EAI Endorsed Transactions
on Context-Aware Systems and Applications
Research Article ICST.ORG

A Framework for Developing Context-aware Systems

Vangalur Alagar[1,*], Mubarak Mohammad[1], Kaiyu Wan[2], Sofian Alsalman Hnaide[1]

[1]Concordia University, Montreal, Canada
[2]Xi'an Jiaotong Liverpool University, China

Abstract

Context-aware computing refers to a general class of mobile real-time reactive systems that continuously sense their physical environment, and adapt their behavior accordingly. Context-awareness is an essential inherent property of ubiquitous computing or pervasive computing systems. Such systems are much richer and more complex than many software systems. The richness mainly comes from context-awareness, the heterogeneity of mobile devices and their service types, while complexity arises in the multitude of adaptations enforced by system policies. In order to provide the true intended effect on the application of services without compromising on the richness, the complexity must be tamed. Towards this goal, this paper proposes a component-based architecture for a Context-aware Framework in which context, awareness capabilities, reactions, and adaptations are formally dealt with. Two formal languages are designed to express context situations and express workflow policies, which respectively lead to a context reasoner and to enforce adaptations. With these formalisms and a component design that can be formalized, this work fulfills a formal approach to construct context-aware applications. Two case studies are explained, of which one is a proof-of-concept case study from service-oriented domain. It is fully implemented to illustrate the expressiveness of the framework design and robustness of its implementation.

Keywords: Context-awareness, Context Reasoning, Adaptation, Framework,Component-based Architecture

Received on 03 February 2013, accepted on 03 April 2014, published on 05 September 2014

doi: 10.4108/csa.1.1.e2

1. Introduction

Computers are increasingly used in a variety of applications that have impact in everyday life of humans, either with or without their explicit knowledge or willing participation. In this computing model, users interact either independently or collectively, and either intentionally or unintentionally, with many different devices to bring about changes or provide services to subjects in the environment reachable by the devices. Both objects and subjects are subject to mobility and sometimes invisibility in this model. Most importantly, interactions need not be initiated by subjects, instead devices go after the subjects to trigger reactions and adaptations. Systems built on such computing models have come to be known as 'context-aware'. This term was first coined in 1994 by Schilit and Theimer [1] to describe the 'ability of a user's applications to discover and react to changes in the environment they are situated in'.

Context-aware systems are rich, notoriously heterogeneous and complex. Consequently, in order to be efficient they require suitable architectural models and computing environments. Richness arises from context-awareness property. Heterogeneity results from the variety of sensory devices used to perceive the environment of concern, diversity of context information, and the multiplicity of actuators used to adapt to environmental situations. One aspect of complexity is a consequence of richness. Context construction and interpretation of interactions within contexts are to be done dynamically as well. Another aspect of complexity is due to heterogeneity. The diversity of relations and connections between devices may result in concurrency. The dynamicity of the environment where contextual changes might demand the addition of some new devices and discarding some old ones will require architectural adaptation. Finally, policies that enforce adaptation have to be uncovered from the application domain properties and institutional laws, and their application should result in correct functioning of the actuators. Towards taming the complexity and retaining the richness in context-aware computing models, we propose a *component-based architecture for a context-aware framework* in which context, awareness capabilities, reactions, interactions, and adaptations are formally dealt with. The framework is sufficiently general to meet the development requirements of applications from different domains, such as health care, transportation, defense, and social networks.

1.1. Motivation of the Current Work

During the last ten years, research in the development of context-aware applications has progressed immensely. In order that the prime focus of our current work is well understood, we need to place our work in the right category of context-awareness literature. So, below we categorize

*Corresponding author. Email: alagar@cse.concordia.ca

the work done under the broad term "context-awareness". In Section 6 we give a detailed comparison of our work with related relevant work and bring out the merits of the framework introduced in this paper.

- *Mobile Applications:* Context-awareness is an essential requirement for mobile applications. Several researchers have offered different kinds of solutions for adapting mobile systems for specific applications. These include Biegel et al. [2], Korpiaa et al. [3], Kramer et al. [4] et al., Lovett et al. [5], and van Setten et al. [6]. We emphasize that they are application-specific approaches.

- *Secure-Critical Systems:* Adding context-awareness to safety and secure-critical systems make them more adaptive. Context-awareness is central to access controls for crisis management [7], authentication [8], and health care delivery [9], [10]. For Cyber-physical system context-aware security solutions have been proposed by Wan et al. [11].

- *Pervasive Computing:* Pervasive computing and ubiquitous applications need to be aware of their environment in order to adapt to changing contexts and provide correct services. Programming environments and context-aware pervasive and ubiquitous computing applications are discussed in [12], [13], [14], [15], and [16].

- *Middleware:* A middleware introduces a layer of abstraction, separating the application (business) layer from lower-level system layer. As such a middleware is not an application development environment. Some of the middlewares proposed for mobile applications are [17], [18], [19], [20] and , [21]. Their application domains and service types are different. A middleware to support context-aware service-oriented systems is discussed in [22].

- *Architecture and Framework:* Framework is a generic infrastructure which simplifies the development of context-aware application. Architecture refers to the design blocks that constructs the infrastructure. A programming framework for service-oriented architecture is discussed in [23]. A conceptual framework for rapid prototyping of context-aware applications is given in [24]. Examples of application domain-specific architectures are [25] and [26]. The two survey papers [27], and [28] discuss middleware, architectures, and frameworks. In the former, architecture principles for context-aware systems, middleware and framework are suggested. In the later the comparison of architectures is based on certain criteria related to pervasive computing.

- *General:* A multitude of papers have introduced context-dependence and context-awareness in many disciplines, such as AI [29], [30] [31], service management in transportation domain [32], [33], web design [34], and agricultural farming [35].

The above categorization is by no means exhaustive. It is given to help us understand where the current work is placed, so that comparison done in Section 6 can be restricted to the literature in this category. Our work described in this paper falls in the category *Architectures and Framework*. The major difference between our work and those referenced above is that we have developed a new framework based on rigorous software engineering principles supported by both theory and a practical robust implementation. Based on the experimental studies reported in Section 5, we claim that our framework is generic and robust enough to support the development of context-aware applications from many different application domains. Based upon the details given in Section 3 we also claim that (1) a projection of our framework architecture provides a middleware for any application and (2) any application-specific middleware can be developed in our framework.

1.2. A Rationale for Our Development Approach

Ten years ago, it was hard to capture the full import of context in context-aware models due to technology constraints. However, the rapid increase in computer power, which is realized in ubiquitous spectrum of high-connectivity, hand-held and light-weight devices, has allowed computers to have a greater awareness to user's context. Therefore, current context-aware applications are expected to interact with different types of sensors in real time, analyze and validate the sensor information, associate it with implicitly perceived user contexts, and reason about situations and policies for a seamless adaptation of it in providing services in different contexts. So, we made the decision to introduce sensor types, which allows introducing many sensors of each type in the system. Since sensors gather heterogeneous information we decided to provide translators which will transform sensor information into a uniform structure. These are explained in Section 3.1.

Contexts that are relevant to an application must be determined by domain experts, independent of how they will be perceived and represented. Proper semantic information from the application domain must be used in defining relevant contexts. Because contexts exist on their own, both in design and implementation they are to be treated as *first class citizens*. This decision requires to model contexts as data types, with representational and operational abstractions. These are formally discussed in Section 2. This formalism supports the context mechanism architectural element in Section 3.

In context-aware system development we need a technique to explicitly define user intentions and a mechanism to infer them based on context information. We formalize user intentions as *situations*, expressed as a well-formed

expression in a context-free language and provide a method to verify it in any given context. This approach, as opposed to a Logic program approach [36], has the primary advantage for macro creation and its plug-in at execution tome. Because context-awareness is a dynamic feature we must rely on methods that are formal for design and efficient during execution. The situation expression language and evaluation, discussed in Section 2, is motivated by this need.

Adapting to dynamically changing context is as challenging as perceiving context. Adaptation requires accurately mapping predefined actions to specific context situations. Predefined actions are usually implemented using actuators. Actuators represent the parts of a computing system which perform actions at the last stage. Actuators can be human beings, or software-based components, such as database transactions, or hardware devices, such as door controllers. Actuators are heterogeneous and only at execution time their availability may be known. When the resources are limited, an actuator availability may not be known until run-time. So, we have introduced actuator types and a mechanism to map a reaction to an actuator of a specific type at run time. Context-aware applications also require a standard mechanism to interact with actuators. Adaptation mechanism is an element of the architecture discussed in Section 3.

In between perceiving context and adapting to it there is a whole process that should be governed with *business rules* and *quality of service policies* that are imposed by the application domains. As examples, in defense applications policies governing security and safety must be fulfilled without fail, and in health-care applications policies governing privacy, safety, and service availability are to be enforced. We insist that no two policies contradict in a given context, although non-determinism is allowed. This is an important requirement for building safety-critical applications, such as health care, defense, and emergency rescue. In a non-deterministic choice any one of several related policies may be chosen in a context. However, in case of contradictory policies it is impossible to resolve a contradiction by automatic analysis during a service execution. In case contradiction of policies is detected in a context at run time, we will require a *supervisory component* to intervene and set a policy to resolve the contradiction. Context-aware applications need an extendable mechanism to define and enforce institutional policies that govern application's behavior to ensure trustworthiness. The representation of policies and the interaction with context information and other application resources are interesting and challenging issues. Policies can be represented in a Logic program style [36], [11]. However, for the sake of efficiency we code a policy in a high-level language, called *Workflow* language. The primary advantages are that policy representation in the language is formal, the semantics of the language structures evaluate every well-formed expression without fail, and the evaluation process is efficient. We discuss a context-free language for expressing policies and discuss their evaluations in Section 2. Adaptation mechanism of the architecture is built on this formalism.

Section 5 discusses the full implementation. To sum up, our framework is bundled up with methods for solving the following issues:

1. Management of sensors to acquire context knowledge from user's environment, and representation and management of context information.

2. Definition of semantic information and rules to infer situations based on context information.

3. Definition of adaptations to contextual situations, and management of policies to regulate adaptations.

4. Management of actuators to perform adaptations in user's environment.

1.3. Contribution

The following is a list of major contributions of our work.

- Formal representation of context situation and a context reasoner that is able to parse and evaluate context situation expressions against context information, as discussed in Section 2.

- Formal definition of adaptations and policies, and extendable Adaptation Workflow Language, implementation of workflow executor engine to parse and execute workflow definitions and enforce defined policies, as discussed in Section 2.

- A component-based architecture for Context-aware Framework, as discussed in Section 3.

- A generic mechanism for the system to interact with both Sensors and Actuators, and a platform-independent implementation of the framework running on different platforms (Phone, Desktop, Web), as discussed in Section 5.

- Two case studies discussed in Section 4, of which one combines context-awareness with mobile devices in optimal service provisioning.

An extensive discussion on related work is given in Section 6. It is clear from this discussion that although some of these aspects have been addressed by researchers and software industry during the last two decades, there does not seem to exist a single rigorous approach to context-aware system development in which all the above aspects have been fully addressed. This is the rationale for the current work. The framework helps software developers in any application domain to empower existing and new applications with context-awareness and adaptation management capabilities.

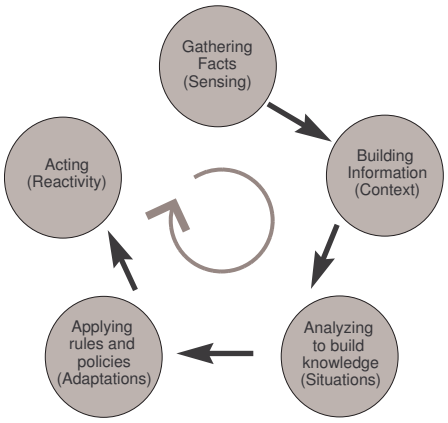

Figure 1. Context-aware systems' execution model

2. Facts, Contexts, Situations, Policies, and Adaptations

Figure 1 depicts our view of the execution model of context-aware systems. It comprises the following sequential operations: 1) Sensing: gather data, facts, from a plurality of sensory devices deployed in an environment, 2) Building Information: extract and combine relevant data to construct current contexts. The construction is based on predefined context definitions, 3) Building Knowledge/Awareness: analyze the resulted contexts to construct current situations. The construction is based on predefined situation definitions. 4) Adapting: deduce valid adaptations based on current situations. And finally 5) reacting: act upon the awareness by executing actions according to adaptation specifications. During system execution, information is transformed from one form to another. In order to implement context-aware systems, there is a need to provide a formal model that describes the syntax and semantics of each form of information which is used at each execution stage. This includes *context*, *situation awareness*, and *adaptation*. This section provides a formal model for context, situation, and adaptation. In this section, we will use the formal definition of context as described by [37]. Then, we will introduce novel formal definitions for situation and adaptation.

2.1. Context

According to the Oxford English Dictionary, context denotes "the circumstances that form the setting for an event". A circumstance is a condition involving, in general, different types of entities. As an example, the setting for a "seminar event" is a condition involving entities *speaker*, *topic*, *time*, *location*. When each entity is assigned a value from the domain associated with that entity, and if the condition is met then the seminar is to be held. A condition involving n entities needs a n-tuple of values for a total evaluation. In general, many different n-tuples may satisfy a condition with n entities. So, we can regard the collection of n-tuples satisfying the condition as a n-ary relation. This is the rationale used by Wan [37] for formally defining context as a *relation*. We call the entities as *dimensions* and associate with each dimension a type. For example, the type of values assigned to *speaker* is $NAME$, whose elements are of type $string$. Therefore, context is a typed relation. This formal notation closely follows the principles behind the original work of McCarthy [38] and the several formal notations compared by Akman [39]. Other definitions which have been proposed by Dey [24], Winograd [40], Wrona [41], and Bettini et al [42] can be used by a developed in our framework. The translators that we have provided will transform such constructs to the formal notation that we use. The significance of our notation becomes clear when we discuss situations in Section 2.2.

It is necessary to fix the set of dimensions and the domain of values for each dimension before constructing contexts. The following syntax definition of context was introduced by [37]. Let $DIM = \{D_1, D_2, \ldots, D_n\}$ denote a finite set of dimensions, and X_i be the type associated with $D_i \in DIM$. The concrete syntax for context is $[D_{i_1} : x_{i_1}, \ldots, D_{i_n} : x_{i_n}]$, where $\{D_{i_1}, \ldots, D_{i_n}\} \subset DIM$, and $x_{i_k} \in X_{i_k}$. Not all dimensions in DIM need to occur in a context, however every dimension used in constructing the context should be a member of DIM.

An important issue is the choice of dimensions. From the perspective of a design and its implementation of a specific application, the set of dimensions that are relevant for the application are to be determined by domain experts. The set of dimensions and the domain of values for each dimension determine context types and hence they should also be fixed before constructing contexts. The dimensions that are most common in ubiquitous computing are (1) *WHO* (to perceive service requests), (2) *WHAT* (to denote the type of service), (3) *HOW* (the service needs to be provided), (4) *WHERE* (to provide the service), (5) *WHEN* (to provide the service), and (6) *WHY* (purpose of request). For each dimension, the domain of values are suggested in a natural manner. For instance, for the dimension *WHY* we can associate the domain of values $\{\text{clinical}, textresearch\}$ for providing hospital services. The dimensions, as suggested above, are neither selective nor exhaustive. The system designer should feel free to choose as many dimension names as are necessary.

The dimensions in a context need not be different, according to this formalism. If the dimensions in a context are all different then it is called *simple context*. A context that is not simple is equivalent to a set of simple contexts. As an example, a simple context of type *SeminarContext* is $c = [SPEAKER : Milner, TOPIC : Bigraphs, LOCATION : EV3.222, TIME : (20/March/1991)]$. If in room $EV3.222$, video conferencing facility is available and the seminar can be viewed and heard in rooms $EV2.111$ and $EV1.222$, then the context $c' = [SPEAKER : Milner, TOPIC : Bigraphs, LOCATION : EV3.222, LOCATION : EV1.222, LOCATION : EV : 2.111, TIME : (20/March/1991)]$ is a non-simple context of type *SeminarContext* and is equivalent to the set

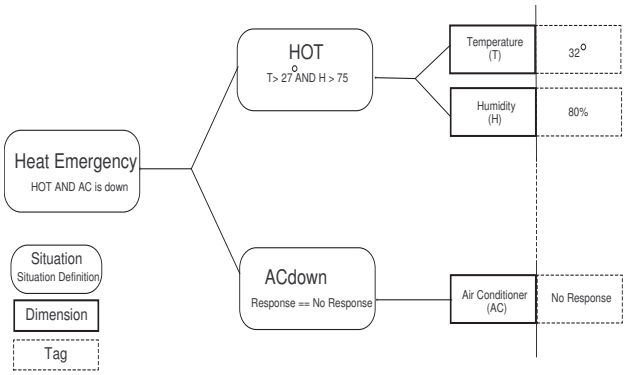

Figure 2. Context Situation

$\{[SPEAKER : Milner, TOPIC : Bigraphs, LOCATION : EV3.222, TIME : (20/March/1991)], [SPEAKER : Milner, TOPIC : Bigraphs, LOCATION : EV1.222, TIME : (20/March/1991)], [SPEAKER : Milner, TOPIC : Bigraphs, LOCATION : EV2.111, TIME(20/March/1991)]\}$ of simple contexts.

2.2. Situation

A *situation* expresses the *knowledge* perceived by an information system according to a specific context. It defines specific semantics to a context constructed by the system. For example, a user context that contains the GPS coordinates of the user location alone may not provide knowledge that is useful. However, providing some meaning to this context information, say by identifying that *the user is at home* or *at work*, will probably be more significant for a meaningful decision making. In this case, the situation *the user is at home* occurs if the current GPS coordinates match the GPS coordinates of user's home.

Formally, we define a situation as an expression that includes a predicate defined over the values of available contexts and situations. This paper introduces a context-free language for defining situation expressions. The language includes extension points which are user-defined functions, higher order predicates. We call this language CSEL, *context situation expression language*. The formal syntax of the grammar is given in the Appendix. The valid strings in the language are situations. The language supports logical, arithmetic, and comparison operators.

The situation expression language provides extension points, user defined functions, to bring user supplied logic to the reasoning operation. Functions should have one or more parameters which are either dimension values (tags) or user supplied values. Syntactically, a function should start with a $ sign. System developers should provide the implementation of the functions in Dynamic Link Libraries (DLL) that should be deployed in a special directory. The system is able to allocate their implementations at the run time, without recompilation. An example of user defined function is $EvenNumber = \{(\textbf{\$EvenNumber}[\underline{\text{Dimension Name}}])\}$

Figure 2 shows three simple contexts, each having one tag-value, and three simple situations. The situation *Hot* is realized if the following two logical conditions are true: 1) in the *Temperature* context the tag value is greater than 27, and 2) in the *Humidity* context the tag value is greater than 75. Thus, the Hot situation depends on the $Temperature$ and $Humidity$ contexts. Thus, it is defined using the expression $\{(T > 27)\ AND\ (H > 75)\}$. Informally, it is easy to check that in the context $[T : 32, H : 80]$, the union of the two atomic contexts shown in Figure 2, the expression $(T > 27)\ AND\ (H > 75)$ is true, and hence the situation *Hot* is realized. Moreover, a heat emergency situation is a conjunction of *ACdown* and *Hot* situations. This means that *Heat Emergency* is a compound situation which can not happen unless it is both hot and AC is down. In general, we define situations as either atomic or compound. The following examples show situation expressions using our CSEL language:

- *Simple Situation:* $HotWeather = \{(\underline{\text{Temperature}} > 30)\}$, where $Temperature$ is a context tag.
- *Situation Token:* $GoOut = \{NOT\ StayHome\}$, where $StayHome$ is a situation expression.
- *Compound Situation:* $NiceWeather = \{Warm\ AND\ Sunny\}; \{NiceWeather\ IMPLIES\ GoOut\}$, where $Warm$, and $Sunny$ are situation expressions.
- *Literal Situation:* $\{(\underline{\text{Role}} == 'Admin')\}$, where *Role* is a context tag and $'Admin'$ is a tag-value.

2.3. Policies

Context-aware systems operate directly in its surrounding environment. Adaptations may affect directly users and their environment. Therefore, it is important to ensure the safety and security of adaptations. In order to ensure predictability and trustworthiness of system adaptations, there is a need to define policies. *Policies* are business and quality assurance rules that restrict and control the behavior of a system. In our proposed architecture, we distinguish two types of policies:

- *Data policies*: They are rules that constrain data values in contexts. For example, a reading of a temperature sensor may have a data policy stating that the temperature should be between -40 and 50 degrees because this is the sensor output range. This means that any other value is an error value and will not cause an adaptation. It is crucial to detect errors at an early stage so the system can ignore bad data instead of carrying unnecessary operations. Another example is context's *time-span validity*. Context information can be valid only for a specific time-span. As an example, the GPS coordinates in a navigation system may be valid only for a few seconds. Some other context information,

such as the date context information in any application, may be valid for 24 hours. More complex policies may exist in applications.

- *Execution policies*: They are related to adaptations. These policies control the behavior of the system when it responds to a change in the context of an application. These policies contribute to selecting the proper reactions that should take place, change the sequence of actions, and enable or disable reactions. For example, an application could check user role to implement different adaptations based on different user authorizations. Another example is when adaptations depend on resources, such as Internet connection which is not always available. Execution policies could be used to check if resources, required for a certain adaptation, are available before execution.

2.4. Adaptation

Context-aware systems are adaptive systems. They sense their surrounding context and adapt to contextual changes. An adaptation is a set of *reactions* that have two properties: 1) they take place in response to a change in situation, and 2) they may affect the surrounding environment. A *reaction* represents an atomic action, which could not be split any further. A reaction is defined to perform an action through an *actuator* deployed in the surrounding environment. The actuator may have a specific system configuration. Therefore a context-aware system should define adaptations and associate reactions to each adaptation.

Context situations are diverse. Consequently, adaptations could be simple or complex. A simple adaptation consists of one reaction. A complex adaptation may require a set of actions either in specific sequence or in parallel. For example, an application could respond to a security threat situation with the following set of actions: (1) setting the fire alarm, (2) closing the exits for critical areas and (3) calling the emergency. Also, complex adaptations may require repeating reactions or controlling them using conditions. Thus, complex adaptations require a workflow expression language. Workflow expressions should support sequencing, repetition, and conditioning on sets of reactions.

We have designed a context-free language, called *Adaptation Workflow and Policy Expression Language* (AWPEL) to express adaptations and domain policies. The language is extensible. It allows system developers to import external functions that execute adaptations that are specific to a domain's security and safety. The grammar of the language is shown in the Appendix. The language supports the following operations.

- Triggering reactions,
- Checking policies,
- Performing policy compositions using logical operations (*AND*, *OR*, *NOT*), and

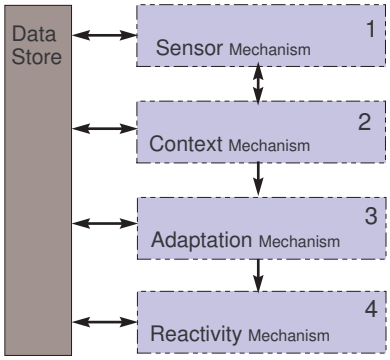

Figure 3. Architecture of Context-Aware Systems

- Constructing rich workflow expressions using the constructs (*WHILE*, *IF ELSE*, and *FOR*,*EXECUTE*).

The two special statements in the language are: *policy check* and *execute*. The former checks execution policies in the chosen adaptation workflow expression. Each policy check statement is a call to a user defined function that accepts zero or more parameters. This function returns a Boolean value to indicate whether the policy has evaluated to true or false. An example of policy check statement is $*IsAuthorized*[$UserID$]. The *Execute Statement* is used for triggering reactions. Reactions are user defined functions, and they are identified by their names. Each reaction may have zero or more parameters that are either user supplied value or dimension context information. Reaction name is a terminal rule that matches a string Identifier.

3. Architecture

This section presents the architecture of our context-aware framework. First, we introduce a high level, component view, of our architecture. Then, details about all components and their interactions will be discussed. The architecture proposed by us not only emphasizes both formalism and components, but lifts the architecture to new heights, in which heterogeneity, distributed and mobile nature of environment, and dynamic application of adaptation policies are seamlessly integrated.

Figure 3 shows the main components of our architecture for context-aware framework. Each component models and implements a mechanism as briefly described bellow:

- Sensor mechanism component is responsible for monitoring the environmental entities and sensing any changes to their *parameters*, dimensions that are of interest to the system. The parameters are scalar or structured data values such as temperature, geographical position, and identity of a person.
- Context mechanism component is responsible for context construction, context modification, and context reasoning. It relates the data received from the sensor

mechanism to events of significance to build awareness. This in turn will assist the system to perform the appropriate adaptation.

- Adaptation mechanism component is responsible for analyzing the collected knowledge about the environment and triggering the appropriate reactions. In the proposed architecture, the analysis is based on predefined rules and policies, which are stored in a shared database. The resulting reactions are regulated by policies.
- Reactivity mechanism component is responsible for performing adaptations in the environment by controlling physical devices, actuators, or displaying results on hardware interfaces.
- Data Store component is responsible for hosting and managing all collected data, context, situations, adaptations, policies, reactivities along with all system configurations.

Figure 4 shows the details of our proposed architecture. A detailed discussion of the architectural elements is provided in the following sections.

3.1. Sensor Mechanism (SM)

The sensor mechanism, illustrated in Figure 4, comprises *sensor*, *connector*, *listener*, *translator*, *verifier*, and *data synchronizer* elements. The main advantage of our proposed architecture is the abstraction and loose coupling between the different elements of the sensor mechanism. This gives flexibility in design and implementation, and increases the maintainability and scalability of the system.

An entity in an environment can behave as an event source. A stimulus is an instantaneous event, fired by an entity, that triggers the system processing. There are two ways to create a stimulus: the occurrence of an external event and a change to a parameter in the environment. For example, the identity of the person is a data parameter associated with the event of entering the room. A parameter is modeled as a *dimension*, associated with a type. A stimulus may be associated with one or more dimensions. Also, the dimension can be carried by one or more stimuli. For every stimulus there is a sensor that detects the occurrence of the event and collects its dimensions. The sensor mechanism contains both hardware and software components. In this approach, we consider a sensor as a black-box architectural unit, such as image recognition unit or a smart card reading device. A sensor can be associated with an environmental dimension to detect any change to its value. For example, a measuring unit can be used to detect the current quality of air in a room. When the value changes, the sensor triggers a stimulus and associates the value as a data parameter to it. Therefore, sensors are data providers. There are many different types of sensors such as physical, chemical, mechanical, biological, or software-based sensors. Sensors share common characteristics such as input range, output range, and accuracy. A *sensor type* is characterized by a set of attributes and a measurement reading data type that represents the language spoken by sensors of a certain type. A *connector* transmits data between a sensor and a *listener*. Connectors may use different communication methods and protocols to transmit data such as serial ports, TCP/IP and network protocols, Bluetooth, wireless, etc. A *translator* is responsible for translating data from one data type to another understood by the system. A *verifier* is responsible for verifying the correctness of the collected data according to predefined *data policies*. A data policy is a logical expression that is used to examine the correctness of the data value received from a sensor. A listener is responsible for managing one sensor's communications. It uses a connector and a translator to communicate with its corresponding sensor.

While the components of the sensor mechanism are shown as being part of the same module, it is to be understood that one or more of these components may be located at different places especially when the connector includes a wireless connection such as Wi-Fi, Bluetooth, or the like. For example, the sensor may be provided at the door while the sensor listener and the remaining components are in the server room.

The data collected by the different sensors is received at a data aggregating module called *data synchronizer*. The data synchronizer is used to update the fact information with the latest sensor readings obtained from sensors. The data synchronizer may also have access to a set of rules and definitions stored in the data store to apply these rules and definitions to the data received from the different sensors prior to outputting this data to the following components of the context mechanism. In a non-limiting example of implementation, the data synchronizer may apply a set of rules that enable it to choose the highest possible fact information when it receives contradicting fact information from different sensor modules. For example, the data synchronizer may give higher priority/value to certain sensor types over others. For example, consider a scenario where different sensors are implemented in a building to identify the location of each individual within the building. Assuming that the different sensor types include biometric sensors which identify a person based on the voice, fingerprint, retina etc. and electronic detectors which detect the identity of the person based on an access code, card reader or the like. If the biometric sensors indicates that a certain person is in room A because his voice has been or is being detected in that room, and the electronic detectors indicate that the same person is in room B because his access code or magnetic card has been used in that room, then priority is given to the biometric data because it is possible to use someone else's card or access code but it is not possible to use their voice or fingerprint. In this case, the data synchronizer may output the location as being in room A. It is to be noted that the set of rules may be set, adjusted, and changed by the user as the needs dictate. For example, the same scenario may lead to different results if the

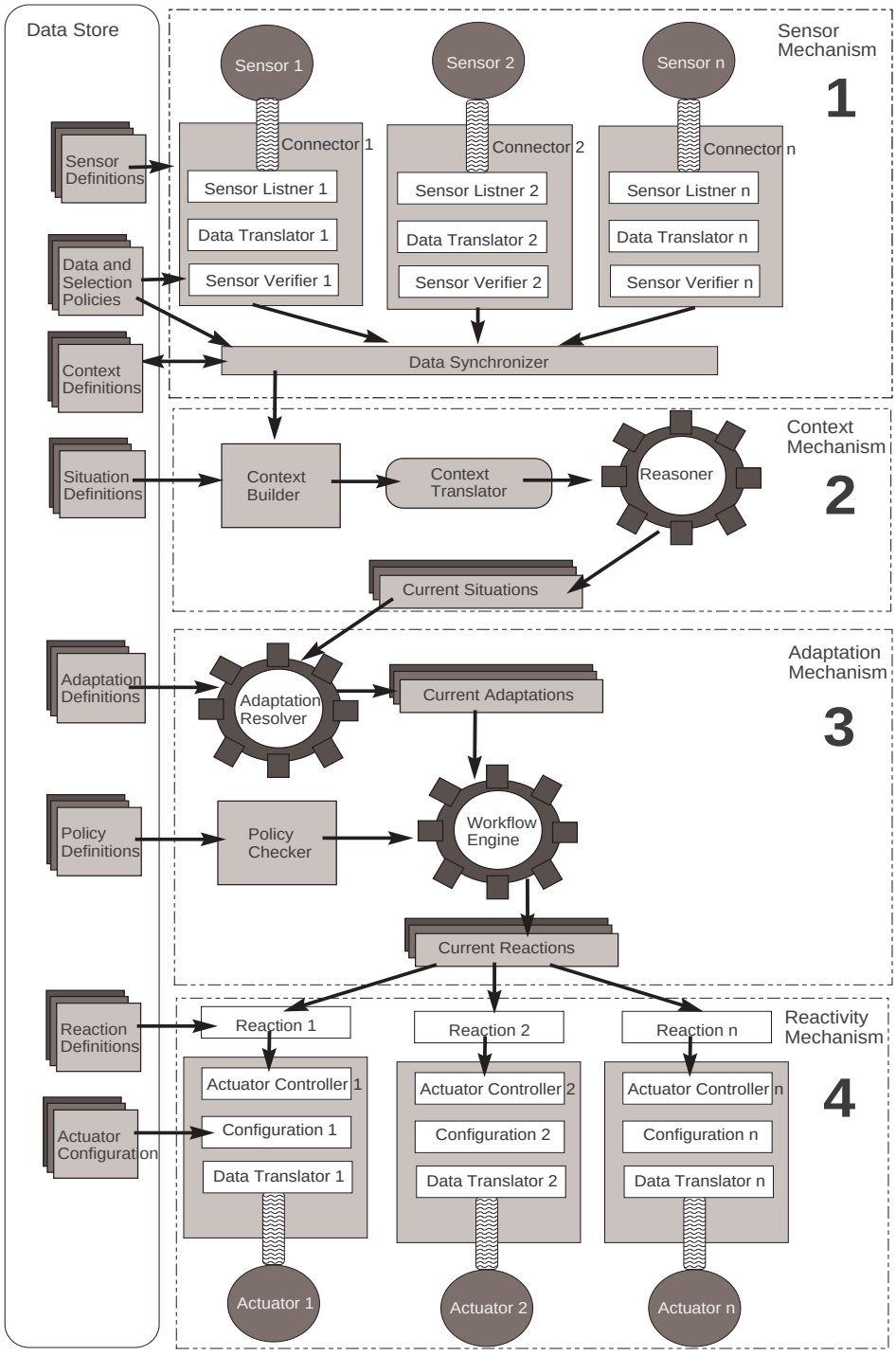

Figure 4. Detailed View of The Architecture

set of rules accounts for the recency of the information. For example, if the data output by the biometric sensors indicates that the person is in room A and the time stamp of that data is 12.54 PM, while the data output by the electronic sensors indicate that the same person is in room B and the data has a time stamp of 1.05PM then, even if the biometric sensors have the higher priority over electronic sensors, the recency of the information from the electronic sensors may provide for more accurate results because it would have been possible for the person to have left room A and entered room B.

In summary, the main features of our sensor mechanism architecture are:

- Communicating with sensors using different connection and communication methods.

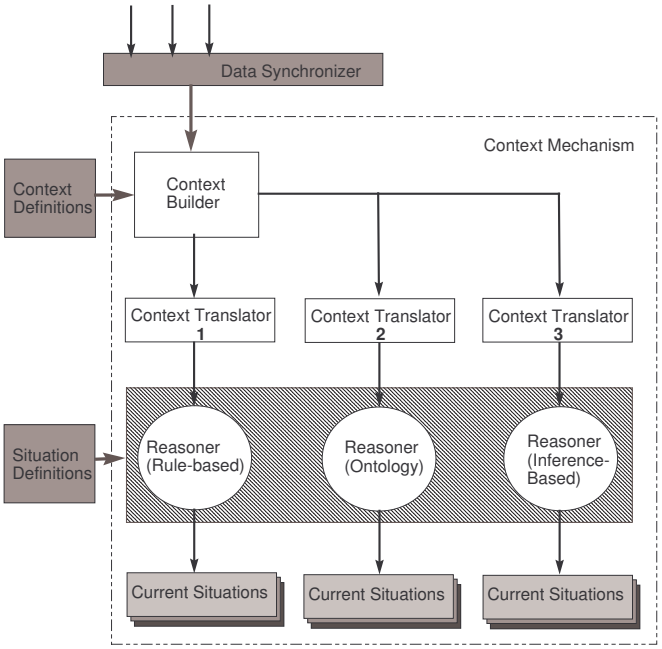

Figure 5. Detailed View of The Context Mechanism

- Managing the translation of sensor outputs using independent architectural elements that could be physically and logically independent from sensors. This relieves the system implementation from knowing details of sensors. Also, it allows the system to connect to simple hardware sensors that produce row data without having to manage its translation.
- Requesting data on demand from sensors and informing any interested entity when new readings are submitted by a sensor. Therefore, the relationship between sensors mechanism and context mechanism is bidirectional.
- Verifying and managing sensor data based on data policies. This enables the system to ignore false alarms, avoid reactions to invalid stimuli, and react only to desired requests. Hence, it ensures the resources of the system are utilized only when required.
- Aggregating sensor data from different sources using predefined rules. This enables choosing different values collected by different sensors at different scenarios, as illustrated in the example above. The selected data are cached and used upon request.

3.2. Context Mechanism

The context mechanism, illustrated in Figure 5, consists of the following architectural elements: *context definitions*, *context builder*, one or more *context translators*, one or more *reasoners*, and *situation definitions*. In Section 2.1 we provided a formal definition of context. It comprises *dimensions* and *tag* values. Since the environment constantly changes, the context builder constructs contexts every time when there is a change to any of the dimensions' tag values. Information about new changes comes from the sensor mechanism through the data synchronizer. Therefore, continuously, the sensor mechanism triggers the context builder to construct contexts. A *reasoner* is used to identify the *situations* that are applicable to the currently constructed contexts. A reasoner uses situation expressions, as defined in Section 2.2, that are stored in the data store. A reasoner evaluates each expression based on the current context. Consequently, the reasoner generates a set of situations that hold in a current context.

There are several context definitions and theories provided in the literature, other than the one we provided in Section 2.1. One example is the use of ontology, utilizing description logic language, for defining context. Other examples of other context definitions and theories will be provided in the related work, Section 6. Therefore, it is important that a context-aware system architecture can work with different formal context definitions. In this case, different types of reasoners might be more suitable based on the type of context definition. For example, in case an ontology-based formalism is used to define context, an ontology-based reasoner should be used to infer situations for a given context. For the context definition that we provided in Section 2.1, a rule-based reasoner is more suitable. Thus, it is possible using our architecture to use a plurality of reasoner engines of different (or similar) types. Figure 5 illustrates an example of a context mechanism including three different types of reasoners: rule-based, ontology, and inference-based. A *context translator* is used for each context theory to translate contexts from its native formats to a format that can be understood by a corresponding reasoner. Therefore, the operations of the context builder are independent from the way context is specified or processed. Also, reasoners may work independently, separately (in a separation of concern manner) and simultaneously using at least a portion of the context generated by the context builder. This is a novel, and exclusive feature of our architecture that no other context-aware system in the literature supports which provides significant improvement over previously known architectures.

It is worth noting that the set of rules and definitions of contexts and situations may be set, changed, and adjusted by system administrators, whereby, sensors, reasoners, and actuators may be added and removed in a plug and play manner. Therefore, the context-aware system build using our architecture may be used in a variety of different and unrelated domains.

In our implementation, a situation expression is internally represented as an *Abstract Syntax Tree*, and evaluated in the usual operator precedence semantics. In order to check if a specific situation is realized or not, the reasoning mechanism takes contexts and their dependent situations as input and gives as output the situations that are realized. Internally,

the reasoning engine extracts relevant context tag values and checks situation conditions against context tag values.

3.3. Adaptation Mechanism

The adaptation mechanism, illustrated in Figure 4, includes: an *adaptation resolver*, a *workflow engine*, *policy definitions*, a *policy checker*, and *adaptation definitions* elements. This unit is responsible for determining suitable reactions for context situations. The chosen reactions are sent to the reaction mechanism.

The current situations are sent to the adaptation mechanism for generating reactions for the situations identified by the context mechanism. Associations between situations and adaptations are defined and managed in the data store using adaptation definitions. A situation can be associated with one or more possible adaptations. The current situations are received at the *adaptation resolver*. The adaptation resolver includes a reasoning engine capable of producing adaptations. It takes as input adaptation definitions and current situations. An adaptation definition is specified as a workflow expression of actions and policies that depend on a set of situations. It is defined using WPEL, as discussed in Section 2.4. A WPEL expression defines a set of reactions to be produced and the order in which the reactions should be executed. Also, an adaptation expression defines execution policies to control reactions. The selected adaptations are sent to a *workflow engine*. The workflow engine is used to parse and execute workflow expressions. It contains implementations for every construct in the workflow expression language (WPEL). It takes as input: 1) situations, 2) adaptation definitions, and 3) execution policies. It uses the *policy checker* to evaluate policies and control execution of reactions. The policy checker takes as input a situation and a policy condition. It evaluates the condition using the context information available in the situation definition. The result controls whether or not an action should be triggered.

3.4. Reactivity Mechanism

The workflow engine manages and orchestrates the resulting reactions that should take place in the environment. Reactions have the following properties:

- *Independence:* A reaction should be executed with no dependency on any other reaction.
- *Atomicity:* A reaction is atomic, and should perform one and only one functionality.
- *Communication:* A reaction communicates with the outside world actors, namely the actuators.
- *Context dependence:* A reaction may have execution parameters which depend on context information. These parameters are passed to actuators. For example, if an action aims to display a message on a screen then the message should be passed from the reaction to the screen actuator.

The reactivity mechanism, illustrated in Figure 4, consists of *actuator controller*, *actuator configuration*, *translator*, *connector*, and *actuator*. Once reactions are decided, their corresponding actuators are determined. Associations between reactions and actuators are defined and managed in the data store. It is possible, using a configuration file, to associate multiple actuators with each reaction. For each actuator, an *actuator controller* is defined. It provides a level of abstraction between the system and actuators. An actuator controller and its corresponding actuator do not have to be in the same physical component or location. It is possible to have an actuator controller in one room and its corresponding actuator at a different or perhaps very far location. Each controller is implemented for a specific actuator. A controller has sufficient knowledge to communicate to its corresponding actuator. The *actuator configuration* is used to specify and manage any necessary configuration for an actuator. Hence, Configurations are abstracted from controllers. This allows using the same actuator to implement different workflow actions based on different configurations. For example, the door actuator can perform open, close, lock and unlock actions through different configurations. A *connector* is used to transmit an adaptation reaction and its relevant context information to actuators. It implements a connection method and a communication protocol. A *translator* is used to translate the command and its information into a format suitable for actuators. A translator is implemented for each type of actuators. Actuators perform the actions to affect the environment. There are many different types of actuators. An actuator type is characterized by a set of attributes and data parameters that represent the required input information necessary for performing actions.

4. Case Studies

In this section we discuss two case studies. The first case study is to explain the steps involved in using the framework to develop a context-aware application. A more complex example discussed in the second case study is on a distributed mobile application for improving the productivity of a traveling salesperson. We have implemented both examples. In order to develop any application using the framework, the main entities that represents the application environment and their intentions at the outset. So the client application would have to define sensor listeners, sensor translators (if applicable), sensor verifiers, actuator controllers, context information, situations and extenders (if applicable), adaptation, reactions, and policies (if applicable).

4.1. Case Study I: Temperature Control System

The system functionality is to increase or decrease or maintain the temperature by sensing the environment parameters. The system architecture, illustrated in Figure 6, contains a thermometer sensor which provides the

temperature, a heater actuator and a cooler actuator. The framework elements for this applications are defined below.

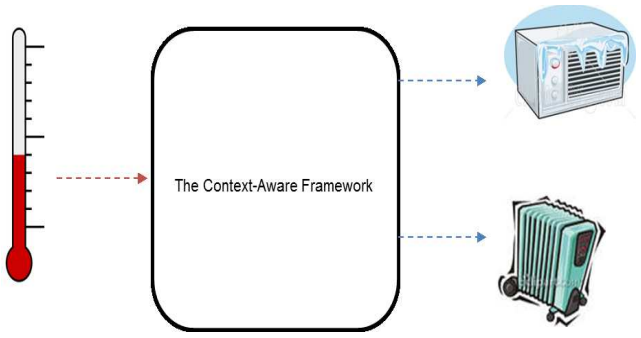

Figure 6. Temperature Control System

Sensors Listener We defined a thermometer listener that is responsible to connect to a thermometer that provides temperature measured in Fahrenheit. For the purpose of testing, we set the connector to a console connector so users can supply random values.

Sensor Translator Since the sensor is providing the temperature measurement in Fahrenheit, we provided a translator that translates the temperature from Fahrenheit to Celsius.

Sensor Verifier The sensor verifier applies the data policies. In this case the verifier will validate the sensor reading. As an example, any sensor reading less than -70C or more than 70C will be filtered out by the verifier.

Actuator Controllers We have two actuators in this case, a heater actuator and a cooler actuator. However, for testing purpose we can also include a console actuator to allow printing.

Context information The only context information we are expecting is temperature. So, we introduce *Temp* as dimension which can also be used as a variable in defining situations.

Situation and Extenders Situations of interest for the client are stated. As an example, "temperature less than 10C and more than 30C" may be the two situations of interest for the application. Thus, we defined the following two situations in CSEL:

1. $Cold : \{ (Temp \leq 10) \}$: This situation refers to the event of observing the temperature transiting below 10C.
2. $Warm : \{ (Temp \geq 30) \}$: This situation refers to the event of observing the temperature transiting above 30C. It is also possible to introduce the Extender $Warm : \{ (\$IsHot[Temp]) \}$, in which the parameter $Temp$ may be substituted at run time.

Since constructs in CSEL may not be sufficient to express complex situations, Extender mechanism is provided to help the application developer to introduce precoded complex situations.

Adaptations and Polices Corresponding to the defined situations, we define the following two adaptation policies:

1. *Adapt to Cold Weather*: In response to the $Cold$ situation, this adaptation policy enforces "temperature increase". The policy is executed by the reaction "increase temperature". The adaptation policy, coded in AWPEL, is

$$if\ (\$IsHeatingSystemWorking[])$$

$$Exec\ (IncreaseTempAction[]);$$

2. *Adapt to Warm Weather* In response to the $Warm$ situation, this adaptation policy enforces "decrease temperature". The policy is enforced by the reaction "decrease temperature". The adaptation policy, coded in AWPEL, is

$$Exec\ (DecreaseTempAction[]);$$

The *guard IsHeatingSystemWorking* is itself a policy to check if the heating system is working before sending an order. In real life situations many other precautions may have to be taken. These can be expressed by using conditional and other branching constructs in AWPEL.

Reactions The two reactions are "increase temperature" and "decrease temperature", which respectively encapsulate the interaction with the heater and cooler actuators.

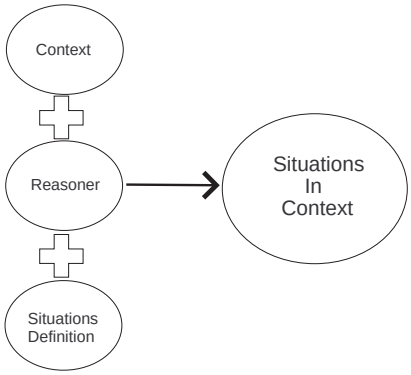

Figure 7. Reasoning Context Information

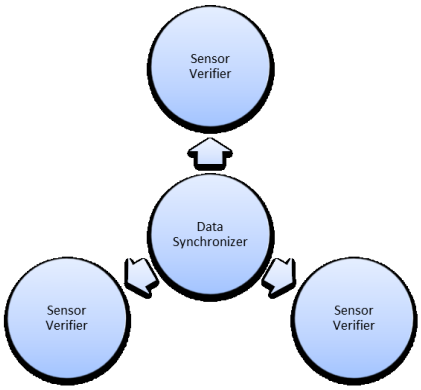

Figure 8. Aggregating Sensor Reading

System process Model

The sequence of steps in the system process model is explained below.

- *Activation:* The system can be activated in two scenarios: (1) the client asks explicitly to check the context and react upon it or (2) new context data is constructed. The second scenario is a subset of the first scenario, so we describe here only the first scenario.

- *Requesting information:* The user request causes the following chain effect: (1) the Adaptation Resolver asks the Context Manager for the updated situations that exists in the context, (2) the Context Manager asks the Data Synchronizer for the aggregated context information, (3) the Data Synchronizer asks all Sensor Verifiers for their verified reading readings and (4) Sensor Verifiers asks Sensor Listeners for their latest reading.

- *Aggregating sensor information:* Once the Sensor Listeners receive the readings, the verifiers verify the data and then notify the Data Synchronizer which waits until all sensors respond and then notifies the Context Manager. The process is illustrated in Figure 8.

- *Reasoning over Context Information:* When all the sensor readings are ready and synchronized, the Context Manager is informed by the Data Synchronizer. Then the Context Manager activates the Reasoner to discover situations in the current context. The process is illustrated in Figure 7.

- *Resolving Adaptations:* When the Context Manager receives the "Situations in Context" from the reasoner, the Context Manager informs the Adaptation Resolver with the discovered situation. The Adaptation Resolver resolves the appropriate adaptations for the discovered context and executes each adaptation. The process is illustrated Figure 9.

- *Reacting:* The Adaptation Resolver determines appropriate adaptations and calls the Workflow Executor to execute each adaptation. The Workflow Executor allocates reactions to Actuators and activates them based on the execution path. That is, reactions are channeled to actuators through actuator controllers and implemented.

- *For-ever loop:* This process cycles forever.

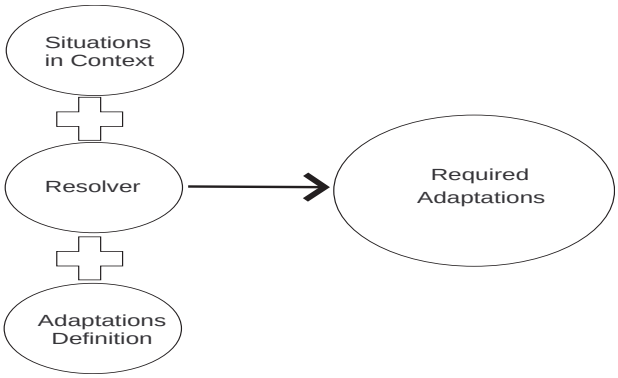

Figure 9. The Adaptation Resolver

4.2. Case Study II: Salesperson Case Study

This case study illustrates the use of the framework for developing context-aware applications which involve mobile devices. The goal of the implemented system is to assist a salesperson improve the productivity in the daily schedule, assuming that it involves traveling in a distributed network of roads, collecting and delivering goods in a manner that earns the trust of the customers and adds economic value to the organization. So, the main functionalities of the implemented application are

- Calculate the best tour route a salesperson needs to take every day, based on the set of customers to be visited, and the network of roads to travel. The status of customers, and environmental conditions (whether, road conditions) are continuously monitored and updated by mobile devices. The tours vary from day to day and are largely driven by contexts observed, situations desired, and knowledge on customer history extracted from a distributed database.

- Suggest items to promote based on customer context and sales history.

- Identify new potential customers in the neighborhood of the tour.

- Facilitate loading items from inventory warehouse, by identifying a salesperson profile and determining quantities based on sales history.

The process model for every context-aware application development is identical to the one given in the previous

section, differing only in details. So, we skip those lengthy details, and describe 1) the sensor types and the contexts constructed, 2) a selection of significant context situations, 3) some sample adaptation policies, and 4) some sample reactions. A complete description of the case study and its full implementation are given in [43].

Sensor Mechanism: Table 1 shows the sensors and the corresponding context tag values that each sensor collects. After validation, the sensor information is used to construct contexts.

Situations: Table 2 presents some situation expressions specified using our CSEL 2.2. These situations are of interest in this application. More situations may be added by the application developer, after ensuring that contexts that can validate them are constructed in the previous step.

Adaptation Mechanism: Adaptations are expressed in AWPEL. In order to express the adaptations, we need to define reactions, policies, and adaptation expressions. Below we describe each of them.

1. Reaction: We implemented four basic reactions. These are: *Notify*, *Check in*, *Check out* and *Recalculate list*:

- *Notify*: This is a general purpose reaction that is used across many adaptations. The reaction is responsible for sending a specific message to a specific destination. This reaction accepts two input parameters: 1) *Destination*, representing the recipient of the message, and 2) *Message*, representing the actual message. The *Notify* reaction can be implemented using SMS Text messages, emails, or method invocations depending on the underlying actuator. The chosen messaging infrastructure is based on the client application. The precondition for this reaction has a valid connection with the destination, which is related to the underlaying actuator. Suppose we are using an email client actuator, the precondition is checked to see if there is a valid Internet connection. The postcondition is either to confirm a successful delivery of the message or to notify the sender of the failure of sending the message.

- *Check in*: This reaction is responsible for triggering a database transaction that checks in items in a specific account. This reaction accepts the two input parameters, *Account Name*, representing the account into which the items are checked in, and *Items*, the set of items to be checked into this account. The precondition is that there exists a valid connection with the database. The postcondition is either to confirm that the operation was successfully completed or to notify the system of the failure of conducting this operation.

- *Check out*: This reaction is responsible for triggering a database transaction that checks out items from a specific account. This reaction accepts the two input parameters, *Account Name*, representing the account from which the items are checked out, and *Items*, the set of items to be checked out from this account. The precondition is that a valid connection exists with the database. The postcondition is either to confirm that the operation was successfully completed or to notify the system of the failure of conducting this operation.

- *Recalculate list*: This reaction is responsible for recalculating a tour plan and the customers list for a salesperson based on specific traffic and weather contexts. The reaction accepts the three parameters, *salesperson ID*, the identity of the salesperson, *Weather*, the weather condition, and *Traffic*, the traffic condition. A precondition for your recalculation might involve weather/road conditions, such as $\{Weather == FREEZING\,RAIN) \wedge (Traffic == CONGESTED\}$, or status change of customers in a certain neighborhood. The postcondition is either to confirm the successful completion of this process or to notify the system of the failure to complete this operation.

2. Policies: A full list of policies for this application is given in [43]. A selected subset from this set of large number of policies is given below. We remark that some policies can be reused with some variation in developing other applications.

- *Salesperson Status Policy* (SSP): *Reactions occur only for active salespersons for a specific duration.*

  A policy of this type is implemented through a policy checker that defines *Is Active* method. The method accepts *SalespersonID* as an input parameter and returns the specific duration for the reaction to be in force if the salesperson is currently active, otherwise returns 0.

- *Customer Financial Standing Policy* (CFP): *Forbid visits and sales for customers with bad credit.*

  A policy of this type is implemented through a policy checker that defines *Is in Dept* method which accepts *CustomerID* and returns whether the customer is in good standing or not.

- *Customer List Recalculation Policy* (CRP): *Recalculate a tour for emergency situations.*

  This policy makes sure that all conditions are met before executing an event. The policy is implemented through a policy checker that defines *Is Necessary* method that accepts *SalespersonID*, *WeatherCondition* and *TrafficCondition* as input parameters and determines whether a recalculation should be executed or not.

Table 1. Sensor Mechanism

Sensor	Context Tag value
System Clock	*Date* and *time*
Salesman GPS	Coordinates
Salesman RFID-Reader	*ShipmentLoaded*, *ShipmentUnloaded*
Warehouses RFID-Reader	*SalespersonID*, *Goods*
Warehouse Loading Manager	*ShipmentIsReady*
Traffic	*RoadCondition*
Weather	*WeatherCondition*
Salesman	*SalespersonID*, *CallingSick*, *OnVacation* or *CarIsBroken*
Business Locator	*NewCustomerInRange*, *NewCustomerAddress*
System Database	*ItemsOnSale*, *AverageSalespersonSales*
Data Warehouse	*SuggestedSale*, *SuggestedCut*

Table 2. Context Mechanism

Situation	Expression
Salesperson in Warehouse	{ ($IsSalesperson[SalespersonID] OR $IsWarehouse[Coordinates])}
Shipment Ready	{ *Salesperson in Warehouse* AND (*ShipmentIsReady*) }
Salesperson on Road	{(NOT $IsWarehouse[Coordinates] AND NOT $IsCustomer[Coordinates])}
Potential Customer	{ (NewCustomerInRang AND $IsNewCustomer[NewCustomerAddress]) }
Salesperson at Customer	{($IsCustomer[Coordinates])}
Customer on Debt	{ *Salesperson on Customer* AND ($IsOnDebt[Coordinates])}
Bad Customer on Debt	{*NOT Good Customer in debt*}
Outstanding Salesperson	{($SalespersonSales[salespersonID] > 2 * AverageSalesersonSales) }

- *Salesperson Authorization Policy*: *Sales managers at the regional and provincial levels or those who have served for more than 5 years may offer special promotional sales to good standing customers.*

  This policy type is applicable to certain class of salespersons and customers with good payment history. This policy is implemented through a policy checker that defines *Is Authorized* method which accepts a *SalespersonID* as input and returns whether the salesperson is authorized or not.

3. Adaptation An adaptation is associated with a situation. Many adaptation policies are implemented using WPEL. The implemented adaptations are (1) *Prepare Shipment*, (2) *Notify salesperson*, (3) *Transfer from Warehouse*, (4) *Transfer from salesperson*, (5) *Recalculate Customers List*, (6) *Suggest Visit*, (7) *Suggest Customer Order*, (8) *offer a discount*, (9) *Offer a waver*, (10) *Pass*, (11) *Notify Nearby salesperson* and (12) *Notify Manager*. Below we give the definitions of four adaptations.

- *3.1 Transfer from Warehouse:* The adaptation workflow shown in Table 3 is triggered once the loading is completed in the warehouse. The quantities of loaded items should be removed from the warehouse account and added to the salesperson account.

- *3.2 Recalculate Tour:* The adaptation workflow shown in Table 4 is triggered to recalculate the tour map, based on road conditions and/or business to be conducted.

- *3.3 Suggest New Visit:* The adaptation workflow shown in Table 5 is triggered when new customer is discovered on a nearby location.

- *3.4 Offer discount:* The adaptation workflow shown in Table 6 is triggered whenever a discount could be offered by a salesperson to customers with good standing.

Reactivity Mechanism: The actuators needed for this case study are *Account Actuator*, *System Functions Actuator* and *Messaging Actuator*. The *Account Actuator* is responsible for managing account transactions for customers, salesmen and warehouses accounts. This actuator uses a database connector. The *System Functions Actuator* is responsible for invoking predefined system functions. They can be implemented as database stored procedures, web services or as code libraries. Consequently, different connectors can be used to communicate with the actuators such as Database Connector, Remote Procedure Call (RPC) Connector, Web Service (WS) Connector, or Remote Method Invocation (RMI) Connector. The *Messaging Actuator* is used by the notify reactions defined previously. The connector used for

Table 3. Transfer from warehouse

Adaptation Name	Transfer from warehouse
Situations	Shipment Loaded

Adaptation Workflow	If ($IsActive[SalespersonID])
	{
	Exec(CheckIn [SalespersonID, Goods]);
	Exec(CheckOut [WarehouseID, Goods]);
	}
	else
	{
	Exec(Notify[SalespersonID, “Your account is not active”]);
	}

Adaptation Policies	Is Active
Adaptation Reactions	Check in, Check out, Notify

Table 4. Recalculate Customer List

Table 5. Suggest Visit

Adaptation Name	Recalculate Customer List
Situations	Salesperson Plan Affected

Adaptation Workflow	If ($IsActive[SalespersonID])
	{
	If($IsNecessary[SalespersonID, WeatherCond, RoadCond])
	{
	Exec(Recalc[SalespersonID, WeatherCond, RoadCond]);
	Exec(Notify[SalespersonID, “Your plan is changed”]);
	}
	}

Adaptation Policies	Is Active Is Necessary
Adaptation Reactions	Re calculate list Notify

Table 6. Offer discount

Adaptation Name	Offer discount
Situations	Good Customer

Adaptation Workflow	If ($IsActive[SalespersonID] AND $IsAuthorized[SalespersonID])
	{
	Exec(Notify[SalespersonID, “Offer a discount”]);
	}

Adaptation Policies	Is Active, Is Authorized
Adaptation Reactions	Notify

this actuator depends on the type of messages, such as email or text message.

5. Implementation

All the functionalities of the framework components described in Section 3 are fully realized in the full implementation [43]. In this section we present a few challenges that we faced in the implementation and explain how we resolved them.

5.1. Platform

We implemented the system using Microsoft.Net platform and Silverlight[1] technology. Technically, it supports multiple platforms by running inside a web browser plug-in. Also, the same .NET code could be reused to write applications for desktop computers and mobile applications that can run on Windows phone.

Smart phones are good platforms to target in this domain. Because a smart phone is portable, it provides a variety of context information and offers Internet connectivity 'on the go'. An important challenge for us was to run the framework on a limited resource environment. We compiled the framework on Windows Phone platform which runs on the .Net Compact Framework, a subset of .Net Framework designed to run on mobile phones and embedded devices. The framework performance was good compared with other time consuming tasks held by the application such as Internet connection and GPS initialization.

Both Silverlight and mobile phone implementations have the same set of capabilities, however there are some differences. An example is connecting with a different GPS sensor, which in Silverlight is a pluggable USB or a Bluetooth based GPS sensor, whereas it was a phone-embedded GPS sensor in Windows Phone. One major difference between the two applications is Internet connectivity, which may not always be available. Unlike the phone application where a 3G connection is available most of the time, in the Silverlight application we had to assume the realistic situation that there is no Internet connection some times. Therefore, we used *execution policies* from our model to check Internet connectivity before executing actions that need connection.

5.2. Component Interactions

We used the observer design pattern [44] to design and implement the interactions between components. The observer design pattern was chosen because (1) we need support for both events and asynchronous calls, and (2) some operations could be either time consuming, such as reasoning, or could include interaction with the outside world, such as reading sensors data. The Observer design pattern allows us to prevent blocking operations and provides a standard mechanism to deal with communication exception handling. Figure 10 and Figure 11 show the communication pattern for the adaptation resolver and reactivity modules. These figures, following the UML sequence diagram syntax, show the interfaces of the component and the temporal order of messages communicated.

The combined behavior of the Resolving and Reactivity module is a composition of the behaviors of the Resolving component and the Reactivity component. Once the *Context Manager* notifies the *Adaptation Resolver* with the situations in context, the *Adaptation Resolver* loads adaptation definitions from the data source through a *Data Provider* (situation may also be cached), and then resolves the appropriate adaptations by matching adaptation definition with the situations in context. The result is a set of adaptations. Each adaptation has a workflow and a set of policies, the adaptation resolver uses the policy checker to enforce the policies, and if all policy's constrains are met, the adaptation resolver activates the workflow executor to trigger the reaction defined in the proper sequence. Once a *Reaction* is instantiated, the *Reaction* activates the *Actuator Controller* passing the appropriate configuration arguments. The *Actuator Controller* then translates the input data from the framework data type to the actuator data type through a *Translator*. Then, the *Actuator Controller* uses a *Connector* to connect to the actuator.

5.3. Challenges

The major implementation challenges are handling the complexity of design and communications, supporting the dynamic nature of context-aware systems, and providing portable solutions to multiple platforms. Our implementation offers effective solutions for these challenges.

1. By using software components as units of implementation and deployment we contain the complexity of context-aware systems design.
2. By using central bootstrapping technique to handle the initialization of the system we provide flexibility in adding and modifying system components at run-time.
3. By using software coding best practices such as using interfaces, inheritance, high cohesion, and low coupling we provide extensibility.
4. By supporting asynchronous method calls for interactions with sensors, actuators and reasoners we effectively prevented deadlocks and interface blocking and allowed proper exception handling.
5. By using reflection mechanisms and dynamic code invocation we allow privileged developers to inject or modify components at the run time without the need to recompile the whole application. For example, in case the developer wants to change the Context Reasoner then that should be done easily in a Configuration file without the need to change the code or recompile it.

[1] http://www.silverlight.net

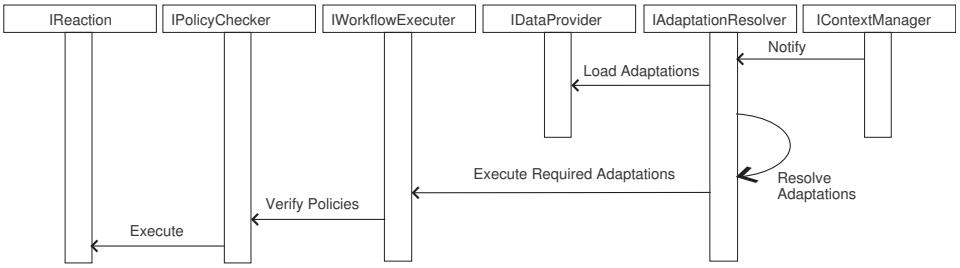

Figure 10. Sequence Diagram for the Resolving Component

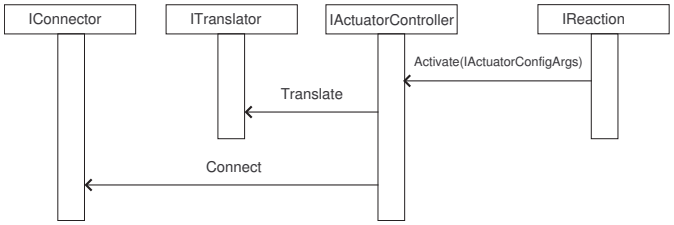

Figure 11. Sequence Diagram for the Reactivity

6. By supporting cross platform environment we allow the deployment of CAF on different platforms such as mobile phones, desktop and laptop computers, and web-based portable devices.

These features affected the choice of the implementation platform and the development environment.

5.4. Case study Implementation

The salesperson case study has been implemented on two different platforms: *Windows phone* and *Silverlight*.

The application connects to the embedded GPS device inside the phone and then connects through the Internet to the distributed system server, where it retrieves a list of the customers that should be visited. Using Microsoft Bing Services[2], the framework reacts by requesting the optimal route taking into consideration time, weather and traffic context information. Figures 12 and 13 show the mobile and silverlight applications. Once the salesperson is within a configurable distance from a customer, 100 meters for example, the framework notifies the salesperson that a customer has been located. It shows detail information about the customer. Also, it shows suggested items to sell and promotions based on analysis of previous orders and payment history. This scenario is illustrated in Figure 12.

When testing this application on road, we faced two important challenges. First, GPS information might not be accurate sometimes. This could result from weather conditions, use of assisted GPS while being far from telecoms towers, and interference. We dealt with this issue by configuring a *Sensor Verifier* for the GPS sensor in our system using a constraint stating that *"in one second a salesperson can not move more than 100 meters"* . Second, improving the usability of the mobile application. We dealt with this issue by utilizing best practices of mobile interface layouts to show navigation tips, screen power saver, and improving touch experience.

[2] *http://www.bing.com*

5.5. Test Results

We implemented a tool for testing and bench marking the context-aware framework performance under different architectural configurations. The following shows the results of running the temperature case study using different number of sensors and actuators. The configurations are (10, 100, 500, 1000, 2000, 5000, 10000, 200000) sensors. Table 7 shows the test results when running the case study on an average machine that has Pentium Dual-Core CPU T4300 2.10GHz, 4GB RAM, and Windows 7 x64 operating system. Figure 14 depicts the performance test results.

Here are our findings:

- Increasing the number of sensors increases the execution time. This is by design as the data synchronizer is awaiting all sensors to provide an up-to-date readings.
- It's a good sign to see the framework providing a fast execution time (under 2 sec for 2,000 sensors, and under 20sec for 20,000 sensors).
- Increasing the number of sensors does not increase the CPU time as the asynchronous (event subscriber) pattern used across the framework is not coupling the processing time with the number of sensors and actuators.

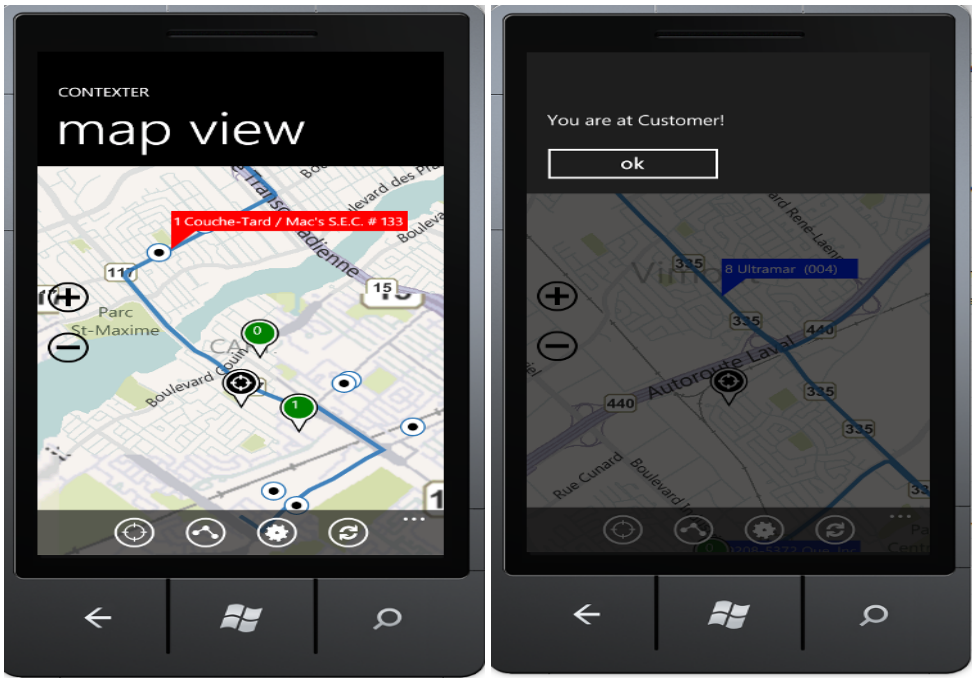

Figure 12. Phone Application-1
Red flag shows next customer, green flag shows potential customers and black flag shows current location

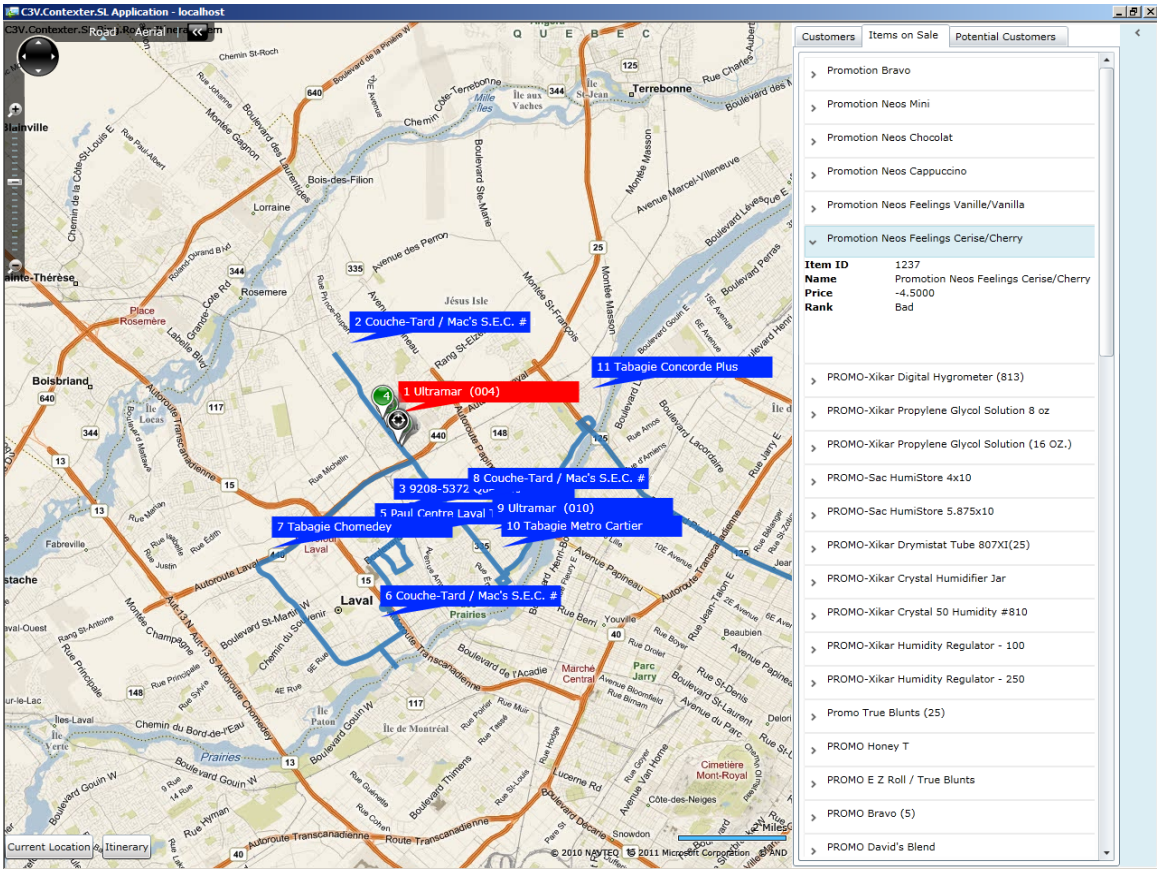

Figure 13. Silverlight Application

Table 7. Test Results

Sensors	Execution Time (seconds)	CPU (%)	Memory (MB)
10	0.04	50	37.24
100	0.04	43	37.52
500	0.12	44	37.79
1000	0.25	91	38.07
2000	0.48	92	38.39
5000	1.89	52	38.59
10000	5.5	55	38.63
20000	17.55	55	39.11

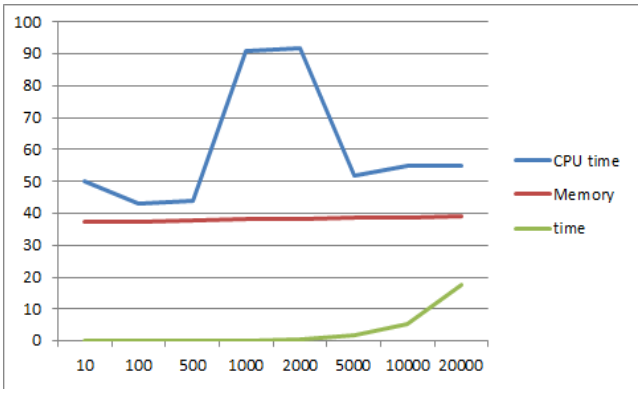

Figure 14. Performance Test

- The memory usage is slightly increasing as we increase the number of sensors, the slop of the increase is related to the case study and how much the sensors are consuming memory. In our case study, the sensors are not doing a memory bound operation and that explains the flat increase.
- There is a slight tipping of CPU usage when using 1000 to 2000 sensors. We tested this scenario several time and found that this tipping is related to some internal operating system operations that are not related to our framework execution.

6. Related Work

In this section we present an extended survey of existing approaches to design context-aware frameworks and context-aware middleware and compare them to our approach.

The framework that we have presented in this paper is supported by formalism at context construction stage, situation validation stage, policy specification stage, and at adaptation stages. In [16] a graphical language is used to model context. Such a graphical representation can be transformed to a relation. In our work we have relational semantics for context representation. Situations are represented as logical expressions in [16] and in some of our previous works [45] [11]. In the current work we decided to specify situations and adaptation-based workflow in two context-free languages. The rationale is such languages are formal, allow extenders, and well-formed expressions can be efficiently evaluated both statically and dynamically. Since the languages admit both standard and abstract types, expressions are well-typed. Any application developer will be quite familiar with these programming constructs. Whereas in [16] no specific architecture or design methodology is explained, we employ the component-based software engineering approach to the design and implementation of the framework. In Section 3 we have explained in detail the different architectural elements and the interfaces to components. Our approach is founded on the trustworthy component design developed by Mohammad and Alagar [46]. So, our framework is fully supported by formal approaches in all stages of its development.

Several researchers are investigating context-aware middleware as a solution to meet some of the challenges, such as heterogeneity of devices and concurrency control, that arise in developing context-aware applications. Whereas in traditional distributed systems the goal of middleware is to *hide* heterogeneity and distribution, in context-aware pervasive computing applications the middleware gets to see its environment in order to provide context management and adaptation to resources in the environment. The traditional middleware is more generic in the sense that it can serve many applications. A context-aware middleware is often specialized. Examples of such middleware include some developed under Gaia [21] and [47] platforms, CORTEX [48]which is specific to pervasive and ad-hoc environments, and CARISMA [17] targeted for mobile applications development. Middleware, whether intended for traditional applications or context-aware applications, can be viewed as a component or set of components that *provide important services to a running system*. A Framework, on the other hand, is a set of components and tools that *facilitate developing systems*. It is easy to understand from our framework description in Section 3 that a context-aware middleware can be viewed as a *projection* of our framework. If necessary such a projected component can be incrementally extended any special needs that are not addressed in our framework. In [18] the two key features *Context Management Services* and *Awareness and Notification Service* are put together to realize the service-oriented

middleware for context aware applications. We emphasize that these two components are part of our framework.

Our framework, being generic, also differs from many specialized middleware architectures. Just to cite one instance, the management of rules and protocols for adaptation in the context-aware middleware CARISMA [17] differ from our framework.

- *Service vs Action:* Our framework enables execution of context-aware actions, where an action might use services. Services in our framework are selected, based on user preferences. In middlewares, the focus is on providing services, where a service itself may denote a family of related services that are not necessarily user-centric.
- *Mutual Exclusion vs Incomplete and Unordered Collection of Rules:* Our framework deals with a collection of mutually exclusive rules that are used with situations. The reasoner in our system infers situations by using flow of rules instead of a flat list. We have extenders in order to embed any logic or function in the execution flow. In middleware an attempt to check a flat list of rules or an unordered combinations of rules might produce conflicts, unless rules are associated with context conditions.
- *Protocol:* When a complete list of rule combinations can not be provided, conflict in rules arise. A greedy protocol used in CARISMA [17] is suitable for dealing with conflicting rules in several applications, such as the traveling salesperson case study that we discussed in Section 4. However, the auction protocol proposed in CARISMA [17] might lead to the violation of safety conditions of safety-critical applications, such as Health Care and Emergency Service. In contrast, the rules that we enforce in our framework are created from user preferences, business and organizational policies, and include trustworthiness criteria defined as a compound property of safety, security, reliability, and availability. As such adaptations corresponding to rule applications can be trusted.

As mentioned in the introductory section, we restrict literature survey and comparison to the category *Framework and Architecture*. Many frameworks for developing context-aware systems have been presented in the literature such as the *context toolkit* [24], SOCAM [22], CoBrA [15], CORTEX [48], Gaia [21, 47], CASS [20], and [3]. A detailed analysis of these frameworks appears in [27]. Recent frameworks include [14] and [12]. Separation of concerns between context sensing mechanism and uses of contexts is one of the suggested design principles of Dey [24]. This principle has been implemented successfully by all the frameworks mentioned above. These frameworks have separate mechanisms for context acquisition using sensors, aggregation and preprocessing of the acquired information either to transform it to a new format or to select from it the required portion, build and store a knowledge base that contains a history of context information, infer high-level contexts by searching the knowledge base, and provide context information to applications. Some frameworks, [8] and [15], provide security and privacy mechanisms to ensure that context information are provided to users who are authorized to view or use it. However, none of the approaches mentioned above have a formal context representation and context conditions for enforcing security and safety rules. For example context modeling in [24] and [20] uses attribute-value model, [48] uses event filters, and [22], [15], [14], [12], and [3] use ontology descriptions based on RDF or OWL. Consequently, adaptation logic could not consider context modifications. The applications contain the logic of reacting to the environment and adapting to the changes of context. However, logic of adaptation is not modeled as part of the framework but rather implemented by the client applications or included in the context management module. For example, in [20] and [48] the context reasoning engine is used to infer a goal or an action. In [22] one method invocation is defined for each context rule. In [14], adaptations are defined using a rule-based language as policies. However, adaptations are implemented by either directly altering context information or by utilizing user defined services (such as imageăcompression services). The ability to change context information directly inside policies confuses context perception with adaptation reactions. Also, it causes conflicts when concurrent processes try to access context information since context information are shared resources. Context-aware adaptation in [4] uses a function that is very specific to a *tour guide* hand held device. The Markov model of adaptation in [49] is specific to enable agents estimate coalition sizes and decide whether or not to join a coalition. In [7] the authors discuss adaptation of role-based access control policies to manage and service crisis situations. The adaptation strategy considers two kinds of constrains, *normal*, and *crisis*, and the grant policies are adapted to these two constraints. Their adaptation mechanism can be extended to accommodate many constraints, yet not general enough to suit any context-aware application. Therefore, our analyses of these approaches reveal a tight coupling between the context mechanism and the required application-specific adaptations. This limits the adaptation component's ability. Given that adaptation is the most significant feature of context-aware systems it is necessary to design this component in a robust manner. In our framework, the low coupling between the context mechanism, situation reasoning, and adaptation mechanisms enables us to have a rich workflow definition language that can be dynamically modified (and extended) to reflect the dynamic nature of context-aware systems. Consequently, we are able to define separate abstractions for adaptations, reactions, and actuation. This has enabled us to (1) define many reactions for each adaptation, (2) control each reaction using policies, (3) and assign reactions to many actuators.

Few frameworks define abstractions for sensors and actuators and allow dynamic plugging of them into the

framework. In [24] a discoverer component is used to keep a registry of available sensors. However, actuators are called using services that are defined inside context wedges. Similarly, in [48] sensors are tightly coupled with the context software abstraction and actuators are defined at design time. In [15] sensors should know about the broker. In [20], sensor listeners are defined to decouple the system from the sensing mechanism. In [3] and [14] sensors are predefined at the design time. Only [20], [12] and [48] provide a mechanism to decouple the sensing mechanism from the communication method used to deliver the sensed context. Some approaches provide a limited validation to the sensed information. For example, [22] and [15] use ontology reasoning to find any conflicts or inconsistencies, and [48] uses a probabilistic scheme to model the uncertainty of sensed data. The heterogeneous nature of devices used in ubiquitous computing imposes the need to use generic data transformers in order to interact with different kinds of sensors and actuators. None of the mentioned approaches provide a dedicated abstraction to perform data translation from native formats to application understandable format.

In addition to the general purpose frameworks discussed before, there exists many domain specific frameworks. Typical domains include Transportation [19], [6], [33], [32], Health care[9], [10], [13], Search engines [26], [31], Social Networks [5], Agriculture [35], Military [30], Security [8], and Mobile phones [50]. Table 8, Table 9, and Table 10 provide a summary of the features included in these frameworks.

Compared to all these works, the framework proposed in this article has many merits, as enumerated below.

1. *Novelty:* The novel features of our framework include (1) the full modeling of both awareness and adaptation, (2) a formal treatment of situation reasoning and adaptation, (3) controlling the adaptation mechanism using business and nonfunctional policies, (4) supporting validation and constraints on context data, and (5) supporting multiple context theories and reasoners. These features enable reasoning about the whole behavior of context-aware systems at architectural level which has been proven to be an effective method to achieve dependability. The modular design of our framework, abstracting context definitions from context processing, and abstracting context reasoning from context aggregation and processing are novel.

2. *Separation of Concerns:* The specification in our framework starts from sensing contexts and extends up to performing the proper adaptation in the environment. Consequently, a developer can exploit this feature to develop a context-aware application with formal basis, without necessarily being an expert in formal methods. The adaptation mechanism specifies the adaptation rules necessary to infer the required changes in the environment and the adaptation workflow, which defines the actions involved in the adaptation. The explicit specification of rules and the separation of the rules from the reactivity mechanism allows us to change the adaptation rules without recompiling the application. It also allows changing the implementation without affecting the rules. Moreover, it allows having different implementations for different environments using a unified adaptation mechanism.

3. *Policy Driven Adaptation:* Adaptation rules are regulated, controlled, and restricted by security, business, and safety policies. Time constraint policies can be defined to regulate self-adaptations and service adaptations. Safety policies can be defined to decide whether or not adaptations should take place and to select one adaptation from many possible ones. Safety policies ensure that the reactions to context changes do not introduce hazards in the environment. Security policies can be defined to ensure that only authorized users will get context information and cause reactions in the environment.

4. *Formal Analysis:* Context, awareness (based on situation evaluation), and adaptation are the basis of context-aware systems. They are defined as first class architectural elements in our framework and are embedded in the framework components. We can utilize the formal component-based development approach presented in [46] to implement a system developed under our framework. The significance of this approach is that behavior protocols of the design components are automatically generated by the methods described in [46]. The behavior model of a component is an automaton, extended with context, awareness (based on situation evaluation), and adaptation rules. The compositions of all these automata can be subject to model-checking, as explained in [46], to verify safety and security properties of the implemented context-aware system. We are not aware of any other context-aware framework that allows formal verification.

7. Conclusion

The essence of this work is in defining a formal CAF and a component based methodology for constructing it. The CAF implementation is robust enough to allow the development of any context-aware application. The methodology introduces a formal process to perceive context and consequently adapt to it. The process consists of (1) identifying *Sensors*, (2) defining *Context*, (3) defining *Context Situations*, (4) identifying *Actuators*, (5) defining *Reactions*, (6) defining *Policies*, and (7) defining *Adaptations*. A summary of our contribution in meeting this process requirements is given below.

- A *Situation Expression Language* is developed to introduce situation expressions in the system. Based

Table 8. Comparison Between Context-Aware Approaches Part-1

Study	[45]	[26]	[25]	[5]
Context Formalism	**Y**	X	X	X
Context Abstraction	**Y**	X	**Y**	X
Sensors Abstraction	X	**Y**	**Y**	X
Actuators Abstraction	X	X	X	X
Communications	**Y**	X	X	**Y**
Data Transformers	X	X	X	X
Approach Type	G	DS	DS	DS
Policies	X	X	X	X
Adaptation Formalism	**Y**	X	**Y**	X

X represents that the study did not address the issue, *Y* represents that it did. *DS*: represents Domain-specific, *G*: represents Generic approaches.

Table 9. Comparison Between Context-Aware Approaches Part-2

Study	[50]	[23]	[34]	[51]
Context Formalism	**Y**	**Y**	**Y**	X
Context Abstraction	X	X	**Y**	**Y**
Sensors Abstraction	**Y**	**Y**	X	**Y**
Actuators Abstraction	X	**Y**	X	**Y**
Communications	**Y**	X	X	X
Data Transformers	X	X	X	X
Approach Type	DS	G	G	G
Policies	X	X	X	X
Adaptation Formalism	**Y**	X	**Y**	X

X represents that the study did not address the issue, *Y* represents that it did. *DS*: represents Domain-specific, *G*: represents Generic approaches.

Table 10. Comparison Between Context-Aware Approaches Part-3

Study	[9]	[8]
Context Formalism	X	X
Context Abstraction	X	X
Sensors Abstraction	**Y**	**Y**
Actuators Abstraction	**Y**	X
Communications	**Y**	X
Data Transformers	X	X
Approach Type	DS	DS
Policies	X	**Y**
Adaptation Formalism	X	X

X represents that the study did not address the issue, *Y* represents that it did. *DS*: represents Domain-specific, *G*: represents Generic approaches.

on the formal syntax valid strings in the language are accepted as situations. Thus, the semantics behind the aggregation of context information is captured.

- A *Situation Reasoner* is implemented for inferring situations that exist in a context.
- The *Sensor Listener*, *Actuator Controller*, and *Translator* defined in component design give the ability to deal with different types of sensors and actuators, and translate data between different formats.
- The *Sensor Verifier* in component design provides the CAF the ability to formally verify sensor readings.
- The communication between all CAF components is asynchronous and all based on the *Observer Design Pattern*. The communication with the sensors and the actuators are done thorough a *Connector*, defined in Section 3, to support different methods and protocols of communications.

- A formal definition of the *Workflow Expression Language* has been provided. The language supports the introduction of execution policies as workflow constraints. The triggering of reactions are always verified with respect to execution policies.

The introduction of *Situation Expression Language* to express sophisticated context situations, and the introduction of *Workflow Expression Language* to formally define the execution flow of adaptations and the domain constraints defined as *policies* are novel, new and quite powerful to deal with dynamic contextual changes. Such programming language descriptions are both formal and allow efficient run time execution.

An analysis presented in Section 6 has revealed many inadequacies in the existing approaches for constructing context-aware applications. Given the current status and the trend in pervasive and mobile computing applications, our generic framework architecture with its default features can satisfy most of the needs of software developers. Our approach also empowers software developers to introduce special awareness and adaptation capabilities as plug-ins, on top of existing features.

We observe that the design and implementation of our framework have three important quality aspects.

- *Reusability*: The component-based architecture for the framework allows us to define each component separately as an autonomous unit of deployment. That gives us the ability to reuse framework components in other systems to perform similar tasks. A prime example is the *Workflow language* which can be reused or adapted for several applications, whether or not they are context-dependent.

- *Testability*: The components in our framework have well defined interfaces, which allows us define independent unit tests for each component with stubs or proxies. Integration testing of the framework will require both incremental testing and formal verification.

- *Scalability*: Our design is scalable in different aspects: The design is interface-driven, which separates the architecture from the implementation and allows developers to introduce an enhanced implementation of specific components without affecting the overall process. The *Situation Expression Language* provides *Reasoner Extender* as extension points to the language capabilities.

- *Performance*: A context-aware application, when deployed in a distributed network might face with resource limitations, say battery power or limited computing power of hand-held devices. In our implementation dedicated servers provide the data and computing power, say for replanning the route for a salesperson. The hand-held devices communicate with servers.

For mobile applications as well as for traditional service-oriented applications that allow multiple sensors and mobile devices an important issue of concern is *power conservation*. This issue needs to be addressed separately. Power-conserving protocols and data sharing strategies between mobile platforms and servers must be developed. These can be plugged in to our framework through different connectors without much effort.

Acknowledgement

This research is partly supported by a grant awarded to Vangalur Alagar from Natural Sciences and Engineering Research Council (NSERC), Canada, under its Discovery Grant Program. Also this research is supported by three research grants awarded to Kaiyu Wan. These grants are awarded by National Natural Science Foundation of China (NSF China Project Number 61103029), Natural Science Foundation of Jiangsu Province (NSFJ), China, and XiŠan Jiaotong-Liverpool University, Suzhou, China (Research Development Funding Project Number 2010/13). We thank the reviewers for their valuable comments.

Appendix

The grammar for CSEL is shown in Appendix A, Figure 11. The root of the grammar is “Situation”, which can be either a *Compound Situation* expressed as a situation expression (rule) over dimensions and other situations, or a *Literal Situation* which is expressed as a situation expression over dimensions only. *SituationToken* is an identifier representing a single situation name. *Literal Situation* is used to define a dimension expression inside a situation expression. The purpose of *Dimension rule* is to make sure that each dimension expression starts with circle braces *()* which eliminates any ambiguity when parsing these roles. The dimension rules allow logical operators, comparison operators, and arithmetic operators. In addition, user defined functions are allowed. The grammar for AWPEL is shown in Appendix B, Figure 12. The root rule for the language is the Workflow rule. The workflow is simply a statement collection, defined as a recursive rule over the statement rule. This means that each adaptation can contain one statement or more. The Statement rule is the main bulk of the workflow language. The Workflow Expression Language supports *IF ELSE* Statement, *While* Statement, *For* Statement, *Execute* Statement and *Brace Statement*, a statement enclosed in braces. Except for *Execute* statement the rest of the statements have semantics as in a programming language. The condition in *IF* and *While* Statements may either enforce a policy check or require that a logical expression defined over other conditions remains true. Policy name is an identifier that starts with a **$** sign. If a policy name is part of a statement then the execution policy referred by it is selected in the workflow. Each policy

Table 11. Context Free Grammar for Situations in CSEL

⟨*Situation*⟩	::=	⟨*SituationRule*⟩ | ⟨*LiteralExpression*⟩
⟨*SituationRule*⟩	::=	⟨*ANDSituationRule*⟩ | ⟨*ORSituationRule*⟩ | ⟨*NOTSituationRule*⟩ | ⟨*LiteralExpression*⟩ | *SITUATIONTOKEN*
⟨*ANDSituationRule*⟩	::=	⟨*SituationRule*⟩ 'AND' ⟨*SituationRule*⟩
⟨*ORSituationRule*⟩	::=	⟨*Situation*⟩ 'OR' ⟨*SituationRule* ⟩
⟨*NOTSituationRule*⟩	::=	'NOT' ⟨*SituationRule*⟩
⟨*LiteralExpression*⟩	::=	⟨*Dimension*⟩
⟨*Dimension*⟩	::=	⟨*DimensionRule*⟩
⟨*DimensionRule*⟩	::=	⟨*BraceDimension*⟩ | ⟨*ANDDimensionRule*⟩ | ⟨*ORDimensionRule*⟩ | ⟨*FUNCDimensionRule*⟩ | ⟨*NOTDimensionRule*⟩ | ⟨*ADDDimensionRule*⟩ | ⟨*DIVDimensionRule*⟩ | ⟨*SUBDimensionRule*⟩ | ⟨*MULDimensionRule*⟩ | ⟨*EqualDimensionRule*⟩ | ⟨*NotEqualDimensionRule*⟩ | ⟨*BiggerDimensionRule*⟩ | ⟨*BiggerOrEqualDimensionRule*⟩ | ⟨*SmallerDimensionRule*⟩ | ⟨*SmallerOrEqualDimensionRule*⟩ | ⟨*DimensionToken*⟩ | ⟨DimensionValue⟩
⟨*ParamList*⟩	::=	',' ⟨*Param*⟩ | ⟨*Param*⟩
⟨*Param*⟩	::=	*DIMENSIONTOKEN* | *DIMENSIONVALUE*
⟨*ANDDimensionRule*⟩	::=	⟨*DimensionRule*⟩ 'AND' ⟨*DimensionRule*⟩
⟨*FUNCDimensionRule*⟩	::=	*FUNCTIONNAME* '[' ⟨*ParamList*⟩ ']'
⟨*ORDimensionRule*⟩	::=	⟨*DimensionRule*⟩ *'OR'* ⟨*DimensionRule*⟩
⟨*IMPLIESDimensionRule*⟩	::=	⟨*DimensionRule*⟩ *'IMPLIES'* ⟨*DimensionRule*⟩
⟨*NOTDimensionRule*⟩	::=	'NOT' ⟨*DimensionRule*⟩
⟨*ADDDimensionRule*⟩	::=	⟨*DimensionRule*⟩ '+' ⟨*DimensionRule*⟩
⟨*DIVDimensionRule*⟩	::=	⟨*DimensionRule*⟩ '/' ⟨*DimensionRule*⟩
⟨*MULDimensionRule*⟩	::=	⟨*DimensionRule*⟩ '∗' ⟨*DimensionRule*⟩
⟨*SUBDimensionRule*⟩	::=	⟨*DimensionRule*⟩ '−' ⟨*DimensionRule*⟩
⟨*BiggerDimensionRule*⟩	::=	⟨*DimensionRule*⟩ ' >' ⟨*DimensionRule*⟩
⟨*BiggerOrEqualDimensionRule*⟩	::=	⟨*DimensionRule*⟩ ' >=' ⟨*DimensionRule*⟩
⟨*SmallerDimensionRule*⟩	::=	⟨*DimensionRule*⟩ ' <' ⟨*DimensionRule*⟩
⟨*SmallerOrEqualDimensionRule*⟩	::=	⟨*DimensionRule*⟩ ' <=' ⟨*DimensionRule*⟩
⟨*EqualDimensionRule*⟩	::=	⟨*DimensionRule*⟩ ' ==' ⟨*DimensionRule*⟩
⟨*NotEqualDimensionRule*⟩	::=	⟨*DimensionRule*⟩ '! =' ⟨*DimensionRule*⟩

check statement is a call to a user defined function that returns true if the policy is valid, returns false otherwise. The *Execute Statement* is used for triggering reactions. Reactions are user defined functions, and are identified with their names. Each reaction may have zero or more parameters that are either user supplied value or dimension context information.

References

[1] SCHILIT, B., ADAMS, N. and WANT, R. (1994) Context-aware computing applications. In *Proceedings of the 1st Workshop on Mobile Computing Systems and Applications* (IEEE Computer Society): 85–90.

[2] BIEGEL, G. and CAHILL, V. (2004) A framework for developing mobile, context-aware applications. In *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications*: 361–365.

[3] KORPIPÄÄ, P., MANTYJARVI, J., KELA, J., KERANEN, H. and MALM, E.J. (2003) Managing context information in mobile devices. *IEEE Pervasive Computing* **2**(3): 42–51.

[4] KRAMER, R., MODSCHING, M., SCHULZE, J. and TEN HAGEN, K. (2005) Context-aware adaptation in a mobile tour guide. In *CONTEXT 2005* (Paris, France: Springer), **LNAI 3554**: 210–224.

[5] LOVETT, T., OÂĂŹNEILL, E., POLLINGTON, D. and IRWIN, J. (2009) Event-based mobile social network services. In *Proceedings of the 11th International Conference on Human-Computer Interaction with Mobile Devices and Services (Mobile HCI)* (Bonn, Germany).

[6] VAN SETTEN, M., POKRAEV, S. and KOOLWAAIJ, J. (2004) Context-aware recommendations in the mobile tourist

Table 12. Appendix B: Context Free Grammar for Adaptations in AWPEL

⟨*Workflow*⟩	::=	⟨*StatementCollection*⟩
⟨*StatementCollection*⟩	::=	⟨*StatementCollection*⟩ ⟨*Statement*⟩
		| ⟨*Statement*⟩
⟨*Statement*⟩	::=	⟨*BraceDimension*⟩ | ⟨*ANDDimensionRule*⟩
		| ⟨*WhileStatement*⟩
		| ⟨*ForStatement*⟩
		| ⟨*IfStatement*⟩
		| ⟨*IfElseStatement*⟩
		| ⟨*BraceStatement*⟩
⟨*ParamList*⟩	::=	*','* ⟨*Param*⟩ | ⟨*Param*⟩
⟨*Param*⟩	::=	*DIMENSIONTOKEN* | *DIMENSIONVALUE*
⟨*BraceStatement*⟩	::=	*'('* ⟨*Statement*⟩ *')'*
⟨*WhileStatement*⟩	::=	*while* ⟨*Condition*⟩ ⟨*Statement*⟩
⟨*ForStatement*⟩	::=	*for '('* ⟨*NUMBER*⟩ *')'* ⟨*Statement*⟩
⟨*IfStatement*⟩	::=	*if* ⟨*Condition*⟩ ⟨*Statement*⟩
⟨*IfElseStatement*⟩	::=	⟨*IfStatement*⟩
		| ⟨*IfStatement*⟩ *else* ⟨*Statement*⟩ *preferred
⟨*PolicyCheck*⟩	::=	*POLICYNAME '['* ⟨*ParamList*⟩ *']'*
		POLICYNAME '[' ']'
⟨*Condition*⟩	::=	*'('* ⟨*Condition*⟩ *')'*
		| ⟨*ANDCondition*⟩
		| ⟨*ORCondition*⟩
		| ⟨*NOTCondition*⟩
		| ⟨*PolicyCheck*⟩
⟨*ANDCondition*⟩	::=	⟨*Condition*⟩ *AND* ⟨*Condition*⟩
⟨*ORCondition*⟩	::=	⟨*Condition*⟩ *OR* ⟨*Condition*⟩
⟨*NOTCondition*⟩	::=	*NOT* ⟨*Condition*⟩
⟨*Reaction*⟩	::=	*REACTIONNAME '['* ⟨*ParamList*⟩ *']'*
		| *REACTIONNAME '[' ']'*
⟨*ExecuteStatement*⟩	::=	*EXEC '('* ⟨*REACTIONNAME*⟩ *')' ';'*

application COMPASS. In *Adaptive Hypermedia and Adaptive Web-Based Systems* (Springer Berlin / Heidelberg), *LNCS* **3137**, 515–548.

[7] SAMUEL, A., GHAFOOR, A. and BERTINO, E. (2007) *Context-Aware Adaptation of Access Control Policies for Crisis Management*. Technical report, Purdue University.

[8] GOEL, D., KHER, E., JOAG, S., MUJUMDAR, V., GRISS, M. and DEY, A.K. (2010) Context-aware authentication framework. In *Mobile Computing, Applications, and Services* (Springer Berlin Heidelberg), *LNICST* **35**, 26–41.

[9] BRICON-SOUF, N. and NEWMAN, C.R. (2007) Context awareness in health care: A review. *International Journal of Medical Informatics* **76**(1): 2 – 12.

[10] VAJIRKAR, P., SINGH, S. and LEE, Y. (2003) Context-aware data mining framework for wireless medical application. In *Database and Expert Systems Applications* (Springer Berlin / Heidelberg), *LNCS* **2736**, 381–391.

[11] WAN, K. and ALAGAR, V. (2014) Context-aware security solutions for cyber physical systems. *Mobile Networks and Application* : 1–18.

[12] GUO, B., ZHANG, D. and IMAI, M. (2011) Toward a cooperative programming framework for context-aware applications. *Personal and Ubiquitous Computing* **15**: 221–233.

[13] KJELDSKOV, J. and SKOV, M.B. (2007) Exploring context-awareness for ubiquitous computing in the healthcare domain. *Personal Ubiquitous Computing* **11**: 549–562.

[14] MALANDRINO, D., MAZZONI, F., RIBONI, D., BETTINI, C., COLAJANNI, M. and SCARANO, V. (2010) MIMOSA: context-aware adaptation for ubiquitous web access. *Personal and Ubiquitous Computing* **14**: 301–320.

[15] CHEN, H., FININ, T. and JOSHI, A. (2003) An ontology for context-aware pervasive computing environments. *Knowledge Engineering Review* **18**(3): 197–207.

[16] HENDRIKSEN, K. and INDULSKA, J. (2006) Developing context-aware pervasive computing applications: Models and approach. *Pervasive and Mobile Computing* **2**(1): 37–64.

[17] CAPRA, L., EMMERICH, W. and MASCOLO, C. (2003) Carisma: Context-aware reflective middleware system for mobile applications. *IEEE Transactions on Software Engineering* **29**(10): 929–944.

[18] DE SILVA SANTOS ANDREMCO POORTINGA-VAN WIJNEN, L.O. and VINK, P. (2007) A service-oriented middleware for context-aware applications. In *Proceedings of MPAC 2007* (Newport Beach, USA).

[19] RAPHIPHAN, P., ZASLAVSKY, A., PRATHOMBUTR, P. and MEESAD, P. (2009) Context aware traffic congestion estimation to compensate intermittently available mobile

sensors. In *Proceedings of the 10th International Conference on Mobile Data Management: Systems, Services and Middleware (MDM '09)* (Taipei, Taiwan): 405–410.

[20] FAHY, P. and CLARKE, S. (2004) Cass - a middleware for mobile context-aware applications. In *Proceedings of MobiSys Workshop on Context Awareness* (Boston, USA).

[21] GAIA (2005). Http://gaia.cs.uiuc.edu/.

[22] GU, T., PUNG, H.K. and ZHANG, D.Q. (2005) A service-oriented middleware for building context-aware services. *Journal of Network and Computer Applications* **28**(1): 1–18.

[23] BARDRAM, J. (2005) The java context awareness framework (JCAF) - a service infrastructure and programming framework for context-aware applications. In *Pervasive Computing* (Springer Berlin / Heidelberg), *LNCS* **3468**, 98–115.

[24] DEY, A., ABOWD, G. and SALBER, D. (2001) A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction* **16**(2-4): 97–166.

[25] GOH, E., CHIENG, D., MUSTAPHA, A.K., NGEOW, Y.C. and LOW, H.K. (2007) A context-aware architecture for smart space environment. In *Proceedings of the 2007 International Conference on Multimedia and Ubiquitous Engineering* (IEEE Computer Society): 908–913.

[26] GUI, F., GUILLEN, M., RISHE, N., BARRETO, A., ANDRIAN, J. and ADJOUADI, M. (2009) A client-server architecture for context-aware search application. In *Proceedings of the International Conference on Network-Based Information Systems (NBIS '09)*: 539–546.

[27] BALDAUF, M., DUSTDAR, S. and ROSENBERG, F. (2007) A survey on context-aware systems. *International Journal of Ad Hoc and Upiquitous Computing* **2**(4): 263–277.

[28] MIRAOUI, M., TADJ, C. and BEN AMAR, C. (2008) Architectural survey of context-aware systems in pervasive computing environment. *Ubiquitous Computing and Communication Journal* **3**(3): 68–76.

[29] GUHA, R. (1995) *Context: A Formalization and Some Applications*. Ph.d thesis, Stanford University, USA.

[30] GRINDLE, C., LEWIS, M., GLINTON, R., GIAMPAPA, J., OWENS, S. and SYCARA, K. (2004) Automating terrain analysis: Algorithms for intelligence preparation of the battlefield. In *Proceedings of the Human Factors and Ergonomics Society 48th Annual Meeting*: 533–537.

[31] CAO, H., JIANG, D., PEI, J., HE, Q., LIAO, Z., CHEN, E. and LI, H. (2008) Context-aware query suggestion by mining click-through and session data. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '08)* (Las Vegas, USA: ACM): 875–883.

[32] DESERTOT, M., LECOMTE, S., POPOVICI, D., THILLIEZ, M. and DELOT, T. (2010) A context aware framework for services management in the transportation domain. In *Proceedings of the 10th Annual International Conference on New Technologies of Distributed Systems (NOTERE)*: 157–164.

[33] ZHENG, Y., LIU, L., WANG, L. and XIE, X. (2008) Learning transportation mode from raw GPS data for geographic applications on the web. In *Proceeding of the 17th international conference on World Wide Web (WWW '08)* (Beijing, China: ACM): 247–256.

[34] BEACH, A., GARTRELL, M., HAN, R. and MISHRA, S. (2010) *CAwbWeb: Towards a Standardized Programming Framework to Enable a Context-Aware Web*. Technical report, Department of Computer Science, University of Colorado at Boulder.

[35] SHAIKH, Z.A. (2008) Towards design of context-aware sensor grid framework for agriculture. *Engineering and Technology* **28**(April): 244–247.

[36] LOKE, S.W. (2004) Representing and reasoning with situations for context-aware pervasive computing. *Knowledge Engineering Review* **19**(3): 213–234.

[37] WAN, K. (2006) *Lucx: Lucid Enriched with Context*. Ph.D. thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada.

[38] MCCARTHY, J. (1984) Some expert systems need common sense. In *Proceedings of a symposium on Computer culture: the scientific, intellectual, and social impact of the computer* (New York, USA: New York Academy of Sciences): 129–137.

[39] AKMAN, V. and SURAV, M. (1996) Steps towards formalizing context. *AI Magazine* **17**(3): 55–72.

[40] WINOGRAD, T. (2001) Architectures for context. *Human-Computer Interaction* **16**(2-4): 401–419.

[41] WRONA, K. and GOMEZ, L. (2005) Context-aware security and secure context-awareness in ubiquitous computing environments. In *XXX Autumn Meeting of Polish Information Processing Society Conference Proceedings*: 255–265.

[42] BETTINI, C., BRDICZKA, O., HENRICKSEN, K. and ET AL. (2010) A survey of context modelling and reasoning techniques. *Pervasive and Mobile Computing* **6**(2): 161–180.

[43] HNAIDE, S.A. (2011) *A Framework for Developing Context-Aware Systems*. Master thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada.

[44] GAMMA, E., HELM, R., JOHNSON, R. and VLISSIDES, J. (1995) *Design Patterns : Elements of Reusable Object Oriented Software* (Addison-Wesley).

[45] WAN, K., ALAGAR, V. and PAQUET, J. (2006) An architecture for developing context-aware systems. In *Modeling and Retrieval of Context* (Springer Berlin / Heidelberg), *LNCS* **3946**, 48–61.

[46] MOHAMMAD, M. and ALAGAR, V. (2011) A formal approach for the specification and verification of trustworthy component-based systems. *Journal of Systems and Software* **84**(1): 77–104.

[47] ROMÁN, M., HESS, C., CERQUEIRA, R., RANGANATHAN, A., CAMPBELL, R.H. and NAHRSTEDT, K. (2002) A middleware infrastructure for active spaces. *IEEE Pervasive Computing* **1**(4): 74–83.

[48] BIEGEL, G. and CAHILL, V. (2004) A framework for developing mobile, context-aware applications. In *Proceedings of the 2nd IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*: 361–365.

[49] LERMAN, K. and GALSTYAN, A. (2003) Agent memory and adaptation in multi-agent systems. In *Proceedings of AAMAS'03*: 14–18.

[50] MCCOLGAN, B., MARTIN-COCHER, G. and SHENFIELD, M. (2010), Method and system for a context aware mechanism in an integrated or distributed configuration, Patent, Canadian Intellectual Property Office. Research In Motion Limited.

[51] GUI, N., FLORIO, V.D., SUN, H. and BLONDIA, C. (2011) Toward architecture-based context-aware deployment and adaptation. *Journal of Systems and Software* **84**(2): 185 – 197.