A Component-Based Platform for Accessing Context in Ubiquitous Computing Applications

Chistophe Jacquet, Yolaine Bourda and Yacine Bellik

Author preprint. Publication details:
Journal of Ubiquitous Computing and Intelligence, 2007, Vol. 1, No. 2. Pages 163-173. American Scientific Publishers.

***Abstract*— Based upon a conceptual model for ubiquitous computing systems, this article presents a component-based platform for building context-aware applications. This platform introduces a high-level service to abstract context and to allow the rapid construction of dynamically reconfigurable applications. Moreover, the inputs and outputs of context components benefit from a strong typing, which permits design-time checks that can detect specification mistakes. At the end of the article, we introduce an implementation for this platform.**

***Index Terms*— Ambient Intelligence, Context-Aware Computing, Ubiquitous Computing, Sensor Fusion, Middleware.**

I. Introduction

PEOPLE interact with "classical" computer systems while sitting at their desks. In contrast, ubiquitous computing systems are meant to be used anywhere, and in a *context-aware* fashion. To this end, these systems must be capable of (1) capturing their *context* of use, and (2) *performing actions* on their environment.

In this article, we mainly focus on the design of a platform to *capture* context. With "context", we mean what Anind K. Dey describes in his PhD dissertation [1]: *"any information that can be used to characterize the situation of an entity"*, where an entity is either a person, a place, or an object.

At first sight, it is very complicated to build a context-aware system because one has to deal with low-level communication protocols with sensors as well as higher-level algorithms to infer information on context.

Indeed, a context-aware system has to capture information about its environment through physical sensors (temperature, light, localization, etc.), and then aggregate these basic data to infer higher-level context information.

This is the reason why several platforms for capturing context have already been proposed. Their primary goal is to ease the tasks of creating, maintaining and extending context-aware systems [2]. They try to handle themselves the most tedious tasks, so as to aid application designers.

However, existing platforms lack (1) advanced methods to access high-level context, and (2) static consistency checks between the inputs and outputs of their building blocks.

In consequence, it is not possible to use context without being strongly tied to one given infrastructure for context capture. In general, changing something in this infrastructure means stopping, and possibly recompiling, applications. Moreover, without static type checking, it is possible to chain components inconsistently, thus leading to runtime bugs hard to investigate.

For these reasons, our platform proposal introduces two original aspects:

- the notion of *object hive*, an abstraction for accessing context, which proposes a weak coupling between context capture and context use, as well as the possibility to reconfigure applications in a dynamic fashion,
- a strong typing for all communications between software components.

Section II summarizes what existing context-aware platforms have already proposed. Section III introduces our conceptual model and the vocabulary in use in this paper. Section IV comprehensively describes our platform and its original aspects. More precisely, Section V deals with the notion of object hive. Finally, Section VI presents a test implementation for our platform.

II. Review of Existing Frameworks for Context-Aware Computing

In this section, we briefly present the main frameworks that have been proposed to support the building of context-aware applications.

Some systems are focused on presenting (and possibly exchanging) information in a context-aware fashion. For instance, the Stick-e notes system [3] can associate multimedia objects to different kinds of context (but mainly localization context). In the example given by the authors, context capture is *ad-hoc*: geographical coordinates are simply acquired by GPS[1]. The system then allows the display of context-dependent information items.

In the CoolTown system [4], real world objects are identified by URLs[2], which allows objects to exchange information. In this system too, context capture components that are capable of retrieving URLs for physical objects are *ad-hoc* and hard-wired. Moreover, this system proposes only a coarse-grained context-capture, and does not allow to retrieve precise context properties.

Conversely, in Bill Schilit's thesis, defended in 1995 [5], *device agents* can collect precise information about the status of common office appliances. However, each device agent is specifically designed to match the characteristics of the appliance, the sensors involved, and the ways to access them.

[1] Global Positioning System
[2] Uniform Resource Locator

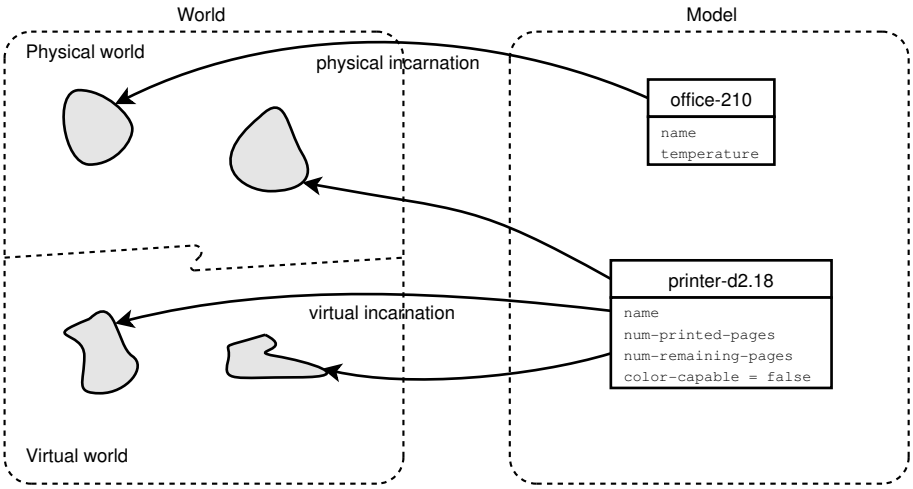

Fig. 1. Relationships between objects of the world and objects of the model.

A decisive step was taken forward by Anind K. Dey [6] with the introduction of the *context toolkit*, whose goal is to ease the design and implementation of context-aware systems. Dey's contribution is built around the concept of *context widget*. In the same way as widgets hide intricate interaction details from graphical user interface designers, context widgets embed either physical sensors or context transformation and aggregation operations. In these cases, context widgets are respectively called *interpretors* and *aggregators*. Context is aggregated by a chain (or a network) of widgets. Applications are directly connected to the outputs of the chain (or network). In consequence, context widgets are assembled specifically for a given application.

The *contextor* abstraction [7] introduced by Rey et al. refines and complements the notion of context widget. This system eases the identification of context components (called *contextors*) needed by applications. However, once useful contextors have been identified, the application is directly connected to a network of contextors. Other platforms have similar proposals, as far as the position of applications is concerned, for instance in the *Sentient Object Model* [8] proposed by Biegel and Cahill.

We think that context capture infrastructures should not be specific to a given application. Instead, it should be possible to specify context capture in a given environment in a *generic way*. It would allow:

- any application to query and use the context,
- the designer to change the context capture infrastructure at runtime, without needing to recompile or even stop the existing applications.

The existing platforms do not completely address these issues, so we designed the *object hive* abstraction to fill this gap. Section V explains how.

III. Conceptual Model

To begin with, we define the concepts underlying our model that is based on two fundamental notions : the *model*, where abstract representations of objects live, and the *world* (both physical and virtual), where *incarnations* of model objects live (see fig. 1).

A. World

We call *world* the set of all the objects in interaction with the user. They can be both physical and virtual. It is therefore possible to distinguish between the *physical world* on the one hand, and the *virtual world* on the other hand (fig. 1).

The term *physical world* refers to all the objects that are governed by physical laws. It is the world human beings live in. Conversely, the term *virtual world* refers to environments composed of imaginary computer-generated objects that the user can interact with through virtual and mixed reality applications.

In *augmented reality* systems, interactions happen mainly in the physical world whose physical objects are augmented by virtual properties. While these systems are quite popular, others have proposed *augmented virtuality* systems, where interactions happen in a computer-generated world that is augmented by elements taken from the physical world [9].

In fact, Milgram has shown [10] that it is very difficult to precisely define the concepts of *reality*, *augmented reality*, *augmented virtuality* and *virtual reality*. Instead, he introduces a continuum that ranges from pure reality (the physical world) to pure virtuality (virtual worlds): the *reality-virtuality continuum* (fig. 2).

Fig. 2. The reality-virtuality continuum ranges from pure reality to pure virtuality.

When thinking of virtual objects, one often imagines only images displayed on head-mounted displays, caves and so on. However, virtual worlds can possibly rely heavily on other senses, such as audio and tactile sensations.

Example — A secretary typesets a letter with a popular word processor. The physical parts of the computer (keyboard, screen, mouse, etc.) are located in the physical world, as well as the secretary herself. Conversely, the letter and the word processor program (as well as the "companion", a small character appearing on the screen and supposed to help the user with her tasks) are located in the virtual world.

However, all these objects belong to the world as a whole. All of them can be sensitively perceived by people.

B. Model and Model Objects

We call *model* the abstract representation of the world (physical world as well as virtual world). The objects of the world are described in the model by a set of characteristics called *attributes*.

Each object of the model describes (at least) one object of the world (fig. 1). Since it is impossible to describe every single detail of the world, the model should then be considered a *partial* representation. Indeed, it is likely that implementors would decide to model only the characteristics of the world relevant to the target applications.

It is not possible to automatically check that the model really represents the world that it is supposed to describe. This is the ambient environment designer's task to check this kind of consistency. No mechanism can automatically detect possible errors.

Example — In the physical world, a printer is located in room D2.18. It is represented in the model by the object `printer-d2.18`. This object has got an attribute called `printed-pages` that represents the total count of pages printed by the real-world printer located in room D2.18. The ambient environment designer must ensure that this attribute is updated according to its semantics. For instance, if one assigned *another printer's* page counter to it, no formal consistency rule would be broken. Thus, only the environment designer can perform consistency checks because they are not feasible automatically.

There are three types of attributes :

- *static attributes*: values are affected to such attributes once and for all for a given object, and they do not change in its lifetime. These attributes can either be defined for a class as a whole or on an individual basis,
- *state variables*: they represent the dynamic state of the world. They are permanently updated,
- *calculated attributes*: a sub-category of state variables, they are dynamically deduced from the values of other attributes. Their value is updated each time that one of the attributes they depend on changes.

Example — The attribute `color-capable` of the printer called `printer-d2.18` is set to `false` for its whole lifetime. Likewise, its `name` attribute is set to `"Printer located in room D2.18"`. A context component regularly queries the physical printer through SNMP[3] requests so as to retrieve the number of printed pages since its toner cartridge was last replaced, and updates the `printed-pages` attribute accordingly. The `remaining-pages` attribute is defined to be equal to `toner-cartridge-capacity` − `printed-pages`. Therefore, it will be updated each time the `printed-pages` attribute is modified.

[3]Simple Network Management Protocol.

C. Incarnations

We call *incarnation* an object in the physical world or in the virtual world that corresponds to an object of the model. In the first case, we call it a *physical incarnation*. In the second case, we call it a *virtual incarnation* (fig. 1).

Example — An incarnation of the `printer-d2.18` object (which is located in the model) is the (physical) printer located in room D2.18. The latter is an object from the physical world.

Up to now, we have implicitly assumed that an object of the model had one and only one incarnation (either physical or virtual). Actually, it is possible for a model object to have *several* incarnations:

1) in *pure reality*, when not interacting with computers, every object has exactly one incarnation, located in the physical world,
2) in *mixed reality*, or simply when interacting with computer tools, some objects of the model can have several incarnations, either physical or virtual,
3) in *pure virtuality*, the objects of the model have one or several virtual incarnations, and no physical incarnation.

IV. A Platform for Ambient Computing

A. Overview

The previous section has shown that objects of the world (either the physical world or a virtual world) can be described by a model. The objects of the world are *incarnations* of the objects that populate the model. Let us now show how this vision can lead to the design of a platform architecture for ambient-computing systems.

In this paper, we only deal with *context capture* : we explain to build and update a model that corresponds to the world. In practice, one would conversely need to *perform actions* on the world. This topic (sometimes called *actuation*) is not tackled here, and may be the subject of future research work. The general layout of this platform is shown on figure 3.

The platform is divided in four layers:

1) **Sensors.** Sensors are the interface between the world (either physical or virtual) and the platform. They permanently track changes in the world in order to update the model accordingly.
2) **Context aggregation.** It is often necessary to combine information from several sensors, so as to deduce relevant and useful context information. That is what we call *context aggregation*.
3) **Assignation.** Information from the preceding layers characterize objects of the model. That is why we *assign* such information to the attributes of the objects of the model.
4) **Object hive.** Model objects are gathered in the platform in a repository that we call an *object hive* (see section V).

On figure 3, *sensors* and *aggregation components* are designated by the generic term *ambient component*. They share

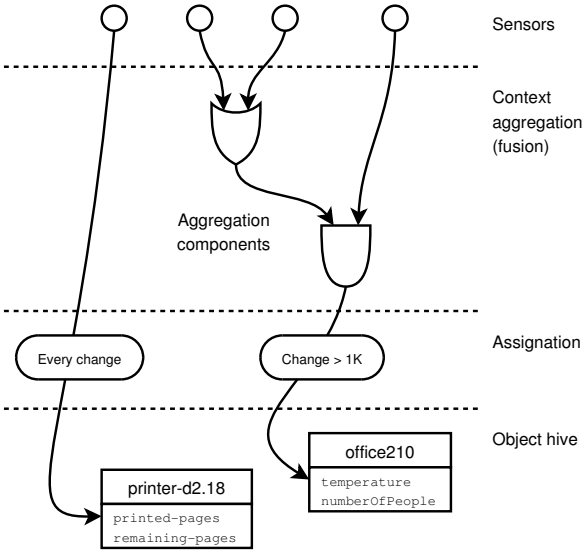

Fig. 3. General layout of the context capture platform for ubiquitous computing applications.

many characteristics with Anind K. Dey's context widgets [1], [11] and Gaetan Rey's *contextors*. The task of these components is to capture information in the world and perform transformations on contextual information.

B. Ambient Components

1) Introduction: an ambient component is a software component that behaves in a relatively autonomous fashion and has got inputs and outputs (fig. 4).

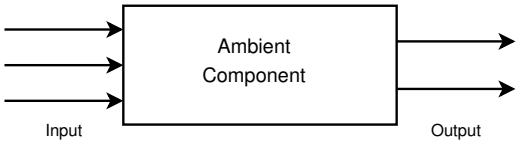

Fig. 4. An ambient component.

For instance, sensors are particular ambient components that have no input and only outputs. They are abstract counterparts of the physical sensors located in the physical world, or information sources located in a virtual world.

More generally, ambient components' outputs are activated or altered in the following two cases:

- when an input changes. In response to this change, the ambient component performs an action, so as to update its internal state as well as its output values. In consequence, the values of some outputs may be modified,
- when an internal *event* happens inside the component, for instance a timeout, or, in the case of a sensor, a change in the world.

Example — The output of a *low-pass filter* component will change every time its input will change. Conversely, we can imagine a *clock* component that will output an event every second. In this case, the component has no input. The cause of output events is totally internal to the component. Likewise, the output of a *thermometer* component embedding a "real" temperature sensor will change depending on the current room temperature. From the platform's point of view, this cause is considered to be *internal to the thermometer component* since it has no input.

Ambient components can be interconnected (inside the aggregation layer or the context capture layer, see fig. 3): the output of one component (called *producer* component) is then connected to the input of another component (called *consumer* component). One given output can be connected to an arbitrary number of distinct inputs. However, one given input can be connected to at most *one* output of another context component.

Indeed, when producing information, it is straightforward to distribute it to an arbitrary number of consumers. Conversely, it is very difficult to *fuse* information from several producers to deduce one unique input. It requires (possibly complex) processing that is *specific* to the information involved. That is why an input can be connected to one producer only. However, it is possible that this producer is in fact a fusion component, able to fuse information from several upstream[4] components, each one being connected to one of its own inputs (fig. 5).

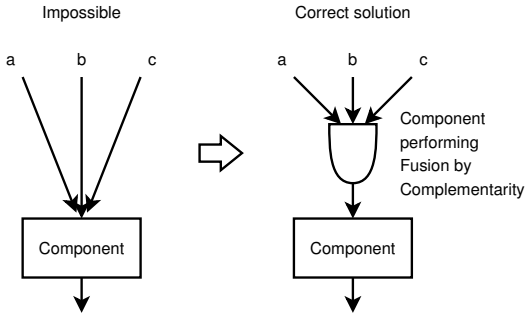

Fig. 5. Since one input can be connected to only one output, we must resort to fusion components.

So as to ensure consistency of data exchanged among components, inputs and outputs are typed. Thus, to be able to connect two ambient components, the type of the upstream component's output must be compatible with the type of the downstream component's input. This rule allows the detection of trivial error cases, but does not allow for the detection of *semantic* mismatches that are more complicated to detect.

However, one can wonder if such a strong type system does not unnecessarily increase the number of components needed to build a system: at first sight, every kind of component would need to exist for every input type / output type pair. This would of course be counter-productive – if not impossible. To avoid such a situation, we introduce two kinds of *polymorphism* (see the next two subsections).

Example — On an input supposed to be fed with the acceleration due to gravity (g, measured in $\mathrm{m} \cdot \mathrm{s}^{-2}$ [meters per second per second]), it is *not* possible to connect the output of a temperature sensor (measured in K [kelvins]). However, it *is* possible to connect by mistake the output of a vehicle's acceleration sensor to it, because this quantity is an acceleration too, measured in $\mathrm{m} \cdot \mathrm{s}^{-2}$.

[4]The terms *upstream* and *downstream* must be understood here in the context of information flow: information is transferred from an upstream component to a downstream component.

In this example, we have informally shown a first category of typing: typing measured quantities with units from the international system of units (SI). Actually, we propose two categories of inputs/outputs: *value* inputs/outputs, and *event* inputs/outputs.

2) First category of input/outputs: value inputs/outputs: *value* inputs/outputs can carry two kinds of values:

- *abstract values* (for instance, user identifiers). They belong to classical computer types: integers, floats, character strings, structures, etc.,
- *physical quantities* (for instance, a temperature, or an acceleration). In this case, we will associate a *unit* (taken from the international system of units) to inputs and outputs (for example, $\mathrm{m} \cdot \mathrm{s}^{-1}$ [meters per second], K [kelvins], etc.)

A value output always has a value. For example, let us consider an abstraction of a physical sensor, for instance a temperature sensor. This sensor permanently measures the current temperature, so it has an output called `temperature`. The value of this output can be read at any moment. A consumer component connected to this output will have several means of retrieving information:

- *probing* the current output value at given moments, for instance on initialization, or at regular intervals,
- *subscribing* to the producer, and being notified when the quantity fulfills a given condition. For instance: "the absolute temperature change since the last notification is higher than $1\ \mathrm{K}$", "the temperature has just risen above $273\ \mathrm{K}$", etc.,
- *subscribing* to the producer and being notified at a given *sampling* frequency. For instance, a given component can ask to be notified two or three times per second.

To use the same components with different input/output types, we introduce *parametric polymorphism* (also known as *genericity* in object-oriented languages), as in ML [12]. Let us see how this works on an example. A maximum detector can have an input of polymorphic type α, and an output of polymorphic type $\alpha-$`maximum`. So, this component can be connected to any value output, which makes it universal, while ensuring strict type-checking. This component can therefore be used to detect temperature maxima as well as maxima of any other physical quantity.

3) Second category of inputs/outputs: event inputs/outputs: *event* inputs/outputs have different semantics. They have no associated value, so one cannot query them at will. Conversely, they punctually send messages to the consumers connected downstream. These messages are called *events*. So, the only means for a consumer to connect to an event output is to *subscribe* to this output. This way, the consumer component tells the producer that it is interested in the events it produces and wishes to receive them until further notice.

Each event type has got a name that is unique throughout the system. For instance, a crossing detector (such as a *light barrier*) sends an event called `crossing-detected` each time someone passes by.

Event types can be classified in a hierarchy, where all events are descendants of a common ancestor, called `generic-event` for instance. This way, an input of one component can be connected to an event output of type T_1 of another component if and only if:

- it is an event input (then, let T_2 be its event type),
- T_1 is a subtype of T_2, ie. an event of type T_1 can be considered as an event of type T_2[5].

This way, we introduce here a kind of type polymorphism (as in object-oriented programming). Indeed, to be able to handle events of different types, a component just has to be able to handle events of a common super-type, i.e. a common ancestor type. The event types can then be classified in a hierarchy (fig. 6).

Example — The event type `crossing-detected` described above can have two subtypes, `fast-crossing-detected` and `slow-crossing-detected` (fig. 6). A component that takes in input `crossing-detected` events will also accept `fast-crossing-detected` and `slow-crossing-detected` events.

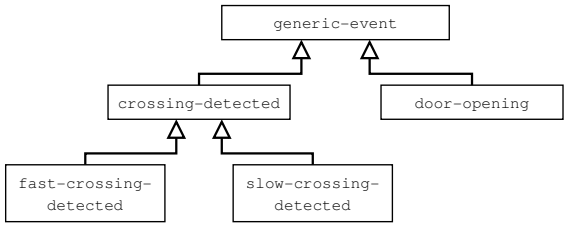

Fig. 6. Event type hierarchy. The arrows mean *"is a sub-type of"*.

Example — We can imagine a generic event counter, that would be able to count all occurrences of every possible type of events. To this end, we only need to create a component that has an output of type `generic-event`. Then, it will be possible to connect it to any kind of event output.

4) Discussion: after studying this example, one can wonder if making a distinction between event and value input/outputs really makes sense. Indeed, why not replace event input/outputs with counters (or even boolean values) that would be incremented (or inverted in the case of boolean values) each time the corresponding event occurs? Event inputs would then simply subscribe to changes of these outputs.

Example — This way, a crossing detector would provide an output value alternatively equal to `true` and `false`. Each time someone would cross the barrier, the boolean value would be inverted. To be notified of crossings, a component would simply need to subscribe to changes of this boolean value.

This solution can seem to be attractive because it suppresses the distinction between value and event input/outputs. However, it suffers from two major drawbacks:

1) this way, the semantics of inputs and outputs are weaker: it is not possible to perform consistency checks between event input and outputs. In particular, one cannot use sub-typing polymorphism that we have yet shown to be useful. Moreover, outputs would not necessarily have a meaning. For instance, in the previous example, the boolean value has no meaning *per se*: only its *changes* are meaningful,

[5]This corresponds to *casting* to a super-class in object-oriented languages.

2) it does not allow events to hold information. However, the STEAM[6] architecture [13] shows that it is quite natural to define *attributes* for events, so that they can be parameterized and carry information.

Example — It can be necessary to measure the durations of the crossings of the light barrier described in previous examples. A *crossing duration* must be associated to each detected crossing. To this end, it is therefore natural to define an event type called `measured-crossing`, that has an attribute called `crossing-duration`, whose type is a physical quantity measured in seconds. This way, crossing events can be *parameterized*.

Resorting to events, and to parameterized events in particular, can offer a clean solution to the issue tackled by this example. Without the notion of event, it would have been possible to find a compromise solution, but this solution would have been semantically obfuscating. In addition, it would not have been possible to perform fine-grained consistency checks on inputs and outputs.

For these reasons, we think that having two kinds of inputs and outputs, namely event and value input/outputs, makes the model elegant and clean, and eases the component specification process very flexible.

C. Context Aggregation

1) Introduction: the role of aggregation components is to transform raw data acquired by sensors into relevant high-level information about the context of use of the system. As described in [7], this transformation can be performed in several steps. Indeed, several layers of aggregation components can transform information step by step.

It is interesting to draw a taxonomy of the different kinds of aggregation components. In [7], Rey et al. propose a taxonomy of contextors, but it is limited to a flat list of six classes. We think it can be valuable to structure such a taxonomy in a hierarchic fashion. To name classes in our taxonomy, we partially use the same vocabulary as used by the CARE[7] properties [14] and in the related ICARE[8] platform [15].

In this section, we consider ambient components from the point of view of only one of their outputs, which amounts to dealing with ambient components with only one output. This assumption can be made without loss of generality, because a component with n outputs can be replaced with n different components with one output each and all the same inputs as the original component (fig. 7).

First, we distinguish between components with one input and components with several inputs. In the former case, we say that they are *conversion* components; in the latter case, we say that they are *fusion* components.

2) Conversion Components: a conversion component has got only one input. From values or events in input, it provides other values or events on its output. That is why we say that it performs a *conversion*.

[6]Scalable Timed Events And Mobility
[7]Complementarity, Assignation, Redundancy, Equivalence
[8]Interaction-CARE

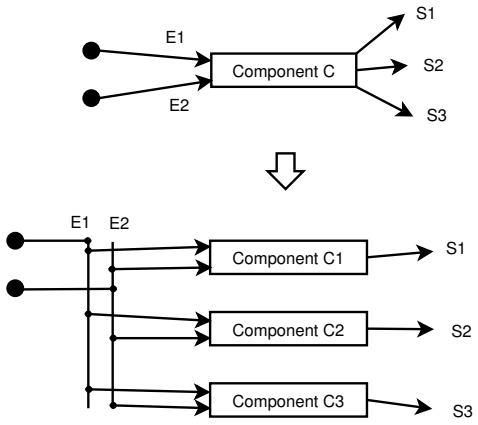

Fig. 7. Equivalence between one component with n outputs and n components with one output each.

Example — Imagine a component that applies a low-pass filter on its input. It gets values as input, i.e. a signal $f(t)$, and provides other values in output. More precisely, its output is a signal equal to $\frac{1}{T}\int_{t-T}^{t} f(u)du$. The input and the output are of the same nature: they are value input/outputs.

Example — Similarly, we can imagine a component that performs a conversion from one event type to another. For instance, one may want to convert from `crossing-detected` events to `person-enters` events if a light barrier is located at the entrance door of a room and therefore detects people coming in.

Example — Imagine a component that detects maxima on its input. It receives values in input but provides events in output: each times it detects a maximum, it sends a `maximum-detected` event. This component's input and output have different natures.

Example — Imagine a component that counts incoming events. Thus, its input is an event input, of type `generic-event`. Each time an event arrives, the component increments a counter, whose *value* is provided in output. Thus, the component's output is a value output. Here too, the component's input and output have different natures.

As we see on the above examples, all combinations of input and output natures are possible. The role of conversion components is *precisely* to perform conversions between all the information types handled by the system.

3) Fusion Components: a fusion component combines several information sources in input, and merges them to produce a unique output. There are two cases, (1) when inputs provide different kinds of information, and (2) when inputs are meant to provide similar information.

(1) — When its inputs provide different kinds of information, an ambient component deduces different new information. In this case, it is called a *complementarity* component, because it combines its inputs in a complementary way to produce its output. The component performs processing very specific both to its inputs and to the expected result. It is therefore very unlikely to be able to design a generic algorithm capable of fusing arbitrary data. It means that there cannot be a generic fusion component, but rather, a multitude of highly

specific fusion components.

Example — Suppose that a system is designed to write the transcripts of meetings. A speech recognition subsystem can provide the raw text of discussions going on, and a video identification subsystem can identify the current speaker. A *specially designed* complementarity component can take in input information from these two subsystems, and provide the complete transcripts of meetings, attributing every statement to the right person.

(2) — When the inputs of one component are meant to provide similar information, we call it an *equivalence* component. For instance, a component can take in input information from three different temperature probes. Several kinds of actions can be performed:

- *redundancy*: in this case, only the inputs that effectively provide values are taken into account, and a "poll" is taken among them. For instance, in the case of three temperature sensors, one can take the *median* value among the three values available. Then, if two sensors out of three provide correct information, the redundancy component provides a correct result, even if the third component provides no or incorrect information. This method is known in the field of system safety as TMR[9]. In this case, it is possible to imagine *generic* redundancy components, that work on arbitrary data and implement generic redundancy techniques.
- *quality enhancement*: redundancy components only compensate for upstream components' failures. One can imagine more complex processing, that would *improve the quality* of incoming similar information. For instance, by combining information provided by three noisy temperature sensors, it must be possible to reduce the noise in output.

4) Summary of Aggregation Operations: in this section, we have seen some classes of context aggregation components. They are summarized on figure 8.

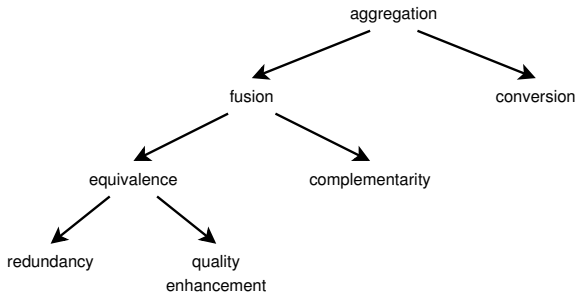

Fig. 8. Classes of context aggregation components.

Among these classes, only fusion can possibly be designed to be generic. As for other aggregation techniques, algorithms are very specific:

- to the inputs and outputs of aggregation components,
- and especially, to the *aggregation technique* that one given component implements.

[9]Triple Modular Redundancy

In our taxonomy, we use notions of *complementarity*, *redundancy*, *equivalence* as in the work of Coutaz et al. [14], but we have not so far introduced a notion of *assignation*, as in their work. We introduce such a concept, but we consider that it is not a fusion operation. Rather, it enables us to establish a correspondence between an output of a context component to an object attribute in the model. This process is discussed in the next session.

V. Object Hive

A. Hive and Assignation

When introducing the platform, we have seen that model objects are stored in an object *hive*. The hive is itself an ambient component, that holds a description of every object of interest. Ambient computing applications can subscribe to the hive so as to be notified about changes in *high level context*. Thus, they receive notifications when model objects' attributes are modified.

Example — an application can subscribe to the `people-count` attribute of the object `office-210` (fig. 3) so as to be kept informed when people enter or leave this office. This is a request on high-level contextual information.

It seems that the *use* of information held in the hive is not problem in itself. However, maintaining the hive in sync with the world is a much more complex problem.

Therefore, an update process needs to be performed permanently. This can be done quite simply, in *connecting* the attributes (*state variables*) of hive objects to the outputs of aggregation components. This connection is called *assignation*: when the designer of an ambient computing system decides that an attribute is meant to receive information from a given component's output, he or she somehow *assigns* these information *to* the attribute.

To some extend, the attributes of hive objects can be considered as consumer components (fig. 3). In consequence, an attribute can only be assigned a *value* output, because an attribute must have a value at any moment. It would have no meaning to connect it to an *event* output because an event happens only at one point in time.

However, we have seen before that attributes have types, exactly in the same way as value outputs have types. As a result, assignation operations must enforce type compatibility, as when connecting one ambient component to one other. An attribute can only be connected to a *value output of the same type*.

As seen before, a consumer connected to a value output must subscribe to this output so as to be kept informed when the value changes and meets a given condition. Thus, attributes too must give a condition when subscribing to information providers. These conditions are given when creating the assignation, and are called the *assignation conditions*. For instance, it is possible to subscribe to every change, or only to changes of a minimum amplitude (see the *assignation* layer on figure 3).

Example — The output of the redundancy component described in section IV, and that combines the outputs of three temperature probes by performing TMR can be assigned to the

`temperature` attribute of the `office-210` object located in the hive.

B. Importance of the Hive

As stated in introduction, the hive does not only provide a means to *name* context attributes within an object model, but it also allows to separate two different processes:

- on the one hand, context capture, and low-level context processing to deduce high-level context. Upstream from the hive, chains of ambient components perform context aggregation at various levels,
- on the other hand, access to the context by applications located downstream.

The hive introduces a weak coupling between context capture and use, which allows a dynamic reconfiguration of systems, without having to recompile, or even stop, applications.

Without the hive, applications would be *directly* linked with component chains. In consequence, when modifying the component infrastructure (for instance, when reconfiguring the system), applications would need to be modified too.

Example — An application running in a lab a few kilometers off Orly airport in France needs to know the exterior temperature. At first, there is no temperature probe available. So people decide to retrieve the data through the network, using METAR[10] information of the Orly airport. Subsequently, a temperature sensor is installed outside of the lab, which will provide more relevant (because more local) information than Orly airport's.

Case 1, without hive: in the beginning, the application had subscribed to a METAR gateway component. After the installation of the exterior probe, the corresponding source code was replaced by code subscribing to the component embedding the probe. *After re-compilation*, the application could be started again, and has provided more precise information ever since.

Case 2, with hive: in the beginning, the output of the METAR gateway component was assigned to the `exterior-temperature` attribute of the object representing the laboratory. After the installation of the exterior probe, the corresponding component was simply assigned to the `exterior-temperature` attribute. The application, located *on the other side* of the hive, was not even stopped. It went on running and using data from the hive *without being bothered by the change, but benefiting from the new sensor's greater accuracy as soon as it was installed*.

This way, the source code of applications holds no direct reference to sensors. All the dependencies can be stated in a *declarative* fashion, and can be modified at runtime, which allows a dynamic reconfiguration of applications.

Moreover, the capture and context aggregation layers can be built independently from the possible applications. Applications can be connected only later, without modifying the context acquisition architecture.

[10]METeorological Aerodrome Routine Weather Report

VI. Implementation

A. Technological Solutions

In this section, we present implementation choices for the conceptual platform introduced in this paper. The main goals of this implementation are the following:

- the system is distributed across a network; the network is transparent for designers of ambient computing components and applications,
- the system is based on simple and open protocols.

From the description of the model, we deduce that an ambient computing system complying with our platform is composed of several *ambient components*. Therefore, our proposal is to distribute these ambient components across the computers of a network. Each computer hosts a server, whose task is to enable communication between its own components and other components, either local ones or remote ones.

To support communication between components, it seems reasonable to exchange fragments of RDF[11] graphs. Indeed, RDF is the new standard for the description of semantic information. In addition, vocabulary description functionalities can be added to RDF thanks to related languages such as OWL[12]. This way, it should be possible to describe vocabularies shared by components in a standard manner. For instance, the data type hierarchy could be modeled using OWL.

RDF is a conceptual model, and it has got several representation formats. However, the representation format called RDF/XML[13] seems to be the easiest to use and the most popular, so we have chosen to use it. In addition, it is well suited for transmission of data over HTTP[14]: this way, server can simply be *web services*. We have chosen to use lightweight web services implemented using the XML-RPC[15] protocol, known to be very simple. However, it would be possible to use more complete (yet more complex) web service standards such as SOAP[16].

B. Components and Component Identification

As we have shown before, servers are meant to host components and enable communication between them. In particular, the object hive can be considered as one of these components. Therefore, it is hosted on one of the servers of the system, exactly in the same way as any other component is hosted on one server.

Besides, it is useful to have a list of all available components at one's disposal, in particular at design time. That is why every server has a particular component called `registry`, capable of providing the list of all components available locally on this server. Registries could even talk with each other so as to build the list of all components available on *the whole network*, and not only locally [16]. Thus, the general architecture of our implementation looks like the example of figure 9.

[11]Resource Description Framework
[12]Web Ontology Language
[13]RDF over eXtensible Markup Language
[14]HyperText Transfer Protocol
[15]XML-Remote Procedure Call
[16]Simple Object Access Protocol

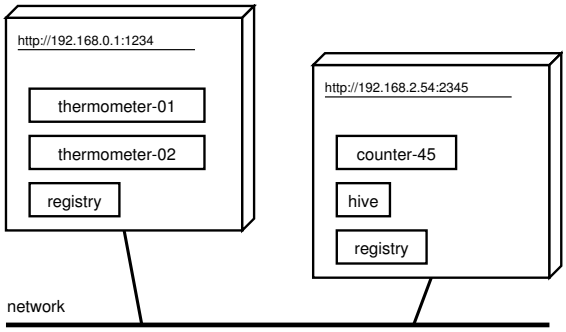

Fig. 9. Example of deployment of an ambient computing system. Every server holds a `registry`. The object hive is located on *one* of the servers (here on `192.168.2.54`).

We associate a URI[17] to every component, composed of its server's physical address (actually, the server's own URI) and of the component's local name within its server. This way, any ambient component can be addressed by its URI, in the same way as every object in CoolTown [4] is identified by its URL.

Example — The `thermometer-02` component, located on the server identified by the URI `http://192.168.0.1:1234`, is itself identified by the following URI: `http://192.168.0.1:1234/thermometer-02`.

Every component has therefore a unique name, its URI. This is necessary when using RDF because all RDF objects (called *resources* in the dedicated vocabulary) *must* be identified by URIs. In our system indeed, URIs are *both logical* identifiers (with respect to RDF) *and physical* identifiers, enabling access to platform components through the layers of a network.

C. Usage Example

Let us consider an example, where two sensors are available: a temperature probe, and a counter that counts the number of printed pages on a printer. The hive contains two objects, representing the current place, and the printer. Assignation can be performed *at runtime* thanks to a graphical editor shown on figure 10. Sensors are on the left; hive objects are on the right.

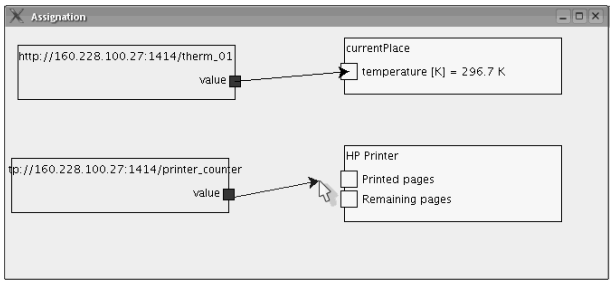

Fig. 10. Graphical assignation at runtime.

On the screen-shot, the designer has already assigned the output of the temperature probe to the `temperature` attribute of the `currentPlace` object in the hive. In consequence, the measured temperature is displayed next to the `temperature` attribute as soon as a value has been received (here: 296.7 Kelvins). At the bottom, the designer is currently making a connection between the output of the printer counter and, presumably, the `printed-pages` of the `HP Printer` object. This operation is performed dynamically at runtime.

It is possible to modify assignations without being obliged to stop running applications: reconfiguration is *graphical*, *declarative* and *dynamic*. Moreover, when creating a connection from an output, the user of the graphical tool (i.e. the application designer) gets a visual feedback only when the mouse cursor hovers a *compatible* input. Indeed, the system performs a type checking and only proposes possible connection, as does the ICON system for interaction customization [17].

VII. Conclusions and Perspectives

In this article, we have first presented a conceptual model of ambient computing systems. We have then proposed a component-based platform for building these systems, that matches the conceptual model.

Our platform is original because:

1) it features a strong typing of messages exchanged between ambient components,
2) it introduces the concept of *object hive* that is a high-level service to access context information, while allowing to dynamically reconfigure applications.

We have implemented a base version of the platform described in this paper using Java. It has already allowed us to try building systems and reconfiguring them dynamically.

Our short term research directions are to handle dynamic context (for instance, allowing objects of interest to change as somebody moves), a better formalization of the model and data types, as well as a more detailed semantic description of components. Later on, we will explore the possibility of handling context history and component synchronization, as it is proposed in some platforms such as the Context Toolkit [6].

[17] Uniform Resource Identifier

References

[1] A. K. Dey and G. D. Abowd, "Providing architectural support for building context-aware applications," Ph.D. dissertation, Georgia Institute of Technology, 2000.

[2] O. Riva, "A conceptual model for Structuring Context-Aware Applications," in *Fourth Berkeley – Helsinki Ph.D. Student Workshop on Telecommunication Software Architectures*, June 2004.

[3] J. Pascoe, "The stick-e note architecture: extending the interface beyond the user," in *IUI '97: Proceedings of the 2nd international conference on Intelligent user interfaces*. ACM Press, 1997, pp. 261–264.

[4] T. Kindberg and J. Barton, "A Web-based nomadic computing system," *Computer Networks (Amsterdam, Netherlands: 1999)*, vol. 35, no. 4, pp. 443–456, 2001.

[5] B. Schilit, "System Architecture for Context-Aware Mobile Computing," Ph.D. dissertation, Columbia University, 1995.

[6] A. K. Dey, D. Salber, and G. D. Abowd, "A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications," *Human-Computer Interaction*, vol. 16, no. 2-4, pp. 97–166, 2001.

[7] G. Rey and J. Coutaz, "Foundations for a theory of contextors," in *Computer-Aided Design of User Interfaces III*. Kluwer Academic Publishing, 2002, pp. 13–32.

[8] G. Biegel and V. Cahill, "A Framework for Developing Mobile, Context-aware Applications," in *Proceedings of PerCom 2004*. IEEE Computer Society, Mar. 2004, pp. 361–365.

[9] L. Nigay, E. Dubois, and J. Troccaz, "Compatibility and continuity in augmented reality systems," in *I3 Spring Days Workshop, Continuity in Future Computing Systems*, Porto, Portugal, Apr. 2001.

[10] P. Milgram, H. Takemura, A. Utsumi, and F. Kishino, "Augmented Reality: A Class of Displays on the Reality-Virtuality Continuum," *SPIE*, vol. 2351, pp. 282–292, 1994.

[11] A. K. Dey, "Understanding and using context," *Personal Ubiquitous Computing*, vol. 5, no. 1, pp. 4–7, 2001.

[12] A. W. Appel and D. B. MacQueen, "Standard ML of New Jersey," in *Proceedings of the 3rd Int'l Symposium on Programming Language Implementation and Logic Programming*, no. 528. Springer, 1991, pp. 1–13. [Online]. Available: citeseer.csail.mit.edu/appel91standard.html

[13] R. Meier and V. Cahill, "Exploiting Proximity in Event-Based Middleware for Collaborative Mobile Applications," in *Proceedings of the 4^{th} IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'03)*. Springer-Verlag, Nov. 2003, pp. 285–296.

[14] J. Coutaz, L. Nigay, D. Salber, A. Blandford, J. May, and R. M. Young, "Four easy pieces for assessing the usability of multimodal interaction: the CARE properties," in *Proceedings of INTERACT'95: Fifth IFIP Conference on Human-Computer Interaction*, 1995, pp. 115–120.

[15] J. Bouchet and L. Nigay, "Icare: a component-based approach for the design and development of multimodal interfaces," in *CHI '04: Extended abstracts of the 2004 conference on Human factors and computing systems*. ACM Press, 2004, pp. 1325–1328.

[16] C. Bettstetter and C. Renner, "A comparison of service discovery protocols and implementation of the service location protocol," in *Proc. EUNICE Open European Summer School*, Twente, Netherlands, Sept. 2000.

[17] P. Dragicevic and J.-D. Fekete, "Input Device Selection and Interaction Configuration with ICON," in *Proceedings of IHM-HCI*, 2001, pp. 543–448.