving Agents), Agent-TCL [6] and Telescript [16]. Similarities with our work include: mobile agents on mobile devices (PDAs [9] and mobile phones [10]) in TACOMA and the concept of *regions* (similar to our *entities*) in Telescript. Unlike our work, none of these previous systems attempted to exploit spatial modal logic to bridge the gulf between the physical world of people and the virtual worlds of mobile agents.

Jiang and Landay [11] 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. IBM Aglets [13] provide a Java-based API for building mobile agents. Their security mechanism is based on the Java-2 security model: code is selectively trusted or not depending on its origin and/or the presence or absence of signatures. The LocALE (Location-Aware Lifecycle Environment) framework provides a CORBA-based mechanism to control the life-cycle (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. The difference between all three of these systems and our work is that none of them allow the specification of spatial mobility security policies.

Our work is inspired by the theoretical work on the Ambient Calculus [2] and the Ambient Logic [3]. Although our model is a great deal simpler than that proposed in the Ambient Calculus, it still allows us to combine together the physical world of people and the virtual world of mobile agents into a single, unified representation. The subset of the Ambient Logic used in our policy definitions remains computationally decidable and simple for humans to understand while still allowing the a great deal of flexibility and expressiveness.

5 Conclusion and Future Work

We have presented a technique for expressing spatial (i.e. location-based) security policies for mobile agents. These policies can be used to make both positive and negative assertions about the dynamic location of agents. Assertions may refer to the location of both physical and virtual objects in the world, a feature useful for location-aware Sentient applications. This technique provides a useful way to constrain the mobility of agents, to use mobile agent technology safely and to simplify the development process of future Sentient applications.

Our work was inspired by recent theoretical work on mobile computation, specifically the Ambient Calculus [2] and the Ambient Logic [3]. In future we would like to investigate how we could enhance our model of the world by adding in Ambient Calculus-style process expressions representing agents. Entities like people could be represented by mobile agents which have the capability to move anywhere at any time. The process expression associated with a mobile agent could be considered a characterisation of its behaviour – a contract with the system – which could be checked for consistency with global policy before the agent is allowed to run.

It is currently possible to do a small amount of up-front static checking of security policies: policies about locations which are known to be fixed can be checked at policy-install time. However, since we only have limited control over the physical environment we cannot do much about a policy which says, “the company laptop never leaves the building”. Clearly this policy could be violated by an individual picking up the laptop and walking home with it.[2] Additionally, some agents may wish to perform limited checking of policies at runtime. For example Alice’s `music player` agent from Sect. 3 may pose the question “if I enter Charlie’s office, will I definitely be allowed to leave?” in an attempt to avoid being trapped.

Perhaps the most interesting avenue for future work is to investigate how to scale the system up beyond one organisation. Our model of the world assumes that everything can be arranged in a single hierarchy, with a world controller in absolute control of everything. This approach might work adequately for a small organisation but to scale any further we need to cope with a multitude of problems: unreliable wide-area network communication, mutual distrust between organisations etc. One possibility is to employ

[2] We could envisage a system in which the doors are under our control and can be locked to prevent the laptop leaving. However consider that Health and Safety legislation would require the doors to automatically unlock if there was a fire!

a two-level approach where within an "intranet" agents are managed using this system while the "internet" case is handled differently.

A further enhancement to this work is to provide support for multiple simultaneous parallel hierarchies, allowing the same object to exist in several places at once. This facility could be used to represent different "views" of the same environment (for an analogy consider that a user may be present and active in more than one Internet "chat-room" simultaneously). When an object moves in one world they may also have to move in another. Reconciling these parallel views is an interesting topic of future work.

In summary, based on the research described in this paper we claim that our work provides a strong foundation for the building of Sentient, location-aware applications with a basis in theory. We believe that Sentient environments are an interesting niche for applications developed using mobile agent technology and our models help design these applications in a less ad-hoc manner.

Acknowledgements. This work was supported by the Schiff Foundation and the Engineering and Physical Sciences Research Council (EPSRC) and sponsored by AT&T Laboratories Cambridge Ltd. The authors would like to thank Richard Sharp for valuable comments and Andy Hopper for all his advice and support.

References

1. Luca Cardelli. Semistructured Computation. In *Proceedings of 7th International Workshop on Database Programming Languages, DBPL'99*, Kinloch Rannoch, Scotland, UK, September 1999.
2. Luca Cardelli and Andrew 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, 1998.
3. Luca Cardelli and Andrew D Gordon. Anytime, Anywhere Modal Logics for Mobile Ambients. In *Principles of Programming Languages (POPL)*, 2000.
4. World-Wide Web Consortium. XML Path Language (XPath) Specification, November 1999. http://www.w3.org/TR/xpath/.
5. Umeshwar Dayal, Eric N. Hanson, and Jennifer Widom. Active database systems. In *Modern Database Systems*, pages 434–456. 1995.
6. R. S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In M. Diekhans and M. Roseman, editors, *Fourth Annual Tcl/Tk Workshop (TCL 96)*, pages 9–23, Monterey, CA, 1996.
7. Andy Hopper. 1999 Sentient Computing. *Phil. Trans. R. Soc. Lond.*, 358(1):2349–2358, 2000.
8. Yannis E. Ioannidis and Timos K. Sellis. Conflict resolution of rules assigning values to virtual attributes. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, 1989.
9. Kjetil Jacobsen and Dag Johansen. Mobile Software on Mobile Hardware – Experiences with TACOMA on PDAs. Technical report, Department of Computer Science, University of Tromsø, Norway, 12 1997.
10. Kjetil Jacobsen and Dag Johansen. Ubiquitous Devices United: Enabling Distributed Computing Through Mobile Code. In *Proceedings of the Symposium on Applied Computing (ACM SAC'99)*, February 1999.

11. Xiaodong Jiang and James A. Landay. Modeling Privacy Control in Context-Aware Systems. *IEEE Pervasive Computing magazine*, 2002.
12. Dag Johansen, Robbert van Renesse, and Fred B Schneider. An Introduction to the TACOMA Distributed System Version 1.0. Technical report, Department of Computer Science, University of Tromsø, Norway, 6 1995.
13. Danny B. Lange and Mitsuru Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley, 1998.
14. K. Schelderup and J. Olnes. Mobile Agent Security — Issues and Directions. *Lecture Notes in Computer Science*, 1597:155–167, 1999.
15. David Scott, Alastair Beresford, and Alan Mycroft. Spatial policies for sentient mobile applications. Draft Manuscript. Available on web page http://www.recoil.org/ djs/papers/spatial02.html, December 2002.
16. J. Tardo and L. Valente. Mobile agent security and Telescript. In *IEEE CompCon '96*, pages 58–63, 1996.
17. Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, 1992.
18. Mark Weiser. The Computer for the 21st Century. *Scientific American*, 9 1991.