A Coordination Model for Sentient Computing

Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo

Dept. of Computer Science
University College London
Gower Street, London, WC1E 6BT, UK
{L.Capra|W.Emmerich|C.Mascolo}@cs.ucl.ac.uk

Abstract. Sentient computing is a computing paradigm that aims at making applications more responsive to their physical world. Environmental information is captured by sensors, and when changes that are relevant to the application (and its user) occur, actions are triggered. Realising this conceptually simple dialogue between the application and the environment poses a number of challenges, ranging from the difficulty and tedium of monitoring a variety of heterogeneous sensors to managing large sets of events triggered by the sensors, from processing raw sensor data to aggregating events in an application specific manner. In this paper we present a coordination model that orchestrates the interactions between applications and their execution environment. We describe a middleware architecture that realises this model, and report on its implementation and evaluation.

1 Introduction

Sentient computing [1] is a computing paradigm that aims at making applications more responsive and useful by observing and reacting to the physical world. Environmental information is captured by sensors, and when changes that are relevant to the application (and its user) occur, actions are triggered. This computing paradigm has been acknowledged to be particularly attractive in mobile computing where applications running on portable devices, such as PDA and mobile phones, are required to react to an always changing context while on the move. For example, information about current temperature, pressure and humidity could be gathered from different sensors and processed by an application on our PDA while we are on holiday to make an accurate weather forecast.

The rapid technological advances we have been witnessing in recent years are making sentient computing a reality. Modern households are flooded with technology and so are our cars. The infrastructure needed to make sentient computing a reality is also getting established, thanks to the rapid spread of wireless networks of increasing bandwidth at a lower cost. However, the development of the conceptually simple dialogue that forms the basis of sentient computing (i.e., 'monitor the environment until some condition holds and then trigger an action'), turns out to be quite complicated instead. Sentient applications may be required to respond to the stimuli provided by a variety of heterogeneous sensors (e.g., location sensors, light sensors, etc.). Large amounts of widely distributed data are produced by sensors, which applications will have to process before deciding whether a relevant change has occurred (e.g., an increase in the room temperature, together with the detection of smoke, may trigger a fire alert, while simple changes in the room temperature may not be regarded as 'relevant'). Processing sensor data is not an easy task in itself, as there is usually a serious gap between raw sensor data (e.g., location information of user A in terms of his x, y, z spatial coordinates) and application-relevant data (e.g., location of A inside a building).

It would be highly ineffective to require application developers to take care of implementing the sentient computing paradigm, thus re-inventing the wheel, each time a new sentient application has to be realised. Rather, a middleware based approach is called for, that takes care of coordinating applications with the physical world, so that developers can concentrate on the application's functionalities. It is the middleware responsibility to: create a mapping between the physical world, made of heterogeneous sensors, and the logical world that is presented to application programmers as a set of uniform abstractions; perform the monitoring of sensors and capture the events they produce at a high rate; correlate the events on behalf of the application in arbitrarily complex ways and, finally, trigger actions only when relevant changes have occurred. To the application programmer, the middleware must only present a simple abstraction of the physical world, together with an interface through which the key elements of this abstraction can be dynamically specified.

In recent years, coordination models for the pervasive environment have started to appear; however, they either do not take the dialogue between a device and its context into consideration, focusing on the coordination between interacting peers only, or they do not provide rich enough abstractions to model complex context conditions. In this paper, we propose a coordination model for sentient computing that exploits powerful abstractions and highly structured data to meet the goals listed above. Section 2 illustrates the abstractions used to close the gap between the logical and the physical environment. We then formally define the semantics for event filtering, event aggregation and reaction, that are the core actions describing the sentient computing dialogue. Because of the possibly high number and variety of sensors and devices that must be coordinated, and the consequently large amount of events that must be processed, issues of scalability and performance must be taken into consideration. Section 3 presents the design of a middleware layer that realises our sentient coordination model, and Section 4 reports on the middleware implementation and evaluation, to prove its suitability to the target scenario. In Section 5 we compare our work with others in the field and, finally, in Section 6 we draw our conclusions.

2 The Sentient Coordination Model

Fig. 1 depicts an overview of the sentient coordination model we have developed. As shown, coordination between the application and the physical world is mediated by a middleware layer.

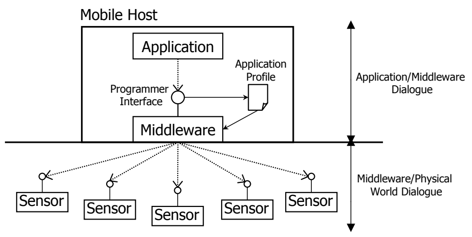

Fig. 1. Sentient Coordination Model Overview.

Dialogue between the Application and the Middleware. The main goal of the middleware is to hide the complexity and the heterogeneity of the environment to application developers, presenting them with an abstract model of the world that is easier to understand and manipulate. In particular, the physical world appears to applications as a collection of *resources*, where a resource may vary from a physical sensor (e.g., temperature sensor), to resources local to the device (e.g., battery), to other devices within reach (e.g., printers). We call *configuration* a snapshot of the resources that make up the execution environment of an application at any point in time. Not all resources are of interest to an application, nor are all possible configurations. For example, a weather forecast application may be interested in gathering information from a temperature sensor and a pressure sensor, while not being interested in the data that an audio sensor provides. Moreover, it may be interested in being informed when both temperature and pressure jointly decrease below certain thresholds (as an indication of possible rain), while not being interested in minor temperature/pressure oscillations. In our coordination model, each application is associated with an *application profile* where applications specify, in a simple declarative way, what are the configurations of interest to their execution, and what behaviour should occur when these configurations are entered.

Dialogue between the Middleware and the Physical World. For each running application, the middleware interacts with the resources the application is interested in (i.e., those that appear in the profile), so to acquire information about their status. These interactions may happen in a variety of ways, depending on the nature of the resources themselves. In this paper, we do not describe these specific, low level dialogues; rather, we assume that each interaction produces data about the current status of the resource. It is the middleware responsibility to collect raw data from the sensors, process them in an application-specific manner, filter out irrelevant information and trigger application-defined behaviours (as specified in the profile) when relevant configurations are entered.

In the reminder of this section, we formalise these two dialogues. First, we describe what information is encoded in application profiles; second, we define the semantics of profiles, with respect to the way they govern the coordination between the applications and the physical world. Finally, we illustrate the programmer interface that application developers are presented with to build sentient computing applications.

2.1 The Sentient Application Profile

As previously said, coordination between an application and the physical world is realised by means of an application profile. Each application profile contains information about *how* the application is willing to behave *when* particular configurations are entered, where each configuration is expressed in terms of conditions on homogeneously represented resources. For example, an application may be willing to know when room temperature increases beyond 24℃ and someone is in the room, so to automatically turn on the air conditioning. An application profile thus establishes *associations* between particular physical world *configurations* that depend on the status of one or more resources that the middleware monitors, and *policies* (i.e., behaviours) that the middleware has to trigger when such configurations are entered. The task of interacting with the actual resources to obtain updated information about their status, of checking if one of the encoded configurations is entered, and then of executing the corresponding behaviour, is automatically performed by the middleware, without the application having to implement these low-level tasks.

Fig. 3 illustrates the abstract syntax of an application profile. As shown, a profile contains hierarchically structured data in the form of associations (*policyList*) between policies (identified by name *pname*) and physical world configurations ($configList$) that determine when each policy should be fired. Each configuration ($config$) is uniquely identified inside the profile (*cid*), and describes the status of one or more resources (*resourceList*). Each resource ($resource$) is identified by a unique name (*rname*), and its status is described by the result of applying a resource-specific operator (*oname*) to a set of associated values (*valueList*). For example, (*temperature*, *inBetween*, $\{20, 25\}$) describes resource *temperature* having one of the possible values $\{20, 21, 22, 23, 24, 25\}$ (captured using a temperature sensor). We use a model for the physical world that is based on boolean algebra, which allows us to easily construct more complex configurations starting from atomic formulae, using the $\wedge$ (logical *and*) and $\vee$ (logical *or*) operators. An atomic configuration is represented by the 3-ary predicate: $<$ *rname oname valueList* $>$. Atomic formulae can then be combined using (implicit) $\wedge$ operators, to form more complex configurations to which a new *cid* is assigned; finally, various configurations can be combined using (implicit) $\vee$ operators and are then associated with a policy. In other words, each configuration expresses the set of resource conditions that must simultaneously hold ($\wedge$ operator) for a policy to be applied; these configurations are then put in $\vee$ relation, as the same policy may be enabled in different contexts.

Fig. 4 shows an example of application profile; the application requires notification (`highTemperatureAlert`) when the external temperature increases above 24℃. Also, whenever specific configurations are entered (`cid` 2 and 3), it requires the re-evaluation (`computeWeatherPrediction`) of the previously computed weather forecast. Note that, once temperature has increased above 24℃ and the corresponding alert has been fired, the application should not keep receiving notifications of the fact that temperature is above 24℃ every t time units (i.e., at the frequency at which middleware processes temperature data). The se-

$$\begin{aligned}
\mathit{\Sigma} &: \text{alphabet}\\
\mathrm{P} \subset \mathit{\Sigma}^* &: \text{set of all policy names}\\
\mathbb{N} &: \text{set of all natural numbers}\\
\mathrm{R} \subset \mathit{\Sigma}^* &: \text{set of all resource names}\\
\mathrm{O} \subset \mathit{\Sigma}^* &: \text{set of all operator names}\\
\mathrm{V} &: \text{set of all values of resources in R}\\
\mathrm{E} \subset \wp(\mathrm{R} \times \mathrm{V}) &: \text{set of all possible execution contexts}
\end{aligned}$$

Fig. 2. Application Profile - Domain Sets.

$$\begin{aligned}
profile &::= policyList \mid \varepsilon\\
policyList &::= policy\ policyList \mid policy\\
policy &::= pname\ configList\\
configList &::= config\ configList \mid config\\
config &::= cid\ resourceList\\
resourceList &::= resource\ resourceList \mid resource\\
resource &::= rname\ oname\ valueList\\
valueList &::= value\ valueList \mid \varepsilon
\end{aligned}$$

Fig. 3. Application Profile's Abstract Syntax. *pname* $\in$ P, *rname* $\in$ R, *cid* $\in \mathbb{N}$, *value* $\in$ V, *oname* $\in$ O, being P, R, $\mathbb{N}$, V, and O the domain sets listed in Fig. 2.

mantics we attach to each association is therefore the following. Configurations are independent: if a policy has more than one configuration associated with it, any of them ($\vee$ semantics) can cause the policy to be fired. Moreover, each configuration determines a partition of the context space; for example, `cid=1` determines a partition between a context where temperature is greater than 24℃, and one where its value is less than 24℃. When the temperature raises above 24℃ (i.e., when the physical world of interest to the application enters the first element of the partition), the policy is fired; before this policy is fired again, changes must have happened that have brought the physical world into a different element of the partition. For example, when temperature increases above 24℃ for the first time, a `highTemperatureAlert` is triggered. Before `cid=1` causes the policy to be fired again, temperature must have decreased below 24℃, and then gone back above 24℃; therefore, while temperature keeps increasing, the policy is not fired repeatedly, thus avoiding an undesired sequence of alerts.

A formalisation of this behaviour is shown in Fig. 6 to 8. The domain sets of the semantic functions presented here can be found in Fig. 5. When an application is started, the middleware loads its profile and processes it, in order to associate a boolean value equal to *true* to each *resource* of the configurations encoded in the profile itself (function *init* in Fig. 6). In order for a configuration to be enabled, and the associated policy to be fired, all resource conditions ex-

```
highTemperatureAlert
    1
        temperature greaterThan 24

computeWeatherPrediction
    2
        temperature greaterThan 20
        pressure greaterThan 1030
        humidity lowerThan 70
    3
        temperature inBetween [15,20]
        pressure inBetween [1010,1030]
        humidity inBetween [70,80]
    . . .
```

Fig. 4. Application Profile Example.

pressed in the configuration must hold, and the associated boolean values set to *true*. As we are going to show, we use these boolean values to prevent cascading policies, as they represent a sort of 'firability' pre-condition.

In order for a policy to be fired, two conditions must be met (see Fig. 7): at least one of its associated configurations evaluates to *true* (i.e., it is enabled) in the current environment (i.e., all resource conditions hold), *and* all the boolean values of resources that make up this configuration are *true*. We use the auxiliary boolean function *eval*, such that $eval((rname, oname, vList), e)$ returns *true* if the value of resource *rname* in environment *e* is among the values obtained by applying the operator *oname* to *vList*. For example, $eval((temperature, inBetween, \{20, 25\}), \{(temperature, 22)\}) = \top$. If the current state of a sensor cannot be obtained (e.g., because of sensor failure), the *eval* function returns $\bot$. Once a policy has been executed, we prevent it from being repeatedly fired by setting the boolean value of each resource in the enabling configuration to $false$ (Fig. 8).

Boolean values representing firability pre-conditions are re-set to *true* by the *update* function. Middleware re-evaluates the status of resources and updates the boolean value previously associated with each resource in the following way: if

$$\begin{aligned}
bool &: \{\top,\ \bot\}\\
resourceStatus &: \mathrm{R} \times \mathrm{O} \times \wp(V) \times bool\\
resourceStatusList &: \wp(resourceStatus)\\
configStatus &: \mathbb{N} \times \wp(resourceStatusList)\\
configStatusList &: \wp(configStatus)\\
policyStatus &: \mathrm{P} \times \wp(configStatusList)\\
policyStatusList &: \wp(policyStatus)
\end{aligned}$$

Fig. 5. Application Profile Semantic Functions - Domain Sets.

$$
\begin{aligned}
init &: policyList \rightarrow policyStatusList \\
init_{cl} &: configList \rightarrow configStatusList \\
init_{rl} &: resourceList \rightarrow resourceStatusList
\end{aligned}
$$

$$
\begin{aligned}
init[\![policy\ policyList]\!] &= init[\![policy]\!] \cup init[\![policyList]\!] \\
init[\![policy]\!] &= init[\![pname\ configList]\!] \\
init[\![pname\ configList]\!] &= \{(pname,\ init_{cl}[\![configList]\!])\} \\
init_{cl}[\![config\ configList]\!] &= init_{cl}[\![config]\!] \cup init_{cl}[\![configList]\!] \\
init_{cl}[\![config]\!] &= init_{cl}[\![cid\ resourceList]\!] \\
init_{cl}[\![cid\ resourceList]\!] &= \{(cid,\ init_{rl}[\![resourceList]\!])\} \\
init_{rl}[\![resource\ resourceList]\!] &= init_{rl}[\![resource]\!] \cup init_{rl}[\![resourceList]\!] \\
init_{rl}[\![resource]\!] &= init_{rl}[\![rname\ oname\ valueList]\!] \\
init_{rl}[\![rname\ oname\ valueList]\!] &= \{(rname,\ oname,\ valueList,\ \top)\}
\end{aligned}
$$

Fig. 6. Application Profile Semantics - (*init*).

$$
\begin{aligned}
fire\ &:\ policyStatusList \rightarrow \mathrm{E} \rightarrow \wp(\mathrm{P} \times \mathbb{N}) \\
fire[\![pStatus\ psList]\!]_e &= fire[\![pStatus]\!]_e \cup fire[\![psList]\!]_e \\
fire[\![pStatus]\!]_e &= fire[\![(pname,\ csList)]\!]_e \\
fire[\![(pname,\ csList)]\!]_e &= \begin{cases} \{(pname, cid)\} \text{ if } \exists\ cStatus = (cid,\ rsList) \in csList\ | \\ \quad \forall\ rStatus = (rname, oname, vList, b) \in rsList, \\ \quad eval((rname, oname, vList), e) = \top\ \wedge\ b = \top \\ \\ \emptyset \text{ otherwise} \end{cases}
\end{aligned}
$$

Fig. 7. Application Profile Semantics - (*fire*).

the boolean value is set to *false*, and the current resource value does not respect the condition expressed by that resource in the profile, the value is changed to *true*; otherwise, the boolean value is not altered (Fig. 9).

Let us refer to the example shown in Fig. 4, where we have encoded a condition $(temperature, greaterThan, 24)$ in configuration 1. When the application is started, a boolean value equal to *true* is associated with the temperature condition; as soon as a change brings the temperature value above 24℃ the `highTemperatureAlert` policy is fired, and the boolean value is changed to *false*; as long as temperature stays above 24℃ configuration 1 does not enable this policy, as the boolean value stays *false*. A change in the environment that brings temperature below 24℃ will cause the boolean value to be set to *true* (*update* function), and the next time temperature increases above 24℃ the `highTemperatureAlert` policy will be fired again.

$$
\begin{aligned}
\mathit{reset} &: \mathit{policyStatusList} \rightarrow \wp(\mathrm{P} \times \mathbb{N}) \rightarrow \mathit{policyStatusList} \\
\mathit{reset}_{csl} &: \mathit{configStatusList} \rightarrow \wp(\mathrm{P} \times \mathbb{N}) \rightarrow \mathit{configStatusList} \\
\mathit{reset}_{rsl} &: \mathit{resourceStatusList} \rightarrow \wp(\mathrm{P} \times \mathbb{N}) \rightarrow \mathit{resourceStatusList}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{reset}[\![\mathit{pStatus}\ \mathit{psList}]\!]_{pcl} &= \mathit{reset}[\![\mathit{pStatus}]\!]_{pcl} \cup \mathit{reset}[\![\mathit{psList}]\!]_{pcl} \\
\mathit{reset}[\![\mathit{pStatus}]\!]_{pcl} &= \mathit{reset}[\![(\mathit{pname}, \mathit{csList})]\!]_{pcl} \\
\mathit{reset}[\![(\mathit{pname}, \mathit{csList})]\!]_{pcl} &= \begin{cases} \cup\{(\mathit{pname}, \mathit{reset}_{csl}[\![\mathit{csList}]\!]_{\{(p_i,c_i)\}})\}\ \forall\ i \mid \\ \quad \exists\ (p_i, c_i) \in \mathit{pcl},\ \mathit{pname} = p_i \\ \\ \{(\mathit{pname}, \mathit{csList})\} \text{ otherwise} \end{cases} \\
\mathit{reset}_{csl}[\![\mathit{cStatus}\ \mathit{csList}]\!]_{\{(p,c)\}} &= \mathit{reset}_{csl}[\![\mathit{cStatus}]\!]_{\{(p,c)\}} \cup \mathit{reset}_{csl}[\![\mathit{csList}]\!]_{\{(p,c)\}} \\
\mathit{reset}_{csl}[\![\mathit{cStatus}]\!]_{\{(p,c)\}} &= \mathit{reset}_{csl}[\![(\mathit{cid}, \mathit{rsList})]\!]_{\{(p,c)\}} \\
\mathit{reset}_{csl}[\![(\mathit{cid}, \mathit{rsList})]\!]_{\{(p,c)\}} &= \begin{cases} \{(\mathit{cid}, \mathit{reset}_{rsl}[\![\mathit{rsList}]\!]_{\{(p,c)\}})\} \text{ if } \mathit{cid} = c \\ \{(\mathit{cid}, \mathit{rsList})\} \text{ otherwise} \end{cases} \\
\mathit{reset}_{rsl}[\![\mathit{rStatus}\ \mathit{rsList}]\!]_{\{(p,c)\}} &= \mathit{reset}_{rsl}[\![\mathit{rStatus}]\!]_{\{(p,c)\}} \cup \mathit{reset}_{rsl}[\![\mathit{rsList}]\!]_{\{(p,c)\}} \\
\mathit{reset}_{rsl}[\![\mathit{rStatus}]\!]_{\{(p,c)\}} &= \mathit{reset}_{rsl}[\![(\mathit{rname}, \mathit{oname}, \mathit{vList}, b)]\!]_{\{(p,c)\}} \\
\mathit{reset}_{rsl}[\![(\mathit{rname}, \mathit{oname}, \mathit{vList}, b)]\!]_{\{(p,c)\}} &= \{(\mathit{rname}, \mathit{oname}, \mathit{vList}, \bot)\}
\end{aligned}
$$

Fig. 8. Application Profile Semantics - (*reset*).

$$
\begin{aligned}
\mathit{update} &: \mathit{policyStatusList} \rightarrow \mathrm{E} \rightarrow \mathit{policyStatusList} \\
\mathit{update}_{csl} &: \mathit{configStatusList} \rightarrow \mathrm{E} \rightarrow \mathit{configStatusList} \\
\mathit{update}_{rsl} &: \mathit{resourceStatusList} \rightarrow \mathrm{E} \rightarrow \mathit{resourceStatusList}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{update}[\![\mathit{pStatus}\ \mathit{psList}]\!]_{e} &= \mathit{update}[\![\mathit{pStatus}]\!]_{e} \cup \mathit{update}[\![\mathit{psList}]\!]_{e} \\
\mathit{update}[\![\mathit{pStatus}]\!]_{e} &= \mathit{update}[\![(\mathit{pname}, \mathit{csList})]\!]_{e} \\
\mathit{update}[\![(\mathit{pname}, \mathit{csList})]\!]_{e} &= \{(\mathit{pname}, \mathit{update}_{csl}[\![\mathit{csList}]\!]_{e})\} \\
\mathit{update}_{csl}[\![\mathit{cStatus}\ \mathit{csList}]\!]_{e} &= \mathit{update}_{csl}[\![\mathit{cStatus}]\!]_{e} \cup \mathit{update}_{csl}[\![\mathit{csList}]\!]_{e} \\
\mathit{update}_{csl}[\![\mathit{cStatus}]\!]_{e} &= \mathit{update}_{csl}[\![(\mathit{cid}, \mathit{rsList})]\!]_{e} \\
\mathit{update}_{csl}[\![(\mathit{cid}, \mathit{rsList})]\!]_{e} &= \{(\mathit{cid}, \mathit{update}_{rsl}[\![\mathit{rsList}]\!]_{e})\} \\
\mathit{update}_{rsl}[\![\mathit{rStatus}\ \mathit{rsList}]\!]_{e} &= \mathit{update}_{rsl}[\![\mathit{rStatus}]\!]_{e} \cup \mathit{update}_{rsl}[\![\mathit{rsList}]\!]_{e} \\
\mathit{update}_{rsl}[\![\mathit{rStatus}]\!]_{e} &= \mathit{update}_{rsl}[\![(\mathit{rname}, \mathit{oname}, \mathit{vList}, b)]\!]_{e} \\
\mathit{update}_{rsl}[\![(\mathit{rname}, \mathit{oname}, \mathit{vList}, b)]\!]_{e} &= \begin{cases} \{(\mathit{rname}, \mathit{oname}, \mathit{vList}, \top)\} \\ \quad \text{if } \mathit{eval}((\mathit{rname}, \mathit{oname}, \mathit{vList}), e) = \top\ \wedge\ b = \bot \\ \{(\mathit{rname}, \mathit{oname}, \mathit{vList}, b)\} \text{ otherwise} \end{cases}
\end{aligned}
$$

Fig. 9. Application Profile Semantics - (*update*).

After an initialisation process that takes place when an application is started (time t_0), coordination between the application and the physical world is determined by the application profile in the following way:

$$\begin{aligned}
t_0 &: PSL_0 = init[\![policyList]\!] \\
t_1 &: P_1 = fire[\![update[\![PSL_0]\!]_{e_1}]\!]_{e_1} \\
&\quad PSL_1 = reset[\![PSL_0]\!]_{P_1} \\
&\dots \\
t_i &: P_i = fire[\![update[\![PSL_{i-1}]\!]_{e_i}]\!]_{e_i} \\
&\quad PSL_i = reset[\![PSL_{i-1}]\!]_{P_i}
\end{aligned} \tag{1}$$

where $PSL_i \in$ *policyStatusList* associates boolean values with resource conditions, and $P_i \in \wp(\mathrm{P} \times \mathbb{N})$ states what policies have been fired, and what configurations have enabled them.

2.2 The Sentient Application Programmer Interface

Application profiles encode how applications (and their users) are willing to coordinate with the physical world. As user's needs may vary over time, applications must be allowed to customise the information here encoded while executing, that is, to dynamically specify what and how environmental changes should be handled. We exploit the principle of *reflection* [2] to attain this goal. Reflection has been recognised to be a powerful and elegant way to make the system it is applied to adaptable to its environment and better able to cope with changes [3]. The approach demands that an explicit representation of middleware behaviour, with respect to the above running applications, is maintained; we do this by means of application profiles. Reflection then allows both dynamic *inspection* and *adaptation* of middleware behaviour by means of a meta-interface that the middleware offers to applications to access this explicit representation.

Fig. 10 lists the semantic functions provided by the meta-interface to dynamically access application profiles. As shown, access to the profiles may occur at various levels of granularity: from the resources associated with a configuration, to the configurations associated with a policy, up to the policies that constitute the root elements of the associations in the profiles. Inspection is based on unique names for policies and resources, and on ids for context configurations; we use *null* to indicate an empty search result. Adaptation takes place by adding, removing and updating the elements of the associations[1].

Note that adaptation of application profiles causes alterations to the *policyStatusList* PSL_i (see equation 1) that is used by the middleware to decide what policies to fire as a result of context changes. In order for a policy to be fired, both the information encoded in a profile, and the 'firability' status of its

[1] The semantics of these functions is trivial and is thus omitted; interested readers may refer to [4] for a complete semantics specification.

$$\begin{aligned}
readPolicy &: profile \times \mathrm{P} \rightarrow policy \cup \{null\}\\
readConfig &: profile \times \mathrm{P} \times \mathbb{N} \rightarrow config \cup \{null\}\\
readResource &: profile \times \mathrm{P} \times \mathbb{N} \times \mathrm{R} \rightarrow resource \cup \{null\}\\
\\
remPolicy &: profile \times \mathrm{P} \rightarrow profile\\
remConfig &: profile \times \mathrm{P} \times \mathbb{N} \rightarrow profile\\
remResource &: profile \times \mathrm{P} \times \mathbb{N} \times \mathrm{R} \rightarrow profile\\
\\
addPolicy &: profile \times policy \rightarrow profile\\
addConfig &: profile \times \mathrm{P} \times config \rightarrow profile\\
addResource &: profile \times \mathrm{P} \times \mathbb{N} \times resource \rightarrow profile\\
\\
updPolicy &: profile \times \mathrm{P} \times policy \rightarrow profile\\
updConfig &: profile \times \mathrm{P} \times \mathbb{N} \times config \rightarrow profile\\
updResource &: profile \times \mathrm{P} \times \mathbb{N} \times \mathrm{R} \times resource \rightarrow profile
\end{aligned}$$

Fig. 10. Reflective Meta Interface.

$$\begin{aligned}
remPSPolicy &: policyStatusList \times \mathrm{P} \rightarrow policyStatusList\\
remPSConfig &: policyStatusList \times \mathrm{P} \times \mathbb{N} \rightarrow policyStatusList\\
remPSResource &: policyStatusList \times \mathrm{P} \times \mathbb{N} \times \mathrm{R} \rightarrow policyStatusList\\
\\
addPSPolicy &: policyStatusList \times policy \rightarrow policyStatusList\\
addPSConfig &: policyStatusList \times \mathrm{P} \times config \rightarrow policyStatusList\\
addPSResource &: policyStatusList \times \mathrm{P} \times \mathbb{N} \times resource \rightarrow policyStatusList\\
\\
updPSPolicy &: policyStatusList \times \mathrm{P} \times policy \rightarrow policyStatusList\\
updPSConfig &: policyStatusList \times \mathrm{P} \times \mathbb{N} \times config \rightarrow policyStatusList\\
updPSResource &: policyStatusList \times \mathrm{P} \times \mathbb{N} \times \mathrm{R} \times resource \rightarrow policyStatusList
\end{aligned}$$

Fig. 11. Implicit Meta Interface.

associated resources are checked. Therefore, each time an operation that changes the profile is performed, the *policyStatusList* is implicitly updated, by internally calling the semantic functions listed in Fig. 11; the new *policyStatusList* obtained is then used in equation 1 to decide what policies to fire[2].

[2] As before, the semantics of these functions is trivial and is not reported; interested readers may find a complete semantics specification in [4].

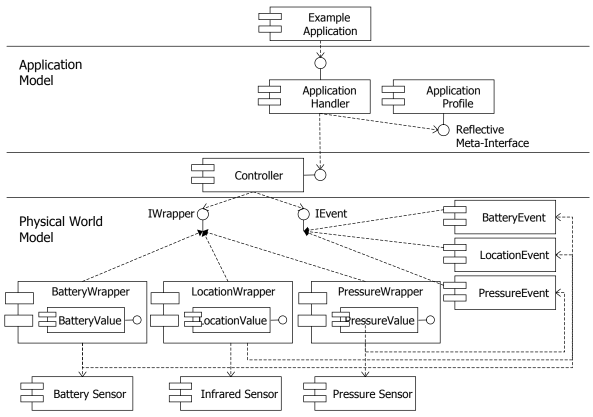

Fig. 12. Sentient Computing Architecture.

3 The Sentient Computing Architecture

Fig. 12 provides an overview of a sentient computing architecture that realises our coordination model for a sample running application. Middleware services (e.g., discovery) that are not the focus of the paper have not been depicted.

For each *abstract* resource an application is interested in (i.e., for each resource listed in the profile), a *wrapper* exists that is able to interact with the *physical* resource (i.e., *sensor* in the picture), and to process the information thus obtaining a *value* that the application understands. Different wrappers may exist that interact with the same sensor, but that process information in a different manner; or, vice versa, the same wrapper may interact with various sensors, to synthesise context information in an application-specific manner. Only wrappers of resources that running applications are interested in are loaded, to avoid wasting computational resources, already scarce on a portable device. Fig. 12 illustrates a Battery Wrapper, interacting with a Battery Sensor (i.e., a primitive of the operating system), a Location Wrapper, interacting with an Infrared Sensor, and a Pressure Wrapper, interacting with a Pressure Sensor.

Wrappers export an interface that is used to register the interest of an application in resource-specific environmental conditions. For example, if an application specifies, in its profile, an interest in the following configuration:

```
1
    temperature greaterThan 24
    humidity lowerThan 65
```

then the Temperature Wrapper is instructed to monitor the first condition, while the Humidity Wrapper is instructed to monitor the second. Wrappers maintain information about the status of the associated resources and independently produce an *event* only when the evaluation of the conditions that they monitor changes (from *true* to $false$ and viceversa).

These events are captured and processed by a *Controller* component whose main tasks are: first, to process the profiles of running applications, both at application startup and whenever a profile is changed, so to break the configurations an application is interested in into their constituents, and then pass individual resource conditions to the relevant wrappers. Second, the controller collects the various individual events that wrappers produce and combine them to check whether an application-defined configuration has been entered; in this case, the corresponding action is triggered, thus realising the coordination model whose semantics has been defined in Section 2.

The dialogue between the application and the middleware is mediated by an *ApplicationHandler*. Each time an application is started, an ApplicationHandler is created to fetch the application profile and to pass it to the Controller component for processing. During the lifetime of the application, the handler is responsible for providing applications access to their own application profile through a well-defined *reflective meta-interface* that realises the sentient application programmer interface described in the previous section.

4 Implementation and Evaluation

The sentient computing architecture described above has been implemented using the Java 2 Micro Edition (Connected Device Configuration, Personal Profile) [5]. Application profiles have been encoded using XML [6], and the KXML2 [7] parser has been deployed as a building block to realise the reflective meta-interface. Additional components that are not strictly related to this work have been implemented, in order to make the middleware usable in practice; in particular, we have implemented a multicast system to advertise and discover remotely available resources[3]. In total, our implementation occupies less than 130KB (compressed), including the KXML2 parser, making it suitable for mobile devices. On top of it, we have developed a smart conference application, that is able to interact with various resources, for example, to provide guidance about where the next talk is going to be, to chat with some attendees when they are connected, and so on. As a proof of concept, we have implemented two sample components to monitor the status of the physical world: a component that emulates the amount of battery power remaining, and a component that interacts with an external location sensor (implemented as an http server). The application and two components occupy 18KB as a Jar archive.

We already pointed out that crucial issues for sentient computing systems are performance and scalability. Applications may express interest in the status of various resources, so that large amounts of data may be processed by the middleware. In order to prove the scalability and efficiency of our middleware (for plausible profile configurations), we have implemented a benchmark application that enables customisation of the information encoded in the application

[3] In this work, we made the assumption that resources (e.g., sensors, devices, etc.) are one hop away from us.

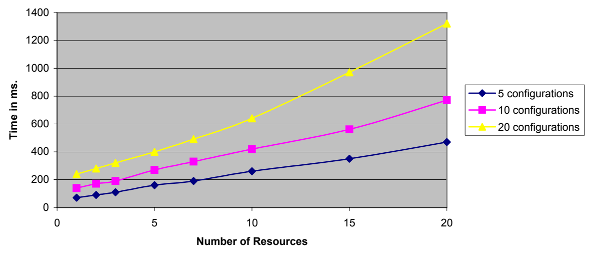

Fig. 13. Performance Evaluation.

profile in terms of: number of associations, number of configurations per association, and number of resource conditions per configuration. In our experiments, we considered the overhead required by the wrapper to process sensor data in an application specific manner to be negligible; we believe this assumption to be acceptable, as sensors greatly vary in nature, and therefore may introduce overheads of different orders of magnitude (e.g., knowing the amount of battery left requires much less time than gathering and processing location information) that we cannot compare.

The chart in Fig. 13 summarises some of the results we observed. All tests were performed on Dell Latitude laptops equipped with 128MB RAM, Intel Pentium II processors rated at 300MHz, and connected in an ad-hoc network using Cisco Aironet 340 10Mbps wireless cards. We simulated a very high traffic of events (every second new events were generate by the wrappers - indicating a change in the individual resource condition satisfiability - and passed to the Controller component). As shown, even for profiles of high complexities (up to 10 configurations of 15-20 different resource conditions each), the time taken by the Controller to process the received events and trigger an action is below or around 500ms. During the evaluation, we noted that, as expected, increasing the number of configurations (while keeping the number of resources associated with each of them constant) had a stronger performance impact than increasing the number of resources associated with a configuration. This is because of the and/or semantics used in the profiles: resource conditions follow the $\wedge$ semantics, so that as soon as one resource condition fails to be true, all remaining resource conditions associated with the same configuration need not be evaluated. On the contrary, configurations associated to the same policy follow the $\vee$ semantics, so that all of them have to be checked as any could enable the associated policy.

5 Discussion

The sentient computing paradigm increases the need for coordination support between the application and the highly dynamic physical world they execute in. Various coordination models have been proposed to date, mainly to support coordination among mobile entities: for example, JEDI [8] offers a publish-subscribe paradigm where nodes interact by exchanging events through a logically centralized event dispatcher; MARS [9] defines coordination among logically mobile agents that migrate from one node to another. A common limitation to these approaches is that they focus on the dialogue happening among peer devices, without taking into consideration the dialogue that happens between a device and its physical environment. Supporting the latter type of coordination is of paramount importance, in order to enable sentient computing applications to adapt to their changing execution environment.

Different aspects of the dialogue between the application and the physical world have been addressed. For example, the Context Toolkit [10] tackles the problem of heterogeneity by providing a uniform interface to perform context sensing; however, it does not address the need for a distributed coordination model. Odyssey [11] provides a reaction mechanism to detect local changes; however, detection of remote changes is not supported. Recently, more comprehensive coordination models have been developed to orchestrate the dialogue between applications and context. Lime [12], for example, manipulates context as data, and supports coordination between applications and their execution environment via tuple-space based coordination primitives. However, each context condition refers to the status of a single resource so that the burden of processing independent events into complex application-relevant configurations is on the application. Similar considerations apply to Limone [13] and to EgoSpaces [14], where the same tuple-space based reactive mechanisms are used. We believe our model moves a step forward in reducing the programming effort required to build sentient computing applications that need coordination with a wide range of physical resources.

6 Conclusions

In this paper we have presented a coordination model that supports the sentient computing paradigm. Developers build applications that can coordinate with the physical world by means of an easy-to-use interface that is based on the powerful abstraction of application profiles. An application profile defines how the application is wishing the dialogue between itself and a (homogeneously presented) physical world to happen, in terms of policies that should be triggered when certain environmental configurations are entered. It is the middleware responsibility to break up this information into its constituents, to interact with the individual resources that compose the environment of interest to the application, to reassemble the data coming from these resources in an application-specific manner, and to trigger application-defined reactions when particular configurations are entered. The semantics of these steps have been defined and an architecture that realises the sentient coordination model has been illustrated and evaluated.

Our experience in building the conference application has shown us that further work is required to correlate environment information in even more complex ways. In particular, besides the boolean logic operators, temporal operators seem to be required, for example, to express an application interest in a sequence of environment changes. It is our plan to extend the coordination model presented in this paper to include these operators. Moreover, rather than considering unsatisfied those conditions that refer to resources whose status cannot be currently obtained (e.g., because of sensor failure or unreachability), we plan to extend our semantics with an *undefined* value, so to treat these conditions differently.

References

1. Hopper, A.: The Royal Society Clifford Paterson Lecture, 1999 - Sentient Computing. Phil. Trans. R. Soc. Lond. **358** (2000) 2349–2358
2. Smith, B.: Reflection and Semantics in a Procedural Programming Language. Phd thesis, MIT (1982)
3. Eliassen, F., Andersen, A., Blair, G.S., Costa, F., Coulson, G., Goebel, V., Hansen, O., Kristensen, T., Plagemann, T., Rafaelsen, H.O., Saikoski, K.B., Yu, W.: Next Generation Middleware: Requirements, Architecture and Prototypes. In: Proceedings of the 7^{th} IEEE Workshop on Future Trends in Distributed Computing Systems, IEEE Computer Society Press (1999) 60–65
4. Capra, L.: Reflective Mobile Middleware for Context-Aware Applications. PhD thesis, University College London (2003)
5. Sun Microsystem, I.: Java 2 Platform, Micro Edition. http://java.sun.com/j2me/ (2000)
6. Bray, T., Paoli, J., Sperberg-McQueen, C.M.: Extensible Markup Language. Recommendation http://www.w3.org/TR/1998/REC-xml-19980210, World Wide Web Consortium (1998)
7. : The KXML2 XML parser. http://www.kxml.org/ (2004)
8. Cugola, G., Nitto, E.D., Fuggetta, A.: The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. IEEE Transactions on Software Engineering **27** (2001) 827–850
9. Cabri, G., Leonardi, L., Zambonelli, F.: MARS: A programmable coordination architecture for mobile agents. IEEE Internet Computing **4** (2000) 26–35
10. Dey, A., Salber, D., Abowd, G.: A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Human-Computer Interaction (HCI) Journal, special issue on Context-Aware Computing **16** (2001) 97–166
11. Satyanarayanan, M.: Mobile Information Access. IEEE Personal Communications **3** (1996) 26–33
12. Murphy, A., Picco, G.P.: Using Coordination Middleware for Location-Aware Computing: A Lime Case Study. In: Proc. of the 6^{th} International Conference on Coordination Models and Languages (Coordination 2004). Volume 2949 of Lecture Notes in Computer Science., Pisa, Italy, Springer-Verlag (2004) 263–278
13. Fok, C.L., Roman, G.C., Hackmann, G.: Lightweight Coordination Middleware for Mobile Computing. In: Proc. of the 6^{th} International Conference on Coordination Models and Languages (Coordination 2004). Volume 2949 of Lecture Notes in Computer Science., Pisa, Italy, Springer-Verlag (2004) 135–151
14. Julien, C., Roman, G.C.: Active Coordination in Ad Hoc Networks. In: Proc. of the 6^{th} International Conference on Coordination Models and Languages (Coordination 2004). Volume 2949 of Lecture Notes in Computer Science., Pisa, Italy, Springer-Verlag (2004) 199–215