WildCAT: a generic framework for context-aware applications

Pierre-Charles David
France Télécom, Recherche & Développement
28 chemin du vieux chêne
F-38243 Meylan
PierreCharles.David@francetelecom.com

Thomas Ledoux
OBASCO Group, EMN / INRIA, Lina
École des Mines de Nantes
4 rue Alfred Kastler
F-44307 Nantes CEDEX 3
Thomas.Ledoux@emn.fr

ABSTRACT

We present WildCAT, an extensible Java framework to ease the creation of context-aware applications. WildCAT provides a simple yet powerful dynamic model to represent an application's execution context. The context information can be accessed by application programmers through two complimentary interfaces: synchronous requests (pull mode) and asynchronous notifications (push mode). Internally, WildCAT is designed as a framework supporting different levels of extensions, from the simple configuration of the default generic implementation to completely new implementations tailored to specific needs. A given application can mix different implementations for different aspects of its context while only depending on WildCAT's simple and unified API.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; D.2.12 [**Software Engineering**]: Interoperability

Keywords

self-adaptive applications, context-awareness, framework

1. INTRODUCTION

The design and implementation of self-adaptive [16] and autonomic systems [13] require a strong knowledge of the system's runtime environment and of its evolutions. This environment includes things like hardware and software resources, but it also increasingly includes informations like geographical position, user preferences, capabilities and activity. For example, mobile applications need to know the context in which they are being used in order to adapt their behavior: use the appropriate network protocol depending on the available bandwidth, choose between visual or aural methods when it needs to notify the user, etc. This ability for a software system to know the environment in which it is used is known as *context-awareness* [12, 8].

More and more applications require this capability, in very different fields from grid management software which needs to know the load of the different machines on the grid to optimize the task deployment, to the video player running on a PDA which adapts the sound level to the ambient noise level, or even turns off the sound and switch to subtitles if the user is in a public place (meeting, bus...). Although these applications are very different, when it comes to context-awareness, their needs are similar. They must be able to

- *discover* their execution context dynamically;
- *reason* about it, beyond raw data acquisition;
- *be notified* when certain events of interest to them happen.

Furthermore, most of the code required to observe the context of an application requires to interact with low-level OS code, or even sometime directly with the hardware. Application developers should not have to write this kind of code, especially since it is essentially the same for many applications, or at least application domains.

For these reasons, we have developed the WildCAT system, a lightweight, general and powerful Context-Awareness Toolkit. It provides an easy way for Java application developers to make their software context-aware, by providing a simple API to discover, reason about and be notified of events occurring in their execution context. Because it is a single well defined framework, it also enables the sharing of the low level code to gather information about the context, so that this code can be written only once and easily reused.

WildCAT was created as a part of SAFRAN [6], an extension of the Fractal component model [2] for the creation of self-adaptive applications, but we designed it so that it can be used independently. Although it does not introduce groundbreaking ideas, WildCAT provides a simple yet powerful interface for programmers to make their applications context-aware, without imposing too many restrictions on the actual implementation of sensors. This should enable the integration of many different kinds of context information (for example local hardware probes or sensor networks), each with its specific implementation techniques, while exposing a simple and unified interface to application programmers.

In this paper, we first give a quick overview of the WildCAT system (Sect. 2) and of its data model, before present-

MPAC'05 November 28 - December 2, 2005 Grenoble, France

ing the two main public interfaces (Sect. 3) used by application programmers. Then, we describe the internal design of the framework (Sect. 4), and in particular the default, generic and configurable implementation of the data model (Sect. 4.3) and the different ways in which the framework can be extended (Sect. 4.4). Finally, we present some related work (Sect. 5) before concluding (Sect. 6).

2. OVERVIEW OF THE WILDCAT SYSTEM AND DATA MODEL

WildCAT is a Java toolkit/framework whose goal is to ease the creation of context-aware applications for application programmers. From the client applications point of view, it provides a simple and dynamic data-model to represent the execution context of the application[1], and offers a simple API for the programmers to access this information both synchronously and asynchronously (*pull* and *push* modes). Internally, it is a framework designed to facilitate the acquisition and aggregation of contextual data and to create reusable ontologies to represent aspects of the execution context relevant to many applications.

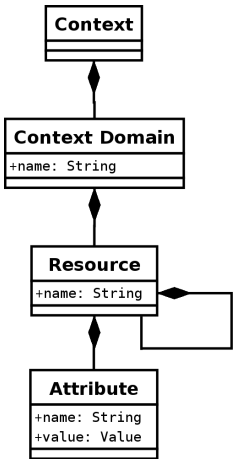

Figure 1: WildCAT's logical data model.

Figure 1 represents the logical model used by WildCAT to represent the execution context. Note that this UML diagram does not imply anything about the actual implementation, except for the `Context` class, which serves as a *facade* [9] for the whole system from the clients' point of view, so as to isolate them from the actual implementation(s). The `Context` class is also a *singleton*; only one instance exists in each application, representing the whole context. It provides the interfaces used by clients (see Sect. 3).

The context is made of several *domains*, each represented by a `ContextDomain` object and identified by a unique name. The purpose of context domains is to separate the different *aspects* of the execution context, and to allow each of these to use a custom implementation, be it for performance reason or for interoperability with existing systems. Examples of context domains include (but are not limited to):

- `sys`: local hardware resources (memory, disks, IO devices...).
- `net`: the topology and performance of the network (maybe using SNMP probes).
- `geo`: geophysical informations (logical and physical location, room temperature...).
- `user`: user preferences, characteristics (disabilities...) and current state (activity, mood...).

Note that WildCAT does not actually implement any of those, but provides a framework in which these heterogeneous informations can be integrated and accessed through a common interface. Because lots of context information is common to many applications, WildCAT is designed so that context domains modelisation (ontologies) and implementation (data acquisition) can be shared and reused like software libraries.

Finally, each context domain is modeled as a tree of named *resources*, each being described by *attributes* (simple *key*, *value* pairs). This simple and generic model was chosen because of its generality, familiarity to programmers and because it does not impose a complex implementation. Note that this is a *logical* model, as seen by clients. An actual implementation of a context domain can use a more complex model internally (for example with sharing of resources, resulting in a DAG), as long as it also provides a tree-like interface.

2.1 Events

Of course, since the context being modeled is highly dynamic, so is WildCAT's data model. Each change occuring in the context model is represented by an event. The possible kinds of events are:

- `RESOURCE_ADDED` and `RESOURCE_REMOVED`, fired when resources appear or disappear in the context, like a new input device, or a network service suddenly not available.
- `ATTRIBUTE_ADDED` and `ATTRIBUTE_REMOVED`, fired when attributes appear or disappear in a resource.
- `ATTRIBUTE_CHANGED`, fired when the value of an attribute changes, like the number of TCP packets dropped by a network interface.
- `EXPRESSION_CHANGED`, fired when the value of a synthetic expression on the context changes (see below), for example the maximum amount of free space on local hard drives.
- `CONDITION_OCCURED`, fired when a boolean synthetic expression becomes true (as a special case of expression change).

2.2 Addressing

WildCAT uses a syntax inspired by URIs to denote elements in the context while being independent on the actual implementation. A *path* (represented by instances of the `Path` class) can denote either a resource, an attribute, or all the sub-resources or sub-attributes of a resource. Note that a path can be syntactically valid but denote a place which

[1]We define *execution context* as "everything external to the application which can have an impact on its execution, as perceived by the end-user". This definition is both very *generic*, in that it is not restricted to a particular list of domains, and *operational*, as it provides an actual criterion to decide if something is part of the context or not.

does not exist. The general syntax is `domain://path/to/resource#attribute`. Here are a few examples:

```
sys://storage/memory              // A resource
sys://storage/disks/hda#removable // An attribute
sys://devices/input/*             // Input devices
sys://devices/input/mouse#*    // Mouse attributes
```

3. EXTERNAL INTERFACE(S)

The `Context` class offers two complementary interfaces to access the actual contextual data of an application, corresponding to the *pull* and *push* modes of interaction.

3.1 Synchronous requests (pull mode)

The first interface allows clients to *discover* the structure of the context and to request the immediate value of an attribute. The methods in `Context` corresponding to this interface are:

```
public class Context {
  String[] getDomains();
  Path[] getChildren(Path res);
  Path[] getAttributes(Path res);
  boolean exists(Path path);
  Object resolve(Path attr);
}
```

The `get*` methods can be used to discover the current structure of the context, as known by WildCAT. `exists()` tests whether the element(s) denoted by its argument path actually exist at this time in the context. Finally, `resolve()` returns the value of the single attribute denoted by its argument. For example,

```
context.getChildren(new Path("sys://storage/disks"));
```

returns the list of all disk drives currently known to WildCAT, while

```
context.resolve(new Path("sys://storage/memory#free"));
```

returns the amount of memory currently available.

3.2 Asynchronous notifications (push mode)

The second interface provided by `Context` follows the publish/subscribe pattern. The client first registers its interest in a kind of event at a particular location in the context, and from this point on, WildCAT will send it asynchronous notifications each time such events occur. The methods in `Context` corresponding to this interface are:

```
public class Context {
  long register(ContextListener listener,
                int eventKinds, Path pp);
  long registerExpression(ContextListener listener,
                          int eventKinds, String expr);
  void unregister(long regId);
}
```

The first method, `register()`, takes a listener object (which will receive the event notifications), a bitmask representing the kinds of events the client is interested in (using constants defined in the `EventKinds` interface), and finally the path on which the events must be detected (this path can include a wildcard as its last element). For exemple, to get notified each time a new input device is plugged *or* unplugged:

```
context.register(myListener,
                 RESOURCE_ADDED | RESOURCE_REMOVED,
                 new Path("sys://devices/input/*"));
```

The next method in `Context`, `registerExpression()`, is used only in combination with the `EXPRESSION_CHANGED` and `CONDITION_OCCURED` event kinds (and the corresponding methods in `ContextListener`). This allows clients to register to more complex events. For example, if a client only wants to be notified when the room temperature goes beyond 30 °C, it can use the following code:

```
context.registerExpression(aListener, CONDITION_OCCURED,
       "geo://location/room#temperature > 30");
```

instead of using a simpler `ATTRIBUTE_CHANGED` event kind, and being notified a many non-interesting events.

An `EXPRESSION_CHANGED` event occurs each time the value of the monitored expression changes. If the expression depends on multiple paths, WildCAT will automatically recompute its value when one of these paths changes. `CONDITION_OCCURED` is a special case of `EXPRESSION_CHANGED` for boolean expressions where only the transation from *false* to *true* are considered.

The expressions usable with `registerExpression()` can use paths, constant numbers and strings, standard comparison, arithmetic and boolean operators, and finally function invocation (`function(arg1, arg2...)`). WildCAT provides some predefined functions working on numbers and strings, but new functions can be added easily to extend the vocabulary of synthetic expression (this feature is probably the best way to extend the power of WildCAT without changing its external interface).

Both `register()` and `registerExpression()` return a numeric identifier (*cookie*) corresponding to this particular subscription. This number should be kept by the client, as it must be provided to `unregister()` in order to disable an event subscription.

4. INTERNAL DESIGN

The UML diagram in figure 2 represents the internal structure of WildCAT. The framework can be decomposed in several different parts. The client interface (in green on the diagram) has already been describe in the previous section. We now describe the other parts in turn.

4.1 The extension interface

The `ContextDomain` interface is the main extension point of WildCAT as a framework. New implementations of the notion of *context domain* can be provided if the default one is not suitable to a particular domain, for example for performance reasons or to create a bridge with an existing system.

```
public interface ContextDomain {
  void initialize(Context ctx);
  String getName();
  boolean exists(Path path);
  Object resolve(Path attr);
  Path[] getAttributes(Path res);
  Path[] getChildren(Path res);
  void register(ContextListener listener,
                int eventKinds, Path path);
  void unregister(ContextListener listener,
                  int eventKinds, Path path);
  void update(Path path, Path cause);
}
```

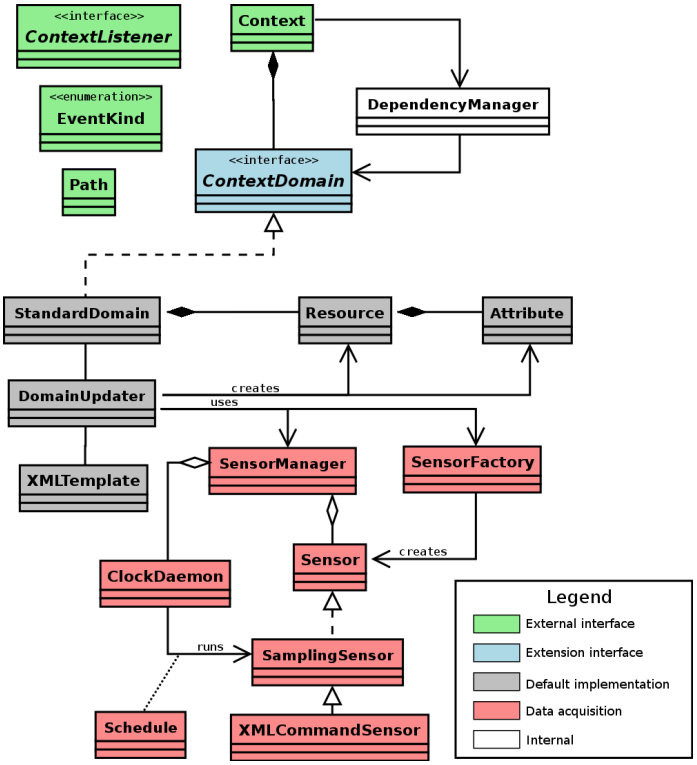

Figure 2: Class diagram of WildCAT internals.

This interface is essentially similar to `Context`, but with a scope limited to one domain, and no constraints on the actual implementation technique. However, new implementations do not have to start from scratch, as WildCAT offers some helper frameworks. Also, WildCAT allow parts of different domains to depend on each other without relying on actual implementation details thanks to the `DependencyManager` class.

4.2 Data acquisition framework

This sub-framework (in red on figure 2) is centered around the notion of *sensor*, which represent the Java objects responsible for the acquisition of raw data. The actual methods used to get the data can be very varied, requiring communication with OS layers or even directly with hardware; WildCAT does not offer any direct help here. However, WildCAT provides the `SensorManager` class to organize all the sensors. Each sensor is identified by a name, and can be either *started* or *stopped*. When it is started, a sensor will send new samples asynchronously to the sensor manager, which gathers all the samples it receives and make them available to its clients. The default context domain implementation, which relies on this data acquisition framework, uses one sensor manager per domain, but nothing prevents other domain implementation to use several ones, or to use sensors directly.

WildCAT distinguishes two kinds of sensors. *Active sensors* implement directly the `Sensor` interface and run in their own thread. They are responsible to send new samples to the sensor manager when appropriate.

```
public interface Sensor {
  String getName();
  void setListener(SamplesListener listener);
  void start();
  void stop();
  boolean isStarted();
}
```

Passive sensors are created by associating a `Sampler` and a `Schedule` to create a `SamplingSensor`. The sensor manager then uses a daemon to invoke the sampler regularly according to its scheduling policy. Samplers are very easy to implement, requiring only one method:

```
public interface Sampler { SampleSet sample(); }
```

When invoked, this `sample()` method should return the raw data corresponding to the current state of the part of the context it is observing.

In order to make it easier to integrate WildCAT with existing systems, the data acquisition framework provides the `XMLCommandSensor` class. This class implements the `Sampler` interface by invoking an external program. The output of the command must be a valid XML document representing a `SampleSet` object, which is then parsed an returned by the `XMLCommandSensor`. As an example, here is a simple Ruby program which can be used by `XMLCommandSensor`:

```
require 'wildcat'
class KernelVersionSensor < Sampler
  def sample
    uname = IO.read('/proc/version').chomp
    version = uname.scanf('Linux version %d.%d.%d');
    return { 'major-version' => version[0],
             'minor-version' => version[1],
             'patch-level' => version[2],
             'version' => version.join('.') }
  end
end
print KernelVersionSensor.new.sense
```

The output of this program looks like this:

```
<?xml version="1.0"?>
<sample-set timestamp="2005-07-19T09:15:36">
  <sample name="patch-level" type="integer">10</sample>
  <sample name="minor-version" type="integer">6</sample>
  <sample name="version" type="string">2.6.10</sample>
  <sample name="major-version" type="integer">2</sample>
</sample-set>
```

Because of the overhead of external program invocations, such sensors are not appropriate for observing very dynamic parts of the context. However, for essentially static data read once on system initialization (like in the example above), it is often the most convenient implementation.

Finally, the `SensorFactory` class helps creating new sensors directly from XML specifications in the format used by the default context domain implementation (see below).

4.3 The default context domain implementation

WildCAT provides a default implementation of `ContextDomain` in the `StandardDomain` class (in grey in figure 2). This implementation follows directly the logical model, with classes for `Resource`s and `Attribute`s. The creation of the initial structure of the domain and its updating is managed by the `DomainUpdater`, which creates and removes resources and attributes according to the specification provided by an `XMLTemplate` (see below). Finally, the raw, unstructured data acquisition is delegated to the previously describe data acquisition framework.

This implementation of `ContextDomain` is generic and can be configured through an XML file. The structure of this file can be seen as a template for the structure of the context domain being modeled. Here is a commented example of such a file.

```
<?xml version='1.0' encoding='ISO-8859-15'?>
<context-domain name="sys">
  <resource name="load">
    <sensor name="load" class="LoadSensor">
      <schedule><periodic period="5000"/></schedule>
    </sensor>
  </resource>
```

The main XML element is `context-domain` and indicates the name of the domain being defined. Inside this element we find a `resource` element, also with a name. The resource is associated with a sensor defined using the name of the Java class implementing it, and an embedded scheduling specification. In this case, the `load` passive sensor will be sampled every 5 seconds by WildCAT. The samples returned by the sensor are automatically mapped as dynamic attributes of the `load` resource.

```
<resource name="storage">
  <resource name="disks">
    <sensor name="hda" class="HardDriveSensor">
      <schedule><on-create/></schedule>
      <configuration>
        <device>/dev/hda</device>
        <mode>static</mode>
      </configuration>
    </sensor>
  </resource>
</resource>
```

The next part of the file defines abstract resources named `storage` and `disks` to serve as categories and organize other sub-resources. The specification of resource `hda`, representing the first hard drive in the host system, shows how to pass configuration information to sensor classes. Such a mechanism is required because a given context can have many resources of the same kind (for example hard drives), and each must be observed by a particular instance of the sensor class. The `configuration` XML element, if present inside a sensor specification, is passed to the constructor of the sensor class (here `HardDriveSensor`) to configure this particular instance. The actual value passed to the constructor is a parsed XML element (using the standard W3C DOM API), and the only constraint is that it is a well-formed XML fragment. Every configurable sensor class can define its own "configuration file" format and is responsible for its interpretation. In the example, the configuration fragment tells the `HardDriveSensor` object to observe the device named `/dev/hda` and to consider it a static device (i.e. non-removable).

```
<resource name="network">
  <attribute name="nb_interfaces">count(*)</attributes>
  <resource name="eth0">
    <sensor name="nic" class="NICSensor">
      <schedule><periodic period="1000"/></schedule>
      <configuration>
        <device>eth0</device>
      </configuration>
    </sensor>
    <attribute name="error_rate">
      #dropped_packets / #received_packets
    </attribute>
  </resource>
</resource>
</context-domain>
```

The last part of the file deals with the network card. It uses `attribute` elements to define *synthetic attributes* (as opposed to the primitive ones corresponding to raw data coming from sensors). The value of these attributes is defined by a expression (using the same simple language as the one used with the `registerExpression()` method) which can reference any part of the context, including elements of other domains. In the example, the expression uses relative paths expressions denoting local attributes. WildCAT tracks the dependencies between these synthetic attributes (using the `DependencyManager` internal class in the architecture diagram) and updates their values automatically. From the point of view of the client programs, these attributes are not different from the others.

The XML template language also supports the definition of dynamic context domains with conditional parameterized branches (enclosed in `<if>` tags) and a more general looping construct (using a `<foreach>` tag), although these features can not be described here for lack of space.

4.4 Framework extension scenarios

Although it provides a default implementation of the notion of context domain, WildCAT has been designed as a framework and supports several extension scenarios:

1. The most simple and directly supported scenario is simply to implement libraries of sensors using the data acquisition framework and then modeling context domains using the default implementation and its template language. Once designed, some of these ontologies (for hardware or software resources for example) could be reused by many applications.

2. The next possibility is to reuse the data acquisition framework and part of the default implementation (the `Resource` and `Attribute` classes), but to discard the `DomainUpdater` and `XMLTemplate` classes to provide custom logic, or even alternate templating languages.

3. The acquisition framework can also be reused by itself with an alternate implementation of the data model, to leverage the particular structure of some domains using an optimized implementation.

4. Finally, it is also possible to implement completely "from scratch" the `ContextDomain` abstraction. Although this is the most difficult scenario to implement, it might be the only alternative, be it for performance reasons or to integrate an already existing system by providing a simple bridge. For example, if an application needs to react to changes in the file system, a `file` context domain could be implemented; however, using the default implementation would be a very bad idea in this case, as it would create thousands of objects to represent all the files and folders. A better approach is to reuse existing systems like Linux's inotify service[2], wrapped using WildCAT's uniform and simple API.

5. RELATED WORK

Several research activities in the ubiquitous computing area have investigated applications for context-sensitive environments and their supporting systems, by focusing in particular on the management of context resources.

The Context Toolkit project [7] was one of the first project allowing to access to context information while hiding the details of context sensing. The Context Toolkit facilitates programming by wrapping sensors with a well-defined widget interface and predefined aggregators may combine several widget outputs to produce context information. WildCAT's data acquisition sub-framework (Sect. 4.2) also hides the data acquisition and we can compare WildCAT synthetic attributes to CT's aggregators.

Like WildCAT, some projects [5, 3, 4] build their models around the concept of *event*, i.e. the generation of a value by a data source can be viewed as an event. The iQL model [5] defines passive and active *data sources*, which are similar to active sensors and samplers in WildCAT. However, WildCAT unifies the treatment of these two kinds of sensors by converting samplers into active sensors using an associated scheduling (polling) policy. The main entities in iQL are *composers*, which are similar WildCAT synthetic attributes in that they compute higher-level or aggregated information from low-level data acquired from data sources. In both systems, synthetic attributes / composers are dynamically and automatically updated when their dependencies change, but iQL provides a more sophisticated language to define composers than the simple expressions supported by WildCAT, although in both systems it is possible to extend the set of primitive operations – and hence the power of the language – using "native" code (for example Java classes). Another difference between the two systems is that iQL provides a language to compose patterns of events into higher-level compound events. This feature is not supported directly by WildCAT, as it is provided by the upper-layer SAFRAN system [6].

The Solar system [3, 4] aims at the same goal as WildCAT and proposes also an event-driven structure with publishers/subscribers where context changes are represented as events. Solar proposes a more sophisticated model that allows context-aware applications to select distributed data sources and compose them with customized data-fusion operators and provides an infrastructure supporting scalability, mobility and reliability.

From a more abstract point of view, the ubiquitous computing community has produced very sophisticated metamodels to represent contextual information like CML [11], taking into account the special characteristics of this kind of data (uncertainty, possible ambiguity when multiple sources are available, etc.). We can also cite work in the knowledge engineering community and the Semantic Web effort on languages to model complex ontologies [10, 15, 14]. Compared to some of these models, the data model used by WildCAT is over-simplified. However, this was a concious decision on our part because we wanted to allow simple and most importantly *efficient* implementations, without too much overhead. Some of the properties of more sophisticated model can be simulated by WildCAT (for example, by using multiple attributes to describe the different facets of sampled values), although it might be necessary in the future to support some of these more directly.

For an in-depth survey of context-aware systems, see [1].

6. CONCLUSION

In this paper we presented WildCAT, an extensible Java framework to ease the creation of context-aware applications. WildCAT provides a simple yet powerful dynamic model to represent an application's execution context. The context information can be accessed by application programmers through two complimentary interfaces: synchronous requests (pull mode) and asynchronous notifications (push mode). Internally, WildCAT is designed to support different kinds of extensions, from the simple configuration of the default generic implementation to completely new implementations tailored to specific needs. A given application can mix different implementations for different aspects of its context while only depending on WildCAT simple and unified API.

The main features of WildCAT are: *(i)* its simplicity from the end-user (i.e. application programmer) point of view, with a familiar hierarchical data model and a small and easy to use API, and *(ii)* its extensibility due to its framework approach which supports different levels of customization (see Sect. 4.4). These characteristics reflect the origin of WildCAT as a part of the SAFRAN system [6] for self-adaptive (autonomic) components. We developed WildCAT as a pragmatic solution to enable SAFRAN components to reason about their execution context and react to its changes. In this context, our main objectives where to impose as little execution overhead as possible (hence the simple data model) and to be as generic as possible in terms of application domains (hence the framework approach).

Future works include:

- support for distribution, either at the level of the data acquisition framework, with probes gathering their raw data from remote machines, or more directly at the context domain level, with the possibility for WildCAT "servers" to export shared context domains to multiple

[2] `http://www.kernel.org/pub/linux/kernel/people/rml/inotify/`

clients;

- re-engineering of the internals to leverage the Fractal component model [2], which should make it easier to monitor and reconfigure WildCAT servers;
- support for storage, retrieval and querying of monitored values and events so as to enable historical and statistical reasoning on the evolution of the context.

WildCAT is available on request at `PierreCharles.David@rd.francetelecom.com`.

7. REFERENCES

[1] M. Baldauf and S. Dustdar. A survey on context-aware systems. Technical Report TUV-1841-2004-24, Technical University of Vienna, 2004.

[2] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani. An open component model and its support in java. In I. Crnkovic, J. A. Stafford, H. W. Schmidt, and K. Wallnau, editors, *Proceedings of the 7th International Symposium on Component-Based Software Engineering (CBSE 2004)*, volume 3054 of *Lecture Notes in Computer Science*, pages 7–22, Edinburgh, Scotland, may 2004. Springer-Verlag.

[3] G. Chen and D. Kotz. Context aggregation and dissemination in ubiquitous computing systems. In *WMCSA'02: Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, page 105, Washington, DC, USA, 2002. IEEE Computer Society.

[4] G. Chen, M. Li, and D. Kotz. Design and implementation of a largescale context fusion network. In *Proceedings of the First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous)*, 2004.

[5] N. H. Cohen, H. Lei, P. Castro, J. S. D. II, and A. Purakayastha. Composing pervasive data using iQL. In *WMCSA'02: Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, page 94, Washington, DC, USA, 2002. IEEE Computer Society.

[6] P.-C. David. *Dveloppement de composants Fractal adaptatifs : un langage ddi l'aspect d'adaptation*. Phd thesis, Universit de Nantes / cole des Mines de Nantes, July 2005.

[7] A. K. Dey and G. D. Abowd. The context toolkit: Aiding the development of context-aware applications. In *Workshop on Software Engineering for Wearable and Pervasive Computing*, Limerick, Ireland, June 2000.

[8] A. K. Dey and G. D. Abowd. Towards a better understanding of context and context-awareness. In *Workshop on The What, Who, Where, When, and How of Context-Awareness, as part of the 2000 Conference on Human Factors in Computing Systems (CHI 2000)*, The Hague, The Netherlands, Apr. 2000. Also GVU Technical Report GIT-GVU-99-22.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Professional Computing Series. Addison-Wesley, Oct. 1994.

[10] M. Genesereth and R. Fikes. Knowledge interchange format version 3.0. Tech. Rep. Logic-92-1, Computer Science Department, Stanford University, 1992.

[11] K. Henricksen. *A framework for context-aware pervasive computing applications*. Phd thesis, School of Information Technology and Electrical Engineering, The University of Queensland, 2003.

[12] K. Henricksen, J. Indulska, and A. Rakotonirainy. Modeling context information in pervasive computing systems. In F. Mattern and M. Naghshineh, editors, *First International Conference on Pervasive Computing (Pervasive 2002)*, volume 2414 of *Lecture Notes in Computer Science*, pages 167–180, Zrich, Switzerland, Aug. 2002. Springer-Verlag.

[13] J. Kephart. A vision of autonomic computing. In R. P. Gabriel, editor, *Onward! proceedings from an OOPSLA 2002 track*, pages 13–36, Seattle, WA, USA, Nov. 2002. ACM.

[14] O. Lassila and R. R. Swick. Resource description framework (RDF) model and syntax specification. W3C Recommandation, Feb. 1999. `http://www.w3.org/RDF/`.

[15] D. Mcguinness, R. Fikes, J. Hendler, and L. Stein. Daml+oil: an ontology language for the semantic web. *IEEE Intelligent Systems*, 17(5):72–80, Sept. 2002.

[16] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. Cheng. Composing adaptive software. *IEEE Computer*, 37(7):56–64, July 2004.