Hindawi
Mobile Information Systems
Volume 2019, Article ID 1209850, 15 pages
https://doi.org/10.1155/2019/1209850

*Research Article*

# CAMeL: A Self-Adaptive Framework for Enriching Context-Aware Middlewares with Machine Learning Capabilities

**Nicola Bicocchi**,[1] **Damiano Fontana**,[2] **and Franco Zambonelli**[2]

[1]*Department of Engineering Enzo Ferrari, University of Modena and Reggio Emilia, Via Vivarelli 10, 41125 Modena, Italy*
[2]*Department of Scienze e Metodi dell'Ingegneria, University of Modena and Reggio Emilia, Viale Amendola 2, Reggio Emilia, Italy*

Correspondence should be addressed to Nicola Bicocchi; nicola.bicocchi@unimore.it

Received 10 July 2018; Revised 14 January 2019; Accepted 20 January 2019; Published 26 March 2019

Academic Editor: Laurence T. Yang



Context-aware middlewares support applications with context management. Current middlewares support both hardware and software sensors providing data in structured forms (e.g., temperature, wind, and smoke sensors). Nevertheless, recent advances in machine learning paved the way for acquiring context from information-rich, loosely structured data such as audio or video signals. This paper describes a framework (CAMeL) enriching context-aware middlewares with machine learning capabilities. The framework is focused on acquiring contextual information from sensors providing loosely structured data without the need for developers of implementing dedicated application code or making use of external libraries. Nevertheless the general goal of context-aware middlewares is to make applications more dynamic and adaptive, and the proposed framework itself can be programmed for dynamically selecting sensors and machine learning algorithms on a contextual basis. We show with experiments and case studies how the CAMeL framework can (i) promote code reuse and reduce the complexity of context-aware applications by natively supporting machine learning capabilities and (ii) self-adapt using the acquired context allowing improvements in classification accuracy while reducing energy consumption on mobile platforms.

## 1. Introduction

The rapid spread of Ubiquitous Computing and the Internet of Things (IoT) technologies is generating a sharp increase in the availability of data somehow representing our living environments [1]. The increasing amount of available data, referred as contextual data, is leveraging the development of applications capable of adapting their behaviour according to the representation of the environment.

In 2001, Dey defined context as *any information that can be used to characterise the situation of an entity. An entity is a person, place, piece of software, software service, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves*. Thus, context-awareness can be defined as *the ability of a system to provide relevant information or services to users using context information where relevance depends on the user's task* [2].

Since then, context-aware applications have been mostly developed using one of the following approaches [3]: (i) applications collecting and processing contextual data in their own manner; (ii) applications collecting and processing contextual data by making use of dedicated and reusable external libraries; (iii) applications built on top of context-aware middlewares providing context management functionalities.

Despite that external libraries provide applications with functionalities while minimising the need of writing original code, they still require to be tailored and assembled in ways that are specific for each application. Instead, middlewares provide in a readily usable fashion, fundamental context management features, such as acquisition, modelling, reasoning, distribution, and visualisation [1]. As a consequence, the third approach frequently outperforms the other two in decreasing the complexity of the development process.

The first efforts for developing context-aware systems focused on exploiting location data, e.g., the Active Badge System [4] and Cricket Compass [5] Later on, context-aware middleware architectures have evolved to achieve more generality and they provide support for more categories of contextual information. Several middleware platforms, including Context Toolkit [2], Gaia [6], Cobra [7], and SOCAM [8], just to name a few, can come in handy for developers for building applications. As a result, making use of context-aware middleware allows developers to focus on designing application functions and business logic instead of managing context.

Recent advances in machine learning and the increased availability of cheap computational power paved the way for acquiring context from complex, information-rich data streams such as audio or video signals [9]. As an example, a camera can be used as a sensor for detecting the presence of people, a location sensor, or a licence plate reader [10]. Alternatively, a microphone can be used as a location sensor or a stress sensor by analysing patterns in sound [11].

This approach is desirable for two main reasons. The former is that a single information-rich data stream contains much more information than a simpler structured stream (e.g., a digital thermometer outputting a floating point number). The latter is that, because of the richer informative content, it is possible to build in software a wide range of sensors thus saving the costs related to the development of dedicated hardware. Because of these reasons, these kind of context sources are getting widespread and highly desirable. However, because current context-aware middlewares only provide mechanisms for accessing sensors producing data in a structured form (e.g., strings, floating point numbers, and Boolean values), developers are still forced to write their own code or use third-party libraries to reduce code reuse or increasing complexity.

Due to these reasons, in this paper we present a framework devoted to context acquisition explicitly supporting machine learning techniques (Source code can be downloaded at https://bitbucket.org/damiano_fontana/awareness) and describe how it can be integrated with off-the-shelf context-aware middlewares. The framework is devoted to transforming data streams into structured data and has been developed to be easily integrated via a compact set of interfaces. By making use of this framework, applications can acquire context from both sensors providing structured data and sensors providing data streams while keeping their internal logic well separated from context management. This framework has been designed around service-oriented, reconfigurable components effortlessly allowing its internal reconfiguration on context changes. The key features of the framework can be summarised as follows:

(i) It can improve the engineering of context-aware applications by introducing machine learning capabilities directly within a context-aware middleware, thus enabling code reuse and preventing the need of using external libraries.

(ii) It can automatically select and reconfigure sensors and machine learning algorithms in a context-based fashion. Experiments showed that this feature might improve classification accuracy while reducing energy footprint on mobile devices.

The remainder of the paper is organised as follows. Section 2 introduces motivations and challenges behind this work. Section 3 presents the global architecture of the framework, while Section 4 details how internal self-adaptation can be configured using state automata. Sections 5 and 6 further detail internal implementation in light of two case studies. Experimental results are presented in Section 7. Section 8 discusses related work, and Section 9 concludes the paper.

## 2. Motivations

Connected objects such as smart phones or smart cameras equipped with increasing computational, connectivity, and sensing capabilities are rapidly being deployed around us. This will eventually contribute to define unforeseen services, ranging from healthcare, transportation systems, to environmental sustainability and participatory governance [12, 13]. The more this process unfolds, the more the services and applications will need to understand the user context in a dynamic and open-ended fashion. Data streams provided by most ICT devices (i.e, cameras, microphones, accelerometers, and magnetometers) are becoming an extremely diffused and inexpensive way for acquiring contextual information. That is, applications increasingly need to collect data streams using available sensors and transfor them into structured information (i.e., context), thus enabling adaptation.

However, context-aware middleware still does not explicitly support the acquisition of contextual information from unstructured data streams. Hence, to reach this goal, applications must rely either on third-party machine learning libraries/modules or on original code written from scratch. Both the approaches have negative implications.

In the first case (Figure 1(a)), developers directly use the context-aware middleware for handling structured data (e.g., $s_1, s_2$). Unstructured data streams (e.g., $s_3$), instead, are routed through a dedicated module, becoming itself a software sensor producing structured information. However, separating the machine learning module from the rest of the context management system implies the inability to adapt its internal functioning based on context. Indeed, it is not possible to dynamically select sensors, classifiers, or modify eventual parameters at runtime. This lack of context-based adaptability negatively impacts pattern recognition in several ways. (i) The combined use of different sensors and classification algorithms is often needed to effectively recognise real-world situations. As an example, crowd-sensing (i.e., a form of data collection distributed among a number of different mobile users) calls for mechanisms for dynamic selection of sensors and classifiers [14]. (ii) Several classification problems require parameters to be modified at runtime. For example, a number of human action recognition systems use different parameters for different groups of people (e.g., males, females, young, and elderly) [15]. (iii)

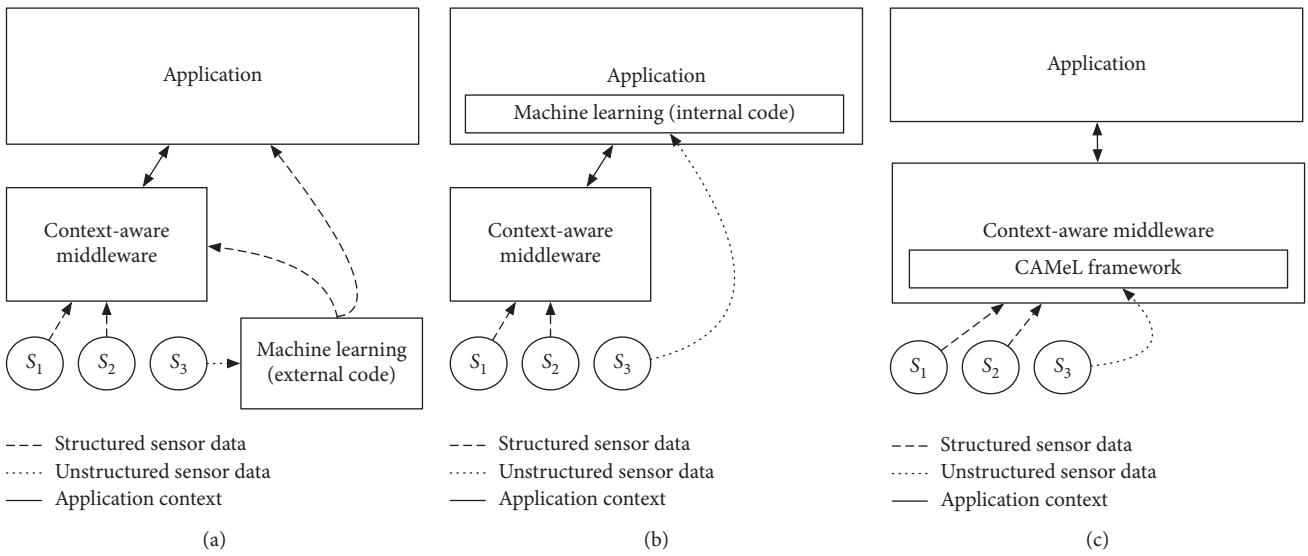


Figure 1: Context-aware middleware acquiring context from both structured data (e.g., a temperature sensor) and unstructured data (e.g., a video stream). Inferring context from unstructured data requires machine learning capabilities (a) to be implemented in a separate module, (b) to be implemented within the application itself, and (c) to be integrated with existing context-aware frameworks achieving benefits both in terms of context-based adaptation and software engineering.

Classification accuracy is inversely proportional to the number of classes to be recognised. A frequent issue in multisensor, multimodal systems is the complexity of the training stage. Different data sources feed a single classifier able to manage a large number of features. Framework allowing programmers to develop specific classifiers and activate them only when needed (i.e., once the proper situation has been identified) is desirable [16].

In the second case (Figure 1(b)), instead, developers implement machine learning code directly within the application. Despite the improved simplicity of the overall architecture (i.e., one component can be removed), this approach lacks the software engineering perspective. In fact, (i) instead of having a collection of well-organised and reusable software components, embedding code within applications reduces its reusability. Furthermore, (ii) this approach cannot fully rely on the underlying middleware. That is, typical features providing middlewares such as context modelling and reasoning must be implemented within the application itself, thus increasing development complexity.

Due to the limitations arising in both scenarios, we propose a modular, reconfigurable framework allowing machine learning to be fully integrated within existing middlewares (Figure 1(c)). It is based on service-oriented and dynamically reconfigurable components [13, 17] allowing to select and reconfigure components depending on the context. The framework is rooted around the following requirements.

*2.1. Software Engineering.* Providing abstractions for both accessing sensors and pipelining classifiers allows the framework to decouple context management from application logic also when unstructured sensor data must be analysed. Furthermore, organising the framework around the idea of reconfigurable components leads to modularity and allows the composition of software ecosystems. Reconfigurable components also allow rapid prototyping of context-aware applications. Developers, in fact, can quickly deploy components (even at runtime) that are already included in the framework or develop original ones, if needed.

*2.2. Extensibility.* The framework is extensible in several ways, without the need of being restarted. In fact, it is possible to deploy, reconfigure, and remove components at runtime. Components can be sensors, classifiers, or modules delegated to fuse different information sources together. In this way, applications can be enriched over time with additional functionalities.

*2.3. Self-Adaptation.* Classifiers based on machine learning algorithms recognise items or events with a level of accuracy which is frequently related with computational complexity. Given a specific classification problem, the framework can be programmed to use specific combinations of sensors and classifiers depending on context. As an example, life-logging mobile applications recognise which vehicle is being used by the user with different approaches (e.g., GPS-based, accelerometer-based, and microphone-based). An application running on an energy-constrained device could use the CAMeL framework to dynamically select the most appropriate trade-off between residual energy and classification accuracy.

## 3. Architecture

This section details the internal architecture of the CAMeL framework in light of the requirements described in Section 2. Its goal is to provide a lightweight, general purpose

software layer for acquiring structured forms of context from unstructured data streams such as video or audio signals. Data streams are classified (i.e., transformed into structured data) and provided to a middleware providing context management to applications.

To deal with the *software engineering* and *extensibility* requirements, it has been developed using OSGi. The OSGi technology is a set of specifications defining dynamic components for the Java language. These specifications enable a development model where applications are (dynamically) composed of different reusable components. OSGi enables components to hide their internal implementation while communicating through services, which are objects shared between components. Furthermore, CAMeL has been organised in a layered fashion, following the separation of concern principle. Indeed, the architecture is structured around three layers, namely, *sensor*, *classifier*, and *awareness* layers (Figure 2). Each layer can host multiple modules (i.e., OSGi components) connected with each other via a dynamic network of queues. Data traverse the whole architecture by means of in-memory queues, enabling decoupling and many-to-many asynchronous communications. On top of these three layers, there are middlewares providing context management (i.e., modelling, reasoning, and distribution) to applications.

To deal with the *self-adaptation* requirement, the three layers are monitored by the *control* layer, a nonfunctional layer in charge of driving internal reconfigurations. Each time a context change is detected, this layer is in charge of verifying if active sensors and classifiers correctly map the current situation and reconfigure the system accordingly.

*3.1. Sensor Layer.* Modules deployed in this layer define which sensors could be possibly activated and used to collect data. These modules (i.e., drivers for actual sensors) hide the complexity and heterogeneity of the hardware and software underneath the framework.

They perform a twofold function: (i) retrieving data from actual sensors by making use of their specific APIs and (ii) preprocessing them in terms of both assessing validity and normalisation. According to the taxonomy proposed in [3], modules deployed within this layer can manage both physical and virtual sensors. The former refers to hardware sensors (e.g., cameras, microphones, and accelerometers), while the latter represents software services. As mentioned above, modules can be configured at runtime on the basis of context. For example, the framework could reduce the frame rate or resize images of a camera to save energy.

*3.2. Classification Layer.* Modules deployed in this layer define the classification capabilities available for any specific application. These modules consume data streams coming from the sensor layer and transform them into structured information. The most common learning techniques (e.g., Naïve Bayes, Bayesian Networks, Hidden Markov Models, Neural Networks, and Support Vector Machines) have been embedded by integrating libraries such as Weka [18], OpenCV [19], and jMir [20] in an OSGi component.

Developers need to deploy the right component (e.g., the Weka component), select the desired classification algorithm, and attach it to specific sensors or a group of them. For example, a module able to recognise human activities from acceleration data could be deployed in this layer. For the sake of experimenting, we have implemented modules for classifying user activity, location, speed, and vehicle used on the basis of Android sensors [21, 22]. Nevertheless, considering the vast range of off-the-shelf algorithms made available from the libraries we have included, most programmers could effortlessly develop the modules they need.

*3.3. Awareness Layer.* Modules deployed in this layer consume structured data (i.e., classification labels) coming from the classification layer and feed context-aware middleware with context. It is not mandatory to deploy modules in this layer in which applications might need only one sensor and one classifier. The main function of this layer is to provide a global representation of the current context by implementing data fusion techniques. These modules receive labels, eventually conflicting with each other, coming from multiple classification modules and apply algorithms to achieve higher semantical levels, remove duplicates, and spot eventual inconsistencies. Data fusion techniques embeddable in this layer such as logic programming, spatial and temporal logic, ontologies, and common-sense knowledge have been surveyed in [23].

*3.4. Control Layer.* For enabling self-adaptation, context is fed to both external middlewares and the internal control layer. This layer provides the framework with self-monitoring and self-reconfiguration capabilities. Given a specific context and a set of rules, this component loads, unloads, reconfigures, and rewires components deployed in all the other layers.

The dynamic selection of the components allows developers to define the most suitable modules to be used in each specific context. It is not feasible, in fact, to deal with a large number of scenarios with a monolithic, static architecture. For example, it is possible to unload sensors and classifiers needed to classify the vehicle being used when the user is located indoor. Alternatively, it is possible to replace a classifier with alternative ones, more lightweight, whenever it is needed.

Reconfiguring components, instead, allows the framework to eventually change parameters of running modules. For example, the sampling frequencies of sensors or the accuracy of classifiers can be modified at runtime according with the computational resources available. Although this feature might not greatly impact classification accuracy, it is useful for reducing energy consumption.

The following section details how internal reconfiguration can be actually programmed.

## 4. Context-Based Reconfiguration

Systems capable of changing their internal functioning must take decisions according to a number of variables. These decisions could be taken on different basis and could be the

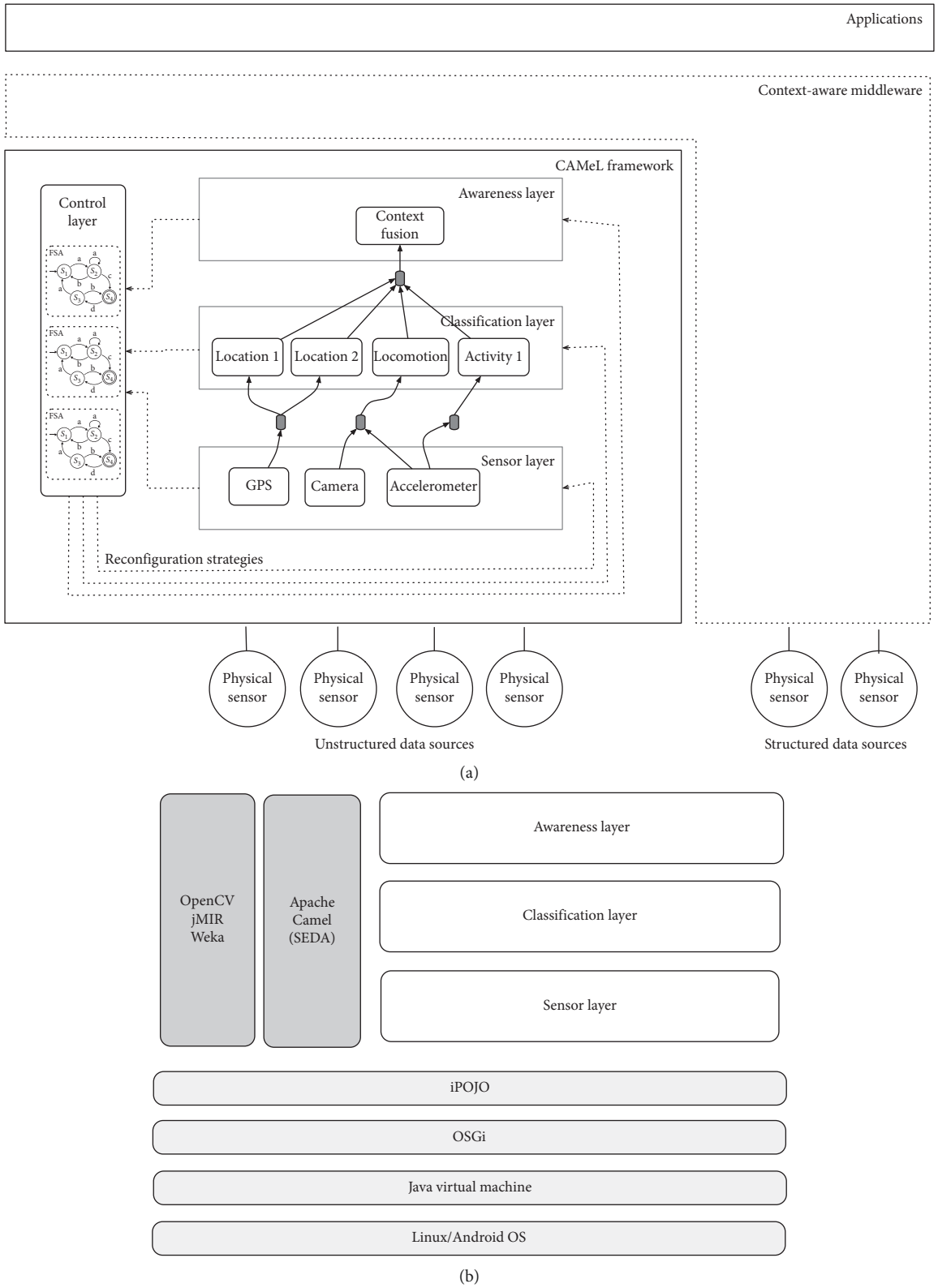


Figure 2: The framework is implemented on top of stable Java technologies. OSGi and iPOJO provide the basis for dynamic execution context and reconfiguration mechanisms, while Apache Camel provides support for the decoupled communication between layers (a). Internally, it is structured around three functional layers, namely, *sensor*, *classifier*, and *awareness* layers, controlled by a *control* layer driving the reconfiguration of the entire framework. The acquired context is then pushed to the above middleware (b).

output of different reasoning processes. Logic, finite-state automata, and machine learning have been used [24]. In this work, we have chosen finite-state machines (FSM) because of their generality. A number of real-world problems, in fact, can be described using this approach. Each FSM represents a set of possible contexts and describe transitions and actions to be taken. Furthermore, FSM are human-understandable. Machine learning techniques, despite their popularity, mostly produce numerical models that cannot be easily understood by humans. FSM, instead, allow driving the reconfiguration using clear diagrams that could be easily monitored and updated by human operators. Furthermore, the computational efficiency of FSM allows real-time applications to run on performance constrained platforms.

FSM are defined with a list of states and transitions. Each state (i.e., node) in the FSM corresponds to a specific context and defines which sensors and classifiers have to be used. It is associated with a set of active modules and a set of possible transitions. Each time the output of an active classifier changes, a reconfiguration is applied. In this way, the overall problem of context recognition is modularised. Each state embeds the knowledge acquired by the previous states and activates more specific classifiers to collect further details. Whenever a transition event occurs (i.e., a specific context is detected), a transition is triggered. More specifically, transitions might imply (i) the reconfiguration of active components, (ii) the deployment of additional components, and (iii) the displacement of active components. Developers only need to define the structure of their scheme by selecting the needed modules and define eventual reconfiguration strategies (i.e., reconfiguration events associated to context states).

To better detail how the framework can be used, we discuss a FSM driving a smart surveillance camera capable of autonomously detecting crowds (i.e., number of people over a certain threshold), as shown in Figure 3. On the right, sensors and classifiers, both active and inactive, for states A, B, and C are represented. The output of each classifier must be standardised in a specific form (e.g., JSON and XML) and is referred in the following as classification label. Each label represents a context change and is fed both to external components (i.e., middleware) and to the control layer. Words over the edges represent the labels produced by active classifiers triggering transitions.

Each state is activated by labels produced in other states. In this case, *state A* is used for detecting the presence of people by making use of a voice classifier, while *state B* activates a camera and an image classifier targeting human bodies. Whenever the number of people exceeds a certain threshold, a label is produced and a transition to *state C* is triggered. Finally, *state C* uses the same sensors and classifiers as *state B* and represents a state in which a crowd is actually present. It allows a transition to *state B* or *state A* whenever the number of people decreases.

It is worth noticing that this programming abstraction is general and not dependent on specific sensors or classifiers. This makes the proposed framework both versatile and compliant with the requirements specified in Section 2. Finally, the actual Java code describing the automata and driving the reconfiguration can be automatically generated by exploiting model-driven engineering techniques. A metamodel describing FSM can be defined and used to generate actual Java code starting from a graphical representation of the system.

## 5. Implementation Insights

From an engineering viewpoint, CAMeL is implemented on state-of-art Java technologies (Figure 2). The reconfiguration mechanisms are provided by OSGi, a well-known Java framework providing service-oriented features. The OSGi specification defines a Java-based service platform for dynamically deploying services into a networked environment. The main abilities contributing to its growing influence are its support of a dynamic service deployment lifecycle and its amenability to remote management.

On top of OSGi, we used an iPOJO layer. iPOJO is a container-based framework handling the lifecycle of *Plain Old Java Objects (POJOs)* and supporting management facilities like dynamic dependency handling, component reconfiguration, component factory, and introspection. Furthermore, iPOJO containers are easily extensible and allow pluggable handlers, typically for the management of nonfunctional aspects. Each CAMeL module is actually an iPOJO component. To provide programmers with off-the-shelf machine learning algorithms, we packed into iPOJO components well-know data-processing libraries such as OpenCV for images and videos, jMir for audio, and Weka for general purpose tasks.

Communications between layers is handled with a staged, even-driven approach provided by the Apache Camel library. This library provides components for asynchronously processing data streams (i.e., SEDA) and in-memory communication with minimum hardware requirements.

The remainder of this section exemplifies the implementation of modules of the layers described so far.

*5.1. Sensor Layer.* Listing 1 shows an example of an iPOJO component representing a sensor. This component is able to acquire frames from a camera and send them to the classification layer. More in detail, the method annotated with *@Validate* is called when the component is deployed. That is, it is up to this method to start the thread fetching images from the camera sensor. The method annotated with *@Invalidate* is called when the component is displaced, and it is up to this method to dispose the thread. Java decorations provide components with a lifecycle completely decoupled from their execution environment. Furthermore, it is worth noticing the small overhead introduced by the framework: only few lines of code are not related to the internal logic of the application but refer to the functioning of the framework itself.

*5.2. Classifier Layer.* Listing 2 shows the code of an iPOJO component for detecting people using the OpenCV API (embedded in CAMeL). The *process ()* method of classes

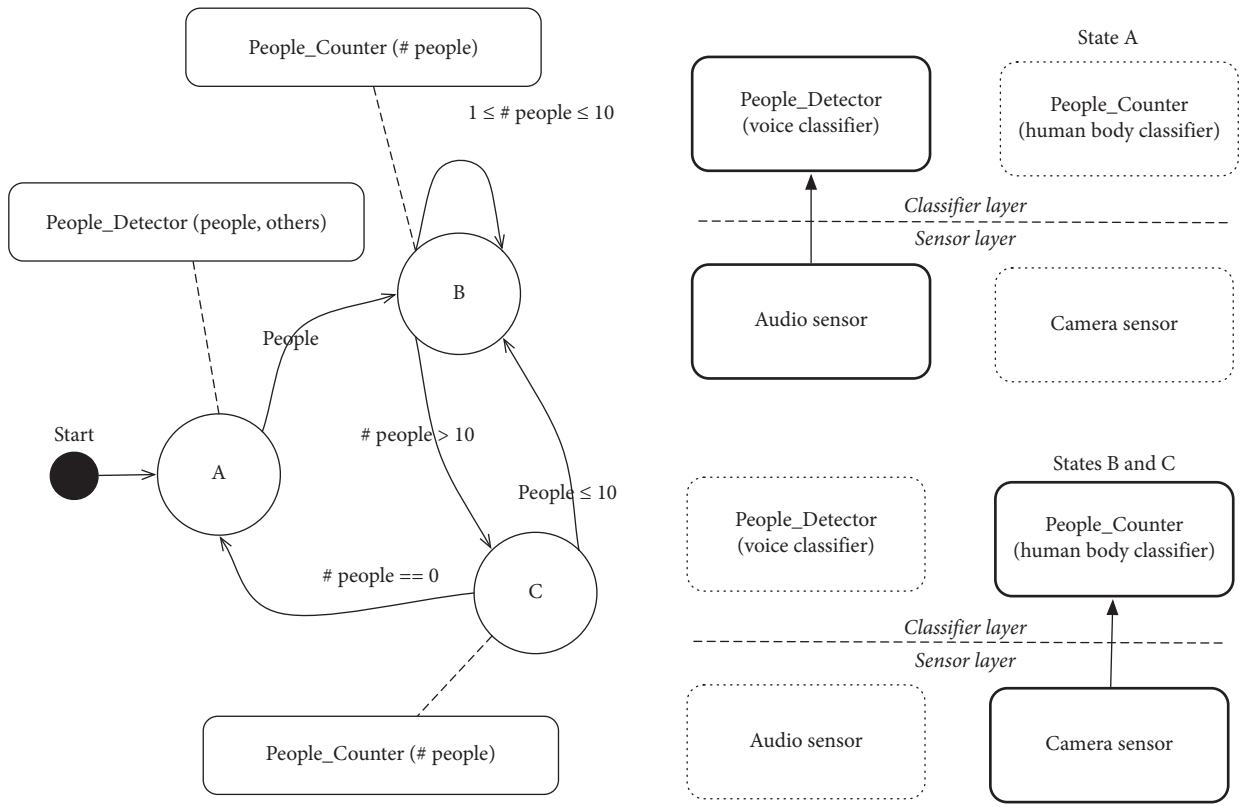


Figure 3: An automaton driving the reconfiguration of a surveillance camera for automatic crowd detection. *State A* is used for detecting the presence of people or not. In *state B* people are counted, while *state C* represents the state in which the system has detected a crowd. Labels over the edges represents the keywords (i.e., the labels produced by active classifiers) triggering transitions.

implementing the *Processor* interface (i.e., in this example *HumanBodyProcessor*) is called every time a new item is available in the input queue (i.e., *seda:image*). When this happens, the image is consumed and processed. In this case, we made use of OpenCV to detect the number of bodies present in the scene. The result of the classification is then sent to the output queue (i.e., *seda:label*) to feed the awareness layer. *HumanBodyProcessor* does not need to be aware of the sensor actually producing images. Indeed, they could be produced by any component able to acquire images from a video source (e.g., smartphone camera, CCTV camera, and webcam) and store them in the *seda:image* queue.

*5.3. Control Component.* Listing 3 describes the code of the control component associated with the FSM used for driving reconfigurations (Figure 3). The *configure ()* method of the *Controller* class translates in Java the states and transitions of the automaton. Actions to be taken, when a state transition is triggered, are defined in the *StateMachine* class.

The method *transitFromAToBOnPeople ()* of the class *StateMachine* shows the reconfiguration undertaken during the state transition triggered by the label *"people"*. In this example, the framework APIs are used for (i) turning off the audio sensor and the voice classifier and (ii) activating the video sensor and human body classifier. Looking at this code, it is worth noticing how reconfiguration has been decoupled from the actual code of the application itself. Reconfiguration strategies are general and can be used in any application independently from the its internal logic.

This approach also supports a development process undertaken by a community. The framework, in fact, allows reconfiguration strategies to be compiled directly from actual state diagrams. Using automatic code-generation techniques, it would be possible to drag and drop sensors and classifiers within a graphical application, wire them to specific states, and let the software to write the corresponding Java code. If eventually the number of available modules becomes significant, the development of the context collection section of a number of diverse applications might be greatly simplified.

## 6. Case Studies

To demonstrate our approach, we describe how two context-aware applications could be implemented by making use of CAMeL. The following applications are implemented in particular:

(i) A smartphone application able to collect data about the life of the user such as her activities, locations, vehicles used, and eventual presence of people talking around

(ii) A surveillance drone application able to detect people within specific areas of interest

```
(1) @Validate
(2) public void start () throws Exception {
(3)     Thread t = new Thread (this);
(4)     t.start ();
(5) }
(6) @Invalidate
(7) public void stop () {
(8)     stop = true;
(9) }
(10) public void run () {
(11)     FrameGrabber grabber = null;
(12)     try {
(13)         ProducerTemplate producer = context.getContext ().createProducerTemplate ();
(14)         //image grabbing via OpenCV API
(15)         grabber = FrameGrabber.createDefault (0);
(16)         grabber.start ();
(17)         IplImage grabbedImage = grabber.grab ();
(18)         while (stop == false && (grabbedImage = grabber.grab()) ! = null) {
(19)             //. . . creating imageData object
(20)             //send data to the "image" queue
(21)             producer.sendBody (imageSedaQueue, imageData);
(22)         }
(23)         grabber.stop ();
(24)     } catch (Exception e) {
(25)         //. . . exception handling
(26)     }
(27) }
```

Listing 1: iPOJO module capable of acquiring a frame from a camera using OpenCV.

```
(1) public class HumanBodyProcessor implements Processor {
(2)     private CvHaarClassifierCascade classifier = new CvHaarClassifierCascade (cvLoad ("body.xml"));
(3)     public void process (Exchange exchange) throws Exception {
(4)         //getting image raw data
(5)         ImageData imageData = exchange.getIn ().getBody (ImageData.class);
(6)         //. . . rgb to gray conversion
(7)          //classification of bodies in the image via OpenCV API
(8)         CvSeq bodies = cvHaarDetectObjects (grayImage, classifier, storage, 1.1, 10);
(9)         int total = bodies ();
(10)        LabelData labelData = new LabelData ("body", total);
(11)        //send classification label to the awareness layer
(12)        producer.sendBody ("seda:label", labelData);
(13)    }
(14) }
```

Listing 2: iPOJO module able to detect faces in images consumed from an SEDA queue.

Both the systems have the goal of collecting data, classifying them, and providing context to applications through a context-aware middleware. As we discussed, in Section 2, developers could implement everything from scratch or make use of external libraries. The first option would require a significant coding endeavour, while the second one would not allow to exploit the typical context management features offered by middlewares. Instead, developers could keep the application simpler by delegating context acquisition and management to a middleware integrated with CAMeL. The middleware could manage the needed sensors and classifiers (in a context driven fashion) to acquire context while providing applications with context management features. In this way, the two applications would become extremely simple: they could receive context from the middleware and act accordingly either by saving the data (data logging application) or by driving the drone (drone application). Thus, the most relevant part of those applications would be the automata used by CAMeL for internal reconfigurations. Figure 4 shows both of them.

Figure 4(a) describes the automata used for the life-logging application. Following the represented states, from

```
(1) @Component
(2) @Provides (specifications = RoutesBuilder.class)
(3) public class Controller extends RouteBuilder {
(4)    ...
(5)    public void configure () throws Exception {
(6)       //state automata defintion
(7)       StateMachineBuilder <StateMachine, State, String, Context> bld = StateMachineBuilderImpl.newStateMachineBuilder
      (StateMachine.class, State.class, String.class, Context.class);
(8)       //definitions of the states and state transitions
(9)       bld.externalTransition ().from (State.A).to (State.B).on ("People");
(10)      bld.externalTransition ().from (State.B).to (State.C).on ("Crowd");
(11)      bld.externalTransition ().from (State.B).to (State.A).on ("No_Crowd");
(12)      bld.externalTransition ().from (State.C).to (State.A).on ("No_Crowd");
(13)      stateMachine = builder.newStateMachine (State.Start);
(14)      stateMachine.setInstanceFactory (instanceFactory);
(15)      from (labelQueue).process (new LabelProcessor ());
(16)   }
(17)   ...
(18) }
(19) public class StateMachine extends AbstractStateMachine <StateMachine, State, String, Context> {
(20)   private InstanceFactory instanceFactory;
(21)   public void transitFromAToBOnPeople (ControllerState from, ControllerState to, String event, ControllerContext ctx) throws
      Exception {
(22)      //destroy Audio Sensor and Voice Classifier
(23)      instanceFactory.disposeInstance ("sensor.Audio", "audio");
(24)      instanceFactory.disposeInstance ("classifier.Voice", "voice_classifie");
(25)      //create Camera Sensor and Human Body Classifier
(26)      instanceFactory.createInstance ("sensor.CameraSensor", "CameraSensor");
(27)      instanceFactory.createInstance ("classifier.HumanBodyDetector", "HumanBodyDetector");
(28)   }
(30) }
```

Listing 3: An example of control component able to take decision based on the current context.

A to F, it is possible to understand the logic of the context acquisition process. First, the system tries to detect the current location ($S_A$). Then, if the user goes outdoor, user activities and speed are assessed using additional sensors ($S_B$). While the user stays outside, if her speed exceeds a specified threshold, ($S_D$) is activated for detecting eventual vehicles. Otherwise, in case of slow speed, user activities are monitored. In case of running, ($S_F$) is activated to recognise the type of location (i.e., street or park). Alternatively, if the user goes back indoor, the system tries to detect if the user is at home, at the office, or in some other place ($S_C$). In case of other places, ($S_E$) is activated for discriminating among pubs, restaurants, and other locations. It is worth noticing how our approach allows context-based applications to be expanded and made eventually more accurate over time. If fact, new states, specialising the existing ones, can be added over time.

Figure 4(b), instead, drives the reconfiguration of a surveillance drone. ($S_A$) is activated as soon as the drone takes off and detects areas of interests. As soon as an area of interest is spotted, ($S_B$) is activated and eventually people are detected. ($S_C$), activated only when people are detected inside an area of interest, analyses audio signals to detect dialogs. Finally, ($S_D$) refines ($S_C$) by inferring the general topic of the conversation. As of before, the actual mechanisms used to detect areas of interest, people, and dialogs can be changed effortlessly among different application versions and even at runtime.

It is worth noticing that the two automata can be defined in terms of either Java code or a graphical representation allowing automatic code generation. This approach allows application code to be almost completely detached from the mechanisms used for collecting context. If fact, one could completely change the application (e.g., user interface, features provided, and storage system) without changing sensors, classifiers, and FSM used. Alternatively, it is possible to change sensors, classifiers, and automaton without modifying the actual application code. Context collection is completely delegated to CAMeL and applications need to only define behaviours associated to the current context. This feature is likely to sensibly speed up the prototyping of pervasive applications.

## 7. A Case Study for Experimental Evaluation

Using a case study, we evaluated (i) how CAMeL internal reconfigurations impact both classification accuracy and energy consumption and (ii) how CAMeL behaves in terms of both performance and scalability (Used dataset can be downloaded at http://www.agentgroup.unimore.it/Bicocchi/files/dataset01.tar.gz).

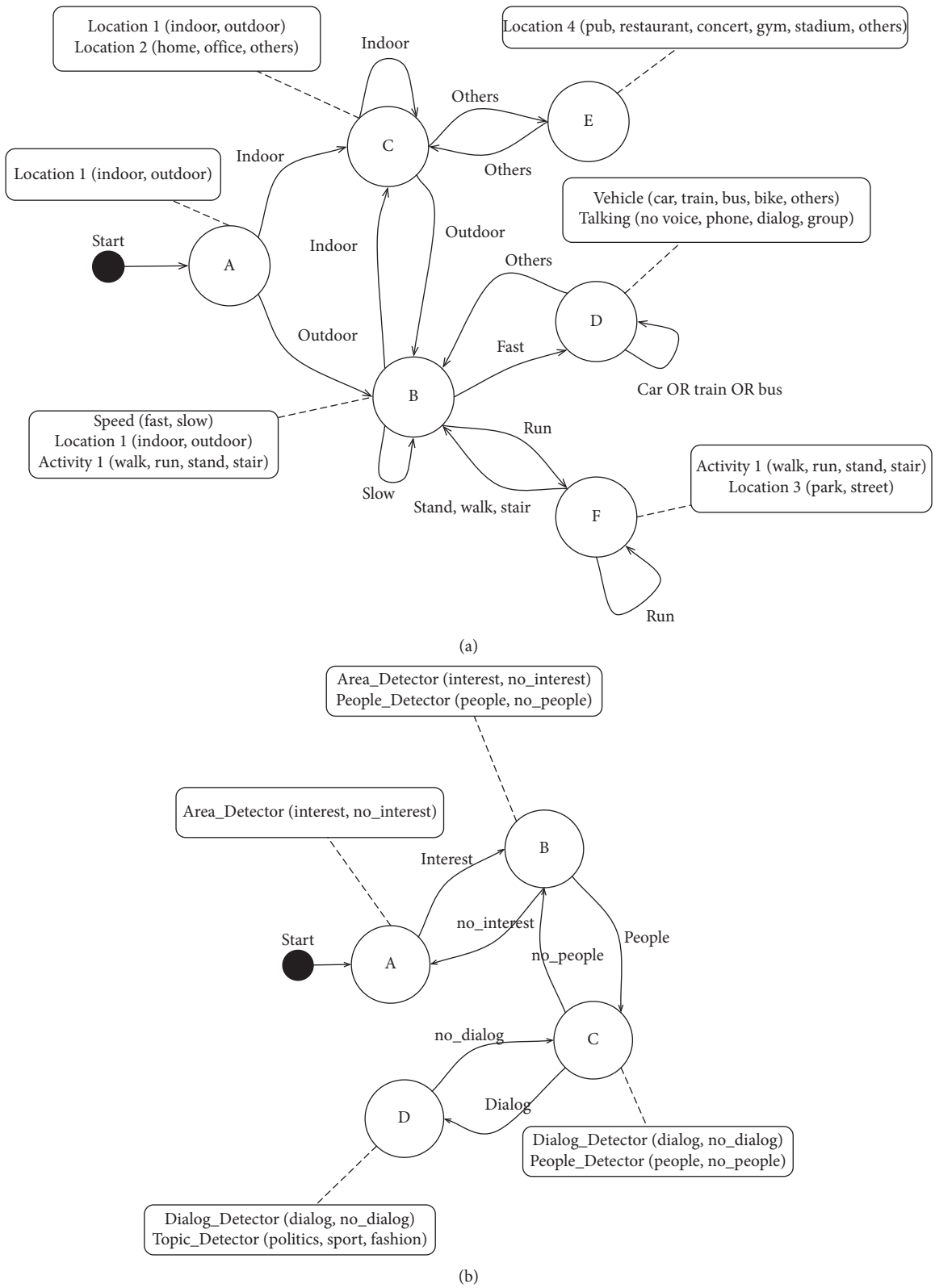


Figure 4: Two examples of reconfigurations of the framework: (a) a smartphone application collecting data about the life of the user, in particular, her activity, location, vehicles used, and people talking around; (b) a surveillance drone application designed for detecting people within specific areas of interest.

Accuracy and energy consumption have been evaluated by implementing the life-logging application described in Section 6 on an Android smartphone. In order to train the classifiers used, we collected and annotated ground truth data with the Funf utility developed at MIT [25]. Performance and scalability have been tested on both desktop and mobile platforms.

*7.1. Accuracy and Energy Consumption.* As introduced above, we wanted to show the benefits of the CAMeL framework by comparing two slightly different versions of the life-logging application. The former making use of the reconfiguration capabilities provided by CAMeL, the latter using a more traditional approach keeping all the needed sensors and classifiers always activated.

We define as context $s = \{\text{activity}, \text{location}, \text{speed}, \text{vehicle}\}$. Each field of the tuple is limited to a specific set of values. In particular activity = {walk, run, $s$ tan $d$, sit}, location = {indoor, outdoor}, speed = {slow, fast}, vehicle = {car, bus, train, other}. The goal of the application is to log, every minute, a tuple $s$ describing the context of the user on a SQLite table.

Each field of the tuple is handled by a specific classifier: (i) an activity classifier using accelerometer and microphone data using a discriminative core based on SVM with a 64 dimensions; (ii) a location classifier using a sliding window of 15 seconds; (iii) a speed classifier recognising fast and slow movements by computing the average speed over a sliding window of 10 GPS samples; (iv) a vehicle classifier using microphone data collected at 44.100 Hz, 16 bit, mono, and divided in windows of 4 seconds, that is based on SVMs and two feature vectors computed for each window: a 13 dimension Mel-Frequency Cepstral Coefficient (MFCC) feature vector and a 10 dimension Linear Prediction Coefficients feature vector. We recorded a trace of 4 hours for all sensors from 5 different users. Results have been 10-fold cross validated.

Considering that the vehicle classifier, associated with $S_D$, is the most active in our dataset in terms of both total activity time and number of on/off transitions, we refer to it for explaining the results (Figures 5(a) and 5(b)). The classification making use of internal reconfigurations achieved in most of the cases (i.e., train, car, and bus) a 10% improvement in precision while recall did not vary significantly. The improvement derives from the fact that automata-driven reconfigurations avoid errors related to classifiers working without context (e.g., a vehicle classifier working while users are walking in a busy street).

Nevertheless, the improvement in classification accuracy has been paired with a substantial reduction of the energy consumption (Table 1). For each status of the automaton, we reported both the sensors active and the percent of the total execution time in which the state was active. Overall, the GPS was active around 46% of time, the microphone 54%, and the accelerometer 6%. These data suggest that the CAMeL framework allows a reduction of the overall energy consumption around 50%. We measured the battery life of the same smartphone running the two versions of the application and obtained an actual improvement around 40% (i.e., 10 h always-on; 14 h using CAMeL).

*7.2. Performance and Scalability.* The inherently portable nature of the CAMeL framework allows it to run on both desktop and mobile platforms. However, given the great performance difference that is frequently present between the two platforms, these results might be more relevant for the mobile domain.

The framework is built on OSGi components exchanging messages via in-memory queues. Although its load on the CPU in frequently low, the queues used for message passing among components might grow and shrink over time due to different processing speeds of the involved components. If the queues reach their maximum or minimum capacity, the number of exchanged messages among components decreases. Because of this, two relevant metrics for measuring performance and scalability of the framework are (i) its memory usage and (ii) its throughput (i.e., average number of messages per second processed) on both desktop and mobile architectures. Both the experiments have been undertaken by increasing the number of parallel components in the classifier layer processing a stream of messages without payload (injected as fast as possible) without further computations. Due to different hardware constraints, we injected to each active classifier 500 and 10000 messages on the desktop and mobile platforms, respectively. Results have been averaged over 10 rounds (Figures 5(c) and 5(d)).

Figure 5(c): The framework's throughput. In this experiment, we progressively increased the number of components in the classification layer and measured the average number of messages per second processed by each of them. The graph shows how the average throughput decreases linearly with an increasing number of components consuming the data stream. This result means that the framework does not imply a significant overhead to the applications. In fact, the throughput of each classifier decreases linearly with the number of classifiers. Furthermore, even in the worst case (80 parallel classifiers), the average throughput remains acceptable for most of the applications (around 400 msg/sec).

Figure 5(d): The memory usage. In particular, the graph shows how the heap memory usage increases linearly with the number of messages processed for both desktop and mobile architectures. As of before, a linear correlation implies a scalable framework.

*7.3. Discussion.* Although this case study does not allow to draw conclusions that can be applied to every possible application, it is enough for formulating some general principles.

Firstly, the framework showed its effectiveness in capturing context from video and audio signals. The life-logging application developed using CAMeL worked properly under different circumstances. The application has been able, in fact, to describe the portion of day the user lived in a meaningful way (i.e., precision and recall around 80%). It is also worth noticing that, being machine learning capabilities integrated with a middleware, the resulting application code has been extremely compact. The application simply received from the middleware a string representing the current context and saved it on a relational database.

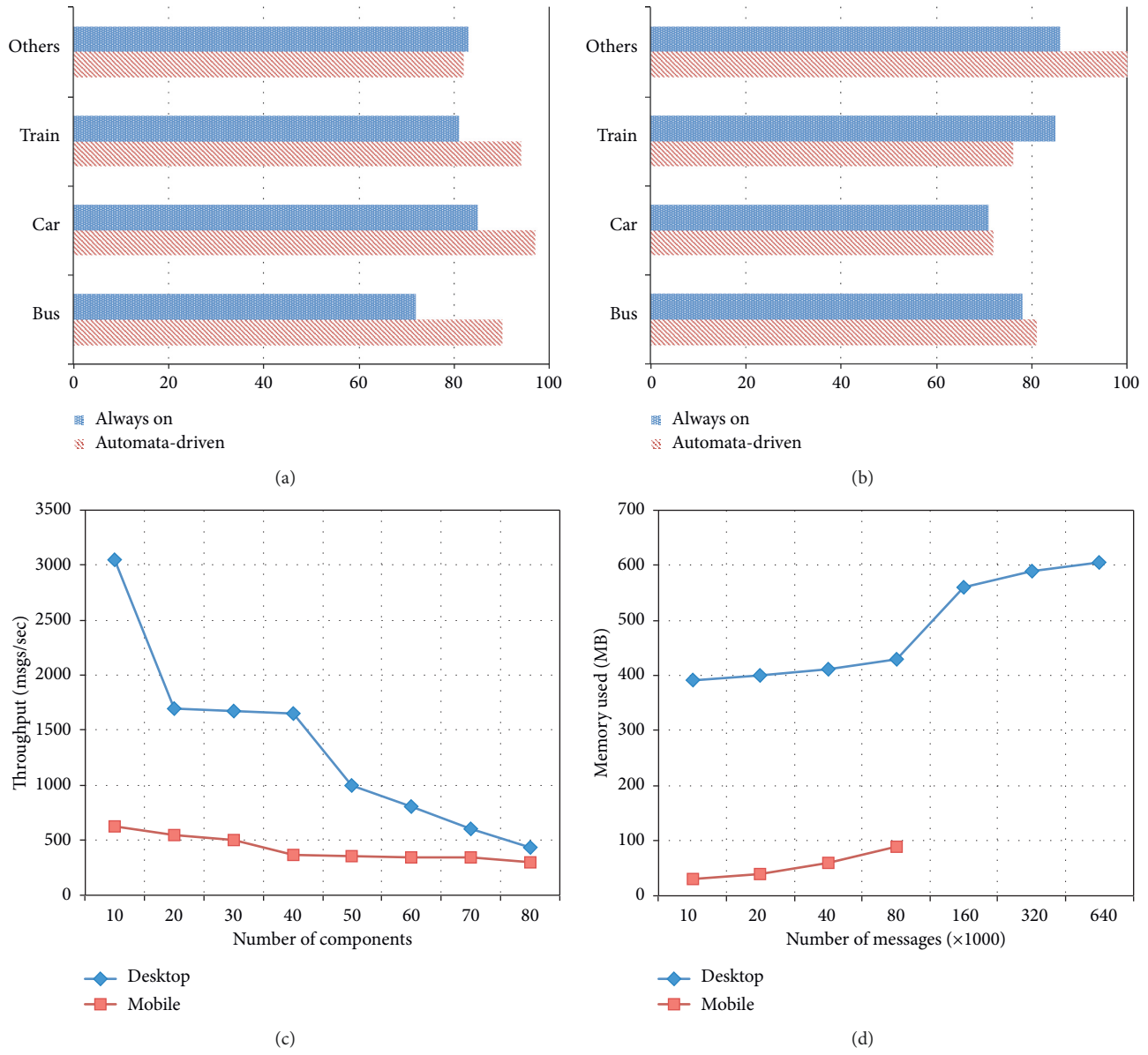


Figure 5: Precision (a) and recall (b) of vehicle classification ($S_D$) obtained with or without automata-driven reconfiguration. The average throughput (c) of classification components decreasing linearly with an increasing number of components. The heap memory used by our framework (d) increasing linearly with the number of messages processed. Both curves have been computed for both desktop and mobile architectures.

Table 1: Power consumption report for each status of the FSM used for reconfiguring the life logging application.

| State | Active sensors | Time (%) |
|---|---|---|
| $S_A$ | GPS | 0.03 |
| $S_B$ | GPS | 0.24 |
| $S_C$ | GPS, accelerometer | 0.06 |
| $S_D$ | Microphone | 0.54 |
| $S_E$ | GPS | 0.03 |
| $S_F$ | GPS | 0.10 |

Note: shown are both the active sensors and the percent of the time the status was actually active.

Secondly, the case study showed how reconfiguration strategies positively affect both accuracy and power consumption. FSM, in fact, allows developers to associate the knowledge of the environment to states. Each state is a refinement of the previous one. Each state encapsulates a specific situation and allows to further refine it or go back to more general cases. This approach allows to use sensors only when needed and to reduce misclassification errors. The case study showed, for example, how the GPS sensor (power hungry) can be effectively replaced with a microphone for more than half the execution time. It is also worth noticing

how FSM allows rapid prototyping and testing. Around such a model, a community of developers might build up large collections of different FSM for handling the vastness of the real world. These models could be also easily connected and integrated with each other due to the simplicity of FSM.

Thirdly, the framework showed its lightweight nature and its scalability in a convincing way. An average Android smartphone, in fact, is equipped with around 10 hardware sensors. We can imagine applications using additional 10 software sensors, each one sampling at 10 Hz. We can also formulate the hypnotises that each sensor is connected to a proper classifiers. Even in this case, with 20 sensors and 20 classifiers, we would need a throughput lower than 100 messages/second. The analysis we conducted showed that, with 40 components, the framework is capable of handling, even a mobile platform, almost 400 messages/second. Similar conclusions can be observed for memory usage as well. The envisioned application should work with around 400 messages/second, thus requiring as little as 50 MB of RAM.

## 8. Related Work

The CAMeL framework relies on established design principles (i.e., separation of concerns, service-oriented architectures), providing developers with a tool for acquiring and managing context inside available middlewares. Its design and development are rooted in both software engineering and context-aware pervasive computing. Related work in both areas is discussed below.

From a software engineering perspective, a number of middlewares and frameworks have been designed for decoupling context management from application code. However, to the best of our knowledge, none of them supports reconfigurable machine learning capabilities with a programmable API.

In [26], a framework capable of managing changes in requirements related to context management without impacting on applications has been proposed. It relies on an RDF model enabling developers to define their own modules. Developers have to comply with the model and the semantic provided, which are used to manage context data and the corresponding management operations. However, the use of ontology-based reasoning makes the framework little scalable in which its performance decreases considerably when the number of managed contexts increases.

In [27], the authors present a platform providing context to applications based on a layered approach using third-party modules to infer knowledge from context data. The platform uses a predefined ontology, allowing developers to supply their inference modules. Context distribution relies on subscriptions to specific contextual data and notifications about the time these data become available. However, the ontology provided by the framework implies a specific semantic, thus limiting the number of target application scenarios.

In [24], a framework for decoupling context management from application code is proposed. Its aim is to reduce the overhead of applications running on resource-limited devices while still providing mechanisms to support context-awareness and application adaptation. The framework structures context by making use of atomic functions. These functions can be designed by third-party developers using an XML-based programming language. Although this approach shares several foundational basis with our approach, it does not provide mechanisms for collecting context from sensors and subsequent classifiers.

Researchers developing context-aware pervasive applications prototyped sensing systems able to acquire detailed contextual information from data streams [9–11]. The most prominent works have been surveyed in [16]. The majority of these studies, however, lacks in generality and addresses specific classification problems by making use of a predefined set of sensors.

Few of them implemented frameworks that are flexible, resource efficient, and robust in a large plethora of situations. For example, Lu et al. [28] propose to use processing pipelines on various sensors (i.e., GPS, microphone, and accelerometer) to show how processing pipelines and dynamic reconfiguration could be used in continuous sensing. However, the framework does not focus on a generic approach enabling runtime reconfigurations. It is tied to the presented case study and cannot be programmed in other ways. Thus, it does not allow the framework to be reconfigurable and programmable in a general way. On the contrary, Cimino et al.[29] describe a framework based on ontologies to deal with heterogeneous user behaviours. However, it does not allow context to drive the reconfiguration of sensors and classifiers.

In [30–32], the authors propose to optimise the sensing process in terms of power saving. In particular, Nath [30] exploits reasoning techniques for learning associations among context attributes for optimising the internal logic of the framework. However, like previous studies, it lacks generality and focuses only on the optimisation of energy consumption in continuous sensing.

The study in [33] demonstrates that our approach of modelling human activities and contexts with finite-state automata can successfully describe a number of real-world scenarios and drive internal reconfigurations. It describes a programming approach for pervasive systems based on high-level models of human activities, so-called situated flows, for uncovering task information embedded in physical environments.

In [34], the authors propose a UML-based, event-driven model for context-aware services based on two views: the former representing contextual data and events triggered by changes in context. The latter representing the dynamic of context-aware services such as different scenes, transitions between them, and service behaviour corresponding to a scene and that could be initiated as a response to a transition. Although the model is event-driven and based on finite-state machines, adaptation is provided for a given scene and not for an event. These state machines are predefined and cannot be customised dynamically for a given application.

Finally, all the previous studies do not focus on easing the development of pervasive applications while CAMeL comes with an FSM programming abstraction embedded. To the best

of our knowledge, our self-aware and reconfigurable framework represents a first attempt to design a programmable awareness module able to meet developer needs and specific requirements. Thanks to its general and self-aware architecture, it is able to implement and make full use of all the strategies and optimisations proposed in the previous studies and it is able to deal with more complex scenarios that require flexibility and adaptability as foundational basis.

## 9. Conclusion

In this paper, we proposed a framework able to enrich general purpose context-aware middlewares with machine learning capabilities. It is capable of collecting data from a number of different sources and classifying them using a variety of algorithms under the only assumption of being encapsulated within as OSGi/iPOJO container. The framework is highly dynamic and reconfigurable, allowing developers to activate and reconfigure sensors and classifiers in a context-based fashion. Its internal reconfigurations are programmable with a simple programming interface based on finite-state machines. Experimental evaluation showed its usefulness for (i) enabling code reuse and reducing complexity of context-aware applications; (ii) self-adaptation using the acquired context, allowing improvements in classification accuracy while reducing energy consumption on constrained platforms.

## Data Availability

The data used to support the findings of this study have been deposited in the following repository (https://bitbucket.org/damiano_fontana/awareness).

## Conflicts of Interest

The authors declare that they have no conflicts of interest.

## Acknowledgments

This work was supported by the ASCENS project (EU FP7-FET, Contract no. 257414).

## References

[1] A. U. R. Khan, M. Othman, F. Xia, and A. N. Khan, "Context-aware mobile cloud computing and its challenges," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 42–49, 2015.

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

[3] X. Li, M. Eckert, J. F. Martinez, and G. Rubio, "Context aware middleware architectures: survey and challenges," *Sensors*, vol. 15, no. 8, pp. 20570–20607, 2015.

[4] R. Want, A. Hopper, V. Falcão, and J. Gibbons, "The active badge location system," *ACM Transactions on Information Systems*, vol. 10, no. 1, pp. 91–102, 1992.

[5] N. B. Priyantha, A. K. Miu, H. Balakrishnan, and S. Teller, "The cricket compass for context-aware mobile applications," in *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking (MobiCom '01)*, pp. 1–14, ACM, New York, NY, USA, July 2001.

[6] M. Roman and R. H. Campbell, "Gaia: enabling active spaces," in *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System (EW 9)*, pp. 229–234, ACM, New York, NY, USA, September 2000.

[7] H. Chen, T. Finin, A. Anupam Joshi, L. Kagal, F. Perich, and D. Dipanjan Chakraborty, "Intelligent agents meet the semantic web in smart spaces," *IEEE Internet Computing*, vol. 8, no. 6, pp. 69–79, 2004.

[8] T. Gu, H. K. Pung, and D. Q. Zhang, "A service-oriented middleware for building context-aware services," *Journal of Network and Computer Applications*, vol. 28, no. 1, pp. 1–18, 2005.

[9] O. Yurur, C. H. Liu, Z. Sheng, V. C. M. Leung, W. Moreno, and K. K. Leung, "Context-awareness for mobile sensing: a survey and future directions," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 68–93, 2016.

[10] C. Nüesch, E. Roos, G. Pagenstert, and A. Mündermann, "Measuring joint kinematics of treadmill walking and running: comparison between an inertial sensor based system and a camera-based system," *Journal of Biomechanics*, vol. 57, pp. 32–37, 2017.

[11] M. Sharmin, A. Raij, D. Epstien et al., "Visualization of time-series sensor data to inform the design of just-in-time adaptive stress interventions," in *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pp. 505–516, ACM, Osaka, Japan, September 2015.

[12] P. V. Krishna, V. Saritha, and S. Sivanesan, "Ubiquitous context aware services," in *Soft Computing Applications in Sensor Networks*, CRC Press, Boca Raton, FL, USA, 2016.

[13] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, "Context aware computing for the internet of things: a survey," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 1, pp. 414–454, 2014.

[14] L. Wang, D. Zhang, Y. Wang, C. Chen, X. Han, and A. M'hamed, "Sparse mobile crowdsensing: challenges and opportunities," *IEEE Communications Magazine*, vol. 54, no. 7, pp. 161–167, 2016.

[15] Y. He and Y. Li, "Physical activity recognition utilizing the built-in kinematic sensors of a smartphone," *International Journal of Distributed Sensor Networks*, vol. 9, no. 4, Article ID 481580, 2013.

[16] W. Z. Khan, Y. Xiang, M. Y. Aalsalem, and Q. Arshad, "Mobile phone sensing systems: a survey," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 1, pp. 402–427, 2013.

[17] J. Ye, S. Dobson, and S. McKeever, "Situation identification techniques in pervasive computing: a review," *Pervasive and Mobile Computing*, vol. 8, no. 1, pp. 33–66, 2012.

[18] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software," *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[19] G. R. Bradski and A. Kaehler, *Learning Opencv*, O'Reilly Media, Inc., Newton, MA, USA, 1st edition, 2008.

[20] C. Mckay and I. Fujinaga, "JMIR: tools for automatic music classification," in *Proceedings of the International Computer Music Conference*, Montreal, Quebec, Canada, August 2009.

[21] N. Bicocchi, M. Mamei, and F. Zambonelli, "Detecting activities from body-worn accelerometers via instance-based algorithms," *Pervasive and Mobile Computing*, vol. 6, no. 4, pp. 482–495, 2010.

[22] L. Ferrari and M. Mamei, "Classification and prediction of whereabouts patterns from the reality mining dataset," *Pervasive and Mobile Computing*, vol. 9, no. 4, pp. 516–527, 2013.
[23] L. Snidaro, J. Garcia, and J. M. Corchado, "Context-based information fusion," *Information Fusion*, vol. 21, pp. 82–84, 2015.
[24] B. Chihani, E. Bertin, and N. Crespi, "Programmable context awareness framework," *Journal of Systems and Software*, vol. 92, pp. 59–70, 2014.
[25] N. Aharony, W. Pan, C. Ip, I. Khayal, and A. Pentland, "Social fMRI: investigating and shaping social mechanisms in the real world," *Pervasive and Mobile Computing*, vol. 7, no. 6, pp. 643–659, 2011.
[26] J. Zhu, H. Pung, M. Oliya, and W. Wong, "A context realization framework for ubiquitous applications with runtime support," *IEEE Communications Magazine*, vol. 49, no. 9, pp. 132–141, 2011.
[27] P. Gutheim, "An ontology-based context inference service for mobile applications in next-generation networks," *IEEE Communications Magazine*, vol. 49, no. 1, pp. 60–66, 2011.
[28] H. Lu, J. Yang, Z. Liu, N. D. Lane, T. Choudhury, and A. T. Campbell, "The jigsaw continuous sensing engine for mobile phone applications," in *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, pp. 71–84, Zurich, Switzerland, November 2010.
[29] M. G. C. A. Cimino, B. Lazzerini, F. Marcelloni, and A. Ciaramella, "An adaptive rule-based approach for managing situation-awareness," *Expert Systems with Applications*, vol. 39, no. 12, pp. 10796–10811, 2012.
[30] S. Nath, "Ace: exploiting correlation for energy-efficient and continuous context sensing," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys '12)*, pp. 29–42, Ambleside, UK, June 2012.
[31] S. Kang, J. Lee, H. Jang, Y. Lee, S. Park, and J. Song, "A scalable and energy-efficient context monitoring framework for mobile personal sensor networks," *IEEE Transactions on Mobile Computing*, vol. 9, no. 5, pp. 686–702, 2010.
[32] M. Schirmer and H. Höpfner, "Senst∗: approaches for reducing the energy consumption of smartphone-based context recognition 6967," in *Proceedings of the International and Interdisciplinary Conference on Modeling and Using Context*, pp. 250–263, Karlsruhe, Germany, September 2011.
[33] F. Kawsar, G. Kortuem, and B. Altakrouri, "Designing pervasive interactions for ambient guidance with situated flows," in *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, pp. 371–375, August 2010.
[34] T. Mo, W. Li, W. Chu, and Z. Wu, "An event driven model for context-aware service," in *Proceedinds of the IEEE 20th International Conference on Web Services*, pp. 740-741, Washington, DC, USA, July 2011.