Entity-Adaptation: A Programming Model for Development of Context-Aware Applications

Sanjin Sehic, Stefan Nastic, Michael Vögler,
Fei Li, and Schahram Dustdar
Distributed Systems Group, Vienna University of Technology
Argentinierstrasse 8/184-1, A-1040 Vienna, Austria
{lastname}@dsg.tuwien.ac.at

ABSTRACT

In recent years, new business and research opportunities have increasingly emerged in the field of large-scale pervasive platforms (e.g., building management systems, pervasive health-care, environmental monitoring). These platforms are characterized by the need to monitor and control a large number of heterogeneous environments with significantly different capabilities and utilize many distributed context sources and actions. Hence, they require additional support in terms of programming models and abstractions that can assist with the development of context-aware applications as generic and reusable components. Previous research studies in context-aware systems have proposed many different programming models, but none of them are enough to support the development of truly reusable applications.

In this paper, we introduce the *Entity-Adaptation* programming model as a novel approach for the development of context-aware applications. The benefit of the Entity-Adaptation model is that it decouples context-aware applications from the underlying physical environments and allows them to be implemented as generic and reusable components. We additionally present the design and implementation of *the CAPA framework*, which provides support for deployment and execution of context-aware applications developed with the Entity-Adaptation programming model. Finally, we evaluate the solution using a case study that demonstrates effectiveness of the approach in a real-world scenario.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Languages, Design

Keywords

Context-awareness, programming model, framework, pervasive systems, mobile computing

SAC 2014, March 24 - 28 2014, Gyeongju, Republic of Korea

ACM 978-1-4503-2469-4/14/03 ...$15.00.
http://dx.doi.org/10.1145/2554850.2555015.

1. INTRODUCTION

Context-awareness is one of the cornerstones of pervasive computing [26, 24]. It refers to the idea that an application can understand its context, reason about its current situation, and perform suitable operations based on this knowledge. Moreover, as the situation changes over time, the application should adapt according to the new circumstances and use new context information to decide which new actions need to be performed. This dynamic adaptation of context-aware applications provides a level of automation that can free users from unnecessary manual labor. For example, context-aware applications in smart-homes can unobtrusively support automatic lightning and heating based on user preferences and current context information like luminosity, temperature and user location.

Recent years showed emerging trends in business and research to utilize large-scale, distributed pervasive platforms. Examples of such trends are building management systems, pervasive healthcare, city traffic scheduling, environmental monitoring, and smart grids. These platforms differ significantly from the conventional context-aware systems that focus on a limited personal context in relatively controlled environments (e.g., smart homes and offices). They are characterized by the need to monitor and control a large number of heterogeneous environments with significantly different capabilities and utilize many, distributed context sources and actions. Because of the size and complexity of such a platform, it is not feasible to develop a separate context-aware application that is specifically suited for each physical environment where it is deployed. Hence, these platforms require additional support in terms of programming models and abstractions that can assist with the development of context-aware applications as *generic and reusable* components.

Previous research studies in context-aware systems have proposed many different programming models for the development of context-aware applications. These programming models support hiding the heterogeneity of context sources [6, 11, 15, 10, 25], defining processing schemes [6, 25], and dealing with mobility [6, 11, 10], privacy [15] and scalability [6, 11, 25]. Unfortunately, none of the them are sufficient to support the development of truly reusable applications that can work in many, heterogeneous environments. The fundamental problem of all these approaches is that information of what is expected from a physical environment is implicitly coded in the application. They either expect that all physical environments are identically configured and provide same context information and actions or that users will manually check that all necessary prerequisites for the physical environment are satisfied before executing an application. Thus, for large-scale pervasive systems none of these programming models provide an adequate solution. Instead, a programming model has to allow applications

to explicitly state which context information and actions they expect from a physical environment. As a result, the context-aware system will be able to automatically validate that an application will work correctly in a physical environment and customize it to the environment and user needs.

In this paper, we present the *Entity-Adaptation* programming model that provides support for the development of reusable context-aware applications. The core idea behind the programming model is based on the concept of entities as a specification that describes which context information and actions are expected from objects (e.g., an area, lights or a person) in an environment. Entities provide an universal programming interface on top of which adaptations can be defined instead of them being implemented to directly use concrete context sources and actuators. Hence, context-aware applications become reusable components that can work correctly in many, heterogeneous environments. Furthermore, we present the design and implementation of the *Context-Aware Programming frAmework* (CAPA) that provides support for deployment and execution of context-aware applications developed using the Entity-Adaptation programming model.

The rest of the paper is organized as follows. The next section describes the challenges in developing context-aware applications and summarizes requirements that emerge out of these challenges. In Section 3, we introduce the Entity-Adaptation programming model. The design and implementation of the CAPA framework is discussed in Section 4. Afterwards, Section 5 presents a case study and Section 6 examines the related work. Finally, Section 7 concludes the paper.

2. CHALLENGES

To better understand all challenges associated with the development of context-aware applications, let us consider applications that are responsible for monitoring and controlling an environment. In general, context-awareness refers to the idea that an application can understand its context and adapt its behavior based on information gathered from the environment without an explicit user intervention. Thus, such applications would use *context information* to determine the current state of the environment, store *user preferences* in order to better understand the current situation in the environment, invoke some *context actions* to adapt its behavior and optionally notify the *user* or update an *user interface* (Figure 1). An appropriate use-case would be an application for automatic control of ambient temperature. This application determines when and for how long to turn on heating and cooling (actions) based on current temperature (context information) and desired temperature range (user preference). Next, we will present four challenges associated with the development of such a context-aware application.

Since instances of a context-aware application will monitor and control heterogeneous physical environments, this provides us with a hint to the first challenge. If we consider different environments, applications will a) depend on different techniques for context information to be gathered from heterogeneous sensors, b) be required to adapt differently to the current situation based on varying user preferences, and c) invoke different actuators through context actions. To cope with such heterogeneity, developers of context-aware applications will need dedicated *programming abstractions* to hide this intrinsic complexity by creating a loose coupling between the environment and the application. As such, these abstractions will have to provide a programming tool to develop context-aware applications using environment-agnostic terms to permit reuse of applications in heterogeneous environments and hence increase their reusability and maintainability. However, this creates a mismatch between programming abstractions that are used during the development and context sources and actuators that are used during the execution. Thus, the underlying context-aware system will be required to associate appropriate context sources and actuators to the programming abstractions before the application can work correctly.

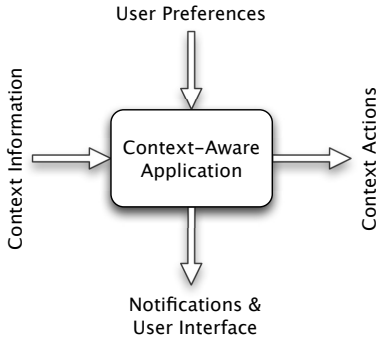

Figure 1: Context-Aware Application

The second challenge in development of context-aware applications arises when we consider how context information is provisioned. More specifically, the context information originates from the outside of the control of the applications. This means that context-aware applications are not able to actively notice changes in the context information. With regard to this drawback, there are two possible models to overcome this challenge: polling and pushing. With the polling model, the context information has to be continuously read in predetermined intervals and sent to applications if the information has changed based on the previously read value. With the pushing model, a source of context information can notice changes in the context information and thus is responsible with sending the new context information to applications. Nevertheless, both models require that the underlying context-aware system *asynchronously* provisions new context information to applications.

The third challenge arises from expectation of latest use-cases. It is not enough anymore to collect context information from a single environment like smart home or office as in many previously proposed systems [10, 3, 5, 16]. Instead, we need to allow context information to come from *multiple, distributed environments* to allow even higher effectiveness of context-aware applications. For example, the automatic ambient temperature control can take advantage of the user location that is collected from a smartphone and use it to calculate the estimated time of the arrival. This information together with the information about the required heating duration allows us to determine if it is at all necessary to immediately turn on heating even when the current temperate is outside the desired temperature range. This context can potentially minimize unnecessary heating and cooling when user is not at home. Furthermore, we can also include information from electric company about future fluctuation of electricity prices. The application can then determine the cheapest strategy to heat or cool the home. For example, we can slightly overheat the home during the daily low price hours and hence be more cost-effective. However, we cannot expect that context-aware applications should be responsible in creating and managing distributed environments. This task must be responsibility of the underlying context-aware system. From the perspective of applications, local and remote context information and actions are indistinguishable.

Lastly, the final challenge emerges from a requirement that users should be able to *customize* their environments to their needs and expectations by deciding how context-aware applications interact

with their environments. For example, we should be able to easily configure the application for the control of ambient temperature to indirectly monitor location of multiple users by provisioning only the location of the closest user to the application (i.e. determining minimum distance from home for set of locations). From the perspective of a context-aware application, this way of provisioning the location of multiple users should be identical to using location of single user and hence not influence its execution in any meaningful way. However, this requires that the underlying context-aware system allows users to determine how context sources and actuators are used by the application in their environments. Furthermore, it should also allow users to change this configuration during the execution of the application without its disruption.

To summarize, based on the these challenges a context-aware solution has to satisfy these four requirements:

- *Requirement 1*: Context-aware applications have to be developed using dedicated programming abstractions that provide an environment-agnostic interface. This allows applications to be developed as generic and reusable components that can work correctly in many, heterogeneous environments.
- *Requirement 2*: The context-aware system must provision new context information to applications in an asynchronous fashion so they can adapt their behavior and control the environment. This requirement is a consequence of the fact that context information originates outside of the control of context-aware applications.
- *Requirement 3*: The context-aware system has to provision context information and actions from many, distributed environments. This requirement can lead to increase in the effectiveness of context-aware applications by providing them with even more context to use for adaptations than they could only get from local context.
- *Requirement 4*: Users of the context-aware system have to be able to configure how context-aware applications interact with their environments. This enables them to customize their environments to their needs and expectations.

3. THE ENTITY-ADAPTATION MODEL

The core concept of the Entity-Adaptation programming model are entities as commonly defined in the context-awareness research field. More formally, "an entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves." [9] In the programming model, we define an entity as a specification that describes which context information and actions are expected from an object (e.g., an area, lights, web service or a person) in an environment. Thus, the set of all entities that are defined in a context-aware application provide the full description of requirements for an environment. The main purpose of entities is to allow context-aware applications to be implemented in an *environment-agnostic* manner. This allows developers to freely define all entities necessary for the correct execution of the application and to implement adaptations on top of them instead of concrete context sources and actuators. More precisely, they provide an *universal* programming interface for monitoring and controlling an environment in context-aware applications.

During development of a context-aware application, developers can freely define any number of entities. Each entity is associated with a *name*, *properties*, and *actions*. The *name* of an entity allows differentiation of one entity from another. Furthermore, it can help during the deployment of the application to provide users with a meaning for the entity so they can easily choose which context sources and actuators to associate with them. *Properties* in an entity represents a context information associated with this entity. They can either be abstract or concrete. An abstract property provides a description of the context information by defining a name and type. During the deployment of the application, each abstract property will be associated with a concrete context source or a static value that matches this description. On the other hand, a concrete property is defined on top of other properties and provides a computation that creates a higher-level context information using the context information from underlying properties. Thus, concrete properties can be used to model situations in which the entity is currently in. *Actions* in an entity represent actuators that can be executed in an environment. They can either be abstract or concrete as well. An abstract action provides a description of an actuator by defining a name and a type for its argument. During the deployment of the application, each abstract action will be associated with a concrete actuator that matches this description. A concrete action defines necessary arguments for the action and is implemented on top of other actions that are appropriately invoked based on the specified arguments.

Because user preferences can be seen as additional context information, we model them as a configuration entity associated with necessary properties. This way, the application can use same mechanism to access properties and context information and hence simplify its design. Furthermore, this allows users instead of just setting preferences as static values to potentially associate these properties to concrete context sources like databases, outputs of other applications and so forth.

```
val state =
  property[State]
      (area("temperature"),
       conf("temperature")) {
    case (current, (min, max)) =>
      if (current < min) COLD
      else if (current > max) HOT
      else OK
  }
val change =
  action[(State, Range)] {
    case (state, (min, max)) =>
      if (state == COLD)
        area.exec("heat", max)
      else if (state == HOT)
        area.exec("cool", min)
  }

val area  = entity("area")
              .hasProperty[Float]("temperature")
              .hasProperty("state", state)
              .hasAction[Float]("heat")
              .hasAction[Float]("cool")
              .hasAction("change", change)
val user  = entity("user")
              .hasProperty[Location]("location")
              .hasProperty("distance")(haversine)
              .hasAction[String]("notify")
val conf  = entity("configuration")
              .hasProperty[Location]("home")
              .hasProperty[Range]("temperature")
```

Listing 1: Definition of Entities

To demonstrate this, in Listing 1[1] we define entities required by an application to control the ambient temperature. This example shows creation of three entities: `area`, `user`, and `configuration`. The `area` entity has one abstract property `temperature` and one concrete property `state`, which uses information about the current temperature from the `area` entity and the range of desired temperature from the `configuration` entity to compute if current temperature is `COLD`, `HOT`, or `OK`. Furthermore, the `area` entity has two abstract actions, namely `heat` and `cool` that both require one `Float` argument representing the desired temperature that should be reached and one concrete action `change` that based on the current `state` determines and executes either the `heat` or the `cool` actions with the desired maximal or minimal temperature respectively. The `user` entity, has one abstract property `location` and one concrete property `distance`, which uses the `home` location from the `configuration` entity to determine how far is the user from the home location using the Haversine formula[2]. Moreover, the `user` entity has one abstract action `notify` that allows the application to send a textual message to the user in case of a problem with heating and cooling.

After entities are defined in a context-aware application, the adaptations can be implemented on top of them. Each adaptation defines, which properties it is monitoring for change and based on the new context information determines which actions needs to be executed. This design is identical in nature to Model-View-Controller (MVC) software design pattern. In our case, properties are equivalent to models, actions to views, and adaptations to controllers. Changes in context information trigger invocation of adaptations which based on the current context information in entities determine which actions to invoke and pass them appropriate arguments. Hence, we named our programming model for the development of context-aware applications as *Entity-Adaptation*.

```
onChange(area("state"),
         user("distance"),
         conf("temperature")) {
  case (state, location, range) =>
    if ((distance < 1000.meters)
        && (state != OK)) {
      change(state, range)
}
```

Listing 2: Adaptation

Lastly, an example of an adaptation is shown in Listing 2. We use the `onChange` method that will monitor changes in given properties and executes specified computation. In our case, the adaptation monitors changes in user's distance from home and state. If user is less than 1000 meters away from home and state is not `OK`, the adaptation will invoke the `change` action, which consequently will determine if heating or cooling is necessary.

4. The CAPA framework

In this section, we will present the design and implementation of the CAPA framework. The main responsibility of the CAPA framework is to provide support for the deployment and execution of context-aware applications developed using the Entity-Adaptation programming model. Figure 2 shows the overview of the CAPA framework and its position in relation to context-aware applications and the underlying infrastructure.

[1]All examples are written in the Scala programming language.

[2]The Haversine formula returns the distance between two points on a sphere from their longitudes and latitudes.

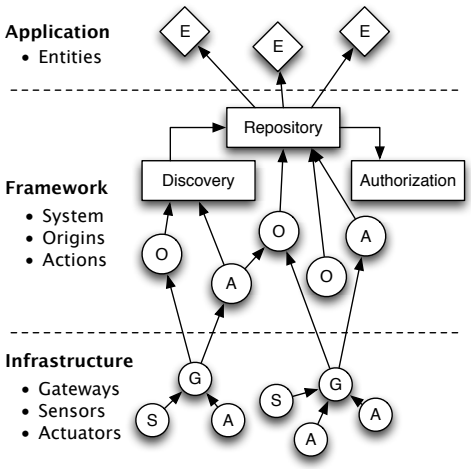

Figure 2: The CAPA framework

4.1 Asynchronicity

In context-aware systems, context changes in an environment are observed as external asynchronous events. Therefore, we designed the CAPA framework on top of the programming paradigm offered by the Actor Model [13, 1] and futures [12, 18].

The Actor Model provides the abstraction for transparent distribution of concurrent computations [1]. In the Actor Model, an *actor* is a universal primitive to represent concurrency [13]. It reacts to messages from the outside by sending other messages, creating new actors and changing its behavior. In the CAPA framework, actors provide a foundation for event-driven reacting to context changes in environments. They allow in a natural way to asynchronously deliver context information as events by sending and receiving messages between actors. Context changes can still be gathered using polling and pushing techniques inside the CAPA framework, but as soon as a change is detected it becomes a message in the Actor Model, which is then furthered processed and enriched with meta-information by actors in the asynchronous fashion. Furthermore, the CAPA framework executes each context-aware application inside an actor which receives context changes, executes its adaptations and sends context actions to appropriate actuators. This alleviates necessity to use synchronization and thread-locking primitives in the application. Thus, developers can focus on implementing correct adaptations of behavior in their context-aware applications without worrying about asynchronous nature of context events.

On the other hand, a future represents "a promise to deliver the value" [12]. This mechanism allows consumers of futures to continue with the execution of other tasks and use the result when it is available in the asynchronous fashion. Moreover, an interesting aspect of futures is that it enables the *promise pipelining* [18]. This concept allows futures and processing operations to be composed in a parallel and asynchronous manner into new futures, which can be further processed and/or consumed. Futures and promise pipelining provides a very powerful mechanism for distributed systems to overcome latency and unreliability of communication [18]. In the CAPA framework, they provide a programming technique to asynchronously retrieve context information, which can possibly require communication with remote and unreliable servers. For example, when in the application that controls ambient temperature an unfavorable change in the temperature is detected, the CAPA framework still has to retrieve the current location of the user from

his smartphone, which might not even be connected to internet, before both pieces of context information can be delivered to the application. In this case, the location information is represented as a future that will be sent together with the current ambient temperature when it is available. This allows the CAPA framework to easily model timeouts, retry and failure logic for each retrieval of context information and thus help us with uncertainty in communication between remote and distributed systems. In the aforementioned case, when location information is not available, the application is still invoked with context information that has no location information and hence it can decide how to proceed with the controlling of the ambient temperature.

4.2 Origins and Actions

To support the provisioning of context information in the CAPA framework, we use the *Origins* model presented in our previous work [25]. The Origins model is a programming model for the development of context-aware applications in large-scale, distributed pervasive systems. Its design allows a pervasive system to provide a flexible infrastructure for the execution of context-aware applications and to easily scale with the increase in the number of applications. The core idea in the Origins model is that *origins* provide an adequate abstraction to represent any type of context source like sensors, web services, databases, files, but also compositions of other origins. They are universal, discoverable and composable components that are associated with name, type and meta-information. Based on the origins, four processing operations are defined, namely *filtering*, *inference*, *aggregation*, and *composition*, that provide a powerful mechanism to express a rich set of processing schemes for context information.

In the CAPA framework, we extend the concept of origins with the additional push-based communication mechanism to support fully asynchronous, reactive programming. Moreover, we implement the concept of *actions* as an actuating counterpart to origins to create a full-fledged programming support for monitoring and controlling of physical environments. Actions provide an *universal* interface to execute operations in an environment. They can invoke the execution of actuators, applications, and remote systems. The actual mechanism that an action uses to invoke these operations is irrelevant for context-aware applications and is hidden from them. This allows reusability in applications because actual implementations of actions can be different in different physical environments but applications can still use them as an universal component that can be invoked. Furthermore, actions can be implemented to be *context-aware*. This functionality allows actions to determine their own behavior based on the current context information and adapt to changes in the environment. We distinguish three cases where context information can be used to improve the functionality of actions: a) determining arguments/settings for the environment operations, b) adapting to changes in the environment, and c) defining end-condition for long-running operations. For example, the action to heat some physical environment can depend on the current temperature to determine the necessary power setting of the heater and to end heating when desired temperature range has been reached. Thus, actions are allowed to use origins to properly function and determine how, when and which operations need to be executed in the environment. Finally, like origins, actions can be reused in other actions and composed together to provide more fine-grained actions. For example, we can define a sequence of actions that should be executed sequentially or a set of actions that should be executed in parallel. This mechanism of composing actions in any particular fashion provides a very powerful method to implement execution schemes.

4.3 Handling Distribution

A system component in the CAPA framework represents a virtual environment consisting of one or more physical environments that can be monitored and controlled. Its main responsibility is to allow access to context sources through origins and invoke context operations through actions. Furthermore, it provides a deployment-time and run-time support for a context-aware application by binding entities with concrete origins and actions, executing the application after all bindings are fulfilled and allowing these bindings to change during the execution of the application. Each instance of a system consists of three subcomponents: *authorization*, *discovery*, and *repository*. The authorization component is responsible for managing and controlling access rights to the system. The discovery component is responsible for finding context sources and actuators in the underlying physical environments. The repository component provides support to create, update, and remove origins and actions in the system. Thus, it allows users to create additional higher-level origins that depend on other, lower-level origins. For example, user can create another location origin that returns the distance of the closest user from his/her home.

In the CAPA framework, we distinguish between three types of systems: *local*, *view*, and *composite*. A *local* system is a system that directly communicates with context sources and actuators. Its main responsibility is to allow users to manage origins and actions. Furthermore, any local system can be accessed remotely by allowing other, remote systems that have proper authorization to access its origins and actions. An example when this functionality can be used is when a system that is deployed on user's smartphone is remotely accessed by a system deployed in his/her home so the home system can access the location information and send user notifications. A *view* system is a special case of a local system that represents a partial view of origins and actions in the local system that can be used to grant separate authorization rights to the view and not whole local system. For example, this allows us to have one major local system for a smart building, which consists of many minor view systems for each apartment and office in the building. Each view systems grants its occupants rights to execute context-aware applications for monitoring and controlling of only this particular apartment or office without them having the same right for the building system. Finally, a *composite* system allows us to compose one or more local or remote systems into one system that provides access to all underlying origins and actions and allows deployment and execution of context-aware applications. Hence, the composite system provides necessary support to create *distributed systems* that can span many physical environments.

4.4 Deployment & Configuration

When a context-aware application is deployed in a system by a user, the first task of the system is to try and find suitable origins and actions for each abstract property and action in the application's entities. Therefore, we devised a binding algorithm (Listing 3) to help users with the deployment of context-aware applications.

The search process in the binding algorithm is similar to the web service discovery mechanisms [8, 21]. It uses the type information from definitions of properties and actions in an entity to find concrete origins and actions that have same type (lines 5 and 12). If the search process is not able to find a suitable origin or action it returns with an error (lines 6 and 13). Furthermore, if there is more than one suitable origin or action definition, the definition is stored in an unresolved collection as it will require a manual intervention from the user by allowing him/her to manually select appropriate origin and action from the list of suitable ones (lines 9 and 16). Otherwise, if only one suitable origin or action is found, the system

```
1  def bind(app: Application, sys: System) {
2    for (entity <- app.entities) {
3      val unresolved = Vector.empty
4      for (p <- entity.properties) {
5        val origins = sys.origins.findAll(p.type)
6        if (origins.isEmpty) return error(p)
7        if (origins.size == 1)
8          entity.bind(p, origins(0))
9        else unresolved += (p, origins)
10     }
11     for (a <- entity.actions) {
12       val actions = sys.actions.findAll(a.type)
13       if (actions.isEmpty) return error(a)
14       if (actions.size == 1)
15         entity.bind(a, actions(0))
16       else unresolved += (a, actions)
17     }
18     if (!unresolved.isEmpty)
19       entity.bind(manualBind(unresolved, sys))
20   }
21 }
```

Listing 3: The binding algorithm

will bind it with the entity (lines 8 and 15). At the end, if the search process did not find all suitable origins and actions, the system will ask the user to choose from a list appropriate origins and actions that should be used by the entity (line 18–19).

Finally, because all bindings between entities in a context-aware application and concrete origins and actions in a system are created and controlled by the system, we can easily allow changing of individual bindings without restarting the application. Thus, users can install a new sensor or change an old one and afterwards easily reconfigure context-aware applications to use the new sensors.

4.5 Implementation

The CAPA framework is implemented in the Scala programming language[3]. Scala is a general-purpose programming language that integrates features of object-oriented and functional programming paradigms. It runs on the Java Virtual Machine (JVM) and is byte-code compatible with Java applications. We choose to implement the CAPA framework in Scala because of its flexible syntax that allows us to easily create domain-specific languages inside the language and package them as libraries. Furthermore, the CAPA framework was developed as a set of reusable components using design concepts described by Odersky and Zenger [20]. This technique allows the CAPA framework to be reprogrammed with minimal changes to its system component and deployed on top of different infrastructures like Wireless-Sensor Networks (WSN), Internet of Thing (IoT) systems, ubiquitous/pervasive platforms, and even other context-aware middlewares.

In the CAPA framework, we use the Akka toolkit[4] as an implementation of the Actor Model and futures. Akka provides a platform to build highly concurrent, distributed, and fault-tolerant event-driven applications. Beside its implementation of actors and futures, we use Akka for its transparent remoting and clustering capabilities, which help us with accessing remote systems (transparent remoting) and creating distributed composite system (clustering).

5. CASE STUDY

As a case study for the CAPA framework, we integrated it as a part of a building management system which is responsible for managing over 10000 buildings with each one having in average 300 sensors and actuators. In this deployment, the building management system uses one Niagara gateway[5] per building to provide a single point of access to sensors and actuators in the building. We use the Niagara gateway as the underlying infrastructure on top of which a system component is deployed. Thus, a single gateway creates a single virtual environment which can be monitored and controlled by a context-aware application. The system's discovery component communicates with the gateway using oBIX standard [19]. The oBIX standard provides the concept of points that represent a single scalar value as an abstraction for sensors and actuators in a physical environment. The oBIX discovery component instantiates one origin or action for each point in the system depending if it is sensor or actuator. Furthermore, we deployed additional origins and actions that utilized origins and actions discovered by the oBIX discovery component.

As part of our case study, we implemented the aforementioned context-aware applications for the control of ambient temperature (Listings 1 and 2). The first system provided access to a temperature sensor and to an AC unit that were both connected to a Niagara gateway. The second system was deployed as a background service on an Android device, which accessed the device's location information through its GPS sensor. By combining these two systems into a composite system, we executed the application which managed to adapt and control the AC unit using the temperature information from the gateway and the location information from the Android device.

During the case study, we dealt with many situations where a smartphone was not able to send the current user location due to the lack of Internet connection. The uncertainty and volatility of communication is an everlasting challenge in distributed system. In this regard, the concept of futures helped significantly by allowing us to easily model timeouts, retry and failure logic without blocking the execution of the CAPA framework in the process. Furthermore, during the implementation of the case study, we also observed a shortcoming of the binding algorithm (Listing 3) with respect to creating entities for temporary objects. This type of entities are transiently created and destroyed by the framework. They allow monitoring and controlling objects in a context-aware application that are temporarily in the environment (e.g., guests, workers). The shortcoming arises when the algorithm requires a manual intervention from the user, which is unsuitable in this situation. Therefore, we plan to extend the binding algorithm with the semantic reasoning to help it overcome the manual binding step.

6. RELATED WORK

Prior research studies in programming models for development of context-aware applications were suitable only for small and static environments like smart homes or offices. Not much effort was put into providing a programming models that allows context-aware applications to be reusable between many different systems and environments. As such, there is a significant gap between the vision of the context-awareness in pervasive computing from [26, 24] and the previously proposed context-aware programming models. A detailed analyses of previous context-aware approaches appear in [2, 4, 22]. In this section we will review some of these previous approaches and compare them to the CAPA framework.

The Context Toolkit provides a programmatic support for the development of context-aware applications [10]. The Context Toolkit incorporates various services related to gathering and provisioning of context information, including encapsulation of context, access

[3]http://www.scala-lang.org/

[4]http://akka.io/

[5]http://www.tridium.com/

to context information, storage, and a client-server infrastructure. The central idea of the system is borrowed from GUI (Graphical User Interface) toolkits and widget libraries that create reusable building blocks and hide specifics of physical system. Thus, context widget represents a dedicated component that is responsible for acquisition of context information directly from sensors. Widgets can then be further combined using interpreters to provide higher-level context information or with aggregators that group related context information. Finally, the toolkit provides a concept of services, which are responsible for invocation of actuators. Although origins have some similarity with context widgets, they additionally provide support for composition that context toolkit does through interpreters and aggregators. Thus, origins can be considered as a more general concept than widgets, interpreters and aggregators. Furthermore, context toolkit does not provide any concept equivalent to entities and as such cannot fully support the concept of reusable, context-aware applications (Requirement 1). Finally, it lacks any support to handle distributed environments (Requirement 3).

JCAF is a Java-based service-oriented run-time infrastructure and API for the creation of context-aware applications [3]. The JCAF run-time infrastructure emphasizes security and privacy in an environment of distributed and cooperating services that acquire context information through Context Monitors and Context Actuators. It enables interested applications to subscribe to relevant context events through an event-based publish-subscribe mechanism. The JCAF programming model provides programmers with tools to create context-aware applications that are deployable in the JCAF infrastructure. The programming model consists of the remotely-accessible API through Java Remote Method Invocation (RMI) for the context services, the model for context information, and the event-based infrastructure. Although their programming model standardizes development of context-aware applications, it also lacks an abstraction equivalent to entities as a specification of what an application expects from an environment (Requirement 1). More specifically, an application can be deployed and executed in an environment that is missing some required context information or actions. Hence, the application will not be able to work properly and user will not be able to know why this happened (Requirement 4).

Solar [5] distinguishes that applications typically need high-level context rather than raw sensor data and that high-level context information can be derived by aggregating data from one or more sensors. Hence, its aim is to make possible to offload this computation from the end-user application device into the middleware, running on one or more servers that host the Solar software. In the system, applications compose data flows, rather than interacting directly with context sources. They instruct Solar on which sensor to use and how the sensor data should be aggregated into desired context. Solar uses the filter-and-pipe software architectural pattern for data-stream oriented processing, which supports reuse and composition naturally. In the CAPA framework, instead of an application explicitly stating how to aggregate and compose context information, we allow creation of composite origins in a system which can be reused by many applications. Thus, applications developed for many heterogeneous, physical environments do not have to implement possibly many different ways of generating higher-level context information (Requirement 1). Furthermore, Solar lacks the support for context actions (Requirement 1) and requires processing schema to be hard-coded in the application logic, which prohibits the user configuration of context-aware applications (Requirement 4).

	Toolkit	JCAF	Solar	C-CAST	COP	CAPA
Req. 1	P	Y	P	P	Y	Y
Req. 2	Y	Y	Y	Y	Y	Y
Req. 3	N	P	Y	P	N	Y
Req. 4	P	P	N	Y	N	Y

Table 1: Comparison of Context-Aware Approaches

More recently, there has been research in developing a broker-based programming model for context-aware systems [6, 16, 17]. For example, the Context Casting (C-CAST) [16] project proposes a broker-based context-provisioning system that is supported by the publish-subscribe mechanism. Components in C-CAST take role of either a context provider or a context consumer. The task of the context broker is to hold registrations of all context providers and offer it as a directory service for context consumers. The broker-based approaches provide a right step towards reusable, context-aware applications by allowing them to discover context providers at run-time. However, all of the proposed approaches lack any support for actuators (Requirement 1). Furthermore, the broker-based approaches rely on the centralization of the knowledge of the whole system in the context broker. Hence, the context broker must mediate almost all communication between context consumers and producers, which inherently limits the ability of the system to scale with increasing number of applications (Requirement 3).

Context-oriented programming [14] (COP) has emerged in recent years as a complementary approach for supporting context-aware adaptations. The primary idea in COP is to provide programming abstractions that support context-adaptability in applications. It focuses on modularizing behavioral variations and supporting dynamic activation of these variations during the run-time. Behavioral variations is a unit of behavior that (partially) modifies the behavior of the application. It is enabled by means of a variation activation. Over the years, many different COP approaches were proposed [14, 7, 23] with majority of them based on a programming concept of layers [7]. They mainly focus on allowing self-adaptation in applications through usage of context information. Thus, they typically implement the context-aware behavior as an orthogonal concern that crosscuts main functionality of an application similarly to aspect-oriented programming (AOP). On the other hand, the CAPA framework focuses on supporting context-aware monitoring and controlling of physical environments. Thus, the COP approaches are more concerned on "internal" adaptations in software rather than "external" adaptations in physical world (Requirements 3 and 4).

Table 1 provides a comparison of aforementioned context-aware approaches with respect to requirements stated in Section 2. In the table, Y represents that an approach fully supports a requirement, P that it partially supports the requirement, and N that it does not support the requirement. Compared to all other approaches, the CAPA framework is the only one that fully implements all necessary requirements.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we presented the Entity-Adaptation programming model and the CAPA framework as an adequate solution for developing, deploying, and executing context-aware applications as generic and reusable components. We started with challenges associated with the development of context-aware applications and enumerated requirements for a context-aware solution. Based on these requirement, we first introduced the Entity-Adaptation program-

ming model. The programming model uses a concept of entities to define a specification that describes objects in an environment and allows adaptations to be implemented on top of these entities in an environment-agnostic manner. Later, we explained the design and implementation of the CAPA framework that provides support for deployment and execution of context-aware applications developed with the Entity-Adaptation programming model. We showed how the programming model and framework fully implement all necessary requirements. Finally, we presented a case study that demonstrates effectiveness of our solution in a real-world scenario. For future work, we will first investigate semantic reasoning for the binding algorithm to help us overcome the manual interventions. We will also examine how functional reactive programming can further ease the development of context-aware applications. Finally, we plan to integrate privacy and security concerns into the CAPA framework.

8. ACKNOWLEDGMENTS

This work is supported by Pacific Controls Cloud Computing Lab[6] (PCCCL) — a joint lab between Pacific Controls LLC, Sheikh Zayed Road, Dubai, United Arab Emirates and the Distributed Systems Group of the Vienna University of Technology.

9. REFERENCES

[1] G. A. Agha. ACTORS: A model of concurrent computation in distributed systems. Technical Report AITR-844, MIT Artificial Intelligence Laboratory, June 1985.

[2] M. Baldauf, S. Dustdar, and F. Rosenberg. A survey on context-aware systems. *Int. Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263–277, June 2007.

[3] J. E. Bardram. The java context awareness framework (JCAF) — a service infrastructure and programming framework for context-aware applications. In *3rd Int. Conf. on Pervasive Computing*, pages 98–115. Springer Verlag, 2005.

[4] P. Bellavista, A. Corradi, M. Fanelli, and L. Foschini. A survey of context data distribution for mobile ubiquitous systems. *ACM Computing Surveys*, 44(4):24:1–24:45, August 2012.

[5] G. Chen, M. Li, and D. Kotz. Data-centric middleware for context-aware pervasive computing. *Pervasive and Mobile Computing*, 4(2):216–253, April 2008.

[6] H. Chen, T. Finin, and A. Joshi. Semantic web in the context broker architecture. In *2nd IEEE Int. Conf. on Pervasive Computing and Communications (PerCom'04)*, pages 277–286, 2004.

[7] P. Costanza and R. Hirschfeld. Language constructs for context-oriented programming: An overview of contextl. In *Symp. on Dynamic languages*, pages 1–10. ACM Press, 2005.

[8] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unraveling the web services web: An introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2):86–93.

[9] A. K. Dey and G. D. Abowd. Towards a Better Understanding of Context and Context-Awareness. In *1st Int. Symp. on Handheld and Ubiquitous Computing*, pages 304–307. Springer-Verlag, 1999.

[10] A. K. Dey, G. D. Abowd, and D. Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16(2):97–166, December 2001.

[11] K. Henricksen, J. Indulska, T. McFadden, and S. Balasubramaniam. Middleware for distributed context-aware systems. In *Confederated Int. Conf. on On the Move to Meaningful Internet Systems*, pages 846–863. Springer Verlag, 2005.

[12] J. Henry C. Baker and C. Hewitt. The incremental garbage collection of processes. In *Symp. on Artificial Intelligence and Programming Languages*, pages 55–59. ACM Press, 1977.

[13] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *3rd Int. Joint Conf. on Artificial Intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[14] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125—151, March–April 2008.

[15] J. I. Hong and J. A. Landay. An architecture for privacy-sensitive ubiquitous computing. In *2nd Int. Conf. on Mobile Systems, Applications, and Services*, pages 177–189. ACM Press, 2004.

[16] M. Knappmeyer, N. Baker, S. Liaquat, and R. Tönjes. A context provisioning framework to support pervasive and ubiquitous applications. In *4th European Conf. on Smart Sensing and Context*, pages 93–106. Springer Verlag, 2009.

[17] F. Li, S. Sehic, and S. Dustdar. COPAL: An adaptive approach to context provisioning. In *6th Int. Conf. on Wireless and Mobile Computing, Networking and Communications*, pages 286–293. IEEE Computer Society, 2010.

[18] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 260–267, 1988.

[19] OASIS. Open building information exchange oBIX 1.0: Committee specification, 2006 December.

[20] M. Odersky and M. Zenger. Scalable component abstractions. In *20th Annual ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications*, pages 41–57, 2005.

[21] S. Ran. A model for web services discovery with qos. *ACM SIGecom Exchanges*, 4(1):1–10.

[22] G. Salvaneschi, C. Ghezzi, and M. Pradella. Context-oriented programming: A software engineering perspective. *Journal of Systems and Software*, 85(8):1801–1817, August 2012.

[23] G. Salvaneschi, C. Ghezzi, and M. Pradella. Contexterlang: Introducing context-oriented programming in the actor model. In *11th Annual Int. Conf. on Aspect-oriented Software Development*, pages 191–202. ACM Press, 2012.

[24] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 8(4):10–17, 2001.

[25] S. Sehic, F. Li, S. Nastic, and S. Dustdar. A programming model for context-aware applications in large-scale pervasive systems. In *8th IEEE Int. Conf. on Wireless and Mobile Computing, Networking and Communications*, pages 142–149, 2012.

[26] M. Weiser. The computer for the 21st century. *Scientific American*, 3(3):3–11, February 1991.

[6] http://pcccl.infosys.tuwien.ac.at/