COPAL-ML: A Macro Language for Rapid Development of Context-Aware Applications in Wireless Sensor Networks

Sanjin Sehic
Distributed Systems Group
Information Systems Institute
Vienna University of Technology
Argentinierstrasse 8/184-1
A-1040 Vienna, Austria
ssehic@infosys.tuwien.ac.at

Fei Li
Distributed Systems Group
Information Systems Institute
Vienna University of Technology
Argentinierstrasse 8/184-1
A-1040 Vienna, Austria
fei@infosys.tuwien.ac.at

Schahram Dustdar
Distributed Systems Group
Information Systems Institute
Vienna University of Technology
Argentinierstrasse 8/184-1
A-1040 Vienna, Austria
dustdar@infosys.tuwien.ac.at

ABSTRACT

Application development on wireless sensor networks is becoming more and more challenging due to increasing complexity of applications and lack of dedicated programming models. Developers should concentrate on the application logic, while network designers should ensure the network and sensor performance. However, in reality, these two roles often overlap because the architectural and programming abstraction between the network and application is missing. Research on middleware and language that bridges these two abstraction levels is still in a preliminary stage.

This paper proposes a macro language based on our previous work COPAL (COntext Provisioning for ALl). COPAL is a runtime context provisioning middleware that, via a loosely-coupled and composable architecture, ensures context information from wireless sensor networks and other sources can be processed for the needs of context-aware applications. COPAL-ML is a macro language that extends Java programming language and is tailored for the application development using COPAL. Its main task is to reduce development efforts, hide the inherent complexity of COPAL API, and separate concerns of the context-aware application from underlining wireless sensor network.

Categories and Subject Descriptors

D.3.2 [**Language classification**]: Specialized application languages

General Terms

Languages

Keywords

Sensor networks, context-awareness, macro language

1. INTRODUCTION

Wireless sensor networks (WSNs) provide a very rich set of information, but current approaches to access and process this information in applications are tedious and require significant implementation efforts. The challenges of development process in wireless sensor networks can be summarized as a *code-and-fix* process without any consideration for the maintenance and reuse, and a "good" programming abstraction for wireless sensor networks is currently missing[7].

In context-aware systems, context information is generated by heterogeneous and lower-level devices, which are unaware of the application requirements and information models. Wireless sensor networks enable dense sensing of the environment and provide multitude of information that can be used to observe the physical world. Additionally, they can provide contextual information about their own state: like power consumption, battery status, quality of sensed data, etc. Therefore, context-aware systems can provide good programming abstraction to build and manage applications in wireless sensor networks. The goal is to provide sufficient information for services to make decisions and adapt themselves to changing environment.

Context provisioning refers to the approach of gathering, transferring and processing context in order to raise context-awareness in applications[5]. We distinguish three general types of components in a context provisioning system: *publishers*, *processors*, and *listeners*. *Publishers* are components that sense the environment and report their findings and *listeners* react based on this stimuli. *Processors* are situated between the publishers and listeners and they process the stimuli and can infer information about the environment.

In this paper, we will present COPAL Macro Language (COPAL-ML) for rapid development of context-aware applications. COPAL (COntext Provisioning for ALl)[1] is a runtime middleware for the context provisioning and is part of the smart homes middleware developed in SM4ALL project[2]. COPAL-ML extends Java programming language with few additional keywords. Keywords significantly decrease the code size needed to implement context provisioning components in COPAL and completely hide the dependency on COPAL API. Most importantly, components in the system act autonomously. Thus, change of one of the components

[1]`http://www.infosys.tuwien.ac.at/m2projects/sm4all/copal/`
[2]`http://www.sm4all-project.eu`

does not require change of others, which ensuring easy maintenance and high code reuse.

The paper is organized in the following way: Section 2 provides a short introduction to COPAL and its components. Section 3 describes a scenario to better understand the logical relationship between the COPAL components. In Section 4, we present the macro language to develop the COPAL components. The related work is surveyed and compared in Section 5. Finally, Section 6 concludes the paper with future work.

2. COPAL

COPAL is situated between communication layer of a wireless sensor network and a context-aware application that is interested in sensory data from the wireless sensor network. Its main requirement is to hide complexity of accessing sensory devices from the application and provide a simple interface for applications to retrieve information about the environment. Additionally, it provides a way for applications to define a processing step. It splits the implementation task into three steps that are independent from each other: publishing, processing, and listening the environmental information.

The wireless sensor network can consist of many heterogeneous sensory devices. Each sensory device provides its sensory data in its specific format. COPAL enforces developers to create a common data model for each type of sensory data. Each type is defined with a unique **ContextType**. In it, we specify which information is contained in each published sensory data. It is the first task for developers to define a common data model that is meaningful for their context-aware application. For example, sensory data for the current temperature can contain information about when a measurement was made, where it was made, how many degrees and in which unit the measurement is. Whenever a change of this information occurs or after some period of time, if change is continuously happening, we publish new sensory data with updated information called **ContextEvent**.

To alleviate the discrepancy between data model provided by a sensory device and data model used by COPAL, a component that bridges these two environments is required. This component is called **Publisher**. Its task is to access a sensory device and retrieve information from it, and to translate information received from the sensory device into an event understood by COPAL.

The second step in publishing an event in COPAL is processing. It can consist of zero or more substeps before the event is consumed by the context-aware application. The task of processing events is done by **Processors** which are used:

- to add, modify or remove attributes in events,
- to filter out malformed or low quality events,
- to translate events from one format to another (e.g. calculate temperature in Fahrenheit from temperature in Celsius),
- and to aggregate events (e.g. aggregating power consumption of each appliance in a house).

Each processor specifies which **Actions** it can perform on an input event and what the result of this action is. The result can be the unchanged input event, a new event or nothing in which case the input event is discarded. In COPAL, an event holds all actions that need to be executed on it. COPAL will dynamically find processors that can perform these actions and send the event to them for processing. Finally, the event is considered *fully processed* and ready to be consumed by the context-aware application when all specified actions have been executed.

Processing is the key concept by which a wide range of operations can be carried out and it provides a solution to build adaptive and customizable processes using common patterns inspired by the work on complex event processing[6]. Processing patterns define the abstract relationships between input and output events of a processor in COPAL. They can help with designing and composing processors to construct complex context provisioning schemes. Five patterns are summarized in Li et al.[5], namely Filter, Aggregation, Differentiation, Enrichment and Peeling.

After processing, events reach the context-aware application that reacts to this change. Context-aware applications register themselves to receive particular events. First, they specify which event types they are interested in. For example, an application that turns on air-conditioning depending on user preferences is interested in different information from an application that turns on light whenever a person enters a room. Additionally, context-aware applications may also specify criteria that are evaluated on received events. This allows further separation of events that are interesting from ones that are not. For example, an application that turns on air-conditioning when it is too cold can specify its interest in temperature events that are less than $10\,^{\circ}\mathrm{C}$. Classes that receive fully processed events are called **Listeners**. In COPAL, each listener is registered to one or more **Queries**. In a query, we specify which events should be caught and optional criteria that is evaluated on them. All registered listeners are asynchronously invoked by the query whenever an event of specified type passes defined criteria.

The relationship between components in COPAL is illustrated in Figure 1.

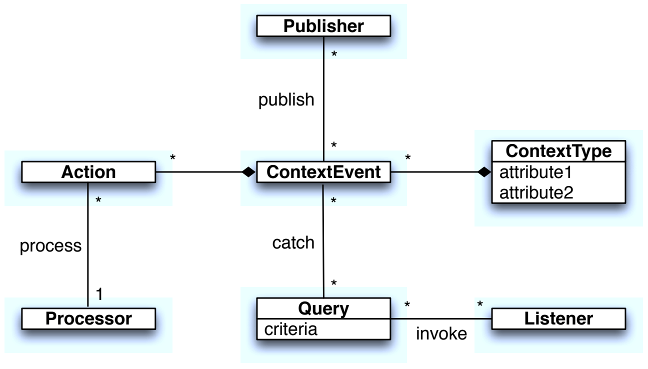

Figure 1: COPAL components

3. SCENARIO

This scenario (Figure 2) is meant to illustrate logical relationship between components in COPAL. In the scenario we examine different sensory information and define a processing phase that infers additional information about the environment.

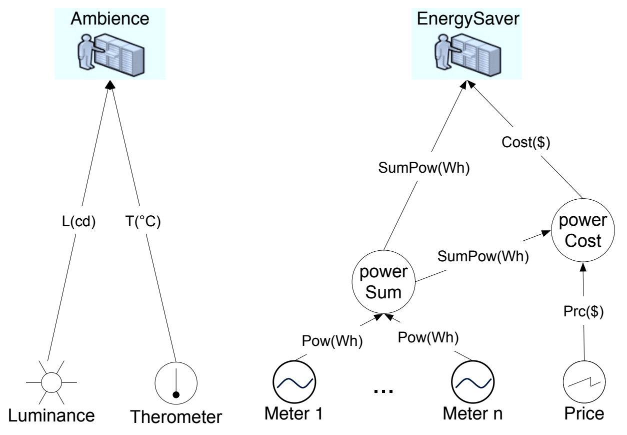

Figure 2: Scenario

Frida lives in a smart-home equipped with a set of sensors, control devices and a context-aware service platform. The basic requirement of context-aware services is to keep a comfortable ambience for Frida. This requirement needs environmental luminance and temperature to decide suitable lighting and settings for air-conditioner respectively.

Furthermore, Frida is a part of an experimental deployment of smart meters that implements a brand new pattern of power service and encourages residential energy saving. Smart meters provide Frida with the current power consumption and are deployed together with a price indicator that receives real-time price information from the power market. Frida has a context-aware service platform and she wants to reduce her power consumption while keeping her home environment comfortable.

From the perspective of COPAL, we have four ContextTypes: temperature, luminance, power consumption and price information from power market, and four Publishers of this information: thermostat, luminance sensor, smart meters and market price indicator. The processing part can consist of two Processors: overall power aggregator and cost calculator. The overall power aggregator receives the current power consumption of all appliances and returns the current overall power consumption in the house. The cost calculator receives the current overall power consumption and price information and returns the current power cost. Finally, we have two Listeners in the system: a service that keeps comfortable ambience in the house and an energy saving service. The ambience service is interested in the current temperature and luminance. The energy saving service is interested in the current overall power consumption and its cost.

4. COPAL MACRO LANGUAGE

This section will describe COPAL Macro Language by implementing parts of the preceding scenario.

In general, a macro language extends a host language by defining additional patterns that expand into constructs of the host language. It is mostly used to define templates for parts of code that follow same pattern and are tedious and error-prone to write. The main advantage of macro languages is that it minimizes code size and reduces possibility to make mistakes.

COPAL-ML works by providing additional macro keywords in Java programming language to define ContextTypes, Publishers, Processors and Listeners that expand into standard Java classes. Developers benefit from using macro keywords based on familiar language instead of learning a new programming language. Furthermore, main advantages of COPAL-ML for developers, comparing it to COPAL API, are:

- that it hides the complexity of using COPAL from developers and allows them to quickly start with the implementation of a context-aware application,
- and reduces size of code needed to implement components in the application.

Code examples in following sections are highlighted as follows: standard Java keywords will be in bold font and COPAL-ML keywords will be in bold font and *underlined* to emphasize the additional non-standard keywords. Figure 3 shows the relationship between entities defined in COPAL-ML and COPAL components.

4.1 ContextTypes

The first step in developing a context-aware application is to define ContextType that are retrieved from a wireless sensor network. Each ContextType consists of one or more attributes that carry information. For example, ContextType that contains just basic information about the current temperature in a room can be defined as follows:

```
public event Temperature {
  attribute String room;
  attribute int degrees; // Celsius
}
```

In this code example, we defined a ContextType called *Temperature* that contains two attributes: *room* and *degrees*. The *room* attribute is of type String and the *degrees* attribute of type integer.

Event defined with COPAL-ML is compiled into a Java class with the same name and a XML Schema file, because each ContextEvent in COPAL must be valid XML document. This transformation between Java instance and the XML document is done automatically in the compiled code. The compiled Java class extends ContextEvent and defines its own ContextType.

4.2 Publishers

When we define publishers, we specify which ContextTypes is this publisher allowed to publish. Afterwards, we define methods in which we can publish ContextEvents using the *publish* method. Methods in publishers do not have to publish any event or they can publish one or more events in each invocation. They are defined as standard Java methods.

```
public publisher Thermostat of Temperature {
  public void publish(Device device) {
    String source = device.getName();
    Temperature t = new Temperature(source);
    // set Temperature attributes using
    // Device proxy object
    publish(t);
  }
}
```

This example defines a publisher called *Thermostat* that is allowed to publish ContextEvents of type *Temperature*. In it, we define an additional method called *publish* that

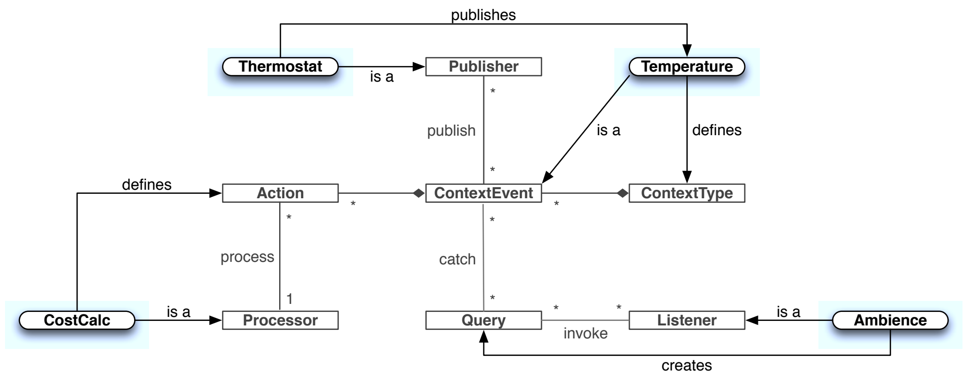

Figure 3: Relationship between entities in COPAL-ML and COPAL components

requires a *Device* argument from which it generates a new *Temperature* instance. We can use an instance of this publisher to periodically publish information about the current temperature by invoking the *publish* method.

Publisher defined in COPAL-ML is compiled into a Java class with the same name that extends BasePublisher defined in COPAL. BasePublisher defines the *publish* method that provides publishing of ContextEvents and implements ContextPublisher interface required by COPAL for all publishers.

4.3 Listeners

In listeners, we define methods that have a ContextEvent as an argument. A method in listener will be invoked whenever an event of same ContextType as the argument is published and fully processed. An additional criteria can be defined with *when* modifier that is appended to the method's declaration. The *and*, *or* and *not* logical operations, beside equality, inequality and comparison operations, can be used to define the criteria. By omitting the criteria, all events of specified type are caught.

```
public listener Ambience {
  private ApplicationController controller;
  private AirConditioning ac;

  public void updateGUI(Temperature t) {
    controller.updateGUI(t);
  }
  public void turnOnHeating(Temperature t)
      when degrees < 10 {
    ac.turnOn(t.getRoom(), 23 /* degrees*/);
  }
}
```

In this code example, we defined a listener called *Ambience* that contains two listening methods: *updateGUI* and *turnOnHeating*. The *updateGUI* method is invoked whenever a *Temperature* event is published because it has no additional *when* modifier. The *turnOnHeating* method is invoked only when published *Temperature* event is less than 10 °C.

Listener defined in COPAL-ML is compiled into a Java class with the same name that implements ContextListener interface required by COPAL for all listeners. Additionally, the compiled Java class creates necessary Queries and registers itself with them, so it can receive ContextEvents that are dispatched to listening methods.

4.4 Processors

Processors define methods that also require a ContextEvent as an argument. Whenever an event of same ContextType as the argument is published, the method is invoked. The result of processing method can be void, an event of same type or an event of different type. Each of these results cause different behaviour with respect to how COPAL will further process the result.

- If result of the method is void, COPAL will just send the input event for further processing.
- If the result of the method is of the same type as the input event, then only the resulting event will be published. Consequently, by returning *null* from the method, the input event is discarded. This allows implementation of methods that can filter out their input events.
- If the result of the method is of different type from the input event, then both the input and the resulting event are published.

If there are multiple processors with methods that have arguments of the same type, then all these methods will be invoked in an undefined order whenever an event of that type is published. This provides us with solution to divide independent tasks among multiple processors that will be executed during runtime.

```
public processor CostCalculator {
  Price price;
  SumPower power;

  public Cost updateCost(Price price) {
    Cost c = getCost(this.price, price, power);
    this.price = price;
    return c;
  }
  public Cost updateCost(SumPower power) {
    Cost c = getCost(this.power, power, price);
    this.power = power;
    return result;
  }
}
```

This example defines a processor called *CostCalculator* that has one overloaded processing method called *updateCost*. This method is invoked whenever an event of type *Price* or *Power* is published. The result of the method is an event of type *Cost*. As previously mentioned, both the input event and newly generated *Cost* event will be subsequently published and further processed before they are caught by interested listeners.

Processor defined in COPAL-ML is compiled into a Java class with the same name that extends BaseProcessor defined in COPAL. BaseProcessor implements ContextProcessor interface required by COPAL for all processors and does necessary steps to register itself with COPAL. The compiled Java class also defines Actions in each input ContextType, so the Processor will be invoked whenever a ContextEvent of any input type is published.

4.4.1 Processing Patterns

As previously mentioned, processors and processing patterns provide us with a solution to compose a complex processing schema for sensory data. In this section, we will show how to implement the processing patterns using the *processor* construct.

- **Filter**

  By returning *null* when a condition fails, we can filter out input events from being further processed and eventually reaching listeners.

```
public processor Filter {
  public Luminance filter(Luminance l) {
    if (l.getDataQuality() >= FINE) {
      return l;
    } else {
      return null;
    }
  }
}
```

- **Aggregation**

  By saving all previously published events, we can aggregate them and publish an aggregated event whenever a new input event is received.

```
public processor OverallPower {
  Map<String, Power> powers;

  public SumPower onPower(Power p) {
    powers.put(p.getSourceID(), p);
    return sum(powers.values());
  }
}
```

- **Differentiation**

  By creating multiple methods that have same input event type but different output event types, we can create different events from the same input event.

```
public processor TemperatureCalculator {
  public Kelvin toKelvin(Temperature t) {
    return calculateKelvin(t);
  }
  public Fahrenheit
      toFahrenheit(Temperature t) {
    return calculateFahreheit(t);
  }
}
```

- **Enrichment**

  By setting an attribute of an event, we can enrich the event with new information.

```
public processor CurrencySetter {
  public Price setCurrency(Price p) {
    p.setCurrency("EUR");
    return p;
  }
}
```

- **Peeling**

  Obviously, by setting an attribute of an event to *null*, we can remove information from the event.

```
public processor ConsumerIDRemover {
  public Price removeID(Price p) {
    p.setConsumerID(null);
    return p;
  }
}
```

4.5 Evaluation

In this section, we compare implementations of Temperature, Thermostat, Ambience, and CostCalculator components in COPAL-ML with their counterparts that use COPAL API[3]. The following table reports the number of files for each component and the number of lines of code.

Component	Files		Lines of code		
	ML	API	ML	API	Δ
Temperature	1	2	6	92	6.52%
Thermostat	1	1	17	54	31.48%
Ambience	1	1	22	104	21.15%
CostCalculator	1	1	57	131	43.51%

For this simple scenario, the code written in COPAL-ML was 255 lines across 17 files in total and the code that uses COPAL API was 1477 lines across 24 files, which is more that five times reduction in code size.

5. RELATED WORK

Applying software engineering approach to develop applications on wireless sensor networks is still in its early stage[7]. Some initiatives have been taken to exercise software models on wireless sensor networks, such as Activity Diagram[3] and State Machine[4]. However, the bridge between models and the actual application execution environment still remains unsolved. Macro languages, like the COPAL Macro Language, are highly promising in this regard. Costa et al.[2] propose TeenyLIME, a WSN middleware that provides developers with the high-level abstraction of the tuple space. Our approach differentiates itself significantly from TeenyLIME. COPAL uses the context provisioning as the programming abstraction over a wireless sensor network and adds an additional processing phase into the sense-and-react paradigm that the both implementations propose. Additionally, TeenyLIME is coupled with the communication layer and poses a requirement that it must overhear messages from neighbouring nodes. Because of this requirement, TeenyLIME inherently can only provide access to information from neighboring nodes. In contrast, COPAL is agnostic w.r.t. the communication layer and provides access to environmental information from all nodes that are connected to it through publishers.

[3]The source code can be downloaded at `http://www.infosys.tuwien.ac.at/m2projects/sm4all/copal/ml/`

Most of past works on context-aware systems have focused on system architecture and context models, but few have studied dedicated programming models for context-aware applications. Several Java-based programming models have been proposed to provide operations on context data based on their underlining middleware architecture. JCAF (Java Context Awareness Framework)[1] is a set of Java API for context management based on a layered context management system. The lowest layer is Context Sensor and Actuator Layer that deals with context acquisition by sensors and actions carried by actuators. The middle layer, Context Service Layer, has the most important API set that provides federated context services. The Client Layer is on top of JCAF, and clients are ultimate consumers of context information. Another approach, along the same line, is Context Toolkit[8] that adopts an idea of widgets from GUI applications. Widgets are encapsulated components, which provide access to abstract context information. They are highly reusable, and the complexity of lower level hardware operations is hidden from application developers. Both approaches define public API that developers use to build context-aware applications. Inherent problem with public API is that once it is published and used by others, it cannot be easily modified. COPAL-ML takes a different approach to programming context-aware applications. It uses keywords to abstract components in context provisioning. This decouples written code from any API specific for COPAL, and even makes it possible to create another compiler which generates code that uses different context provisioning framework instead of COPAL.

In our previous work[5], we have developed a domain-specific language that is able to generate code skeletons for COPAL components. The macro language proposed in this paper extends the idea of designing a dedicated language to provide the programming abstraction on top of COPAL. COPAL-ML improves our previous DSL in several aspects. It employs Java grammar by extending its keywords set, thus the programming on it can be learned quickly. Furthermore, the embedding of Java code is seamless in the macro language. Finally, the DSL generator focuses heavily on the deployment of COPAL components, while COPAL-ML is intended to provide focus on implementing them.

6. CONCLUSION AND FUTURE WORK

This paper introduced COPAL Macro Language in order to help developers with creating context-aware applications. It is based on COPAL middleware, which hides the complexity of a wireless sensor network and provides programming abstraction for the context provisioning. It is centered on defining a common data model in an application using context types. Furthermore, it separates publishers of sensory data from listeners and adds an additional processing phase between them. COPAL-ML helps with rapid development of components in COPAL by decreasing code size and removing burden for developers to learn COPAL API. The language offers a deliberate abstraction over COPAL middleware, thus, developers can focus on describing key components and context provisioning logic by using keywords and concise grammar of COPAL-ML, while retaining the possibility to implement functionality of each component.

We plan to further improve COPAL-ML in several aspects. At first, we plan to improve the developer's control over the processing phase. This will inherently make the language more complex, but it will also enable a more fine-grained, runtime control over how an event is processed. Further more, we will move COPAL-ML from the compile-time phase into the runtime phase of COPAL. This will enable a dynamic redeployment of components and an incremental development without a need to restart an application and lose its current context. Finally, we will combine COPAL-ML and COPAL-DSL into one coherent development tool where COPAL-ML will describe the logic of components and COPAL-DSL their deployment.

7. ACKNOWLEDGMENTS

This work is supported by:

- EU FP7 STREP Project SM4ALL (Smart hoMes for ALL) under Grant No. 224332,
- Pacific Control Systems LLC, Techno Park, Sheikh Zayed Road, Dubai, United Arab Emirates.

8. REFERENCES

[1] J. E. Bardram. The java context awareness framework (JCAF) – a service infrastructure and programming framework for Context-Aware applications. In *Pervasive Computing*, pages 98–115. 2005.

[2] P. Costa, L. Mottola, A. L. Murphy, and G. P. Picco. Programming wireless sensor networks with the teenylime middleware. In *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, Middleware '07, pages 429–449, New York, NY, USA, 2007. Springer-Verlag New York, Inc.

[3] G. Fuchs and R. German. UML2 activity diagram based programming of wireless sensor networks. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications*, pages 8–13. ACM, 2010.

[4] N. Glombitza, D. Pfisterer, and S. Fischer. Using state machines for a model driven development of web service-based sensor network applications. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications*, pages 2–7. ACM, 2010.

[5] F. Li, S. Sehic, and S. Dustdar. Copal: An adaptive approach to context provisioning. In *Wireless and Mobile Computing, Networking and Communications (WiMob), 2010 IEEE 6th International Conference*, pages 286–293, October 2010.

[6] D. Luckham. The power of events: An introduction to complex event processing in distributed enterprise systems. *Rule Representation, Interchange and Reasoning on the Web*, pages 3–3, 2008.

[7] G. P. Picco. Software engineering and wireless sensor networks: happy marriage or consensual divorce? In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications*, SESENA '10, pages 1–1, New York, NY, USA, 2010. ACM. ACM ID: 1809113.

[8] D. Salber, A. K. Dey, and G. D. Abowd. The context toolkit: aiding the development of context-enabled applications. In *Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit*, pages 434–441, Pittsburgh, Pennsylvania, United States, 1999. ACM.