A FRAMEWORK FOR DEVELOPING CONTEXT-AWARE SYSTEMS

SOFIAN ALSALMAN HNAIDE

A THESIS

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE AND SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE

CONCORDIA UNIVERSITY

MONTRÉAL, QUÉBEC, CANADA

APRIL 2011

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Sofian Alsalman Hnaide

Entitled: A Framework for Developing Context-aware Systems

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Bipin Desai

_____ Examiner
Dr. Dhrubajyoti Goswami

_____ Examiner
Dr. Joey Paquet

_____ Supervisor
Dr. Vangalur Alagar

Approved by _____
Chair of Department or Graduate Program Director

_____
Dr. Robin A. L. Drew, Dean
Faculty of Engineering and Computer Science

Date _____

ABSTRACT

A Framework for developing Context-aware Systems

Sofian Alsalman Hnaide

In ubiquitous computing the environment constraints are often regarded as static and software applications are allowed to function in a mobile ecospace. However, in context-aware systems the environment attributes of software applications are dynamically changing. This dynamism of contexts must be accounted for in order to provide the true intended effect on the application of services. Consequently, context-aware software applications should perceive their context in a continuous manner and seamlessly adapt to it.

This thesis investigates the process of constructing context-aware applications and identifies the main challenges in this domain. The two principal requirements are (1) formally defining what context is and expressing the enclosed semantics, (2) formally defining dynamic compositions of adaptations and triggering their responses to changes in the environment context.

This thesis proposes a component-based architecture for a Context-aware Framework that would be used to bring awareness capabilities into applications. Two languages are formally designed. One is to formally express situations, leading to a context reasoner, and another is to formally express workflow, leading to timely triggering of reactions and enforcing policies. With these formalisms and a component design that can be formalized, the thesis work fulfills a formal approach to construct context-aware applications. A proof-of-concept case study is implemented to examine the expressiveness of the framework design and test its implementation.

ACKNOWLEDGMENTS

It is a pleasure to thank those who made this thesis possible. First and foremost, I would like to express my sincere gratitude to my supervisor Prof. Vangalur Alagar for his patience and motivation, his wide knowledge and perspective had profoundly inspired this research. I would also like to extend my gratitude to Dr. Mubarak Mohammad for his guidance and support, his help was truly indispensable for this work.

Finally, I would like to dedicate this work to my family. There never-ending love and unconditional support ever since I was a child is the reason that keeps me going. I can't find enough words to express how fortunate I am for being part of their lives.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Context plays an important role in our lives. It helps us better comprehend the surrounding environment of a specific situation. It exists in our everyday activities. For example, in conversations, humans have the ability to go back and forth between different topics that may seem irrelevant to an observer who lacks knowledge of the context of the conversation while it sounds perfectly normal to someone who is aware of the context. Unfortunately, this does not translate optimally in Human Computer Interaction (HCI). When a person asks a friend about a restaurant, the friend may respond with good answers keeping in mind the preferences of the requester such as quality of service, price and ambience. Hence, the answer provided is relevant. On the other hand, the same query when presented to a search engine would give various results that may or may not match the requester's preferences. Enhancing the relevance of the results can be a tedious task without explicitly specifying the preferences which are mostly subjective in nature.

Context can be defined as the circumstances that characterize an event [Dey00]. It is hard to capture context due to technology constrains. However, the rapid expansion in computer power which is realized in ubiquitous spectrum of high-connectivity, handheld and light-weight devices allowed computers to have a greater insight to user's context. Therefore, computer applications are expected to implicitly perceive user context and seamlessly adapt to it.

Perceiving context requires defining how to present it. When we look at context as a

set of properties such as time, location and user preferences we lose the essence of context which is the semantic that lies behind these properties. For example, GPS coordinates may not bring relevant knowledge about user context, but information such as home, office or school are more profound even if the accurate physical position is not identified.

Context information are captured using sensors. Sensors are entities that provide measurable responses to changes in an application's environment. Context-aware applications are required to interact with sensors. Sensors can be hardware devices such as GPS sensor or software applications such as authorization provider. Identifying the common characteristics of sensors helps defining a generic interface and a mechanism to deal with them. One important issue when dealing with sensors is that their output is not always accurate and their trustworthiness varies in respect to other environment aspects. For example, GPS data accuracy is related to the number of satellites in range, weather information and whether the receiver is placed indoor or outdoor.

Presenting semantics behind the aggregation of atomic properties of contexts is an issue that has been addressed in other domains as well such as Artificial Intelligence or Semantic Web [BLHL01]. The added value of using contextual information heavily depends on defining semantics which requires a technique to explicitly define user intentions and a mechanism to infer them based on context information.

Adapting to the constantly changing context is equally challenging as perceiving context. Adaptation requires accurately mapping predefined actions to specific context situations. Moreover, it requires dynamic composition of these actions which underlines the importance of having a flexible yet formal definition of adaptation. Predefined actions are implemented using actuators. Actuators represent the parts of a computing system which perform actions at the last stage. Just like sensors, actuators could be software based such as database transactions or hardware devices such as door controllers. Context-aware Applications require a standard mechanism to interact with actuators.

In between perceiving context and adapting to it there is a whole process that should be governed with business and quality policies that are imposed by application domains. Context-aware applications need an extendable mechanism to define and enforce policies

that govern the behavior of applications to ensure higher quality. The representation of policies and the interaction with context information and other application resources is an interesting issue that needs to be addressed.

Therefore, developing context-aware systems requires essentially supporting (1) the representation and management of context information, (2) management of sensors to acquire context knowledge from user's environment, (3) definition of semantic information and inference rules to infer situations based on context information, (4) definition of adaptations to contextual situations, (5) management of policies to restrict adaptations and (6) management of actuators to perform adaptations in user's environment. Some of these requirements have been addressed by researchers and in software industry during the last two decades. However, there exists not a single approach that addressed a complete solution to context-aware system development that involved all the essential requirements. This is the motivation behind this thesis. We propose a framework for developing context-aware systems which incorporates all the essential requirements. The framework helps software developers empower existing and new application with context-awareness and adaptation management capabilities.

1.1 Thesis Contribution

The major contributions of this work are: (1) introducing a component-based architecture for Context-aware System Development, (2) defining a rich and extendable expression language to define context situations, (3) implementing an inference engine that is able to parse and evaluate context situation expressions against atomic context information, (4) formally defining adaptations and policies and introducing a rich and extendable Adaptation Workflow Language, (5) implementing a workflow executor engine to parse and execute workflow definitions and enforce defined policies and (6) introducing a generic mechanism to interact with both Sensors and Actuators.

The minor contributions of this work are: (1) providing a rich library for interaction with sensors and actuators, (2) providing a platform-independent implementation of the

Framework running on different platforms (Phone, Desktop, Web), and (3) a full implementation of a case study in the sales domain with a phone and desktop interfaces.

1.2 Thesis Outline

The structure of the thesis is as follows: Chapter 2 contains a survey of the related works on this domain and an analysis of these studies. Chapter 3 presents the main concepts and requirements in the domain of context-awareness. Chapter 4 presents a formal definition of context-aware systems (CAS). In Chapter 5 we introduce the main architecture of the Context-aware Framework. In Chapter 6 we provide a full documentation of the detailed design. Chapter 7 reviews main implementation aspects and decisions and presents the case studies used to test this framework. Finally Chapter 8 contains the conclusion and future works.

Chapter 2

Related Work

In this chapter we present related work in the domain of context awareness. In Section 2.1 we present domain specific studies conducted in specific application domains such as health care and transportation. In Section 2.2 we present generic frameworks designed to address different application domains. In Section 2.3 an analysis of these studies and a comparison between them are presented. The comparison is done with respect to different aspects that are identified as crucial for this research.

2.1 Domain Specific Studies

This section discusses context-aware applications that target specific domains. Studies surveyed here propose a domain-specific architecture for managing context information. For example, using users search history to optimize search engines results or real time traffic information to optimize routing. We briefly present the usage of context information and the architecture proposed in each domain of interest.

2.1.1 Search Engines

In search engines the ultimate goal is to achieve a higher relevance in search results. That requires, in addition to good ranking and indexing algorithms, taking into consideration all possible factors that might affect the quality of search results. Context information is one

important factor. A simple example is 'searching for a seafood restaurant'. While Vancouver has a great reputation for seafood, searching for such a restaurant from Montreal reduces the significance of presenting Vancouver restaurants in the search results. Since restaurant in Montreal are more likely to interest users from Montreal rather than restaurants in other cities. Location here is an example of context information that might change the entire relevance criteria.

In [GGR+09] an architecture for context-aware search engine is presented. The authors argued that mobile search is gaining more attraction and the traditional search techniques are not efficient for mobile devices.

This study is proposing an architecture that acts as a mid-tier between mobile search clients and search engines. This architecture, illustrated in Figure 1, contains the following modules.

- Context interpreter: It is responsible for interaction with sensors and aggregating their output.
- Service registry: It keeps track of all registered services that may be used to answer user queries.
- Service interpreter: It is responsible for interacting with mobile applications and interpreting their request.
- Context manager: It is responsible for parsing formatted requests and checking if any service could answer these request. In case this is not possible it directs the request to the Context-aware Search manager.
- Policy Registry: It contains the policies for managing underlying network communications.
- Context-Aware Search Manager: It is a search engine capable of taking context information into consideration.

The term context reasoning was used but without concretely defining what type of reasoning is being held. The adaptation capability is limited since reactions are embedded inside

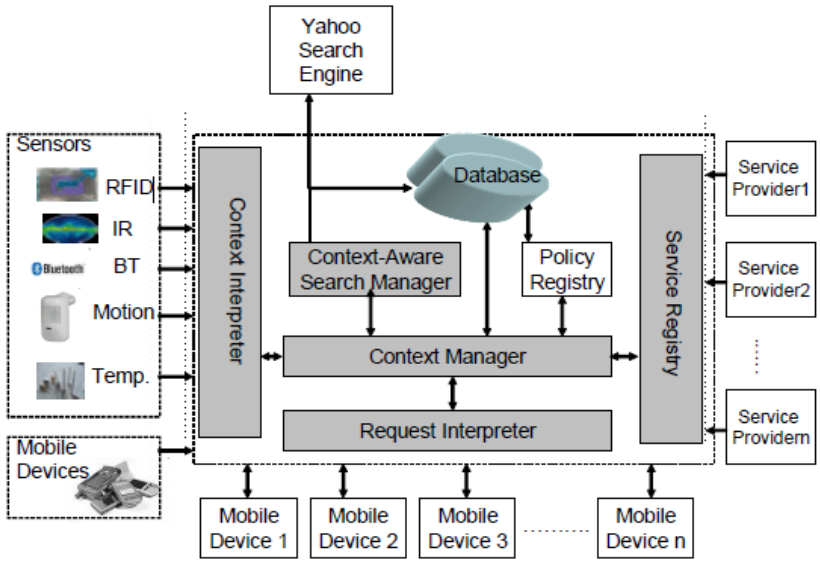

Figure 1: Search Engine Framework Architecture [CJP+08]

the relevant algorithm in the search engine. However, from a design perspective there was a clear decoupling between aggregating sensors reading, managing context information and searching using this information.

Other context aware studies are conducted on this domain. However, some of them considered a different aspect of user context. In [CJP+08] context is considered only as the historic search data, in this study the authors argued that being aware of search history could help search engines provide a better relevance. However, their approach did not contain any formalism for context or a specific architecture to model either how to collect information or how to react upon it.

2.1.2 Smart Places

Smart places are environments empowered with sensors and computer applications that can anticipate user actions and react by accommodating the surrounding environment to user need. Consequently, context awareness has a vital part in research and applications of smart places. This domain is attractive to big companies who have established several projects in context-aware applications, such as HP ® Cool Town project [BBKK01] and Microsoft ® Easy Living project [BMK+00].

In [GCM+07] a context-aware architecture is proposed. The architecture introduces a Context Engine (CE). This engine is responsible for reasoning over context information based on semantic web technologies. The reasoning is done through an ontology reasoner. This architecture, illustrated in Figure 2, contains the following components.

- Device/User Component - This component contains users and the pervasive devices.
- Reasoner - It is the core engine which contains the reasoning system.
- Context Storage - This module contains the most up-to-date environment information.
- Rules Repository - The inference rules stored in this module are used by an ontology reasoner to draw inferences.

In the process of domain analysis the appropriate ontology is constructed to formally model the expected output of the device and to define inference rules used in the system. The adaptation for the changing context is represented using Device Workflow Management System (DWFMS). The framework's main contribution is in introducing the semantic web technology as a suggested solution to uncover the hidden semantics in context information. However, this approach is not proposing a generic interface to interact with sensors or actuators. The formalism for context information is heavily dependent on domain analysis and it's not reusable in different domains. The adaptation techniques are also not discussed and completely dependent on (DWFMS).

2.1.3 Social Networking

Social networks are gaining an increased attraction in the IT domain. With more than 500 million users on Facebook [1], social networks are becoming an important part of everyday life. Social data such as events, friends and interests are packed with context information. Appropriate usage of such information could add considerable value to the quality of service provided to the users.

[1] http://www.facebook.com

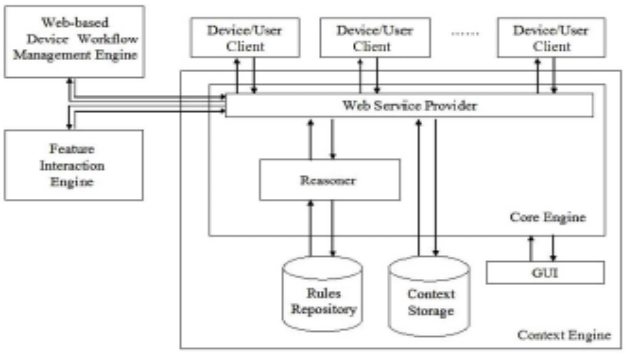

Figure 2: Smart Place Framework Architecture [GCM+07]

In [TL09] a platform to detect user events through user context is introduced. The platform uses mobile phones as sensors that provide information such as location, time, social networks, phone calls log, voice mail, text messages and others.

The platform, illustrated in Figure 3, contains four modules. The first module is the co-presence module which is responsible for detecting physical co-presence through detecting location information for a group of people. The second module is the Social Network. This module connects to a social network and searches through events or calendars of people. The social network module identifies events that may be happening at any moment in time and analyzes social ties between people. It uses social relations, such as friends or family, in combination with other context information, such as location and time, to detect events. The planning module is responsible for calling services needed for that event such as ordering cake for a birthday. The last module is the event module which declares the event when it is ready.

Services provided in this study could be used to organize social events and other social services such as suggesting familiar strangers (people who attend same social events, but not on the friends list). This study suggests some interesting aspects of using context information. However, the platform targets a very detailed problem and no solution was provided to address more complicated situations.

It was interesting to see some other studies that used different kinds of context information as tagging pictures with contextual information beyond time location and people, such

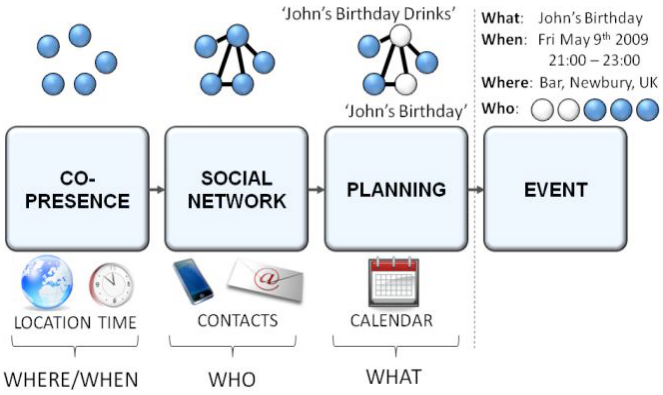

Figure 3: Social Network Framework Architecture [TL09]

as snake rate in [CL07].

Sensors in social networks domain have a unique nature, since basic context information is not directly provided by software or hardware components. Context information needs to be extracted from knowledge base such as calendars or personal notes. Some studies [LOIP10] tackled this issue by integrating and refactoring sensors output such as integrating calendar with social networks data to better investigate events.

2.1.4 Health care

Health care is an interesting and rich domain for context awareness research [BSNc07] whether in managing hospitals or in helping medical staff taking timely decisions. Context is embedded in almost every little detail in the medical domain. Electronic Patient Records (EPR) that show all the medical history of patients and devices readings such as temperature and blood pressure are examples of context-dependent information.

The study in [BSNc07] addresses the issue of communication in hospitals. The architecture is proposed as an extension to regular messaging techniques such as SMS text messages through using context information in determining critical factors in messaging systems like destination, time and validity period of the message. Context information was categorized in three main aspects location, time and role.

Location is important in determining what the medical staff is doing. If a doctor is near a patient bed, the doctor is probably checking on the patient, so it will be helpful to

view information on a nearby screen. This information could contain EPR and notes by the previous doctor on-call. Timeliness is also important in hospital communication, since a message that was sent a day before may not be valid anymore. The medical condition of the patient changes and the proposed medication may not be effective anymore. The last category of context information can be labeled as role. In hospitals there are always roles regardless of who is filling them, such as nurse on-duty or doctor on-call. This piece of information is vital to deliver messages. The communication system in role-based system is all about viewing the right message to the right person in the right place and time.

The architecture proposed in this study is presented in Figure 4 that contains a context-aware client, context aware agent, messaging server and a hospital agent directory. The context-aware client is responsible for providing context information and sending and/or receiving messages. Mobile phones, Personal Device Assistance (PDA) or even monitors are examples of such devices. Context information can by dynamic, such as location of a doctor, or static, such as location of a screen. The context-aware agent is the abstraction that represents the identity of the device on the network. The agent is responsible for contacting the hospital agent directory that keeps track of all agents and routes the messages through the messaging server to the appropriate agent.

Research in context-awareness also uses medical data as a testing prototype. In [VSL03] a context-aware data mining method is proposed. This framework uses context information in tuning data mining algorithms and the authors argue that contextual information have a profound impact in the efficiency of the algorithms.

2.1.5 Security

Security is an increasing concern in computer systems. Authentication techniques such as strong passwords, hardware dangles, RFID cards and their combinations are getting more complicated and negatively affecting usability. Context information helps systems better understand user states and situations. Therefore, using context information as additional tokens provides a more secure, yet less complicated, mechanism for authentication and authorization.

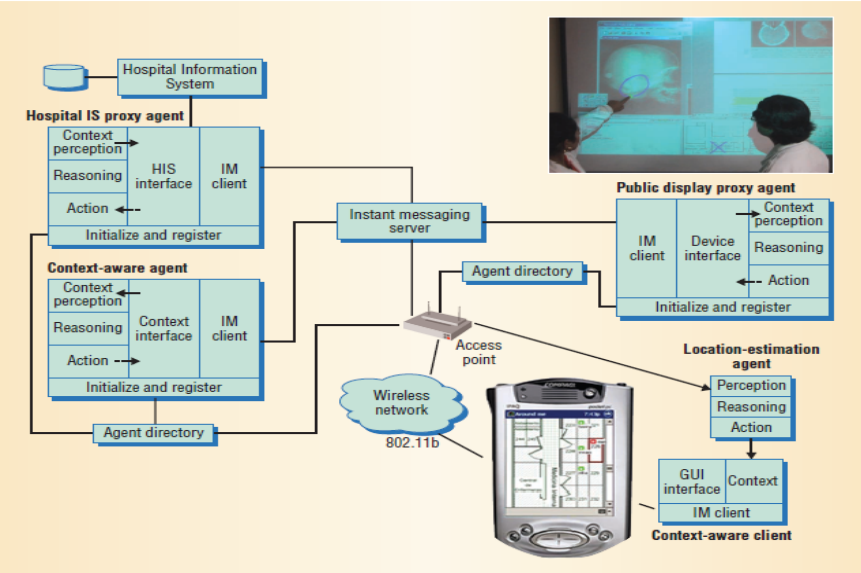

Figure 4: Context Aware Mobile Communication Framework [BSNc07]

In [GKJ+10] the authors propose a Context-aware Authentication Framework for using contextual information extracted from database and gathered by real-time sensors to help systems take well-informed decisions. The authors argued that enhancing security should not involve users explicitly providing additional information. The information collected from other sources should be sufficient.

The framework suggested QR (Quick Response) barcodes [OPB+99] as an authentication technique since it's easy to generate and read, it's also robust compared to other techniques such as RFID (Radio-frequency identification) and other radio techniques which expose vulnerability for sniffing. The framework architecture, illustrated in Figure 5, contains the following modules.

- Core Access Management Module (CAMM) - It manages the QR presented to users and the usage patterns such as logging all login attempts, times and results.
- User Database and Policy Store - It contains user information and policies used to authenticate users.
- Client Mobile Device - It is a network enabled mobile device equipped with a camera. It is used to run the framework mobile application.

- Authentication Site - This is a place where authentication is needed. It is either equipped with computer screens viewing dynamically changing QR or static QR printed on papers.

- Additional Context Cures - This module adds other aspects of context information, such as detailed location inside buildings and calendar information to include user events.

They present a case study on a school campus for managing access to rooms based on context information. Students and staff are admitted to rooms based on their context information, such as events and roles. However, the solution suggested here is tied to a specific technology (QR barcodes) and adaptations are limited only to the authentication operation.

We also reviewed other studies in the Security domain were the security context is formally defined. In [WA07] a context enhanced security architecture is proposed. The authors formally defined *security context* based on the box notation introduced in [WAN06]. The suggested multi agent implementation of the architecture enforces self-protection in Autonomic Computing Systems (ACS) through utilizing context information.

2.1.6 Mobile Phones

Context-awareness is also a hot research topic in telecommunications industry. Research In Motion Limited (RIM) ® has recently registered a context-aware platform in the Canadian patent database [MCC10]. This patent introduces a context-aware server and client. Context is abstracted as aspects, which include location and time. Rules and logic are referred as the techniques to compute context but no further clarification was provided to describe what type of rules and how to represent them. Policies define how the combination of reactions are presented. The client queries the context-aware server with respect to at least one aspect.

The architecture, illustrated in Figure 6, shows a generic environment for context- aware services. The architecture contains mobile phones that connects through the carrier network

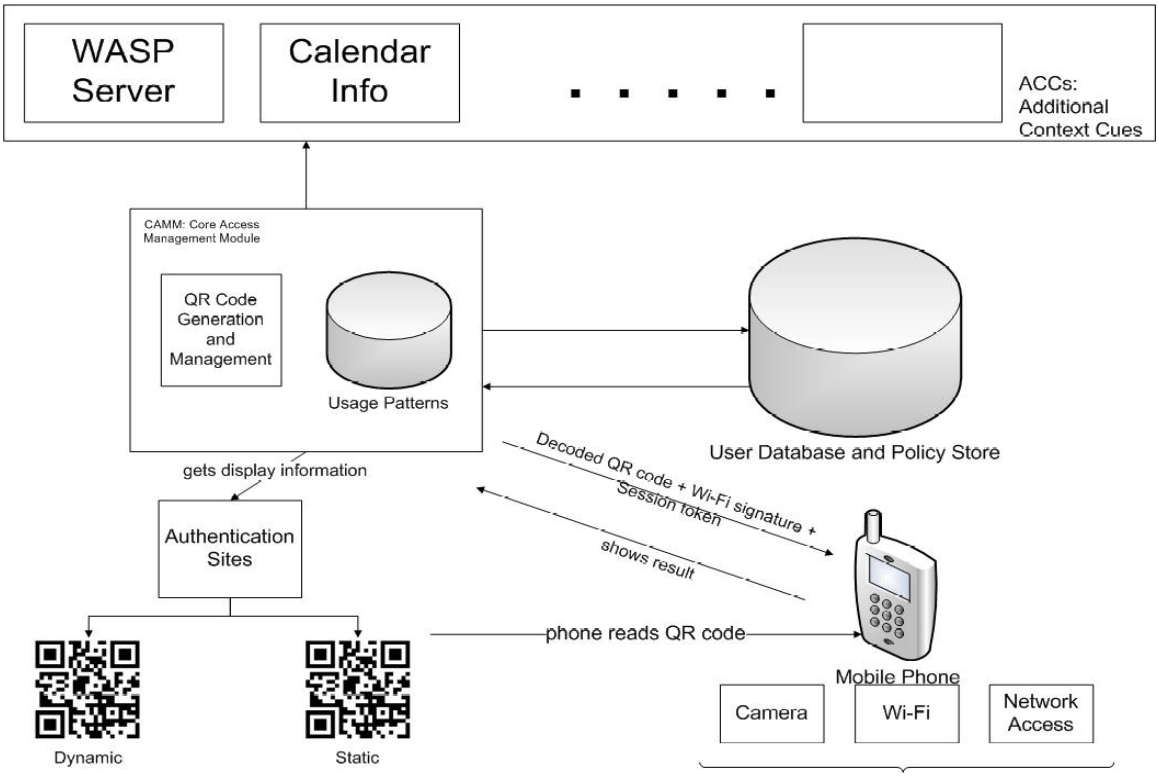

Figure 5: Context-Aware Authentication Framework [GKJ+10]

to the Generic Platform. The Generic Platform calculates the context through the rules and instantiates the appropriate service through the policies.

The architecture provided was very generic. No discussion is given on how to represent logic, policies and context. However, the method is very specific in determining how connection is established in devices and what network protocols are used.

2.1.7 Other Domains

To expand the horizon of our research we broaden our survey to cover a spectrum of other domains. We encountered domains where context-awareness is a hot research topic such as transportation ([RZPM09], [vSPK04], [ZLWX08], [DLP+10]) where context information, such as traffic and historical data, has a vita role. Defense is another domain where context information has important part, as shown by the study in [CM04]. We also encountered context-awareness research in domains like agriculture [uRS08] where context

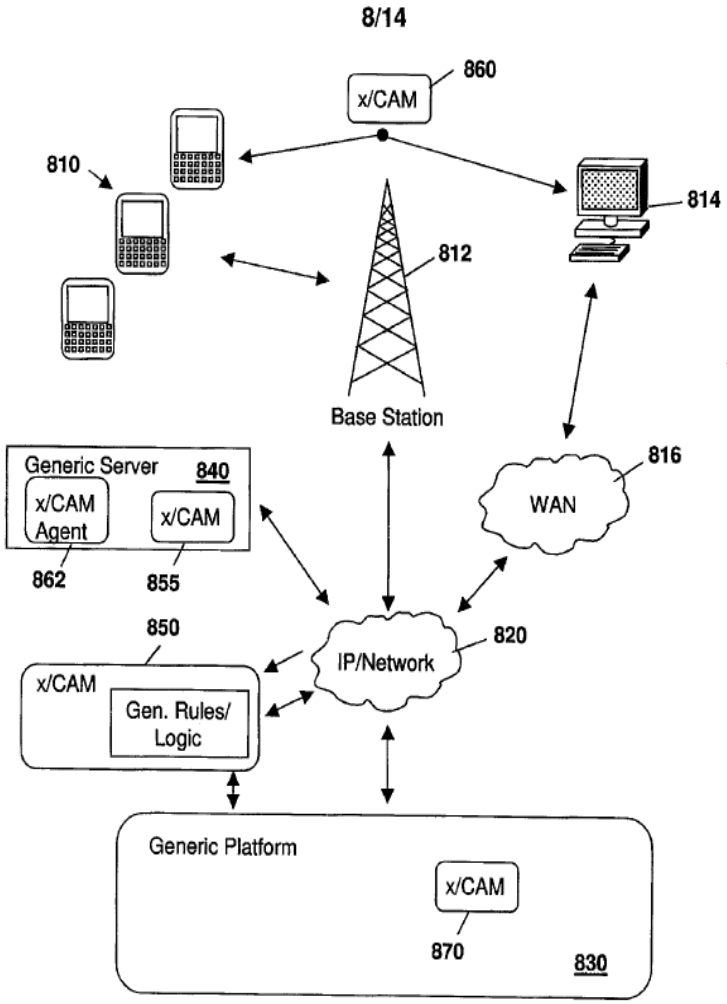

Figure 6: Generic Context Aware Mechanism [MCC10]

information includes soil condition, weather, water-flow, pesticides and crop. From a software engineering perspective, analyzing all different approaches of representing, handling, and interacting with context information gave us a good starting point to identify the main requirements for our framework.

2.2 Generic Frameworks for Context Awareness

Generic Frameworks are general-purpose software frameworks built to facilitate the process of creating context-aware applications in different domains. It represents the main motive behind this work. Surveying this area shows that there are not many solutions that address this specific issue compared to domain-specific solutions. That finding gave us the confidence to go forward and investigate a generic, flexible, component-based architecture in this thesis. The thesis work modifies and enriches an early work [WAP06] suggests in which a three-tiered component-based architecture has been introduced for context-aware systems.

2.2.1 The Java Context Awareness Framework (JCAF)

This research [Bar05] claims to propose a service-oriented, event-based and secure infrastructure suitable for the deployment and development of context-aware applications. The interface-driven design represents a framework that could be extended by developers.

Context is defined on the three main levels *item*, *entity* and *context*. Context item represents one piece of contextual information, such as location or time. An entity is a logical grouping of one or more context items. A Person, as a context entity, contains location, name and job as context items. Context is the biggest container, which contains all context entities such as hospital context, work context or home context.
The framework architecture, illustrated in Figure 7, consists of the following layers.

- Context Client layer (CC)- It contains the applications that are using the framework through subscribing or requesting context information.

- Context Service layer- It contains a context service API which provides client applications with the sensed information. This API is provided through web services.

- Context Monitor and Actuator layer- It contains the entities responsible for communications with sensors and actuators.

This framework supports both synchronous and asynchronous calls. However, the communication mechanism is not decoupled from sensors and actuators. Context representation proposed by this framework does not provide the ability to define other abstractions over simple context information. The added value of the logical grouping of context information in entities was not discussed.

The adaptations in this framework support only actuator calls, but do not meet any realistic application requirements which often demand for sophisticated adaptation scenarios. Although a data access policy is supported, no native support exists for any other type of business policies. Implementation-wise, this framework enforces certain technologies especially in communication with sensors and actuators which should have been abstracted to support evolving technologies in future.

2.2.2 Context-Aware Web (CAwbWeb)

This study [ABM10] proposes a framework for context-aware mobile cloud computing. There is a growing mobile application market (iTunes ® store contains more than 400,000 applications for iPhone ®) and ongoing focus on cloud computing (Amazon EC2 ®, Microsoft Azure ®, Google Apps ®). These two domains are growing side by side to deliver an enhanced user experience by providing thin clients with high connectivity. However, applications should identify cloud services in order to work properly. The framework facilitates the process of mapping mobile applications with cloud services by providing a middle layer that matches services with demands.
The methodology proposed in this study divides the problem space into the following aspects.

- Intentions: It represents the applications purposes of using the services.

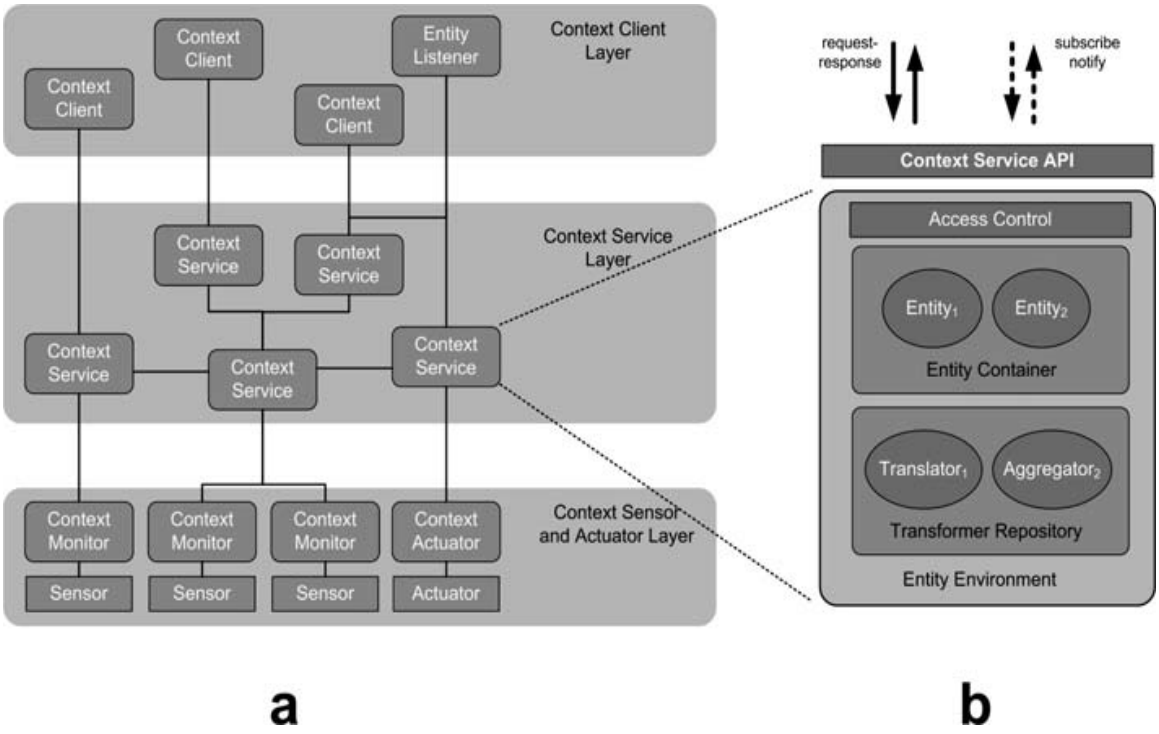

Figure 7: JCAF Framework [Bar05]

- Context: It represents the contexts of application executions.
- Action: It represents the appropriate actions that should be taken in a specific context to meet a specific intention.
- Actuation: It represents the detailed operations of each action.

The framework, illustrated in Figure 8, contains four layers (1) Mobile application, (2) Contextual lookup Service, (3) Context-aware Intention Compiler and (4) Actuation Program Interpreter. The mobile application is responsible for collecting context information and specifying the application intentions. The mobile application then contacts the contextual lookup service which in its turn maps the appropriate cloud web service to the context. Afterwards, the intention complier suggests actions in the execution context that meet the application intentions. Once the action is chosen, the actuation interpreter is responsible for executing the instructions.

The framework proposes a Context Description Language (CDL) using ontologies [W3C10],

an XML based Action Description Language (ADL) and an XML based Actuation Instruction Language (AIL). However, the main structure of this ontology is not specified. As a consequence applications are *forced* to define their own context information. This in turn might introduce conflicts and inconsistencies in mapping ontologies. For example, if two applications are using two separate ontologies, in order for them to cooperate they need to map their ontologies. Ontology mapping is done by matching each concept in the first ontology with the corresponding concept in the second ontology. This operation is time consuming if done manually and challenging to automate. The role of ontology reasoner in defining context was not discussed, and consequently inconsistencies introduced by ontology mapping cannot be detected.

The studies made a clear separation of concerns in terms of how to represent context and how to represent reactions. However, although they mentioned security and privacy concerns, no actual solution was proposed for representing execution policies.

2.2.3 Architecture-Based Context-Aware Deployment and Adaptation

This study [GFSB11] presents a framework for context-aware development. The framework underlines the importance of runtime in order to capture context information and to adapt to the changing context. The framework proposes a dynamic run time methodology to deal with the continuously changing context information.

The architecture, illustrated in Figure 9, contains five main models: (1) Event Monitor, (2) Adaptation Actuator, (3) Structural Modeler, (4) Context-specific Modeler and (5) Context Reasoner. Event Monitor is responsible for measuring what they described as system states. It is worth to mention that the term sensor was not used in their paper. Adaptation Actuator is responsible for communicating with actuators. Structural Modeler is responsible for managing run time dependency and instantiating the right workflow of components on runtime. Once the components are ready, the Context Reasoner is responsible for choosing the appropriate Context-specific Modeler that implements the right adaptation based on the current context.

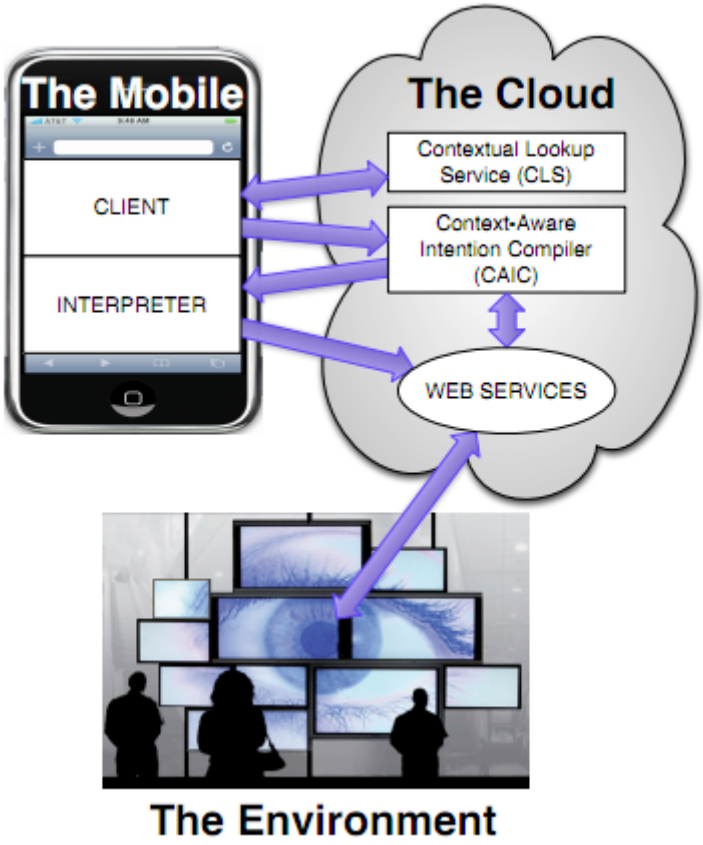

Figure 8: CAwbWeb Framework [ABM10]

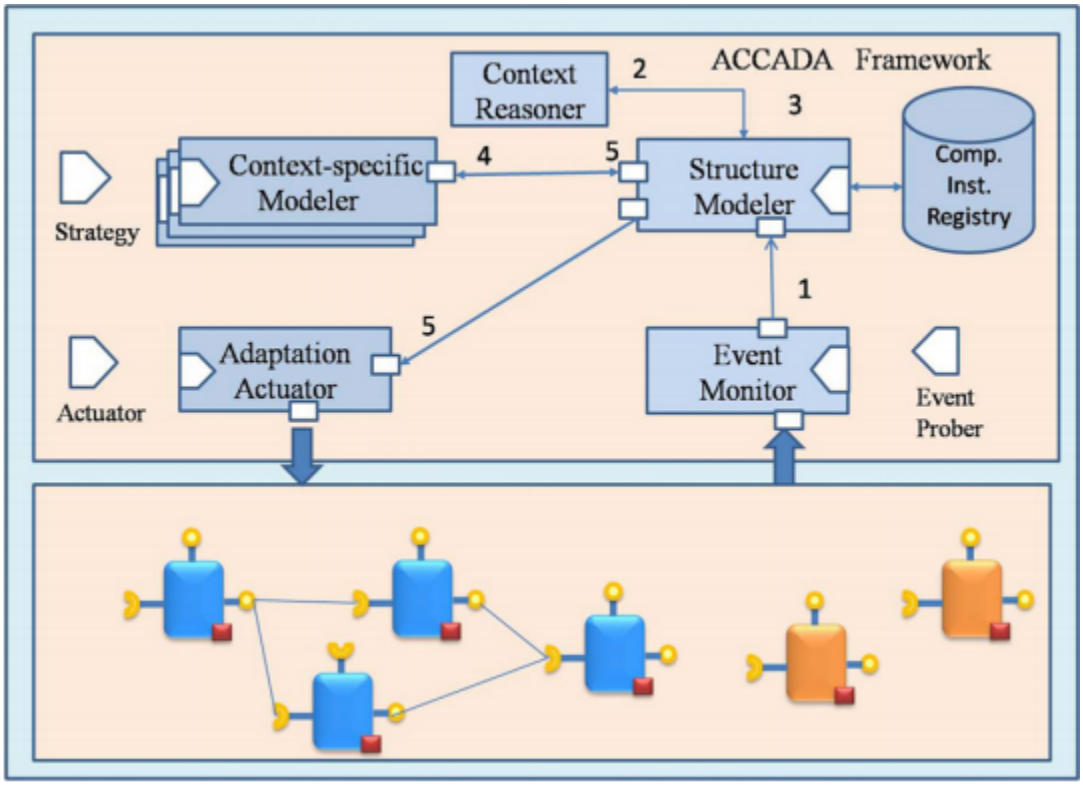

Figure 9: ACCADA Framework [GFSB11]

The study mentioned multiple types of context reasoners. However, no concrete design or implementation was suggested. While the concepts of context and adaptation were introduced, no formal definitions were provided. Consequently, they did not discuss how "context-specific services" were registering interest on context information. On the other hand, they discussed thoroughly the dynamic composition of software components.

2.2.4 Context Toolkit

Context Toolkit [Dey00] is a conceptual framework that facilitates the design and implementation of context-aware applications. They identified the following requirements of the framework.

- Separation of concerns and context handling (SEP).
- Context interpretation (I).

- Transparent distributed communications (TDC).
- Constant availability of context acquisition (CA).
- Context storage (ST).
- Resource discovery (RD).

The component model for the Context Toolkit, illustrated in Figure 10, contains the components *BaseObject*, *Widgets*, *Services*, *Aggregators*, *Discoverer* and *Interpreter*.

BaseObject component provides the communication infrastructure used to communicate with widgets, services, aggregators, discoverers and interpreter. As all other components are sub-components of BaseObject, the communication mechanism is consistent all over the framework. Widget is the framework abstraction that separates how context is acquired from sensors and how it is used. Widgets notify applications whenever contexts change. Service is the framework component responsible for interacting with actuators, it is considered as a sub components of Widget. For example, the widget responsible for monitoring the light in a room connects with a light sensor and a light actuator through a service. Aggregator is responsible for notifying interested components with multiple context information in oppose single context information like to what widgets do. Discoverer is the component responsible for detecting interested components in a specific context change. Finally, Interpreter is! responsible for mapping row context information to meaningful information, such as translating GPS coordinates to street location.

The Toolkit makes a clear separation of how to communicate with sensors and translate sensor information. However, context abstraction is discussed but no concrete solution was provided and no abstraction for adaptation was provided as well.

2.2.5 Other Studies

Other studies in this domain have been surveyed, for time and space constrains we could not describe them all in this thesis. Some of these studies address specific platforms, such

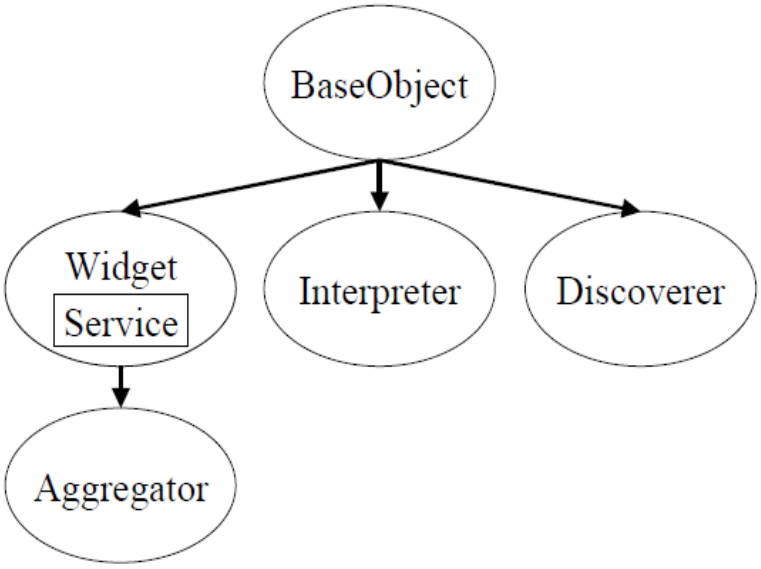

Figure 10: The Context Toolkit Architecture [Dey00]

as Mobile devices [BC], or technology specific approaches, such as targeting specific deployment platforms [RS06].

2.3 Analysis

A comparison of the main criteria behind the different approaches surveyed above is presented in Table 1, Table 2 and Table 3. The comparison focuses on the following aspects.

- Context Formalism: It represents how context is defined. Since context representation is not intuitive, the process of formally defining what is context and how to represent is an important factor in evaluating solutions in this domain.

- Context Abstraction: It represent the mechanism used to uncover the hidden semantic beyond atomic context information. Presenting context information in a raw level does not meet the requirements of applications that needs to identify high and abstract context definitions. For example, while GPS coordinates might mean nothing to applications, information such as home, work or school have more impact even if it's not accurate.

Study	[WAP06]	[GGR+09]	[GCM+07]	[TL09]
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

Table 1: Comparison Between Context-Aware Approaches Part-1
X represents that the study did not address the issue, *Y* represents that it did. *DS*: represents Domain-specific, *G*: represents Generic approaches.

- Sensors abstraction: It represents the abstraction layer for dealing with sensors.
- Actuators abstraction: It represents the abstraction layer for dealing with actuators.
- Communication: It represents the communication mechanism. It is a critical operation since frameworks need to deal with sensors and actuators. Specially that different sensors may use same communication technique such as Serial Port, USB or Bluetooth. Abstracting the concept of communication is important factor in this domain.
- Data Transformers: It represents how different approaches where addressing the heterogeneity issue caused by using different software components. For example, while all GPS sensors provide the same content, there exist different formats for GPS information which makes it critical to separate content from presentation.
- Approach type: It is either *G*eneric or *D*omain-*S*pecific
- Policies: It represents the mechanism used for enforcing application constrains.
- Adaptation formalism: It represents the formalization methodology used to present adaptations.

Study	[MCC10]	[Bar05]	[ABM10]	[GFSB11]
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

Table 2: Comparison Between Context-Aware Approaches Part-2
X represents that the study did not address the issue, *Y* represents that it did. *DS*: represents Domain-specific, *G*: represents Generic approaches.

Study	[BSNc07]	[GKJ$^+$10]	[Dey00]
Context Formalism	X	X	**Y**
Context Abstraction	X	X	**Y**
Sensors Abstraction	**Y**	**Y**	**Y**
Actuators Abstraction	**Y**	X	X
Communications	**Y**	X	**Y**
Data Transformers	X	X	**Y**
Approach Type	DS	DS	G
Policies	X	**Y**	X
Adaptation Formalism	X	X	X

Table 3: Comparison Between Context-Aware Approaches Part-3
X represents that the study did not address the issue, *Y* represents that it did. *DS*: represents Domain-specific, *G*: represents Generic approaches.

Based on the analysis provided in Table 1, Table 2 and Table 3 we can conclude that no single approach has the features to address all the issues that was identified as crucial for context-aware modeling. The main focus of this work is the development of a context-aware framework that provides feasible solutions to *ALL* the issues that we have discussed.

Chapter 3

Requirements of Context-Aware Systems

Context-aware systems (CAS) are the class of computing systems which maintain continuous monitoring of the surrounding environment and adapt their operations based on the changes in current context without direct user intervention.

In recent years the emergence of ubiquitous computing with a plethora of sophisticated, hand-held, and lightweight devices amplified the importance of the evolving studies in context-awareness. The environment information provided by cell phones, tablets, music players, PDAs and eBook readers can be used in different aspects to provide a seamless user experience. It facilitates the process of Human Computer Interaction (HCI) with adaptive applications that anticipate and react to user context. This transition from stationary computing to ubiquitous computing made it possible to deliver some of the implicit user contexts that were hidden in the former mode. As a consequence, the need for a precise formal definition of what context is and how to interpret and react upon it have become inevitable questions. Any application with context-awareness capabilities should have clear guidelines on these issues.

Context-aware systems are notoriously heterogeneous and complex. Heterogeneity results from the variety of sensory devices used to perceive the environment of concern, diversity of context information and adaptations, and the multiplicity of actuators used to adapt to environmental situations. Complexity results from the diversity of relations and

connections between devices, context construction and interpretation, and the dynamic nature of the environment where contexts change requiring new devices to be added and some old ones to be discarded. Heterogeneity and complexity can be handled by following software engineering principles such as separation of concerns which includes modularity and abstraction, low coupling, generality, and reuse.

This chapter contains the definitions of the main concepts in the domain of context awareness. We analyze context-aware systems and define their essential requirements.

3.1 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*, and *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 for formally defining context as a *relation* in [WAN06]. The entities are called *dimensions* and the values assigned to them are called tag values. Note that the tag values have a *type*. For example, the type of tag values assigned to *speaker* is $string$. Therefore, context is a typed relation. If the dimensions in a context are all different then it is called *simple context*.

Example 1 *The location context of a user (tourist) perceived by a hand held device has the four dimensions: (1) LPS (geographical position of the user), (2) $TIME$ (local time), (3) NS (north-south coordinate), and (4) EW (east-west coordinate).*
Assume that the tag set for LPS is the value determined automatically by the geographical positioning system and the tag set for $TIME$, NS, and EW be finite sets of positive integers. Thus, the context $c = [LPS : NewYork, TIME : 13, NS : 5, EW : 3]$ is

a reference to the current user location. It may also be interpreted as a characterization of some event that happens at the intersection of 5^{th} *north-south street and* 3^{rd} *east-west avenue in New York city at time* 13 *hours local time. Hence, the definition of context is independent of what it references.*

This formal representation of context does not reflect the underlying semantics. If we have a user context that contains the GPS coordinates of the user location, this information alone may not be useful. However, providing the semantics of this context by identifying if the user is at home or at work is probably more significant for a meaningful decision making. Therefore, in order to represent semantic information based on context atomic properties we introduce *Context Situation*.

Context Situation is a custom state that occurs when predefined environment conditions are met. Situations are represented as expressions evaluated against context dimensions. **Therefore, an expressive expression language should be used to define sophisticated context situations.**

An application can have one context at any one instant. This context is built from all the gathered information at that particular moment. Figure 11 shows an example of context. A set of context situations are defined as follows.

- Context contains a set of key-value pairs, the key is the dimension and the value is the tag.

- Situation represents a state of interest to an application. Situations can have relations between each other. For example, a *Hot* situation is realized whenever the temperature degree and the humidity are higher than a certain level. Moreover, a heat emergency situation *depends* on the *Hot* situation. This means that *Heat Emergency* can not happen unless it is *Hot*.

The following discusses the requirements of context-aware systems for handling context information.

Context information is the basis on which context-aware systems operate. A context tag value can be of any data type. Therefore, a context information may have a heterogeneous

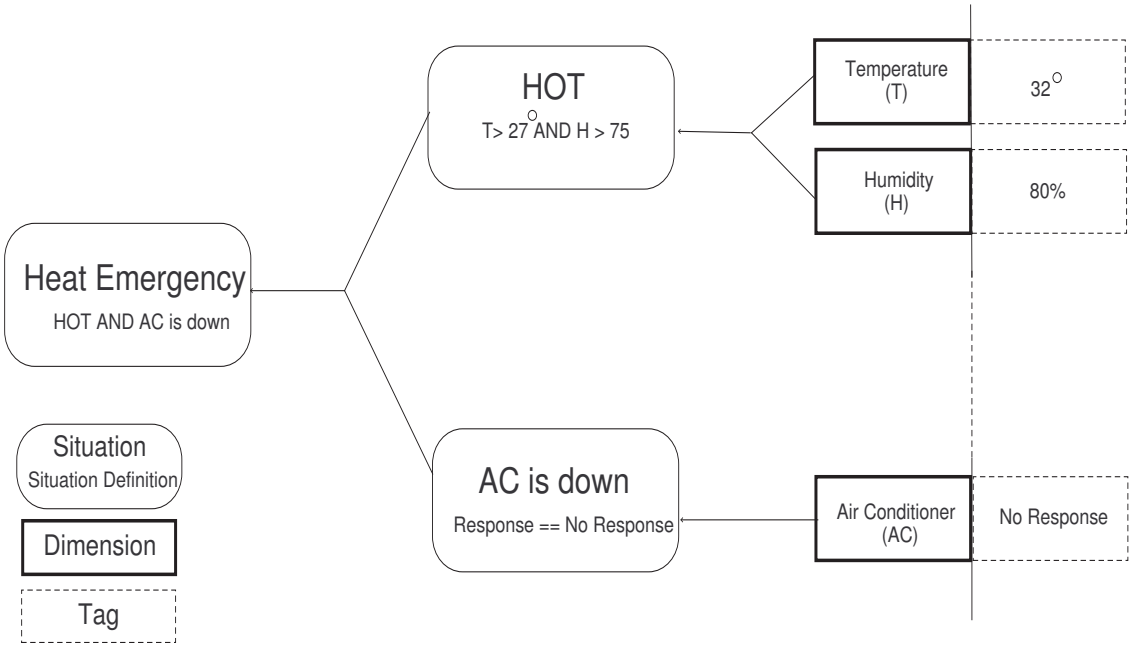

Figure 11: Context Situation

collection of tag value data types. Consequently, there is a need to model context information in a generic way that supports heterogeneous data types. The solution should allow efficient handling of context information. Efficiency involves compact representation that can take small memory storage of context information, fast lookup and access to tag values by knowing its dimensions, and ease of adding and updating the context tag values. **Therefore, a context-aware system should provide an efficient solution for representing and handling context information**.

Context-aware systems are concerned mainly with context situations rather than specific context tag values. There is a dependency relationship between context situation and context. A context situation is realized if there exists a context whose tag values evaluate the context situation conditions to true. For example, in Figure 11 the *Hot* context situation is realized if in the *Temperature* context the degree tag value is greater than 27 and in the *Humidity* context the tag value is greater than 75. Thus, the Hot situation depends on the $Temperature$ and $Humidity$ contexts. In order to check if a situation is realized or not, there is a need for a reasoning mechanism that takes as input contexts and their

dependant situations and gives as output the situations that are realized. Internally, the reasoning engine should extract relevant context tag values and check situation conditions against context tag values. **Therefore, a context-aware system should include a reasoning mechanism to infer situations based on other situations and context information**.

Context situations are defined using logical conditions over contexts. Therefore, there is a need for a rich expression language that empowers context-aware systems with the ability to define context situations. The expression language should contain logical and mathematical operators. To enrich the expression language and make it extensible, it should support also custom user defined functions. These functions can perform complex calculations and evaluate application-specific and business-related conditions. In order to process expressions, there is a need for a parsing engine to parse and interpret these expressions. Also, there is a need for an execution engine that evaluate expressions. **Therefore, context-aware systems should include a powerful situation expression language and evaluation engine**.

3.2 Sensors

Sensors are data providers that produce measurable responses to changes in application's environment. There are different kinds of sensors such as physical, chemical, mechanical, biological, or software-based sensors. Sensors share common characteristics such as *input range*, *output range*, and *accuracy*. Input range is the maximum measurable range that a sensor can accurately measure. The input range of a temperature sensor could be between -30 and 100 degrees Celsius. Output range is also closely related to input range, which refers to output format such as Fahrenheit or Celsius in case of temperature sensor. Accuracy refers to output-specific measures that are related to sensor type. For example, for a GPS sensor the number of decimal point places for Longitude and Latitude data may be used to refer to accuracy, whereas for a clock an error threshold could be used to refer to error data. Sensors can be categorized as either hardware-based or software-based. Examples of hardware-based sensors are GPS navigation sensors, temperature readers or motion

detectors. Examples of software-based sensors are databases and files that reside in storage mediums. The following examples present the most common types of device sensors that are used in ubiquitous computing.

- Motion sensor: It is the device used to detect moving objects. It uses a combination of different technologies such as Passive Infrared (PIR) or Ultrasonic to detect movements.

- Light sensor: It is the device used to detect light. The capabilities of light sensors vary between detecting light to detecting specific attributes of lights such as color and brightness.

- Geographic Position Sensor (GPS): It is the device used to detect geographic location. GPS sensor usually uses satellite signals in order to specify the location.

- Accelerometer: It is an electromechanical device that measures acceleration forces. It is used to detect device orientation with respect to earth or speed of moving objects.

- Optical sensor: It is the device used to detect objects or specific attributes of objects using lights, photos or video. For example, Photoelectric sensors are used to detect distance and absence or presence of an object. Video Sensors often are used in surveillance and facial recognition systems.

- Compass: It is a navigational instrument for determining directions and orientation.

- Temperature sensor: It is used to detect the environment temperature.

- Fingerprint reader: It is used as an authentication technique to augment normal passwords with the fingerprint.

- Barcode reader: It is an electronic device that reads printed barcodes. It is widely used in warehouse management, and in sales and distribution systems.

- RFID sensor: It is the device that uses radio waves to read the information printed on an electronic tag. It is used in different domains, such as transportation and assets management. It is gaining more attention since it is more flexible than barcode readers which require a clear line of sight in order to work.

In the following we discuss the characteristics of sensors. Then, we derive the essential requirements of context aware systems that are related to sensors.

Each sensor type uses a different measurement method to respond to a change in the environment. For example, a GPS sensor locates four or more satellites, figures out the distance to each one of them, and uses the information to deduce its own location based on mathematical principles. The measurement method happens internally inside the device. Software systems interact with sensors as black-box components. Sensors provide public interfaces to allow software systems consume their readings. The interface defines the data types of the reading. This helps the software system communicate with the sensor and consume its data. For example, NMEA 0183[1], RTCM[2], and RINEX are common GPS formats used by sensors to represent their readings. In order for a context-aware system to interact with a sensor and consume its data, it should build translation methods to translate sensor data in different formats into a standard format that can be used by the system. The translation should support as much data formats as possible so that a system can interact with a wide variety of sensors. **Thus, a context-aware system should be able to use different sensor types, and it should have a translation mechanism to translate sensor readings from its native format into a format that can be understood by the system**.

Sensors continuously monitor their environment and register new readings. For example, a GPS sensor may register a new reading every second. This poses a challenge to maintain the huge amount of data that result from sensor readings. **Therefore, a context-aware system should contain a mechanism to handle the continuous flow of data**.

The accuracy of sensors readings varies according to conditions of the environment in which a sensor operates. For example, the accuracy of a GPS sensor's data is affected by

[1] http://www.nmea.org
[2] http://www.rtcm.org

weather conditions, interferences, and whether or not the device is in closed building or in an open space. Since sensor readings cause adaptations in the environment, inaccurate readings can be misleading and cause undesired adaptations. This poses a challenge to check sensor readings and validate them before using it. **Therefore, there is a need for a validation mechanism that validates the accuracy of sensor readings and selects only readings with high accuracy**.

Different sensor types can have different connection methods to communicate data to a context-aware system. Examples of connection methods are USB-cable, Bluetooth, Infrared, Network, Serial and Parallel cables. Each connection method has a different communication protocol. For example, HTTP and TCP/IP protocols can be used to connect to sensors via networks. **Therefore, a software system that is required to get data from sensors should support different connection and communication methods**.

Sensors monitor the environment and communicate their readings to the system. It is possible that a system requests sensor readings at a specific point in time. In this case there are two possible scenarios. First, the system will request its sensors to perform measurements and get current readings. Second, the system caches the latest readings of each sensor and uses the cached value whenever needed. In the later case, the system should maintain expiration dates to its cashed information and needs to update the cached values each time a new sensor data is received. **Therefore, there is a need to support two way communications between a system and its sensors, sensors sending data to the system, and the system requesting data from its sensors. Also, context-aware systems should provide cashing mechanism for sensor data**.

A tag value for a context might come from different sensors. For example, a user's location can be figured out using a GPS sensor, or its IP address, or the user address information. If many readings that are related to the same context tag value are available then the system should be able to select one value from the available ones. Moreover, knowing a tag value of one type it should be able to automatically transform it to an equivalent tag value of another type. **Therefore, a context-aware system should have a mechanism to aggregate sensors' data and select only one value for each context tag**.

Context-aware systems are heterogeneous and flexible. New sensors can be either added or removed continuously. New sensors should be registered in the system as soon as they are available. They get registered as sources for context tag values. When a sensor becomes unavailable, the system should adapt itself and remove the sensor from the registered sources. **Thus, context-aware systems architecture should be flexible. It should allow sensors to be added in a plug-and-play manner**.

3.3 Actuators

Actuators represent the parts of a computing system which perform actions at the last stage of system processing. End results of system processing are realized at actuators. Monitors, printers, mechanical locks, and lids are examples of actuator devices. Actuators could also be software-based. For example, a service responsible for putting user's account on hold after a certain number of failed login attempts or a service responsible for closing all ports when a security hole is discovered can be regarded as actuators. Actuators can have different behaviors after performing their actions. Some actuators, such as printing devices, release control after execution is completed. Some others, such as security hole detectors, may not release control after execution is complete. Actuators are essential parts of any context-aware system. The actuators are at the other-end of the system in which the reactions to any change in the context are realized.

The following discussion is on the characteristics of actuators and the requirements of context-aware systems that are related to actuators.

A context-aware system can interact with different types of actuators. Therefore, it should build translation methods to translate adaptation results into a format that can be understood by actuators. The translation should support as much data formats as possible so that a system can interact with a wide variety of actuators. **Thus, a context-aware system should be able to use different actuator types, and it should have a translation mechanism to translate adaptation results into formats that can be understood by actuators**.

Similar to sensors, actuators are used as black-box components. They expose public interfaces and perform their actions internally. Different connection and communication methods and protocols can be used to interact with actuators. **Therefore, a system which requires interactions with actuators should be flexible and support a variety of connection and communication methods**.

Context-aware systems are adaptive systems. Adaptations can be performed by different actuators. New actuators can be added or removed continuously. New actuators should be registered in the system as soon as they are available. They get registered as actors which perform adaptations. When an actuator becomes unavailable, the system should adapt itself and remove it from the set of registered actors. **Thus, context-aware systems architecture should be flexible. It should allow actuators to be added in a plug-and-play manner**.

3.4 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 take place in response to a contextual change and affect the environment. A reaction represents an atomic action, which could not be split any further. A reaction is defined to do an action through an actuator with a specific 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. The workflow expressions should support sequencing, repetition, and conditioning on sets of reactions. In order to

process adaptations workflow, there is a need for a parsing and execution engine. **Therefore, a context-aware system should include an expressive workflow language to express complicated adaptation scenarios required by client applications. In addition, a workflow execution engine is required.**

3.5 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. Below we define data policies and execution policies.

Data policies: They are rules that constrain data values in contexts. For example, a temperature sensor may have a data policy stating that the temperature should be between -50 to 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. Context time span validity could be presented as data policy. 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, is valid for 24 hours. More complex policies may exist in applications involving network protocols that employ large integers as cryptographic session keys. In general, data policies can be used to express specific time span of validity for different pieces of context information.

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.

In order for a context-aware system to implement policies and control its behavior, there is a need to include the following language related features.

- **A policy expression language** is necessary to specify business and quality assurance rules. The language should be expressive, extensible, and easy to use. By expressive we mean that the language constructs should enable designers to express data policies and execution policies. It should support sequencing, branching, conditions, and loop rules.

- **A parser** is necessary to read and interpret policy expressions, and transform them from their native expression language to constructs that can be understood by a processing engine.

- **A verifier function** is necessary to ensure the syntactic correctness of policies.

- **A processing engine** is necessary to evaluate policies and execute reactions.

3.6 Summary of Requirements

From the above discussion, the essential requirements of context-aware systems are summed up as follows.

- *data model* An efficient data model for representing and handling context information is necessary.

- *context situation handler* This includes an expression language, parser, and an evaluation engine.

- *a reasoning mechanism* The ability to infer situations based on context information is the core of the reasoning system. The reasoning mechanism should be independent from the situation expression language. This means that the reasoning mechanism should be able to reason about contexts defined using different expression languages.

- *different sensor types* The system supports different sensor types and provides a translation mechanism to translate sensor readings from its native format into a format that can be understood by the system.

- *continuous data flow* The system should handle the continuous flow of data.

- *validation* The accuracy of sensor readings should be validated in order to accept only readings with high accuracy.

- *communication and connection* The system should support different connection and communication methods to sensors. A two-way communication between a system and its sensors should be provided. A cashing mechanism for sensor data is essential. Actuators must be supported by a variety of connection and communication methods to actuators.

- *context aggregation* A mechanism must exist to aggregate context data from different sensors and select only one value for each context tag.

- *flexible architecture* A flexible architecture, that allows sensors and actuators to be added in a plug-and-play manner, is essential.

- *actuator types* Different actuator types are to be supported. A translation mechanism should be provided to translate adaptation results into formats that can be understood by actuators.

- *adaptation* Adaptations must be specified and reactions must be associated to each adaptation. An expressive workflow language should exist to express complicated adaptation scenarios that may be required by client applications. An workflow execution engine must be defined.

- *policy language* A policy expression language, a parser, a verifier, and a processing engine are necessary to support data and business policies which constrain system behavior.

Chapter 4

Formal Definition of Context-aware Systems

This chapter presents formal definitions of the Context-aware Framework components. Also, it defines context-free grammars for the *Situation Expression Language* and the *Adaptation Workflow and Policy Language*. These languages are responsible for generating the expressions used to define *Context Situations* and *Adaptations*.

4.1 Sensor Mechanism

We use the following notation in all subsequent definitions:

- $\mathbb{T}$ denotes the set of all data types.
- $\mathcal{D} \in \mathbb{T}$ means $\mathcal{D}$ is a data type.
- $\nu : \mathcal{D}$ denotes that ν is either a constant or variable of type $\mathcal{D}$.
- χ_ν is a logical expression that is defined over the value of ν. If ν is a constant then χ_ν is true.

An attribute qualifies a semantic information associated with an element. The set of attributes is $\mathcal{A} = \{\alpha = (\mathcal{D}, \nu_\alpha) \mid \mathcal{D} \in \mathbb{T},\ \nu_\alpha : \mathcal{D}\}$.

Sensors are data providers. There are many different types of sensors. 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. We define the set of sensor types as $ST = \{st = (\mathcal{A}_{st},\ \mathcal{D}_{st}) \mid \mathcal{A}_{st} \subset \mathcal{A},\ \mathcal{D}_{st} \in \mathcal{D}\}$, where $\mathcal{D}_{st}$ is the measurement data type and $\mathcal{A}_{st}$ defines a set of attributes.

A *sensor* is defined using a sensor type and a set of attributes. Therefore, the set of sensors can be defined as $S = \{s = (ST_s,\ \mathcal{A}_s) \mid ST_s \in\ ST,\ \mathcal{A}_s \subset \mathcal{A}\}$, where ST_s is the sensor type such that $s : ST_s$ and $\mathcal{A}_s$ is a set of attributes.

Connectors transmit data between sensors and listeners. The set of connectors CN can be defined as $CN = \{cn = (\mathcal{A}_{cn},\ \mathcal{CCM},\ \mathcal{CP}) \mid \mathcal{A}_{cn} \subset \mathcal{A}\}$, where $\mathcal{CCM}$ defines a connection method and $\mathcal{CP}$ defines a communication protocol. The connection method and the communication protocol are enumeration types.

A *translator* is responsible for translating data from one data type to another understood by the system. The list of translators are defined as $TR = \{tr = (\mathcal{D}_i,\ \mathcal{D}_o,\ translate) \mid \mathcal{D}_i , \mathcal{D}_o \in \mathcal{D}\}$, where $\mathcal{D}_i$ is a source data type and $\mathcal{D}_o$ is a destination data type, and $translate : \mathcal{D}_i \rightarrow \mathcal{D}_o$ is a function that translates a data of type $\mathcal{D}_i$ to a data of type $\mathcal{D}_o$.

A *Verifier* is responsible for verifying the correctness and validity of sensor's data using data policies. The list of verifiers is defined as $V = \{v = (\mathcal{A}_v,\ S_v,\ \mathcal{D}_{S_v},\ \chi_{\mathcal{D}_{S_v}},\ verify) \mid \mathcal{A}_v \subset \mathcal{A},\ S_v \in S,\ \mathcal{D}_{S_v} \in \mathcal{D}\}$, where $\mathcal{D}_{S_v}$ denotes the output data of the sensor S_v, $\chi_{\mathcal{D}_{S_v}}$ is a data policy defined over the sensor's measurement data, $verify :\ S \times \chi_{\mathcal{D}_S} \rightarrow Boolean$ is a function that validates the correctness of a sensor's reading using a data policy.

Sensor listeners are responsible for managing sensor communications. It uses a connector and a translator to communicate with sensors. A listener is defined for each sensor type. The set of sensor listeners is defined as $L = \{l = (ST_l,\ TR_l,\ CN_l,\ V_l,\ \mathcal{A}_l) \mid ST_l \in ST,\ TR_l \in TR,\ CN_l \in CN,\ V_l \in V,\ \mathcal{A}_l \subset \mathcal{A}\}$, where ST_l is a sensor type which the sensor listener can communicate with, TR_l is a translator which is used to translate sensor data, CN_l is a connector used to connect to a sensor, V_l is a verifier which is used to verify sensor data, and $\mathcal{A}_l$ is the set of attributes associated with the sensor listener.

A *sensor mechanism* is defined as $SM = (ST,\ S,\ CN,\ TR,\ V,\ L)$.

4.2 Context Mechanism

The following formal definition of context is taken from [WAN06]. The set of dimensions and the domain of values for each dimension are fixed before constructing contexts. Let $DIM = \{D_1, D_2, \ldots, D_n\}$ denote a finite set of dimensions, and X_i be the tag set associated with $D_i \in DIM$. Let C denote the set of contexts such that the concrete syntax of a context definition is $c = [D_{i_1} : x_{i_1}, \ldots, D_{i_n} : x_{i_n}]$, 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. It is the application that suggest the set of dimensions. 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 *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.

A *situation* represents a set of contexts which satisfy certain conditions. A situation expression language SEL is used to specify the valid conditions in which a situation holds. The language, also, describes the resulting situation information based on context information. The set of situations is defined as $U = \{u = (C_u,\ SEL_u) \mid C_u \subseteq C\}$ where SEL_u is an expression specified using SEL. Details of the situation expression language will be explained later in this chapter.

A *context reasoner* is responsible for reasoning about situation definitions against context information. It uses a translator to translate situation from one type to another. The set of context reasoners is defined as $R = \{r = (C,\ U,\ TR_r,\ reason) \mid TR_r \in TR\}$ where $reason : \mathbb{P}C \times \mathbb{P}U \rightarrow \mathbb{P}U$ is a function that reasons a set of situations against a set of context and returns the set of situations that are satisfied by the contexts.

A context mechanism is defined as $CM = (C,\ U,\ R)$

4.3 Adaptation Mechanism

Adaptation is a workflow of actions and policies in response to a non empty set of situations. The set of adaptations is defined as $P = \{p = (U_p, \mathcal{W}_p,\ \mathcal{A}_p) \mid U_p \subseteq U,\ \mathcal{A}_p \subset \mathcal{A}\ \}$, where U_p is a set of situations that cause an adaptation and $\mathcal{W}_p$ is a workflow expression that describes the execution of actions. The workflow expression language will be explained later in this chapter.

We define the function $resolve :\ \mathbb{P}U \rightarrow P$ that resolves the required adaptation in response to a set of situations of current context.

Let $AN = \{\mathcal{D}_a \mid \mathcal{D}_a \subset \mathcal{D}\}$ denote the set of actions where an action is an event which denotes an information flow from the system to its environment. An event carries a set of parameters where each parameter has a data type. We define a policy checker function $check :\ \mathcal{W} \rightarrow \mathbb{P}AN$ which returns only the actions in a workflow which satisfy execution policies. We define an adaptation execution function as $execute :\ P \rightarrow \mathbb{P}AN$ that takes an adaptation and defines a set of actions.

An adaptation mechanism is defined as $AM = (P,\ AN,\ resolve,\ check,\ execute)$.

4.4 Reactivity Mechanism

Actuators perform actions. There are many different types of actuators. An actuator type is characterized by a set of attributes and data parameters that represents the required input information necessary for performing actions. We define the set of actuator types as $AT = \{at = (\mathcal{A}_{at},\ \mathcal{D}_{at}) \mid \mathcal{A}_{st} \subset \mathcal{A},\ \mathcal{D}_{at} \subset \mathcal{D}\}$.

An actuator is defined using an actuator type, actions, and a set of attributes. Therefore, the set of actuators can be defined as $AC = \{ac = (AT_{ac},\ \mathcal{AN}_{ac},\ \mathcal{A}_{ac}) \mid AT_{ac} \in\ AT,\ \mathcal{AN}_{ac} \subset AN,\ \mathcal{A}_{ac} \subset \mathcal{A}\}$, where AT_{ac} is the actuator type such that $ac : AT_{ac}$.

Actuator configuration holds setting information for actuators. It is defined using a set

of attributes. The set of actuator configurations can be defined as $F = \{f = (AC_f,\ \mathcal{A}_f) \mid AC_f \in AC,\ \mathcal{A}_f \subset \mathcal{A}\}$.

Actuator controller is responsible for managing communications with actuators. It uses a connector and a translator to communicate with an actuator. A controller is defined for each actuator type. The set of actuator controller is defined as $O = \{o = (AT_o,\ F_o,\ TR_o,\ CN_o,\ \mathcal{A}_o) \mid AT_o \in AT,\ F_o \in F,\ TR_o \in TR,\ CN_o \in CN,\ \mathcal{A}_o \subset \mathcal{A}\}$.

A reactivity mechanism is defined as $RM = (AT,\ AC,\ AN,\ F,\ O,\ CN,\ TR)$

4.5 Context-Aware System Formal Model

Based on the formal definitions presented in the previous sections, we are able to formally define a context-aware system as follows.

Definition 1 *A context-aware system is defined as $CAS = (SM, CM, AM, RM)$ where SM is a sensor mechanism, CM is a context mechanism, AM is adaptation mechanism, and RM is reactivity mechanism.*

4.6 Situation Expression Language

The *Situation Expression Language* defines the syntactic structure of situation expressions. It allows defining situations based on logical and mathematical operations over dimensions, tag values and other situations. The language supports the following operations.

- Logical *AND*, *OR*, & *NOT* operations between situations.
- *Equal, Not Equal, Bigger, Smaller, Bigger or equal, & Smaller or equal* between dimensions.
- *Logical AND, OR, & NOT* operation between dimension expressions.
- The basic arithmetic operations, namely *Addition, Subtraction, Division, and Multiplication* defined on numeric tag values.

The situation expression language is defined using a context free grammar (CFG) as follows.

The root grammar in this CFG is called Situation. Situation should be defined between curly braces {}. Situation can be either a *Hybrid Situation* which contains a situation expression over dimensions and other situations, or a *Literal Situation* which contains a situation expression over dimensions only. The root grammar is presented as follows.

$$<\mathbf{S}ituation> ::= \{<\mathbf{S}ituationRule>\} | \ <\mathbf{L}iteralExpression>;$$

$$\begin{aligned} < SituationRule > ::= & < ANDSituationRule > | \\ & < ORSituationRule > | \\ & < NOTSituationRule > | \\ & < LiteralExpression > | \\ & < SituationToken > \end{aligned}$$

The logical operations used in the situation expression language are AND, OR, & NOT. The operations are defined recursively over the *Situation Rule* notation as follows.

$$\begin{aligned} < ANDSituationRule > ::= & < SituationRule > AND < SituationRule > \\ < ORSituationRule > ::= & < Situation > OR < SituationRule > \\ < NOTSituationRule > ::= & \ NOT < SituationRule > \end{aligned}$$

Example 2 *Nice Weather = { Warm AND Sunny } shows an expression with a logical operator.*

Context Free Grammars contain either terminal or non-terminal rules. Non-terminal rules contain other rules, whereas terminal rules are rules that match a token presented as a

regular expression. All rules eventually end up with terminals. All leafs in the expression tree are terminals.

As mentioned before, a situation could depend on other situations. In order to represent that in the Situation Expression Language, situations are identified with their names, and consequently are referenced by them. *SituationToken* is an identifier representing a situation name.

$$<SituationToken> \ ::= \ Identifier$$

Example 3 *Go Out* = { *NOT Stay Home* }, *where Stay Home is a SituationToken.*

Literal Situation is the mechanism used to define a dimension expression inside a Situation expression. A situation can be a simple situation which contains only dimension expressions in its definition. For example, *Hot Weather* situation is defined as { (Temperature > 30) }. This means that *Hot Weather* is a simple situation. Literal Situations can be used to define anonymous situations in the body of other situations. In Example 4, { (Role == 'Admin') } is a *Literal Situation*, however that could be presented differently by explicitly defining a *Role Situation* and then referencing it in the *Admin* definition.

Example 4 *Admin* = { *Authenticated AND* { *(Role == 'Admin')* } }

$$<\mathbf{L}iteralExpression> :: =\{<Dimension>\}$$

Dimension Root represents the operations on context dimensions. These operations evaluate to either true or false, which determines if a situation exists in a given context or not.

$$<\mathbf{D}imension> \ :: = (<DimensionRule>)$$

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 contains all the following operations.

1. Logical Operations: *AND, OR, NOT, Equal, NOT Equal, Bigger, Smaller, Bigger or Equal, Smaller*, and *Equal*.

2. Mathematical Operations: *Addition, Multiplication, Division,* and *Subtraction*.

3. User defined functions: They contain domain specific logic constructed to perform operations over dimensions. For example, users can define a function to check if a specific dimension value is a prime number.

$$
\begin{aligned}
< DimensionRule >:: & < BraceDimension > | \\
& < ANDDimensionRule > | \\
& < ORDimensionRule > | \\
& < FUNCDimensionRule > | \\
& < NOTDimensionRule > | \\
& < ADDDimensionRule > | \\
& < DIVDimensionRule > | \\
& < SUBDimensionRule > | \\
& < MULDimensionRule > | \\
& < EqualDimensionRule > | \\
& < NotEqualDimensionRule > | \\
& < BiggerDimensionRule > | \\
& < BiggerOrEqualDimensionRule > | \\
& < SmallerDimensionRule > | \\
& < SmallerOrEqualDimensionRule > | \\
& < TokenDimensionRule > | \\
& < DimensionValue >
\end{aligned}
$$

The situation expression language includes logical dimensions operations. Logical operators at the dimension level are richer than their counterpart at the Situation level. In addition to AND, OR & NOT, compare operators can be used over dimensions since they

have concrete values. Following is an example of the operations definitions.

$$< ANDDimensionRule > ::=< DimensionRule >$$
$$AND < DimensionRule >$$
$$< BiggerOrEqualDimensionRule > ::=< DimensionRule >$$
$$' >=' < DimensionRule >$$

The situation expression language includes mathematical dimension operations. Mathematical operations are used over dimension values or the values supplied by users in the expression. All the four basic operations are supported.

$$<DIVDimensionRule> ::= < DimensionRule > "/" < DimensionRule >$$

Example 5 *Hot Celsius*= { *(Fahrenheit Temperature - 32) > 30)* }

The situation expression language provides an extension point to bring user supplied logic to the reasoning operation. User defined functions provide a mechanism to extend the situation expression language. Functions should have one or more parameters which are either dimension values (tags) or user supplied values. Functions should return a Value. Syntactically, functions should start with an **$** sign and the parameters should be in square braces **[]**. Users 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. The following example illustrates user defined functions.

Example 6 *Even Number* = { *(**$EvenNumber[** 56, Dimension Name **]**)* }

The situation expression language includes dimension terminals. Terminals, in case of dimensions, are either dimension name (identifier) or a dimension value (number or string). The following example shows a situation expression with the dimension name Temperature.

$$<\mathbf{D}imensionValue> ::= < NUMBER > \mid < STRING >$$

Example 7 *Freezing* = { *(Temperature =< **0**)* }

The full documentation of the context-free grammars of the *Situation Expression Language* is provided in Appendix A.

4.7 Workflow and Policy Expression Language

Adaptations contain a workflow of reactions that should be executed with respect to a set of policies. The *Workflow and Policy Expression Language* is used to define the adaptations. This language supports the following operations.

- Triggering reactions.
- Checking policies.
- Logical Operations (*AND*, *OR*, & *NOT*). and
- Control constructs (*WHILE*, *IF ELSE*, and *FOR*) to allow rich workflow expressions.

The workflow and policy expression language is defined using context free grammar as follows.

The root rule for the language is the Workflow rule. The workflow is simply a statement collection.

$$< Workflow > ::= < StatementCollection >$$

The statement collection is a recursive rule (star rule) over the statement rule. Which means that each adaptation can contain one statement or more.

$$< StatementCollection > ::= < StatementCollection >< Statement > \mid$$
$$< Statement >$$

The Statement rule is the main bulk of the workflow language. The Workflow Expression Language supports the following statements.

- *IF ELSE* Statement,
- *While* Statement,

- *For* Statement,
- *Execute* Statement and
- *Brace Statement*: It represents that a statement can be encapsulated in braces to enforce operator precedence.

The definition of the Statement rule is presented as follows.

$$\begin{aligned} < Statement > ::= & < WhileStatement > | \\ & < ForStatement > | \\ & < IfElseStatement > | \\ & < ExecuteStatement > | \\ & < BraceStatement > \end{aligned}$$

The Condition Statement is used as a part of the *IF Statement* and *While Statement*. Conditions are either a policy check or a logical expression over other conditions.

$$\begin{aligned} < Condition > ::= & < Condition > | \\ & < ANDCondition > | \\ & < ORCondition > | \\ & < NOTCondition > | \\ & < PolicyCheck > \end{aligned}$$

The logical condition statement is a logical aggregation of other statements. The following shows the definitions of these rules.

$$\begin{aligned} < ANDCondition > ::= & < Condition > AND < Condition > \\ < ORCondition > ::= & < Condition > OR < Condition > \\ < NotCondition > ::= & NOT < Condition > \end{aligned}$$

Example 8 *$IsAuthorized[User ID] AND $IsInRole[User ID]*

The policy check statement is the mechanism used to check execution policies in the workflow. 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. Policy Name is a Terminal rule that matches string identifier which starts with an **$** sign to eliminate ambiguity when parsing the language.

$$< PolicyCheck >::= < PolicyName > [< ParamList >]|$$
$$< PolicyName > [\ ]$$

Example 9 *$IsAuthorized[User ID]*

The *While Statement* is used to repeatedly execute a set of actions as long as a condition is met. The While Statement contains two parts (1) a condition, and (2) a body which is a statement.

$$< WhileStatement >::= while < Condition >< Statement >$$

The For Statement is used to repeatedly execute a set of actions for a fixed number of times. The For Statement contains two parts (1) a number indicating the number of iterations, and (2) a body, which is a statement.

$$< ForStatement >::= for(< NUMBER >) < Statement >$$

The *If statement* is used to execute different branches based on a condition. The *IF Statement* is a typical example of a shift-reduce conflict, the preferred behavior in this case is the shift (whenever the parser encounters an “else” the parser should shift rather than reduce

with the IF rule).

$$< IfElseStatement > ::= < IfStatement > \mid$$
$$< IfStatement > else < Statement > *prefered$$

$$< IfStatement > ::= if < Condition >< Statement >$$

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. Reaction Name is a terminal rule that matches a string Identifier.

$$< ExecuteStatement > ::= \ Exec\,(\, < ReactionName >)\,;$$

$$< Reaction > ::= < ReactionName > [\, < ParamList >] \mid$$
$$< ReactionName > \ [\,]$$

The *Param List* is a plus closure rule that matches one parameter or more, each parameter is either a string value in single quotation, or an integer value.

$$< ParamList > ::= < ParamList > , < Param > \mid$$
$$< Param >$$

The full documentation of the context-free grammars of the *Workflow Expression Language* is provided in Appendix B.

Chapter 5

Architectural Design

In this chapter the context-aware architecture proposed in [WAP06] is first reviewed. Next, its merits and inadequacies are discussed. Finally, a new improved component-based architecture built around the formal trustworthy components [MA11] is proposed for context-aware systems.

The primary goal of a software architecture is to define software building blocks and their relationships. It defines the blue-print based on which a system should be developed. Component-based development methodology (CBD) for developing context-aware systems promises many advantages including adaptive reuse, containing complexity induced by mobility, and reducing development time. A comprehensive survey in [MA11] identifies a multitude of advantages of CBD methodology, especially for embedded systems deployed in safety critical environments. They have shown with case studies how their formal approach effectively manages complexity and promotes a formal analysis. Because of its underpinning formalism to construct trustworthy component-based systems, it is a suitable architectural style to follow for building a context-aware platform. The results of this chapter come out of significant improvements made to the three-tiered architecture [WAP06] through extensions, generality, and adaptation of trustworthy component development methodology [MA11].

5.1 A Quick Review of Past and Present Architectures for Context-aware Systems

A three-tiered model was introduced in [WAP06] as a solution to contain the complexity created by the heterogeneity problem in context-aware systems. The model is illustrated in Figure12. The functionality for each tier and the nature of information flow across the three tiers were proposed. The architecture was founded on the three-tiered formalism, however not fully realized.

The three tiers separately dealt with perception, context modification, and adaptation. Tier 1 is a description of "*see*, *gather*, *control*, and *modify*" features of perception abstractions. Perception will involve the objects perceived, the devices used for observing the objects, and the observational measurements. We emphasize that information related to a user in the environment was assumed to be either conveyed directly by the user to one of the devices or to be perceived automatically by some devices. This aspect was not made quite clear. Tier 1, which describes the assembled *symbolic* representations of the observations, will notify it to Tier 2. The exact structure of this representation and the semantics for translating it to the internal representation was not specified. Tier 2, after receiving the notification from Tier 1, will construct internal context representations that reflect the current awareness. The formal basis of Tier 2 functionality is the context theory in [WAP06]. Tier 2 will use context calculus provided by the theory to construct general contexts, de-construct and modify the contexts as might be demanded by the application. Tier 2 will notify current context information to Tier 3. Tier 3, after receiving current context from Tier 2 will determine how the system has to adapt itself. The system, modeled as an *Extended State Machine* exists only in Tier 3 and is supposed to interact with Tier 1 in order to convey reactions and with Tier 2 in order to exchange modified context information. However, no details on the relationship of reactions to adaptations, and adaptation policies were discussed. However, a model of *Anti-lock Braking System* was discussed in Tier 3 to explain a specific adaptation for that application. This work produced a coarse architecture using components, but failed to describe the architectural elements and their

specifications. In fact, the architecture looked like a 'tightly-coupled' system, whereas in practice a context-aware system is supposed to work in distributed and mobile environment. Yet, the distinguishing features of this approach are their resort to formalism for analysis and components for design.

The architecture proposed in this thesis 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. The architecture is portable to different platforms, extendable to new emerging devices, and can be used for a wide variety of applications ranging from business to defense systems.

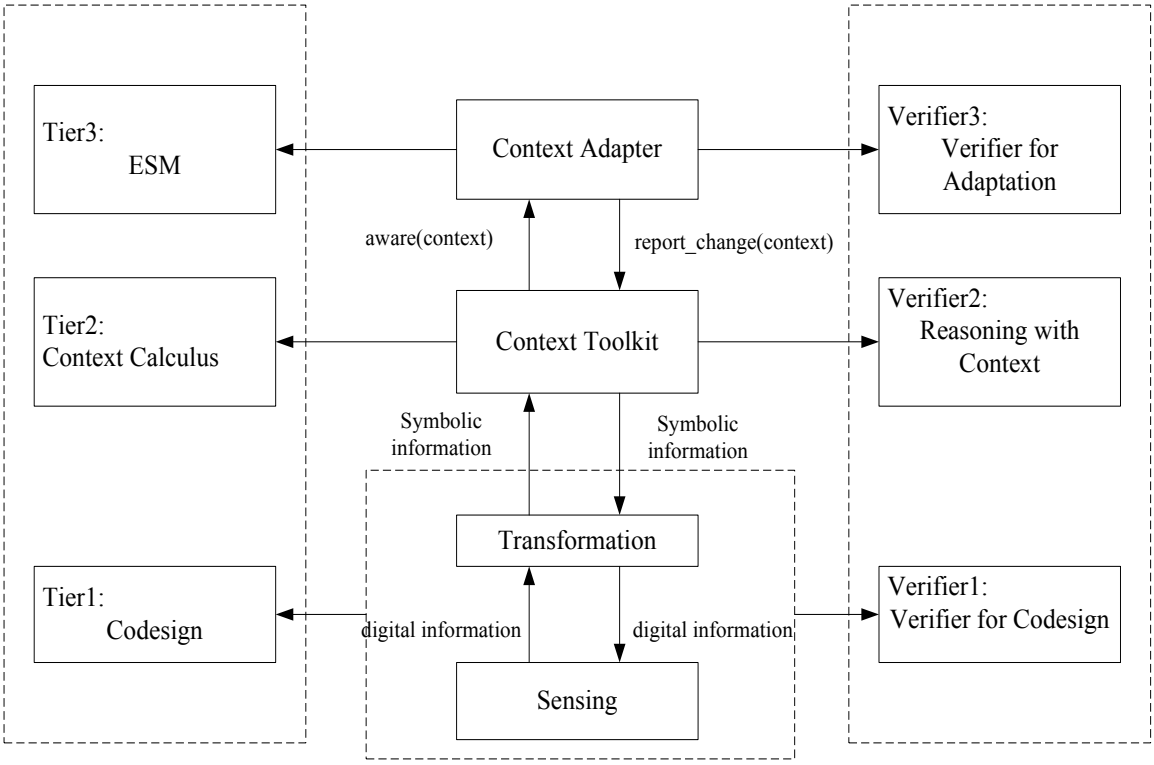

Figure 12: The Three-Tiered Formalism of Context-Aware System [WAP06]

5.2 Proposed Architecture

Figure 13 shows our proposed architecture for context-aware framework. The architecture retains the spirit of 'separation of concerns', and hence the original three-tiered structure is preserved, while its internals are enriched. A new architectural element *Data Store (DS)* is added to the architecture. Essentially $T1$, the new *Sensor Mechanism*, corresponds to Tier 1, $T2$, the new *Context Mechanism*, corresponds to Tier 2, and $T3$, the new *Adaptation Mechanism*, and *Reactivity Mechanism*, correspond to Tier 3. This comparison is just to illustrate the extent of similarity between the new and old architectures. However, they vary greatly in detail.

In principle, the new architecture should be viewed to consist of four essential modules *Sensor Mechanism*, *Context Mechanism*, *Adaptation Mechanism*, and *Reactivity Mechanism*, each implementing an essential mechanism. A module defines a package of components that are grouped together. Each module contains a set of components that interact with each other. Hence, the architecture clearly abstracts, and loosely couples context sensing, building awareness, deciding adaptations, and reacting to the environment. This loose coupling is more suitable for implementing a wide variety of context-aware applications. The following sections describe in detail the proposed architecture and its modules, components, and the relationships between components.

5.3 Architecture

Context-aware System environment consists of a set of entities, where an entity is a person, place, object, etc. Sensor mechanism 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, position, and identity of a person. Context mechanism is responsible for combining related events and data, once they are sensed, to build awareness, which will assist the system to perform the appropriate adaptation. Adaptation mechanism 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. The resulting reactions are regulated by policies. Reactivity mechanism is responsible for performing the reactions and adaptations in the environment by controlling physical devices, actuators, or displaying results on hardware interfaces. For example, raising or lowering the temperature by adjusting the thermostats controller in a room is a possible reaction to the event "person entering a room". A detailed discussion of the architectural elements is provided below. To sum up, the novel features are separation of concerns, isolating the interaction of components for achieving a specific task, allowing interactions between different types of functionalities subject to context relations, and regulating adaptations based on specific policies. To the best of knowledge, no previous work has achieved all these results.

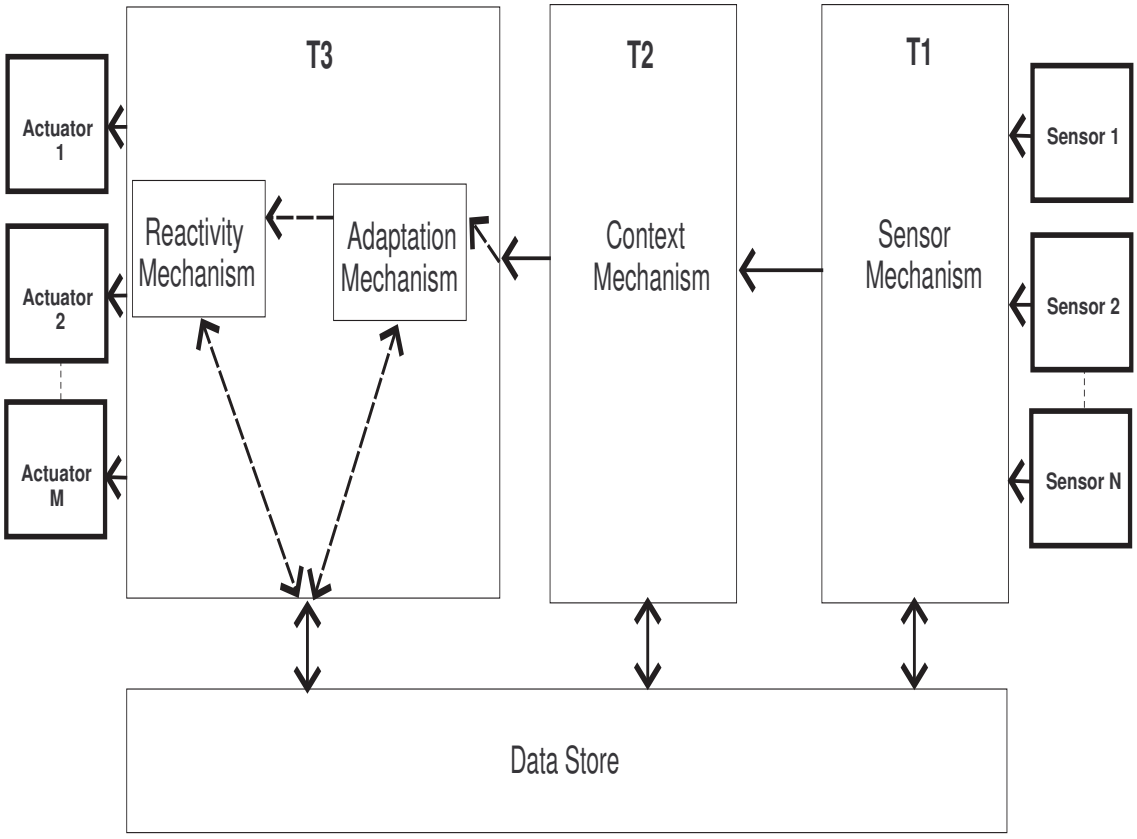

Figure 13: The CAF Architecture

5.4 Sensor Mechanism (SM)

The sensor mechanism SM, illustrated in Figure 14, comprises *sensor*, *connector*, *listener*, *translator*, *verifier*, and *data synchronizer* elements. 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, when a person enters the room, a stimulus event is created. Also, when the value of the current temperature in the room changes then a stimulus is created. A stimulus may come with data parameters. 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*, a typed data value. 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 is a subsystem that contains hardware and software components. In this thesis, 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.

For every sensor, the system defines a *listener*. Listeners form a level of abstraction between sensors and context-aware systems. Listeners are software components that monitor continuously the activities of sensors and subscribe to any event triggered by sensors. Listeners may use different connection and communication methods to interact with sensors. A *connector* implements the communication method through which the data will be communicated from its source to the context mechanism. There are many possible implementations to connectors such as method call, remote procedure call, SOAP, etc. The selection of the appropriate implementation depends on the deployment specification of a system. For example, if a sensor and a context mechanism are deployed on the same machine and loaded into the same application domain, normal method invocation could be

used. However, if a sensor is deployed on another application domain on the same machine or on another machine then there is a need to establish a communication channel using a protocol such as TCP or HTTP. Therefore, there is a connector defined for every sensor-listener relationship.

Sensors collect raw data which may need interpretation and translation into formats understandable by the system. Therefore, a translator is used to perform the translation process. When a listener receives a sensor's raw data, it looks up the appropriate *translator* for sensor's data type and performs translation. Then it passes the data to a *verifier*. A verifier could be assigned to verify the correctness of sensory readings using data policies. Data policies are constraints on sensors' data. For example, a range constraint such as "a temperature should not be less than -60 degrees and not more than 70 degrees" could define a data policy. Another type of data constraint is data validity. For example, weather forecast information could be valid for up to one day and GPS information could be valid for only a few minutes. Every type of sensors can have one defined verifier. The associations between sensors and verifiers are defined in a system configuration setting that is stored and managed in the data store DS.

A tag value for a dimension of a context might come from different sensors through different listeners and verifiers. *Data synchronizer* aggregates dimension information from all sensors that are associated with a dimension and keeps only the latest received value. Associations between sensors and dimensions are defined in a system configuration setting that is stored and managed in the data store DS. The data synchronizer informs the context mechanism whenever a new sensor data is received. Also, whenever the context mechanism requires sensor data, it requests it from the data synchronizer. Therefore, the data synchronizer is the interface component of the sensor mechanism.
In summary, the sensor mechanism is responsible for the following.

1. Communicating with sensors,

2. Translating output of sensors to a format that can be understood and processed by a system,

3. Requesting data on demand from sensors and informing interested entities when new readings are submitted by any sensor,

4. Verifying sensor data using data policies,

5. Aggregating sensor data from different sources. Rules should be specified to help choose the best tag value in case there are multiple sources, and

6. Caching the latest value of every context dimension and providing it to the context mechanism upon request. Thus, it serves like a buffer in which the simple context tag values are stored. When any sensor sends a new gathered data then this component will replace the value after validating it.

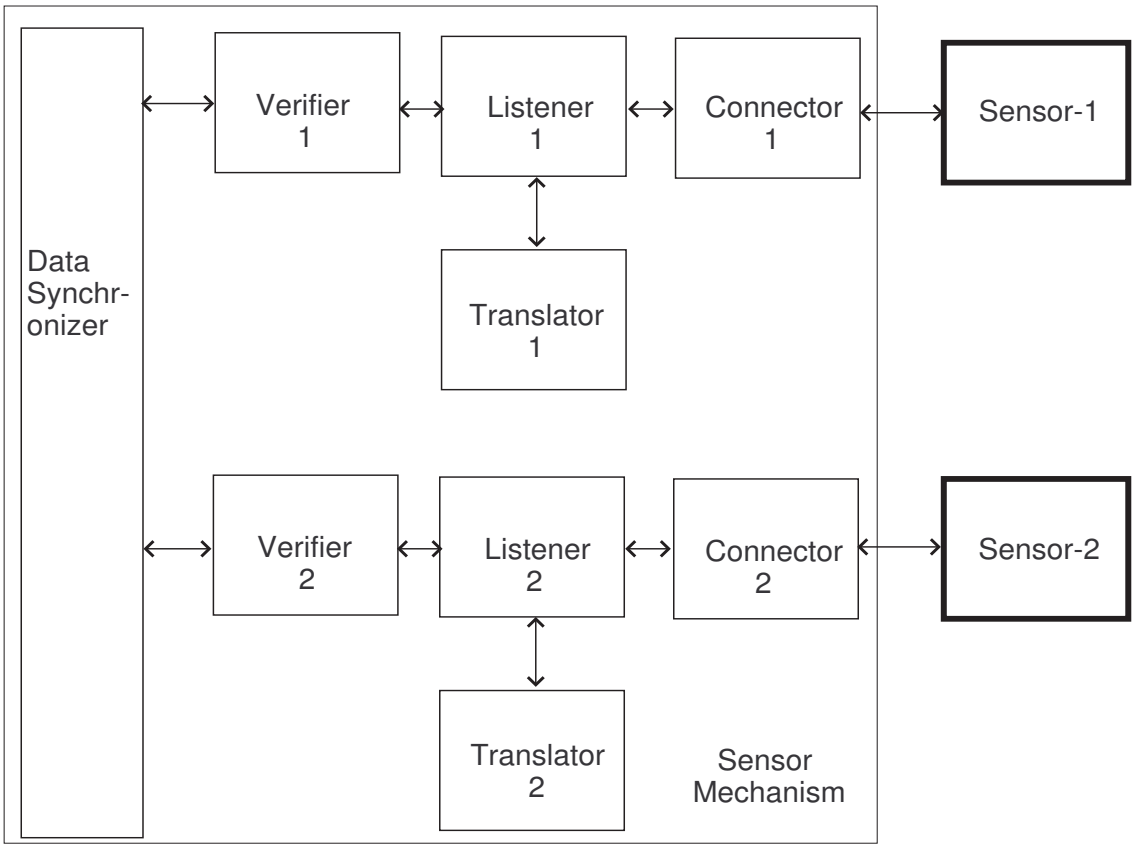

Figure 14: The Sensor Mechanism

5.5 Context Mechanism

The context mechanism, illustrated in Figure 15, consists of *dimension*, *context*, *situation*, *context manager*, *context translator*, and *reasoner* elements. Since the environment is in constant change, the context manager rebuilds contexts every time there is a change to one of the dimensions. Information about new changes come from the sensor mechanism. Therefore, continuously, the sensor mechanism triggers the context manager to rebuild contexts.

The context manager builds and updates contexts by aggregating sensors' readings. Then, the context manager uses the *reasoner* to identify the context *situations* that are applicable to the current context. The reasoner uses situation expressions that are stored in a data store and tries to evaluate each expression against the current context. Consequently, the reasoner generates a set of situations that are inferred by the current context.

In the proposed architecture context situation can be implemented based on different theories. One possible way to implement a context situation is the Box notation [WAN06]. Another way to represent context situation is Ontology using Description Logic (DL). In this case a standard Ontology reasoner could be used to infer situations for a given context. Therefore, the context manager can use different types of reasoners. This is a significant improvement over previously known architectures. 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 reasoner. Thus the operations of the context manager are independent from the way context is specified or represented.
In summary, the context mechanism is responsible for the following.

1. Defining context and context situation,

2. Translating contexts and situations from different context theories, and

3. Evaluating contexts and reasoning about situations.

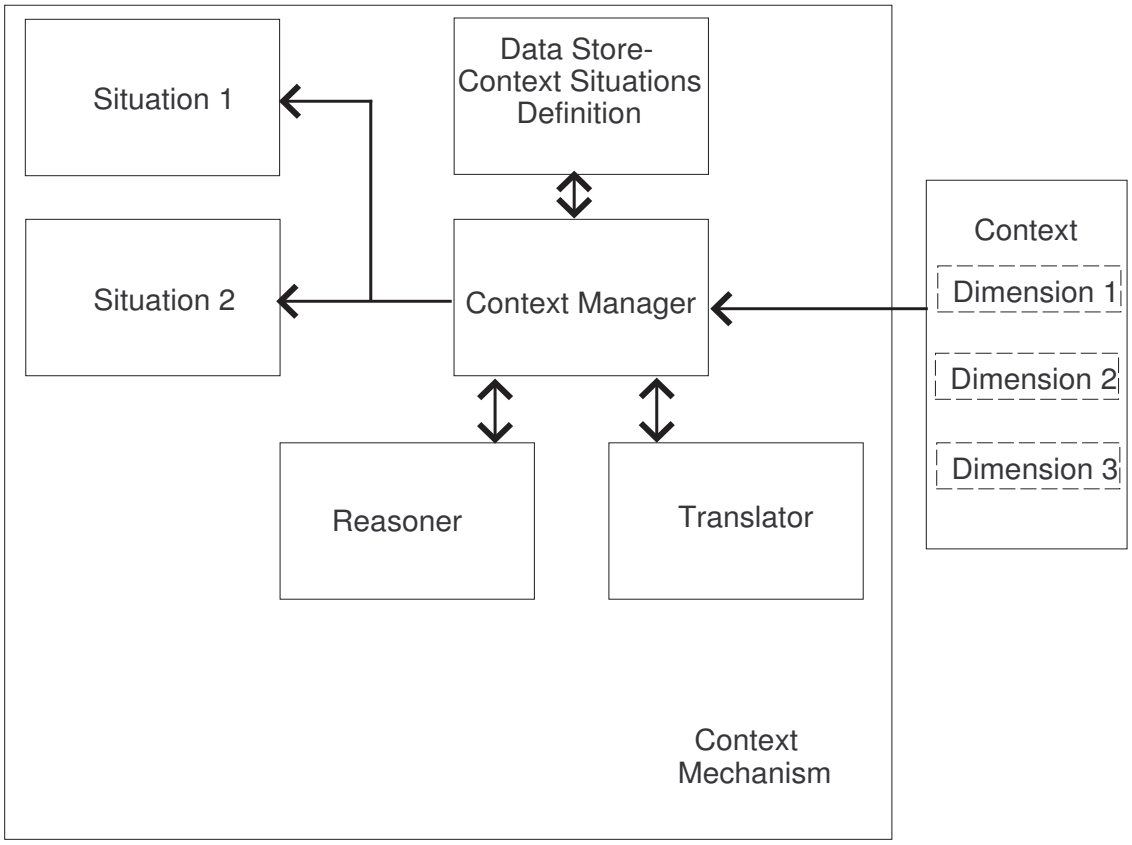

Figure 15: The Context Mechanism

5.6 Adaptation Mechanism

The adaptation mechanism, illustrated in Figure 16, includes *adaptation resolver*, *adaptation*, *workflow executer*, *policy checker*, and *reaction* elements. It is responsible for determining suitable reactions for context situations. When a situation is realized, its relevant adaptation is searched and selected by the *adaptation resolver*. Associations between situations and adaptations are defined and managed in the data store. A situation can have one or more possible adaptations. By knowing the situation, the adaptation resolver scans the list of adaptations to see which adaptation is related to the current situation. An adaptation is defined using a workflow expression language. The expression language defines a set of reactions that form the adaptation and define the sequencing in which the reactions

should be executed. Also, the expression language defines execution controls over the set of reactions. The execution controls include loops and conditions. Execution policies are used as conditions to control reactions. A policy is a constraint defined over the situation information that is associated with an adaptation. A policy is checked before selecting a *reaction*.

The *workflow executer* is an engine that is used to execute adaptations' workflow. It contains implementations for every construct in the workflow expression language. It takes as input a situation and an adaptation. Then 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.

Reactions are atomic system actions that could not be split any further. Reactions have the following properties.

- It should be executed with no dependencies on other reactions.
- It should perform one and only one functionality.
- It communicates with the outside world actors, namely the actuators.
- It 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.

In summary, the adaptation mechanism is responsible for the following.

1. Determining an appropriate adaptation for a special situation,
2. Verifying the execution polices of adaptations before executing,
3. Executing the corresponding workflow of reactions for a determined adaptation, and
4. Communicating with actuators and passing any necessary context information.

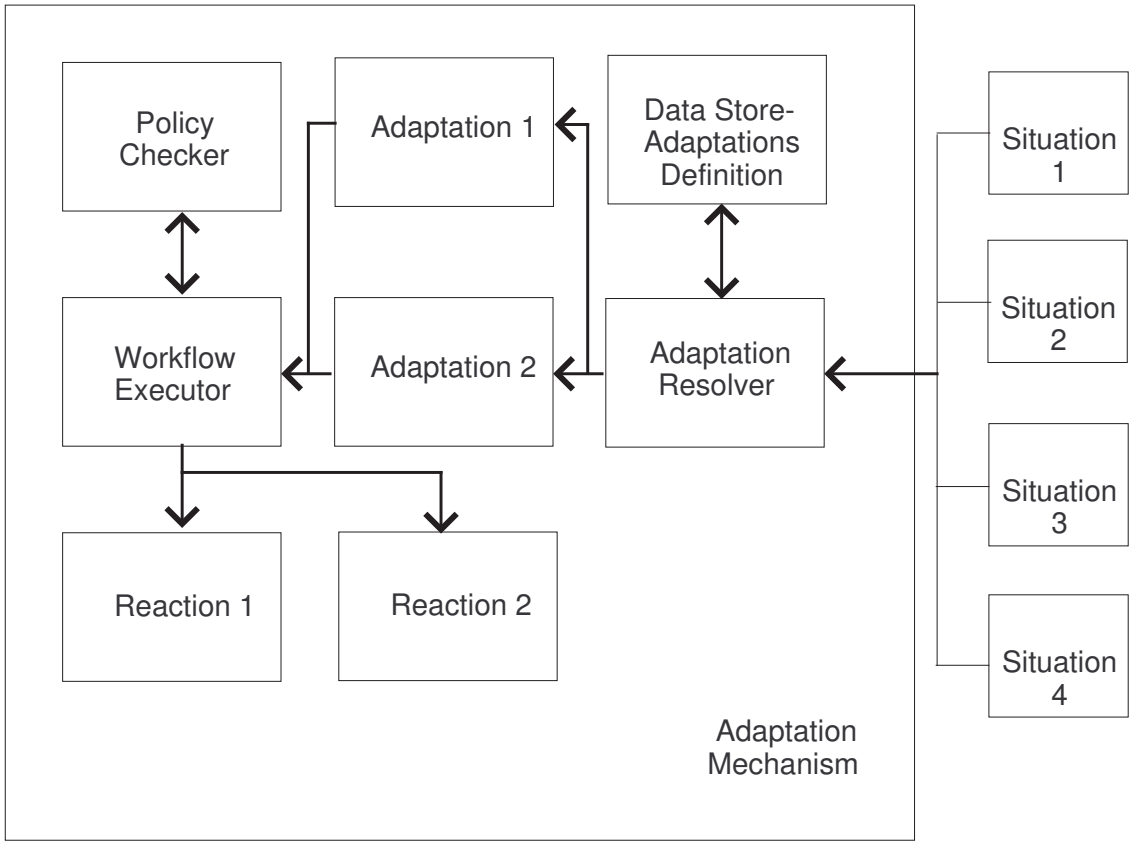

Figure 16: The Adaptation Mechanism

5.7 Reactivity Mechanism

The reactivity mechanism, illustrated in Figure 17, 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 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. Each controller is implemented for a specific actuator. A controller knows how to communicate to its corresponding actuator. The *actuator configuration* is used to specify any necessary configuration for an actuator. Configuration are abstracted from controllers. This allows using the same actuator to implement different actions based on different configurations. The door actuator can perform

open, close, lock and unlock actions through different configuration.

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.

In summary, the reactivity mechanism is responsible for the following.

1. Communicating with actuators,
2. Managing actuator configurations, and
3. Translating reaction information into formats that can be understood by actuators.

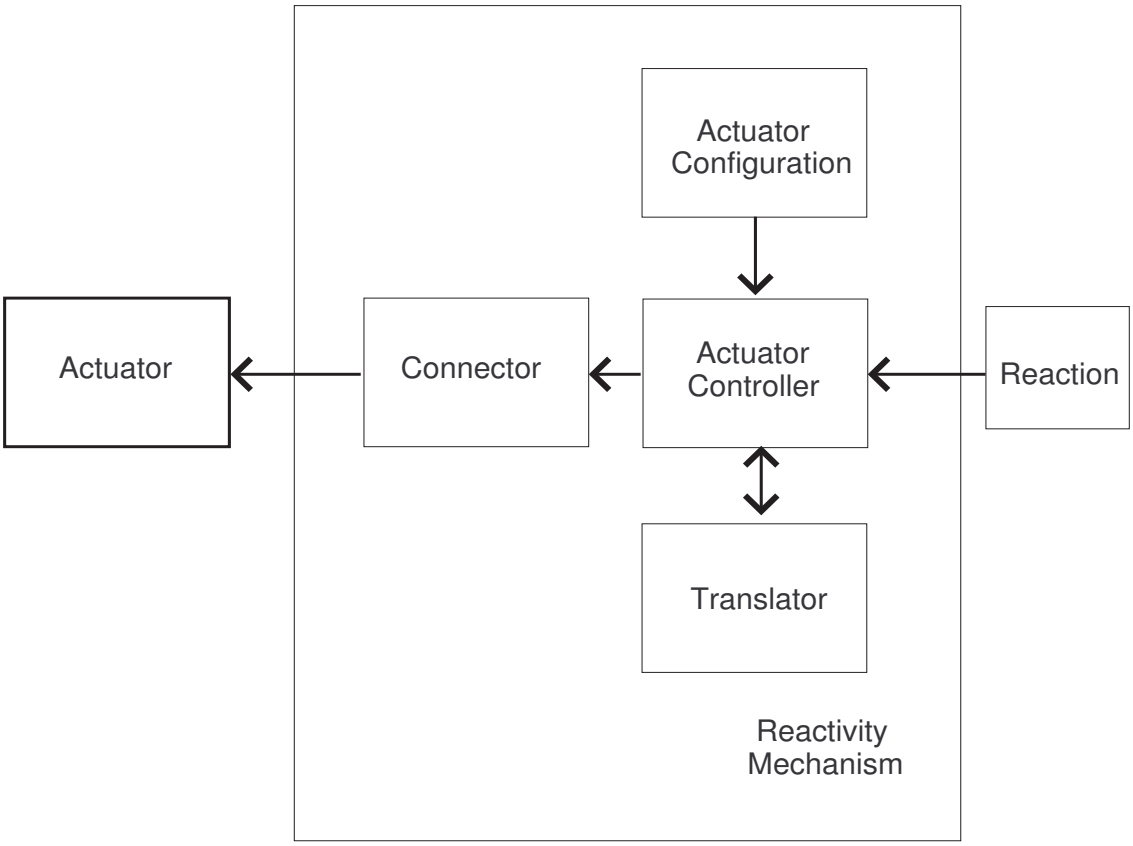

Figure 17: The Reactivity Mechanism

Chapter 6

Detailed Design

In this chapter the detailed design of the framework is presented. The Framework design is interface-driven, and the decision of constructing the main architecture as interfaces is taken to abstract the entities's properties and responsibilities from the actual implementation. This abstraction allows a greater flexibility in implementation. Below we review all the implementation components that comprise the modules described in Chapter 5. For each component all the interfaces and their subsequent properties and operations are discussed. The specific mechanisms used to parse expressions defined for context situations and adaptations workflow are also discussed.

6.1 Framework Module Components

The Framework contains the Sensor Module (illustrated in Figure 18), the Context Module (illustrated in Figure 19), and the Reactivity Module (illustrated in Figure 20). These modules consists of components, some of them shared across different modules. The design is an interface-driven design, where interfaces are used to allow maximum flexibility and to increase decoupling and modularity. Below, each component design and its responsibilities are discussed.

Core Component

The core component contains all general-purpose interfaces that are shared between different modules. All the interfaces in the core component are generic and can be used for different purposes. The elements in core component design are discussed below.

1. *IConnector* Interface: This defines the interface methods used to connect asynchronously to an external entity such as sensor or actuator. There are multiple possible implementations for *IConnector*, such as Database connector, or a Web service connector, or a Serial port connector, or an OS registry connector. The connector does not care *WHO*, *WHY* or *WHAT* it is connecting to, but rather it implements asynchronously methods to connect to a specific external entity. The interface *IConnector* implements the interface *INotifyChange*. The fields and methods that are part of *IConnector* definition are explained in Table 4.

2. *IConnectorConfigArgs* Interface: It is an empty interface which represents objects responsible for holding a specific configuration to connect to a specific type of connectors.

3. *ITranslator* Interface: It defines a method to translate data from one format to another. The interface is described in Table 5.

4. *INotifyChange* Interface: Following *Observer design pattern* [GE95], this interface is the standard mechanism used to notify observers with any data change. This interface has two basic usages. First, sensors can proactively inform the framework with a new reading. Second, it is used to enable asynchronous calls all across the framework. The interface is described in Table 6.

Field	Type	Description
Data	Property	The result data of the connection (Object)
Connect	Operation, input: IConnectorConfigArgs, output: void	To establish the connection and get the result back, returns void.
Logger	Property	ILogger value.

Table 4: IConnector Description

Field	Type	Description
Translate <*From, To*>	Operation, input: an object from a generic type *From*, output: an object from a generic type *To*	a generic method to translate data from one format to another
Logger	Property	ILogger value.

Table 5: ITranslator Description

5. *IData* Interface: It is the data type that holds sensors' data, which is a collection of a key-value pairs in which the key is a string, and the value is an object. The interface is described in Table 7.

6. *IExpression* Interface: This interface represents any generic expression that may be used in this framework. A generic expression is used to express objects such as workflow, policy, and situation. The interface is described in Table 8.

7. *IDataProvider* Interface: It represents the object responsible for interacting with a

generic data source. This data source can be either a situation data source or an adaptation data source. The interface is described in Table 9.

8. *ILogger* Interface: It represents the object responsible for logging the events in the framework. The logger can be either a console logger or file logger. The interface is described in Table 10.

Field	**Type**	**Description**
NotifyChange<T>	Operation, input: an object from a generic type T, output: void	Translate data from one format to another

Table 6: INotifyChange Description

Field	**Type**	**Description**
Data	Property	Key-Value List (Disctionary), key is string, value is object.

Table 7: IData Description

Field	**Type**	**Description**
Expression	Property	String value

Table 8: IExpression Description

Field	Type	Description
List Items<T>	Property	List of items from type T
GetItemsAsync	Operation, input: void, output: void	Get data from the data source asynchronously
Logger	Property	ILogger value.

Table 9: IDataProvider Description

Field	Type	Description
Log	Operation, input: string message, output: void.	Used by clients to log messages.

Table 10: ILogger Description

Listener Component

The Listener component contains the interface responsible for communicating with sensors. The elements of this component are discussed below.

1. *ISensorListener*: It represents the framework abstraction of sensors, and contains the properties of the sensor and its output. It implements the interface *INotifyChange*. The interface is described in Table 11.

2. *SensorType*: It is an enumeration type that represents all types of sensors. The reason why sensor type is hard-coded and not just simply a string is because we need to be aware of all sensor types in the design time to be able to implement a type-specific verifier.

Field	Type	Description
Data	Property	IData Value, the sensor's output
Type	Property	SensorType value, the type of the sensor that will be verified
Translator<object, IData>	Property	A translator responsible for translating data from object formate to IData which is the generic way of representing context information in the framework.
Connector	Property	IConnector value, the connector used to connect to the sensor.
ConnectorConfig	Property	IConnectorConfigArgs value, represents the connector specific configuration
Name	Property	String value.
GetDataAsync	Operation, input: void, output: void	To ask the sensor listener for an updated data.
Logger	Property	ILogger value

Table 11: ISensorListener Description

Aggregation and Verification Component

The Aggregation & Verification Component is illustrated in Figure 18. It is responsible for verifying sensors' output and aggregating the output of all sensors to generate the most up-to-date and consistent context. The component consists of the following elements.

1. *ISensorVerifier*: This represents a type-specific sensor verifier. Verifiers are responsible for executing data policies over sensors' data to make sure that the data is valid and up to date. Examples of possible verifiers are for weather monitoring, and location verification. Ideally, location verifier should be able to deal with different types of location sensors such as GPS, and IP address. Sensor Verifier implements the interface *INotifyChange*. The interface is described in Table 11.

2. *ISensorVerifiersManager*: This interface represents the objects responsible for mapping a collection of sensors to a collection of verifiers. The mapping is done by matching sensor types with verifiers types. The interface is described in Table 13.

Field	**Type**	**Description**
Data	Property	IData Value, the sensor's output
SensorType	Property	SensorType value, the type of the sensor
Name	Property	String value.
GetDataAsync	Operation, input: void, output: void	To ask the sensor listener for an updated data.
Logger	Property	ILogger value.

Table 12: ISensorVerifier Description

3. *IDataSynchronizer*: This unit is the facade interface for the Aggregation and Verification Component. It is responsible for interacting with all sensor verifiers. The main responsibility of this unit is to keep track of the verifiers in order to make sure that data is aggregated and synchronized before the context component is notified about the context change. The interface *IDataSynchronizer* is used as a buffer for sensors'

reading. Whenever a new context is constructed, this object is responsible for verifying that the cached value is up-to-date, otherwise it asks the sensor to provide an updated value. It implements the interface *INotifyChange*. The interface is described in Table 14.

4. *SensorAccuracy*: It is an enumeration data type that represents the accuracy of the sensor.

Field	**Type**	**Description**
SensorListeners	Property	List of Sensor Listeners
SensorVerifiers	Property	List of Sensor Verifiers
GetRequiredVerifiers	Operation: input: void, output: List of verifiers mapped to their corresponding sensors	For mapping each sensor with a verifier
Logger	Property	ILogger value.

Table 13: ISensorVerifiersManager Description

Field	**Type**	**Description**
Data	Property	IData Value, the *aggregated* sensor's output
Buffer	Property	IData Value, the temporary buffer used to aggregate sensor's output
SennsorVerifiersManager	Property	ISensorVerifiersManager value
Verifiers	Property	List of IVerifier
GetDataAsync	Operation: input: void, output: void	To ask the sensor verifiers for an updated data.
Logger	Property	ILogger value.

Table 14: IDataSynchronizer Description

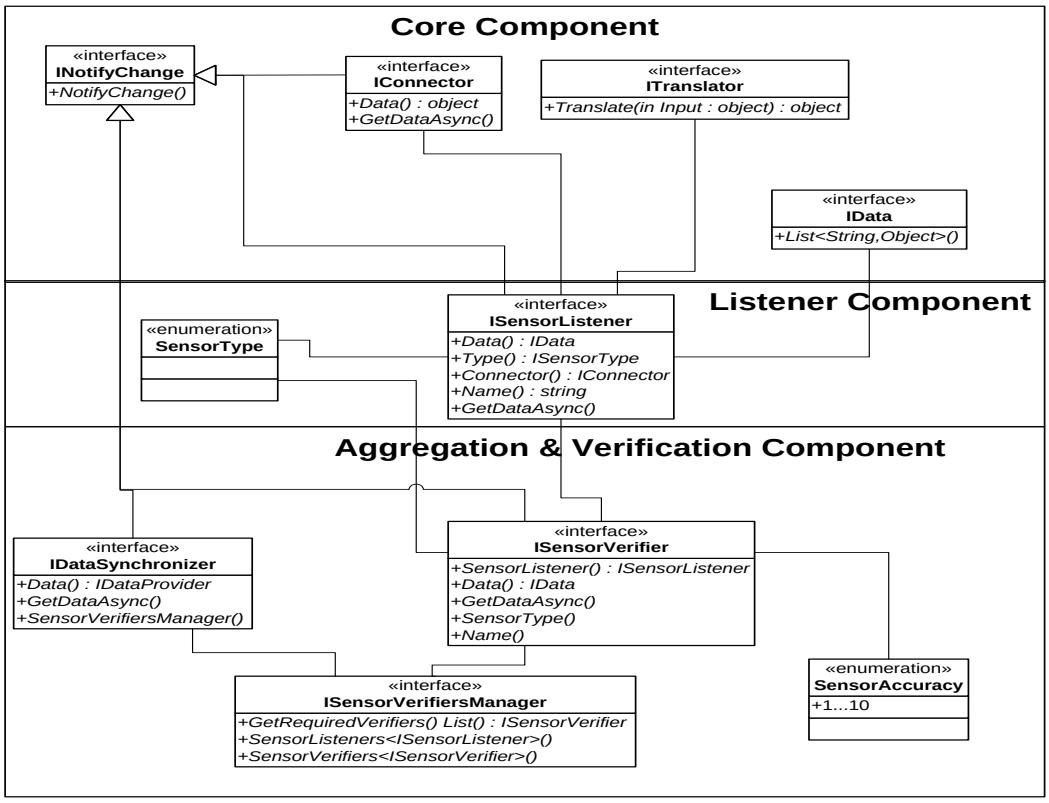

Figure 18: The Sensor Module

Context Calculus Component

Context component is responsible for inferring context situations which requires loading situation definitions and activating the reasoner whenever a new context in constructed. The component receives the context from the *Aggregation* component and notifies the *Resolving* component with the situations in the current context. The component interfaces are described below.

1. *IContextManager*: It is responsible for observing notification when a new context is calculated, loading the definition of situations from a Data Provider, activating the reasoner to reason against the predefined situations and notifying the *Resolving* component with the situations that exists in the current context. It is the facade interface

for the Context calculus component and it implements the interface *INotifyChange*. The interface is described in Table 15.

Field	**Type**	**Description**
DataSynchronizer	Property	IDataSynchronizer value
List<ISituation>	Property	The Situation exists in the current context
DataProvider	Property	IDataProvider value.
ContextReasoner	Property	IContextReasoner value.
GetSituationAsync	Operation, input: void, output: void	To ask the reasoner to evaluate the current context
Logger	Property	ILogger value.

Table 15: IContextManager Description

2. *IContextReasoner*: It is responsible for discovering existing situations in the current context. The reasoner evaluates the calculated context against a set of predefined situations. Context reasoners can be based on different theories. Consequently, different formats may be needed to present both context and situations. To serve that purpose the interface *IContextReasoner* uses *ITranslator* to translate context and situations from the framework generic format to the reasoner specific format. Reasoning is a time consuming operation, some technologies can take minutes and even hours to complete reasoning, such as ontology reasoners. In order to support all types of reasoners in our framework *IContextReasoner* implements the interface *INotifyChange* to enable asynchronously performing the reasoning operation. That would prevent reasoners from blocking other tasks done by the framework and would also provide

a better error handling mechanism.

3. *ISituation*: This interface is used to represent context situation in a generic way so it can be used across different reasoners. The interface is described in Table 15.

Field	**Type**	**Description**
List<ISituation>	Property	The Situation exists in the current context
ReasonAsync	Operation, input: IData value represents context and List of ISituation represents the list of predefined situations, output: void	To reason simple context against predefined situations
ContextTranslator	Property	ITraslator value. To translate context to the reasoner specific formate
SituationTranslator	Property	ITraslator value. To translate Situations to the reasoner specific formate
Logger	Property	ILogger value.

Table 16: IContextReasoner Description

Field	Type	Description
Name	Property	Situation Name
Expression	Property	IExpression value. That will be evaluated when testing if this situation exists in a given context

Table 17: ISituation Description

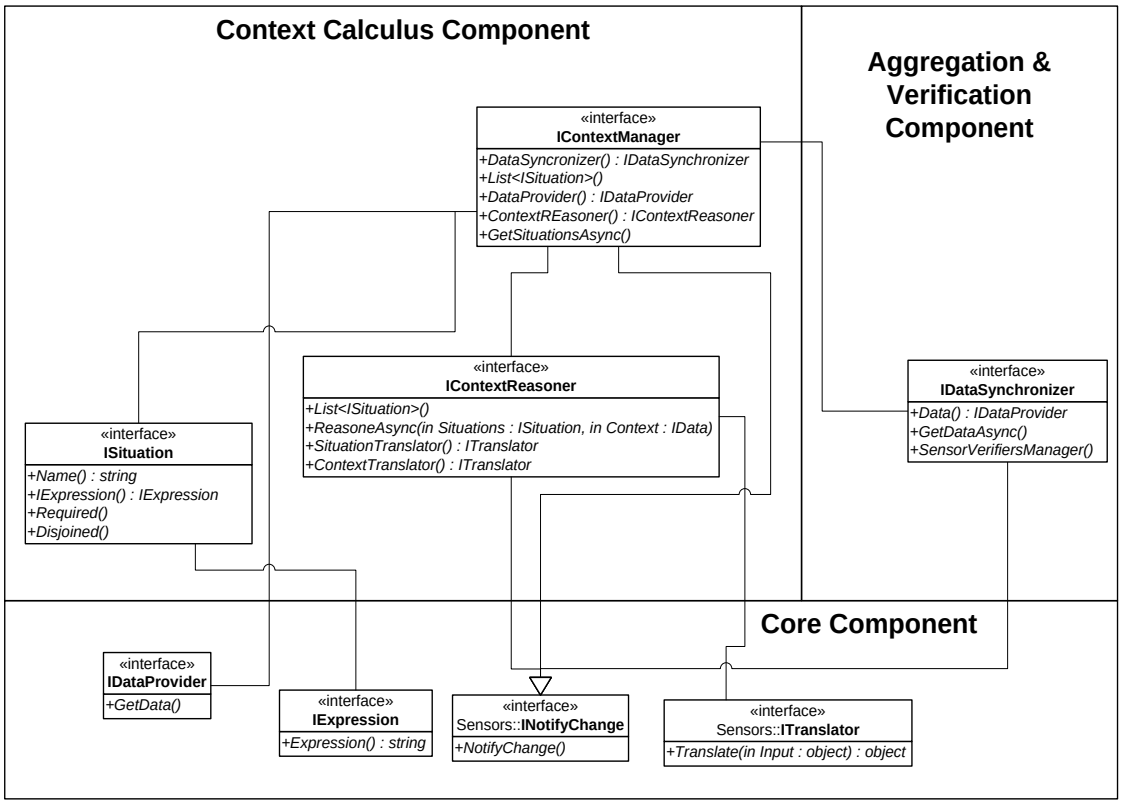

Figure 19: Context Module

Resolving Component

The Resolving Component is responsible for resolving the appropriate adaptations that should be taken in response to existing context situations. It is also responsible for implementing the workflow for these adaptations and enforcing the policies defined in them. The component consists of the following interfaces.

1. *IAdaptationResolver*: It is responsible for coordinating the process of resolving required adaptations. The resolver activates the data provider to fetch the definition of adaptations, resolves the required adaptations in response to the situations that exist in context, and activates the workflow executor to execute the required adaptation. The interface is described in Table 18.

Field	**Type**	**Description**
ContextManager	Property	IContextManager value.
DataProvider	Property	IDataProvider value. To get the definitions of adaptations
WorkflowExec	Property	IWorkflowExecutor value, the object responsible for executing workflow.
PolicyChecker	property	IPolicyChecker value, the object responsible for verifying policies.
React	Operation, input: void, output: void	This operation is used when the client wants to force reevaluation of the context, and the adaptation
Logger	Property	ILogger value.

Table 18: IAdaptationResolver Description

2. *IAdaptation*: It is responsible for representing adaptation in our framework, The interface is described in Table 19.

3. *IWorkflow*: It represents the workflow of an adaptation, which contains an execution plan of reactions. The interface is described in Table 20.

4. *IWorkflowExecutor*: It represents a workflow executor, which is responsible for realizing the workflow. The interface *IWorkflowExecutor* is implementing the visitor design pattern [GE95] to isolate the functionality of executing the workflow from the actual representation of the workflow. The interface is described in Table 21.

Field	**Type**	**Description**
Name	Property	The adaptation identifier
List<ISituation>	Property	The list of situation this adaptation will react against
Workflow	Property	IWorkflow value, the work flow of the adaptation
Policy	Property	IPolicy Value, the policies governing the execution of this workflow.

Table 19: IAdaptation Description

Field	**Type**	**Description**
Expression	Property	IExpression value. That will represent the workflow.

Table 20: IWorkflow Description

Field	Type	Description
Execute	Operation, input: IWorkflow value, output: Boolean value represents whether the operations was successful or not	The method for executing a given workflow.

Table 21: IWorkflowExecutor Description

5. *IPolicyChecker*: It represents policy checker, which is responsible for verifying that the conditions defined by policies are met before execution. Just like in workflow, the *IPolicyChecker* interface implements the visitor design pattern [GE95] to isolate the functionality of verifying the policy from the actual representation of the policy. The interface is described in Table 23.

6. *IPolicy*: It represents a policy that constrains an adaptation. The interface is described in Table 22.

Field	Type	Description
PolicyName	Property	This represents the unique name of the policy.
Params	Property	This represents an array of input items needed to evaluate the policy.

Table 22: IPolicy Description

Field	Type	Description
Name	Property	This represents the unique name of the checker.
Check	Operation	The method responsible for verifying a given policy.
Logger	Property	ILogger value.

Table 23: IPolicyChecker Description

Reactivity Component

The Reactivity Component is responsible for implementing predefined system reactions. Each reaction has an Actuator Controller to control the actuator responsible for executing the reaction. The component consists of the following interfaces.

1. *IReaction*: It is responsible for representing a specific reaction. The interface is described in Table 24.

2. *IActuatorController*: It is responsible for interacting with actuators. It uses the *ITranslator* to translate the data from the framework generic type to the actuator specific type. The controller uses *IConnector* to connect with actuators. The interface is described in Table 25.

3. *IActuatorConfigArgs*: It is an empty interface to represent the specific configuration used in the actuator. An actuator responsible for controlling a door would have the configuration to open, or close, or lock the door. These actuator-specific configurations are presented in *IActuatorConfigArgs*.

Field	Type	Description
Name	Property	The Reaction identifier
ActuatorController	Property	IActuatorController Value, represents the object responsible for dealing with the actuator.
ActuatorConfig	Property	Represents the configuration for the corresponding actuator
Result	Property	The result of the action.
DoWorkAsync	Operation, input: an array of input object items, output: void	The method for activating the action.
Logger	Property	ILogger value.

Table 24: IReaction Description

Field	Type	Description
Connector	Property	IConnector Value, the connector responsible for dealing with the actuator.
Result	Property	The result of the operation.
DoWorkAsync	Operation: input: IActuatorConfigArgs value, output: void	The operation responsible for doing the action.
Logger	Property	ILogger value.

Table 25: IActuatorController Description

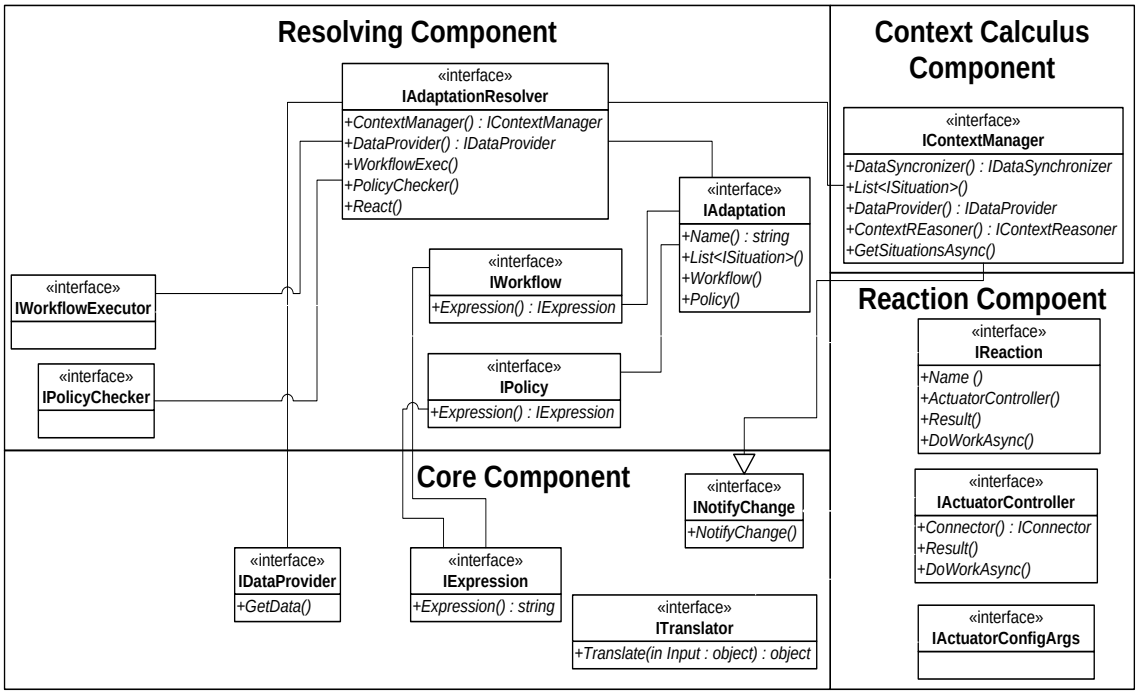

Figure 20: Resolving and Reactivity Module

Inner-and-inter communication in components

The design of interaction between components is all based on the observer design pattern [GE95]. The observer design pattern was chosen because (1) we need a support for both events and asynchronous calls, (2) some operations could be either time consuming (reasoning) or could include interaction with the outside world (reading sensors data). The Observer design pattern allows us to prevent blocking operations and provides a standard mechanism to deal with communication exception handling.

The interaction between system components is event driven. Once the system is up and running, the operation of context reasoning and consequently triggering adaptations could happen in one of two cases.

1. Either the client asked for it, say to update its current context, or
2. New data have been detected.

The components inner and inter communication is illustrated in the form of UML Sequence diagrams.

Listener component interaction The interaction is illustrated in Figure 21. When the *Connector* receives new data it notifies the *Sensor Listener* which in turn translates the data through a *Translator*, and then notifies the corresponding *Verifier* with the new data.

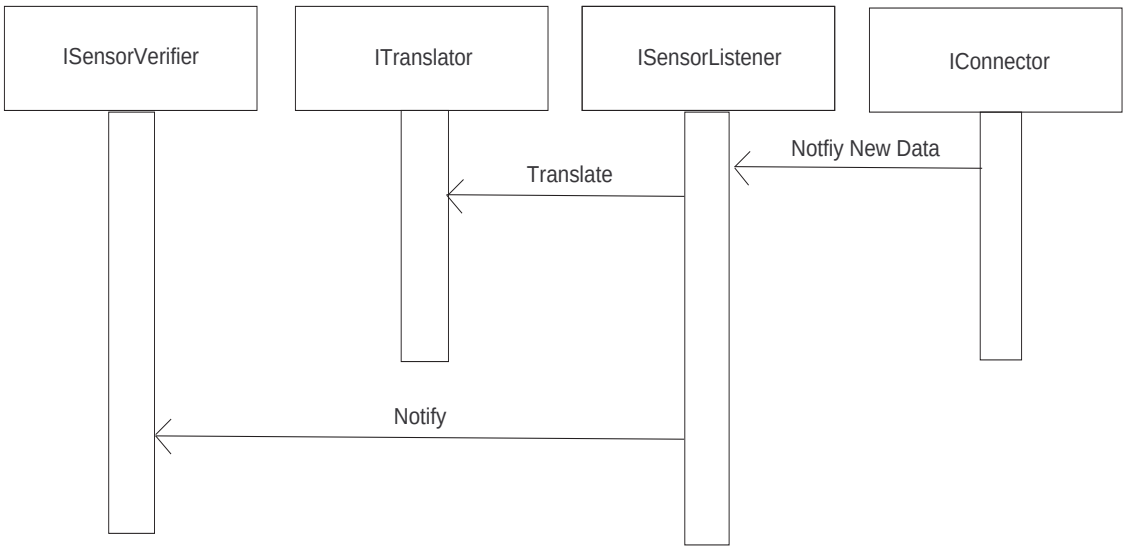

Figure 21: Sequence Diagram for Listener Component

Aggregation and Verification component interaction The interaction is illustrated in Figure 22. When the *Verifier* receives information from the *Sensor Listener* the *Verifier* notifies the *Data Synchronizer*, which in turn asks all other sensor verifiers to get the updated data. A *Sensor Verifier* has a buffer that saves the last sensor reading. If this reading is valid and up to date the verifier returns the value, otherwise the verifier asks the sensor for an up-to-date information. When data is updated, the *Context Manager* is notified.

Context calculus component interaction The interaction is illustrated in Figure 23. The *Context Manager* receives a notification when context data are ready. The *Context Manager* uses a data provider to connect to the data source and get situation definitions. Following that, the *Context Manager* activates a *Reasoner* to reason about context against the predefined situations. The *Reasoner* uses two translators to translate the context and the situations. Then, the *Reasoner* calculates and infers the situations that exist in the current context. The *Adaptation Resolver* is activated once situations in context are calculated.

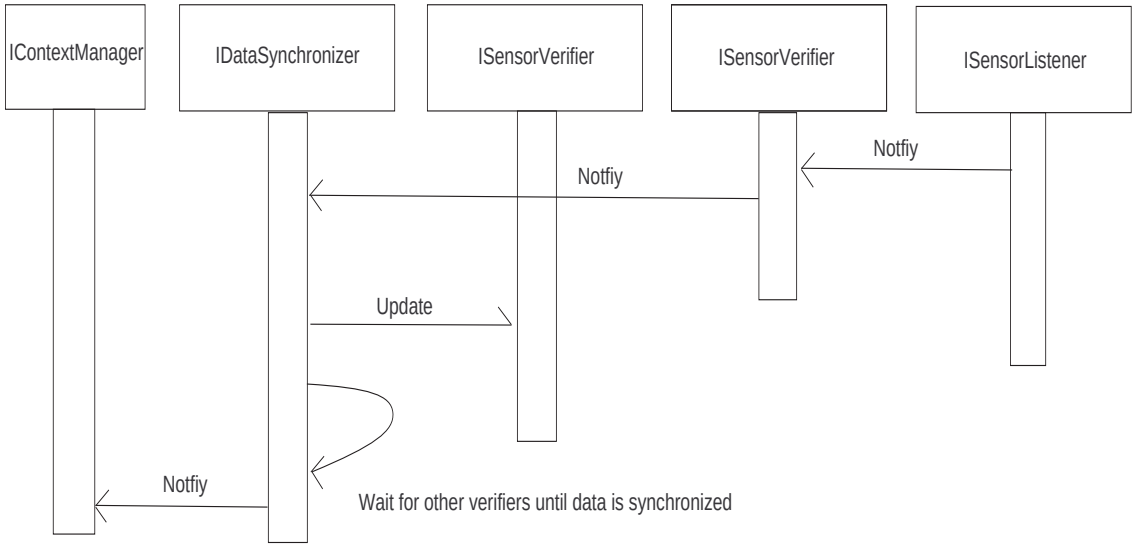

Figure 22: Sequence Diagram for Aggregation and Verification Component

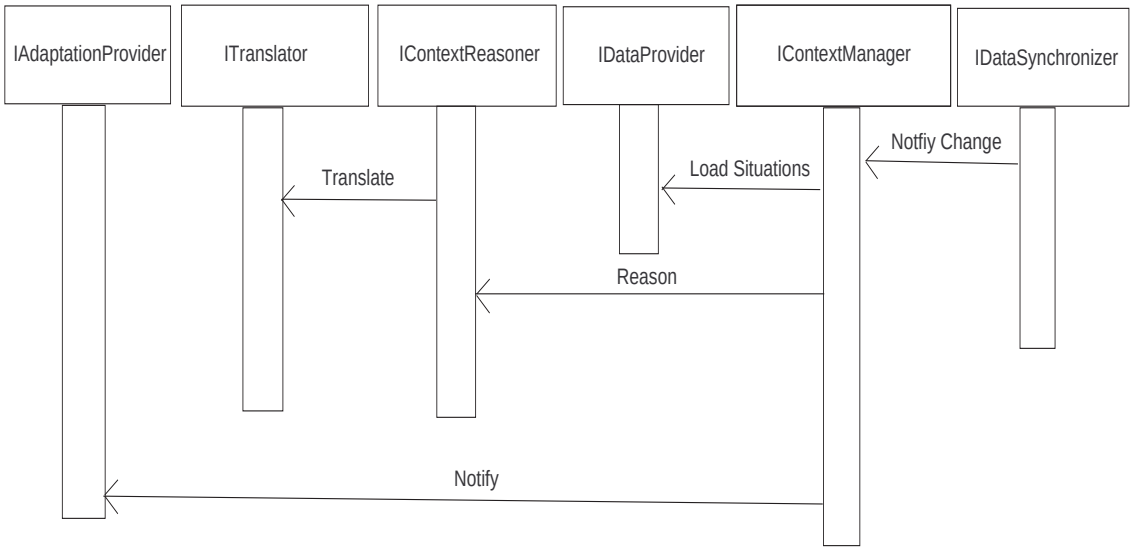

Figure 23: Sequence Diagram for Context Component

Resolving component interaction The interaction is illustrated in Figure 24. 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 implement the reaction defined in the proper sequence.

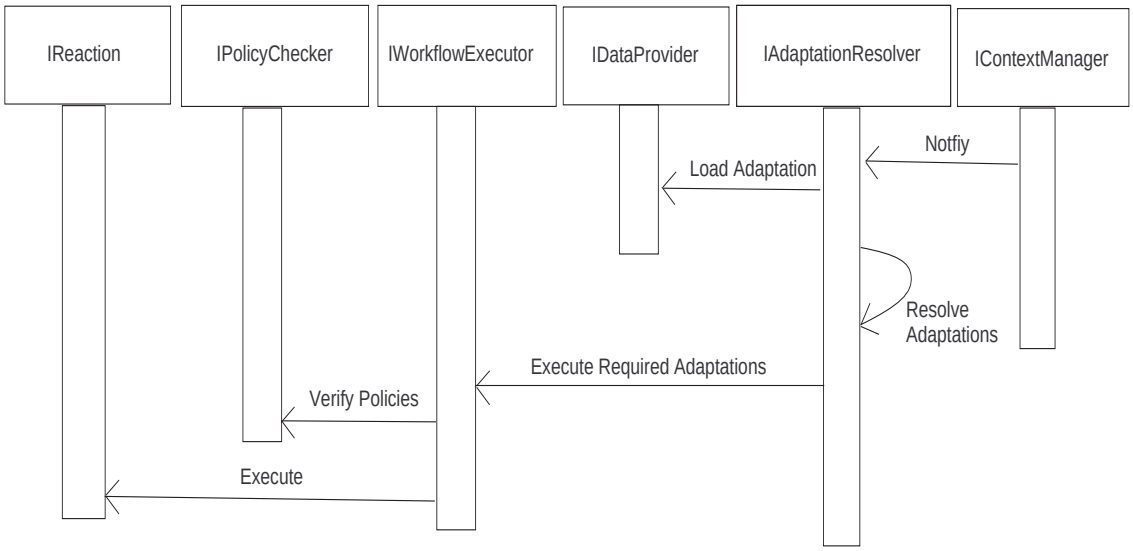

Figure 24: Sequence Diagram for the Resolving Component

Reactivity component interaction 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. The interaction is illustrated in Figure 25.

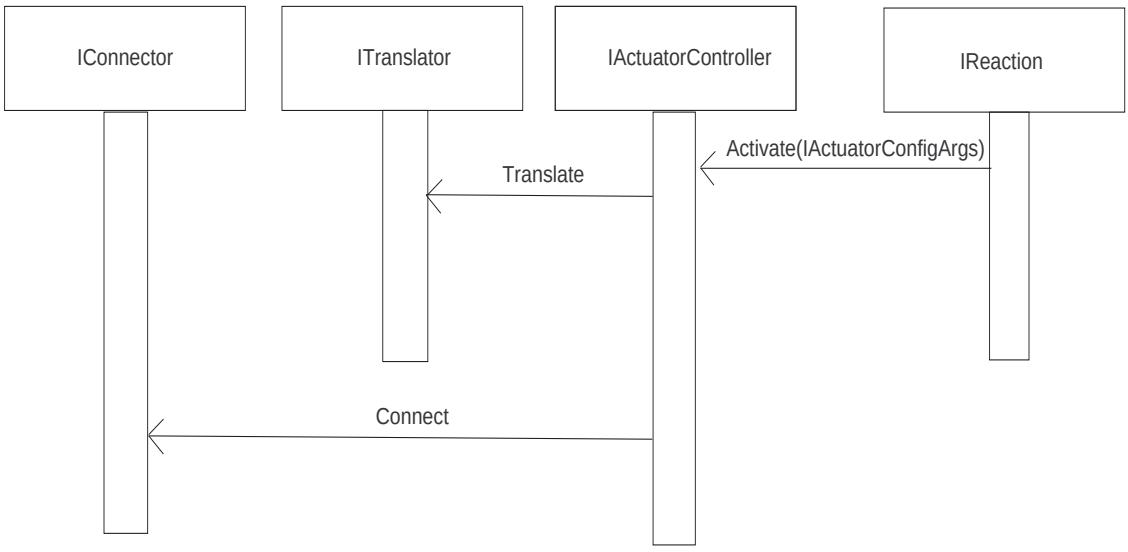

Figure 25: Sequence Diagram for the Reactivity

6.2 Context Reasoning

Presenting context information as a set of dimension-tag pairs hides the semantics that lies behind. In order to express the semantic information resulting from the aggregation of context information we have to present context in a yet more abstract mean. We introduce Context Situation as an abstraction that is presented as expressions of Situation Expression Language defined in 4.6. Context Situations needs to be parsed and evaluated against context information. The detailed description of how this operation was conducted is presented here.

Situation Representation and Parsing

A *Situation* is defined based on the box notation introduced in [WAN06]. Each Situation contains an expression, which identifies the conditions over context information and over other situations as well. For that purpose we defined the Situation Expression Language in Chapter 4.6.

Situation Parsing Push Down Automaton (PDA) is a proven technique for parsing Context Free languages [Har78]. Since the Situation Expression Language is a Context Free language, using a PDA is an efficient technique to parse those expressions. PDA tools give us great flexibility for both defining and parsing languages. In addition, it enables checking the grammars for ambiguity and making sure that expressions are accurate.

We explored different tools for that purpose. We finally decided to use Irony[1]. Irony is an open source tool built using .Net to parse Context Free Grammars (CFG). Unlike most of the other tools, Irony's grammars are not provided as plain text, they are written using C# in a compiled grammar class. This give us the privilege of writing grammars in a compiled environment taking advantage of Visual Studio and using IntelliSense.

In Addition, Irony provides a tool to read grammars and verify it for any conflicts (shift-reduce and reduce-reduce). This tool also provides a graphic user interface (GUI), illustrated in Figure 26, to test expressions against the grammars. The graphic experience is enhanced with syntax highlighting and visualized Abstract Syntax Tree (AST).

Abstract Syntax Tree Abstract Syntax Tree (AST) is the object model of parsing a context free language. The tree represents the corresponding expression defined in the language. This tree is used to perform operations over the defined expression such as Semantic Checking, Code Optimization or Code generation. In our case we are using it for reasoning.

Irony provides a default implementation for an Abstract Syntax Tree (AST) that corresponds to the defined grammars. Also, it provides the ability to define custom AST types through passing the AST Node to the parser.

We decided to use Irony's default AST and then use a converter that will navigate through the tree and generate our own expression tree. This decision was made to decouple our AST from Irony's. In Irony, to be able to make its parser create an application specific AST one should either inherit from a base class or implement an interface. In both cases a static dependency is bounded. For that reason we decided to keep our AST decoupled from

[1] $http://Irony.codeplex.com$

Irony's AST. So if we decided in the future to move to another tool our logic in terms of the structure of the tree and behavior, i.e. the reasoning, will not change.

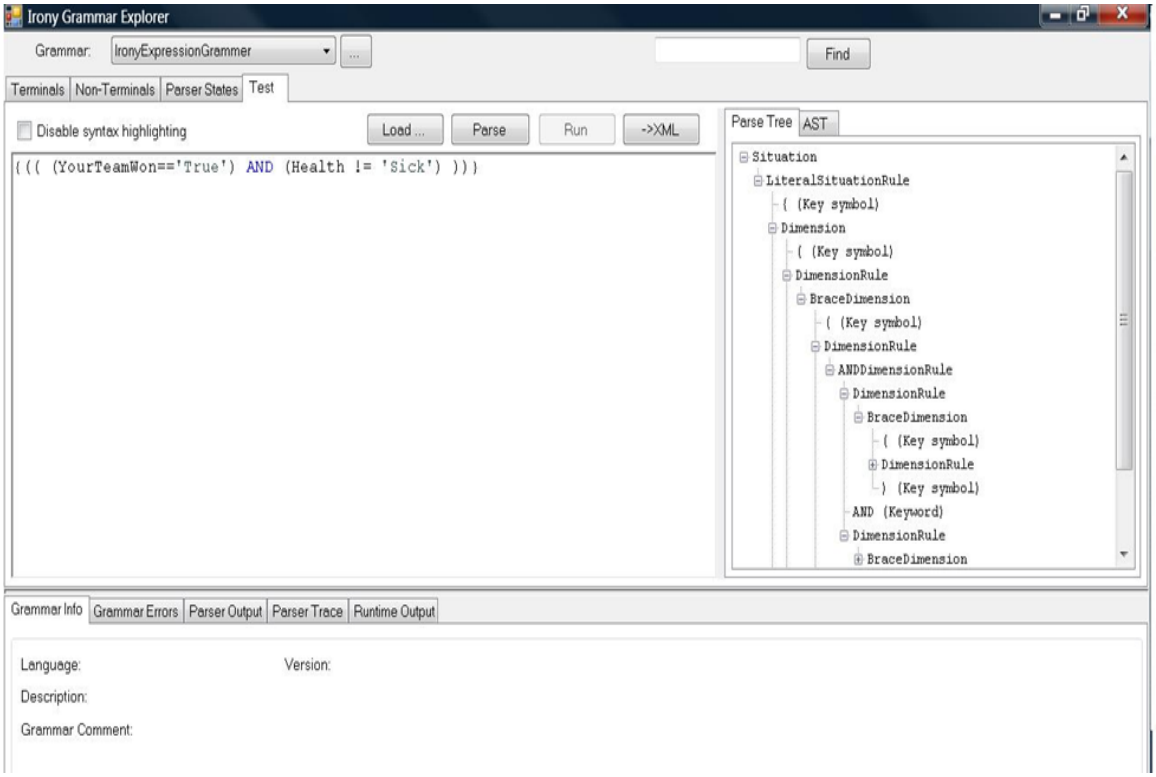

Figure 26: Irony Grammar Explorer

Situation AST Design The Expression tree corresponds to the language defined in Chapter 4.6. Each node in the tree represents an operation or a terminal. An operation node is held between the node children. For an OR operation the node holds the value resulting from conducting logical OR on the children nodes. The Situation exists in context if the expression root node evaluates to true. The tree structure is closely related to the way we designed our grammars.

The Situation Tree Structure is illustrated in Figure 27. Bold edged squares represent terminals in the tree, whereas, normal edged squares represent non terminals.

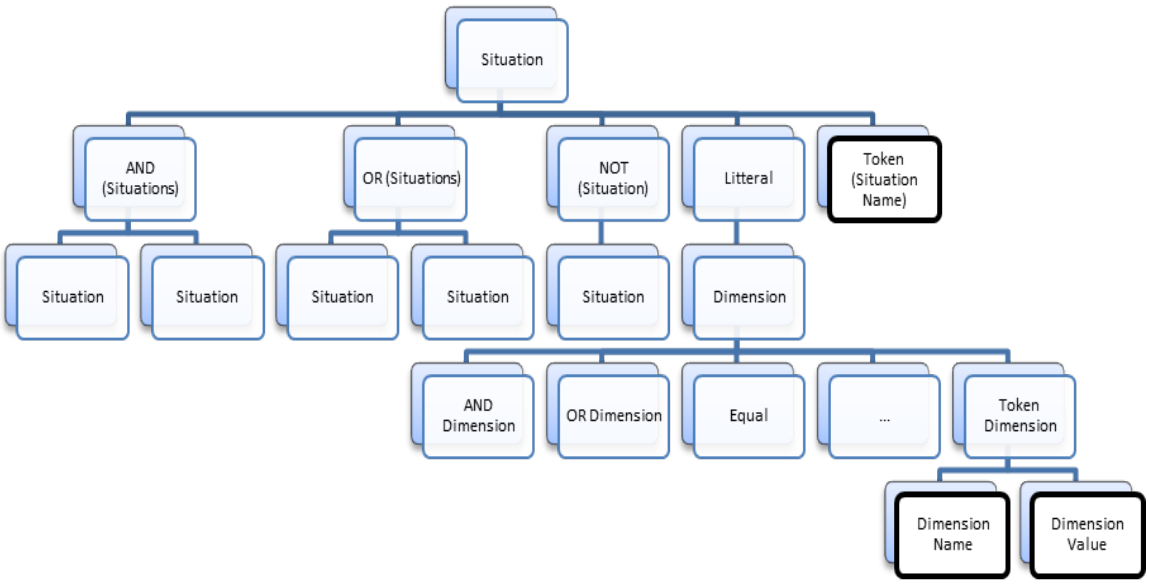

Figure 27: AST Tree Structure

Bold squares represent terminals, others are non terminal

Reasoner Component Implementation

The Reasoner component, illustrated in Figure 28, is responsible for (1) loading the Situation Expression Language grammar definitions, (2) parsing the situation definitions, and (3) evaluating the defined situations against context information. This is done through incorporating the framework component with Irony's. The component contains Irony AST *Converter*, *Situation Translator*, *Expression Evaluator* and *Reasoner*.

Irony AST Converter The converter, as we described earlier is responsible for converting the AST from the parser-specific format to the reasoner-specific format.

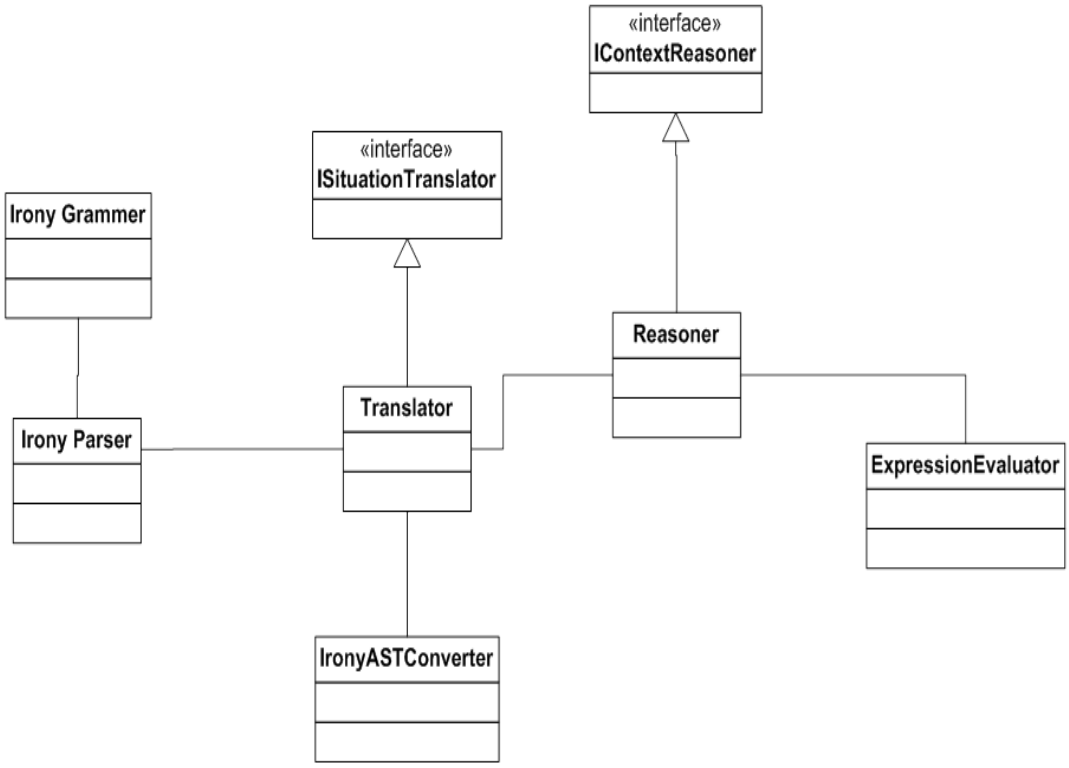

Figure 28: Reasoner Component Design

Situation Translator The Situation translator encapsulates Irony's parser. The Translator that implements *ISituationTranslator* is responsible for translating the textual expression of Situation into the reasoner-specific format which is in this case the Situation AST. The situation translator execution workflow instantiates an instance of Irony Parser, feeds the parser with the grammars, passing the Situation Expression string to the parser, converts the AST from Irony's format to the reasoner format, and returns the result to the reasoner.

The Expression Evaluator Building the Expression tree was for the purpose of verifying an expression against a specific context. Design-wise, to do that we had two options. The first option is to define a virtual method *"Verify"* which should be implemented by each class that inherits from Situation Expression. Then each class can implement its own logic for verification. The second option is to create a visitor (visitor design pattern [GE95]) to group specific application logic of verifying the nodes in a single visitor class. For example,

instead of separating the logic of evaluation among different classes one class is allowed to contain all the logic necessary for all verifications.

Each option has its own advantages and drawbacks. We chose to go with the visitor design pattern to decouple the behavior from the structure of the tree. Since the structure is related to the way we defined our expression and somehow, to the way we parse them. However, the actual reasoning or verifying mechanism is indifferent to all that. That is the main motivation to decouple the structure from the behavior. By using the visitor design pattern we are still taking advantage of the polymorphism in the design since the navigation is done through the tree but the actual action is outsourced to the visitor.

The Reasoner The reasoner implements *IContextReasoner* defined in the framework. The reasoning operation is asynchronies. First, the reasoner uses the translator to translate the situation to the reasoner specific format. Then it uses the evaluator to travel through (visit) the tree and evaluate each expression in the given context. Then the reasoner informs the Adaptation Resolver with the discovered situations.

6.3 Workflow & Policy component

The Adaptation is a set of governed system reactions in response to a changing context. They constitute a critical part of any context aware system. The system reacts by firing automatic reactions. In order to assure the trustworthiness and predictability properties of such actions these actions should be accurate, precisely defined and most importantly governed by execution conditions that we call polices. To meet all these conditions, an adaptation is presented as a logical grouping of yet a finer system tasks that we have called reactions. Each adaptation also has policies that should be checked before triggering the reactions. The reactions and policies are constructed using the Adaptation Workflow expression language defined in Chapter 4.7. In this section we present the workflow component detailed design, and illustrate how workflows are parsed and executed.

Workflow Representation and Parsing

The Adaptation Workflow Language, described in Section 4.7, is also a context free language. Therefore, we need a Push Down Automaton (PDA) tool to parse the workflow expressions. Consequently, we used the same tool Irony, described earlier in Situation parsing, for parsing the workflow language.

Workflow Abstract Syntax Tree The same approach chosen for building the AST for Situation is used here. We use a Converter to convert Irony's AST workflow into our own AST. The driving motivation for that is to decouple the tool's logic from our own. The workflow tree corresponds to the language grammar presented in Section 4.7. The AST architecture is illustrated in Figure 29. Nodes with bold edged squares represent terminal rules, whereas nodes with normal edged squares represent non terminal rules.

Workflow & Policy Engine implementation

The Workflow & Policy Engine is responsible for implementing the defined workflows. The engine first parses the workflow definitions. The parsing is done through Irony parser. Then the AST is converted to our framework object model. Once our AST is constructed, the workflow executor navigates through the tree and implements the reactions after checking the policies. The component, illustrated in Figure 30, contains in addition to the framework abstractions *Irony Workflow AST Converter*, *Workflow translator* and the *Workflow Executor*.

Irony Workflow AST Converter The converter converts the parsed AST from Irony's default format to the framework format. The converter traverses through the tree and converts each node to its counterpart in the framework.

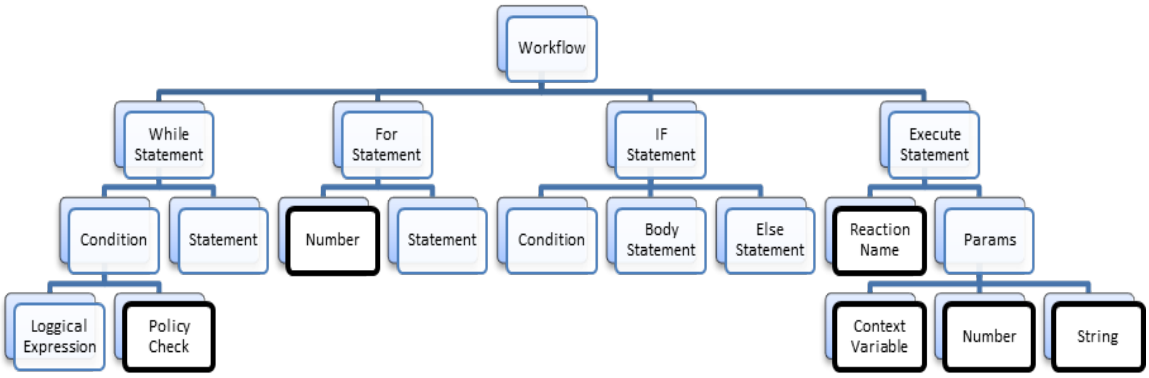

Figure 29: AST for Workflow

Bold squares represent terminals, others are non terminal

Workflow Translator It is responsible for translating the workflow expression into our AST object model. The translation is done using Irony's parser to parse the expression into Irony's default AST format, and then translating the AST into our framework format. This translator implements the interface *IWorkflowTranslator*.

The Workflow translator instantiates an instance of Irony Parser, feeds the parser with the grammars, passes the expression string into the parser, and converts the AST from Irony's format to the framework workflow executor format.

Executor It implements *IWorkflowExecutor*, which is responsible for executing the corresponding workflow using the visitor design pattern [GE95]. Depending on the AST deign of the framework the executor implements the logic behind each node type. For *While* node

the executor checks the condition before executing the body. As for the *Exec* node, the executor allocates the appropriate reaction and instantiates it.

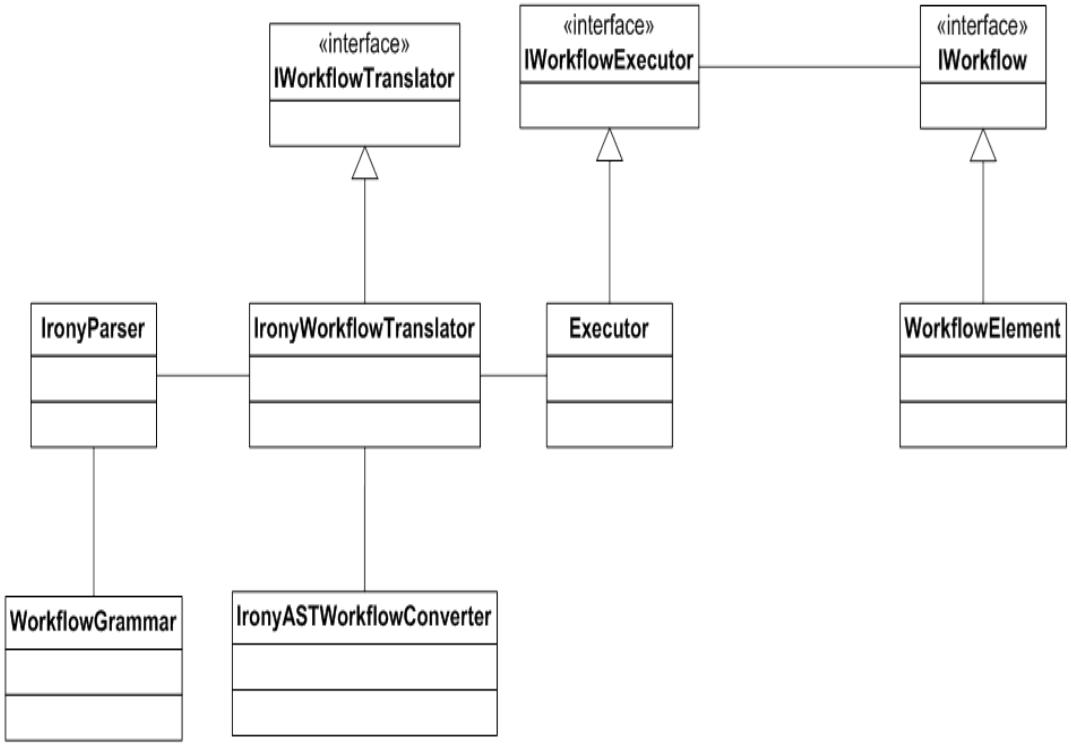

Figure 30: The Workflow & Policy Engine

Chapter 7

Implementation

In this chapter a synopsis of the Context-aware Framework (CAF) implementation is given. The reasons behind the choice of the implementation framework (IF) and specific technologies for realizing the CAF implementation are discussed. Two example case studies are explained. The first example is taken from reactive systems domain and the second example is taken from business domain. Since the second example is more complex, a thorough description of implemented elements and their mechanisms are given. Finally, test results on the implemented prototype are shown.

7.1 Implementation Platform

The minimum set of requirements for developing the CAF are first identified. Based on that set and the available technology the implementation platform that best suits the requirements is chosen.

7.1.1 Requirements and Analysis

The four essential aspects to be considered are a faithful implementation of the component-based design, the communication need between components, need for dynamic changes and portability. From this we extract the following set of requirements for an efficient implementation of CAF.

1. *Support for Object-Oriented design* The framework architecture is component-based, and an Object-Oriented platform is acceptable for its implementation. Many aspects including but not limited to inheritance, polymorphism and interfaces are useful for CAF implementation.

2. *Support for asynchronous method calls* Inner-component calls and interactions with outer world, such as Sensors, Actuators and Reasoners, should be done asynchronously to prevent interface blocking and for exception handling.

3. *Support for code reflection and dynamic code invocation* The abstraction proposed in CAF offers privileged developers with the ability to inject the right implementation in 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.

4. *Support for cross platform environment* It is intended to deploy CAF on different platforms such as mobile phones, desktop and web.

With respect to the aforementioned requirements we had the chance to choose between two development platforms: Microsoft .Net or Java based platform. Both platforms provide a rich Object-Oriented experience and support for asynchronous calls. Reflection capabilities are mostly similar in providing the same functionalities of loading classes, attributes and invoking methods at run time. As for portability, Java is supported on Mac machines, several UNIX flavors and Microsoft Windows. No formal support exists for .Net in any platform other than Windows (Windows 7, Vista, and XP). However, with the introduction of Silverlight[1] a big portion of the .Net framework is supported on the mentioned platforms. In addition, with the mono project[2] the .Net code is supported in a various other platforms such as UNIX, Mac, iOS (iPhone and iPad) and even Android. That have been said, the same code could be reused to write applications for desktop in Silverlight and for phones

[1] http://www.silverlight.net
[2] http://www.mono-project.com/

in Windows phone, Android based phones and iPhone. On the other hand Java is only partially supported in Android and not supported on either Windows Phone or iPhone.

Another aspect we had to take into consideration when choosing the development platform is the integrated development environment (IDE) used in CAF development. While eclipse[3] is providing a rich development experience in Java compared to other IDEs such as NetBeans[4] and JBuilder[5], Visual Studio[6] is far superior when compared in integration with other platforms such as Database and phone.

In respect to all the requirements previously identified, we conclude that both .Net and Java are suitable for the development of the Context-aware Framework (CAF). However, minor aspects such as IDE and integration with other services (maps) made us prefer to use Visual Studio 2010 with C#, .Net 4.0, & Silverlight 4 as the development platform.

.Net Framework (NF) Characteristics

The software framework NF, developed by Microsoft, supports the development of applications using different programming languages. The Base Class Library (BCL) in NF provides a reach set of tools that support developers in implementing applications which require Graphic User Interface (GUI), data access, database connectivity, cryptography, web application or data structure.

The decoupling between the language capabilities and the framework capabilities is provided through a Common Language Interface (CLI). Which is a technique to provide the main framework functionalities including the NF Base Class Library (BCL) through a language-neutral interface. Microsoft .Net Implementation of CLI is called the Common Language Runtime (CLR). There exist other implementation for CLI such as Silverlight CLR and Mono CLR.

Different language Compilers in NF (C#, Visual Basic, J#, etc...) compile high level

[3] $http://www.eclipse.org/$
[4] $http://netbeans.org/$
[5] $http://www.embarcadero.com/products/jbuilder$
[6] $http://www.microsoft.com/visualstudio$

code to a common intermediate language formally known as Microsoft Intermediate Language (MSIL). The MSIL then interpreted by a Just-In-Time complier (JIT) to machine level code. The process is illustrated in Figure 31. Therefore, developers don't have to write a platform-specific code or explicitly targeting specific hardware architectures, such as 32bit or 64bit, since the CLR generates the right machine code at runtime.

Silverlight

Although the NF was designed to work on different platforms, Microsoft only provides the NF on Microsoft Windows OS. However, a portable version of NF, called Silverlight, was introduced. Silverlight[7] is an application Framework that was first intended to target web application with heavy multimedia, graphic and animation content. Later, it evolved (especially in version 3.0 and 4.0) to be a full application framework for business type applications.

Silverlight Framework, starting from version 2.0, implements the Common Language Interface (CLI) with a different Common Language Runtime (CLR) than the one shipped with NF. The Silverlight framework design is illustrated in Figure 32. Silverlight runs as a power plug-in inside web browsers, such as IE, Firefox and Safari. Consequently, Silverlight runs on different platforms, namely Windows, Mac, and Unix flavors. The Silverlight framework uses an XML based annotation language, namely the eXtensible Application Markup Language (XAML), for representing the application interfaces. XAML was first introduced in Windows Presentation Foundation (WPF) as a part of NF version 3.0.

Version 4 of Silverlight introduced an out-of-browser experience, which allows users to install applications on their desktops and run them outside the web browsers. This brought Silverlight to a whole new level to act as a full portable application Framework. However, for security concerns Silverlight is still running inside a *sandbox* which limits the accessibility of Silverlight applications. Although many workarounds exist to safely pass these security concerns, future versions of Silverlight are believed to provide more flexibility. In

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

this aspect Silverlight applications are evolving to be business applications with ability to connect to client resources, such as Barcode readers, GPS, and Printers.

7.2 CAF Implementation

CAF is implemented using C# on .Net 4.0. The components described in 6.1 are implemented with seven projects as follows.

1. *CAF.Framework*: It contains the framework core, which includes all the generic interfaces and data types. This project is the only dependency for all the other projects which are mutually independent. This is due to the abstraction made in design phase which allows us to group all shared parts in one component. This structure provides flexibility in the implementation and clear separation of concerns, increasing the testability of the whole framework and thus the trustworthiness.

2. *CAF.Aggregation* It contains the framework default implementation of *Verifier Manager* and *Data Synchronizer* and the exceptions thrown from this component.

3. *CAF.Context* It contains the framework default implementation of *Context Manager* and defines the exceptions thrown from this component.

4. *CAF.Adaptation* It contains the framework default implementation of *Adaptation Resolver* and defines the exceptions thrown from this component.

5. *CAF.SituationReasoner* It provides the implementation of the suggested technique proposed by this framework for reasoning over context information. As described in Chapter 6.1 the framework may support different kinds of reasoners. By default we implement one type of Reasoners, called *SituationReasoner*, based on the Situations Expression Language defined in Chapter 4.6. The *SituationReasoner*, as any reasoner, parses the definitions of Situations defined by the client application, then converts it into its own format, and finally returns Situations that exist in the current context.

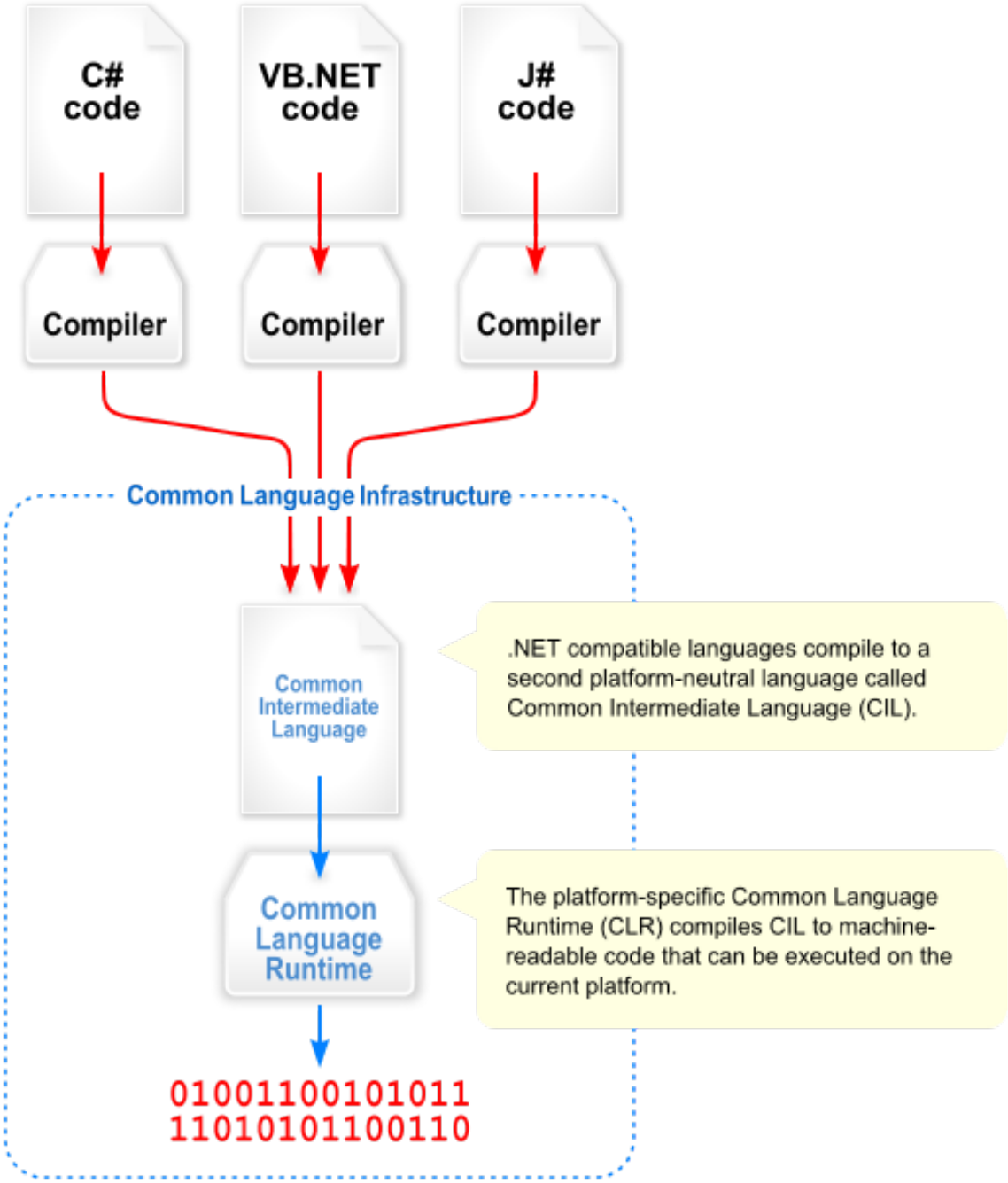

Figure 31: .Net Framework

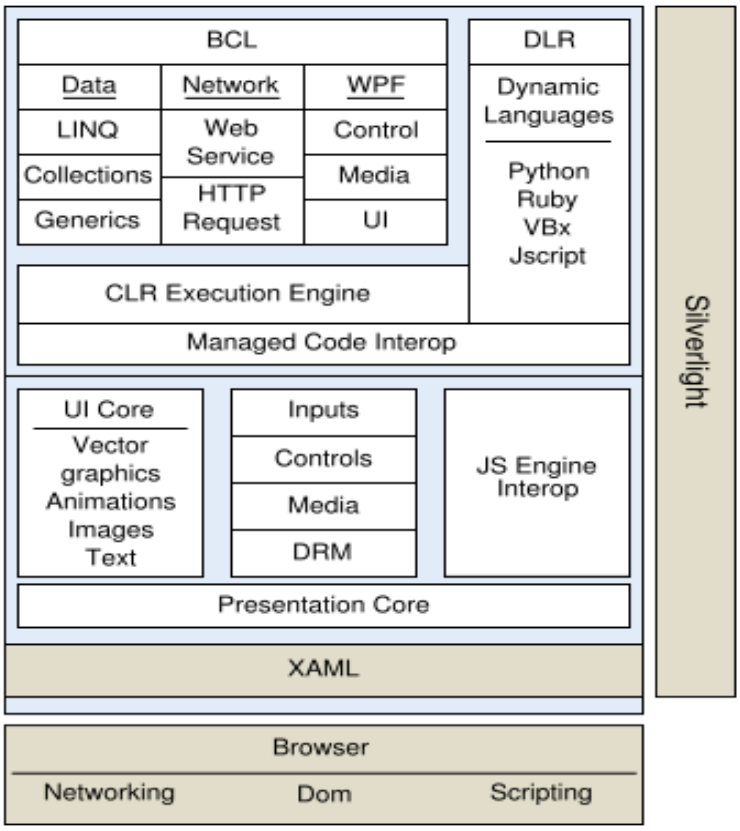

Figure 32: Silverlight Framework

6. *CAF.Workflow*: It contains the framework default implementation of Workflow Executor. The Workflow Executor parses the workflow defined in Chapter and then implements the actions with respect to the defined policies that are appropriate for situations in the current context.

7. *CAF.Tools*: It contains a set of Connectors, Actuators, Translators and other tools that may be used in different projects such as Serial Connectors, and GPS Sensors.

Framework Bootstrapping

Framework initialization is a challenging problem, since it's intended to wire all abstractions with there concrete implementation. It is like solving a puzzle by putting each piece in its appropriate position. All the abstractions exist in one component as interfaces, yet

the right implementation needs to be wired with the corresponding interfaces in order for the framework to function properly.

In computer science the entity that is responsible to address the previous issue is called *BootStrapper*, which is responsible for the initialization of the applications that contain multiple components. The two techniques used by *BootStrapper* to initialize software applications are *static strapping* and *dynamic strapping*.

Static Bootstrapping At *compile time* interfaces are wired with the concrete implementation, which means any change will cause a complete recompilation. Although the changes in the code are minimal, recompilation may not be possible at all times.

Dynamic Bootstrapping At *run-time* interfaces are wired with the right implementation. This could be achieved through a configuration file that maps each interface with the right implementation or through other techniques such as code annotation.

One of the requirements of CAF is to be able to **dynamically add sensors and actuators** and furthermore **dynamically plugging and unplugging reasoners and workflow executors**. Hence, we decided to use *Dynamic Bootstrapping*. In order to do that we wrote our own entity *Container* which is responsible for instantiating objects based on Reflection and C# Code Attributes. This provides the ability to annotate code with specific attributes that could be used by compilers or through reflection. The *IContainer* interface described in Table 26 provides the ability to retrieve two types of entities, *objects* and *classes*. While the framework contains entities with only one possible implementation such as *Reasoner* and *Workflow Executors*, it also contains entities with possible multiple implementations, such as *Sensor Listeners* and *Actuators Controllers*.

The *Container* is asked to retrieve a specific object, the *Container* loads the assembly using reflection and then searches for dependencies, dependencies are identified through a customized code attribute, called "*Dependency*". This attribute has one of the two following types.

- *Item* This attribute is set when the dependency contains one item. For example, any

entity in CAF needs one logger to report activities or possible errors.

- *List* This attribute is set when the dependency contains multiple items. For example, the *Sensor Aggregator* needs to hold references of all *Sensor Listeners* in the framework, and the *Workflow Executor* needs to hold references for all *Reactions*. *Sensor Listeners* and *Reactions* are List dependencies.

The *BootStrapper* feeds the *Container* with all the concrete implementation of the CAF component interface based on a configuration file. Then the container is asked to *resolve* an abstraction, resolving an abstraction means returning the right implementation of an interface and resolving all the dependencies of the interface. Example 10 shows the definition of the *IWorkflowExecutor* interface. The interfaces declares three dependencies described as follows.

- Logger: Every entity in CAF depends on a logger in order to register all event happing, the logger is an example of an item dependency since there exist one logger in all the framework.

- Reactions: The workflow executor holds a reference to all *Reactions* in order to call the reactions when specified in a workflow. Since CAF contains more than one *Reaction*, reactions are declared as list dependency to inform the container that there may exist multiple possible implementation of the *Reaction* interface.

- Checkers: The workflow executor holds a reference to all *Policy Checkers* in order to execute the workflow. Just like *Reactions*, Checkers are declared as list dependency for the same reason.

For example, the container returns a reference of *CAFWorkflowExecutor* (which is CAF default implementation of *IWorkflowExecutor*) when asked to *resolve IWorkflowExecutor*. Resolving *IWorkflowExecutor* requires resolving *ILogger* and *IReaction*.

Example 10

```
public interface IWorkflowExecutor
    {
        [Dependency(DependencyType = DependencyType.Item)]
        ILogger Logger { get; set; }

        [Dependency(DependencyType = DependencyType.List]
        List<IReaction> Reactions { get; set; }

        [Dependency(DependencyType = DependencyType.List]
        List<IPolicyChecker> Checkers { get; set; }

        List<ISituation> Situations { get; set; }

        bool Execute(IWorkflow Workflow);
    }
```

The framework was designed to cause a domino effect when resolving objects. If the container was asked to resolve *IAdaptationResolver* all the dependencies of the Framework is resolved at once. The Figure 34 shows a sample of the Framework dependency tree.

Field	Type	Description
RegisterInstance	Operation	To register an instance with a specific type Ex: GPSSensor – ISensorListener
RegisterType	Operation	To register a type with a specific type Ex: SituationReaosner – IReasoner
Resolve	Operation	To resolve a specific type by returning the right implementation
GetAllinstances	Operation	To resolve a specific type by returning all implementations.
ILogger	Property	Event Logger

Table 26: IContainer Description

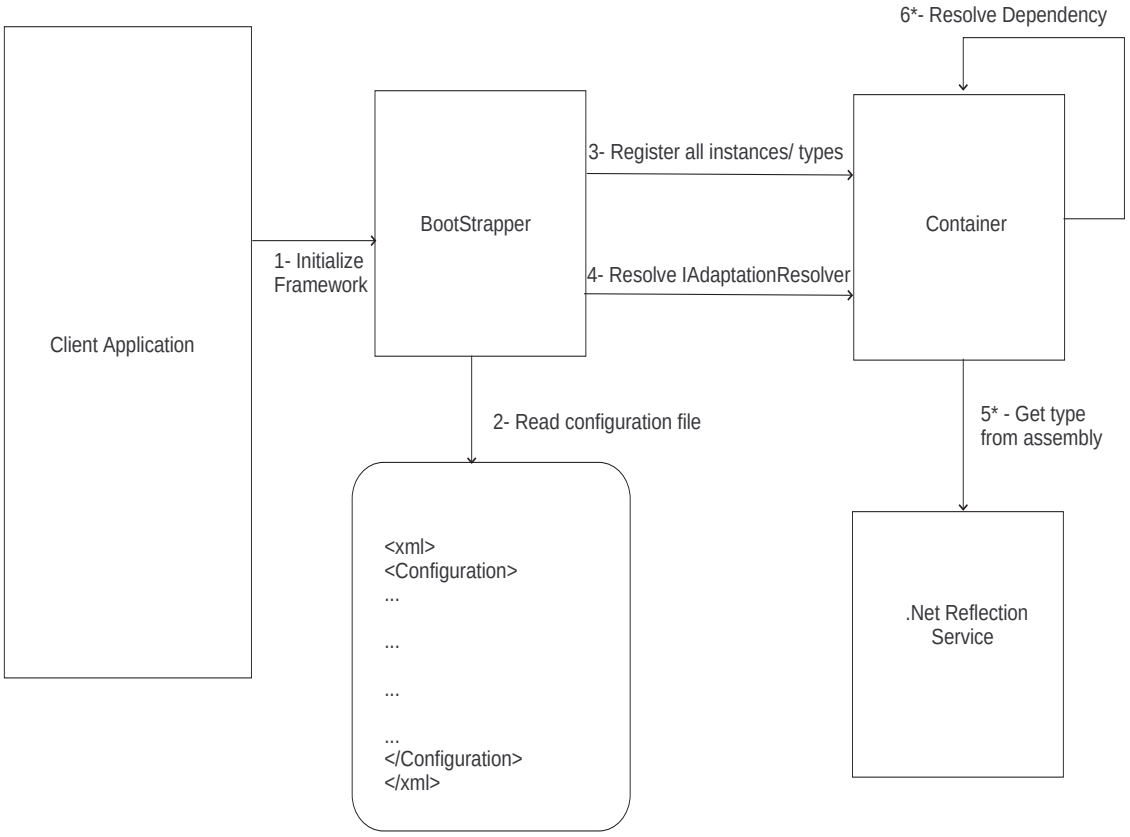

Figure 33: Initialization Process

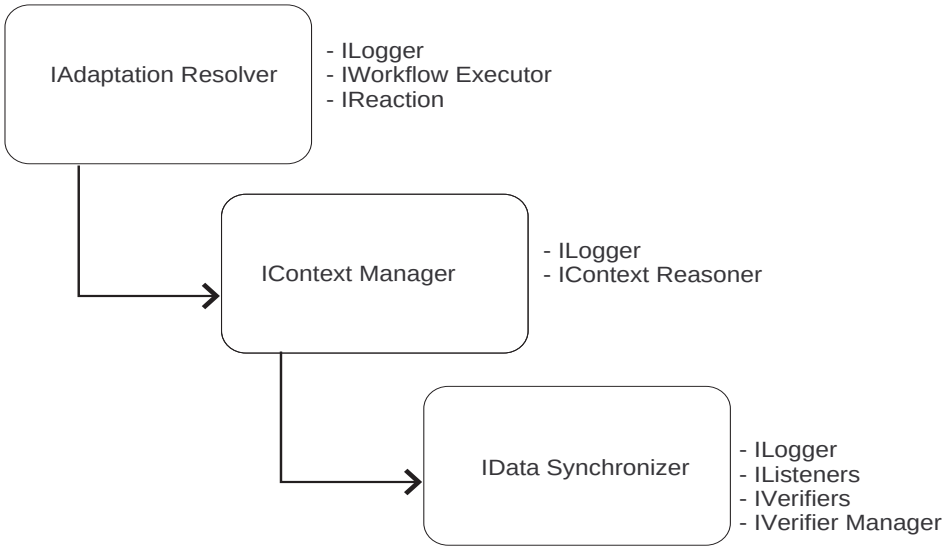

Figure 34: Dependency Tree

7.3 Case Studies

Two case studies were developed using the CAF and implemented according to the methodology described above. The first case study is chosen from the domain of 'reactive systems' and the second case study is chosen from 'business services' domain. The first example is simple enough to be done thoroughly, whereas the second example is complex enough to be adequately handled.

7.3.1 Temperature Control and Cooling System (TCCS)

A naive temperature control system is modeled in CAF. The system is required to increase the temperature when the environment's temperature is below a certain level and to decrease the temperature when the environmental temperature is above a certain level. The actuators used are a heater and a cooler. The architecture of TCCS is shown in Figure 35.

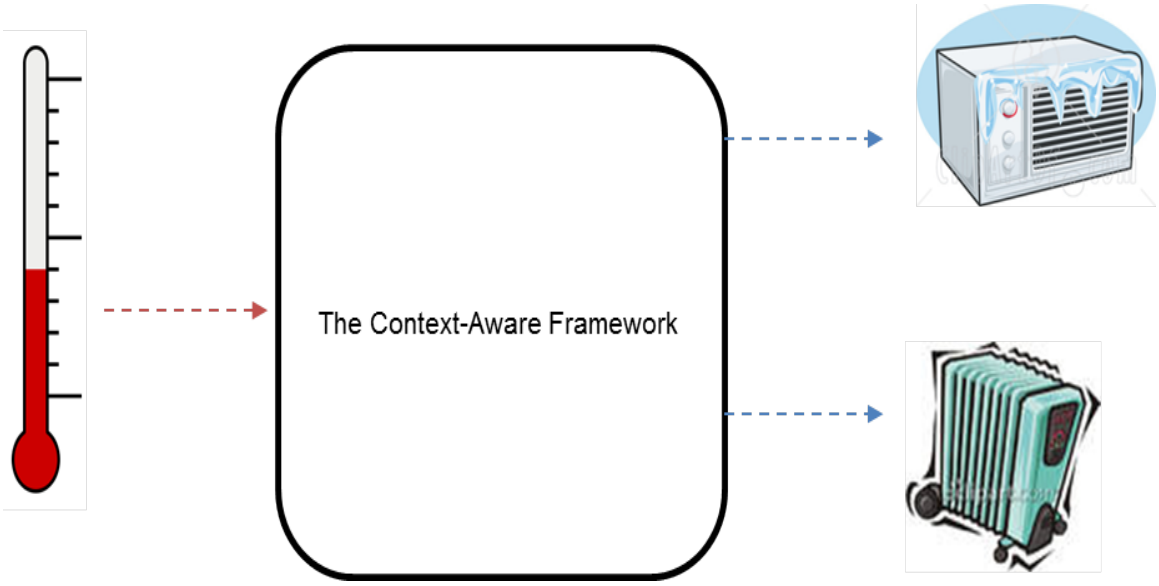

Figure 35: Temperature Control System

System Design

Modeling applications using the CAF requires defining the main entities that represents the application environment and intentions. The following client applications are defined.

1. **Sensor Listener** A thermometer listener is defined. It is responsible to connect to a thermometer that provides temperature degrees in Fahrenheit.

2. **Sensor Translators** Since the sensor reading of the temperature is in Fahrenheit, a translator that translates the temperature from Fahrenheit to Celsius is provided.

3. **Sensor Verifiers** The sensor verifier applies the data policies, which in this case is making sure that the sensor's reading is reasonable. Any reading, less than -70C or more than 70C indicates an error, and is filtered out by the verifier.

4. **Actuator Controllers** We have two actuators in this case, a heater actuators and a cooler actuator.

5. **Context Information** The context in this example has one dimension which is temperature (*Temp*).

6. **Situations and Extenders** Situations defined here reflect the state when the temperature is less than 10C or more than 30C. For that purpose we defined two situations *Cold* and *Hot*. The *Cold* situation refers to the state when the temperature degree is less than 10C. It is presented in the Situation Expression Language as follows.

$$Cold : \{\ (Temp < 10\ )\ \}$$

This *Hot* situation refers to the state when the temperature degree is more than 30C. It is presented in the Situation Expression Language as follows.

$$Warm : \{\ (\$IsHot\,[Temp])\ \}$$

‘*IsHot”* is a situation extender that provides the ability to extend the reasoning capability of the framework by triggering custom user defined functions that takes context information as an input and returns a boolean value. Since the situation expression language is limited, situation extenders are used whenever the user could not express its intentions using the predefined operations. For our situation, the extender checks

the value of temperature and returns true if it's more than 30. The Situation Expression Language *does* supports the ($>$) operation. However, this example is presented as an extender just for illustration purpose.

7. **Adaptation & Policies** We define the following two adaptations to adapt to each of the defined situations.

   (a) The adaptation in response to the *Cold* situation is *Adapt to Cold Weather*. This adaptation increases the temperature by triggering the reaction to *increase temperature*. The workflow of this adaptation is presented using the Workflow Expression Language described in Section 4.7 as follows.

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

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

   (b) The adaptation in response to the *Warm* situation is *Adapt to Warm Weather*. This adaptation decreases the temperature by triggering the reaction *decrease temperature*. The workflow of this adaptation is presented using the Workflow Expression Language described in Section 4.7 as follows.

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

   The *IsHeatingSystemWorking* is a policy to check if the heating system is working before sending an order. The policy checker for this policy is provided by the user.

8. **Reactions** As we previously mentioned the two reactions are *increase temperature* and *decrease temperature*, which respectively encapsulates the interaction with the heater and cooler actuators.

System Process Model

The system process model of TCCS is illustrated in Figure 36. It contains the sequence of steps (1) activation, (2) requesting information, (3) aggregating sensor's information, (4) reasoning over context information, (5) resolving adaptations, and (6) reacting.

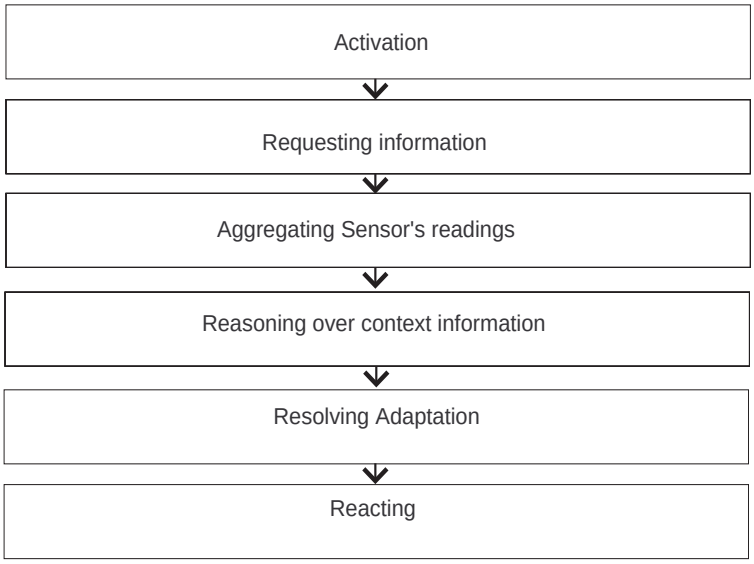

Figure 36: Framework Process Model

Activation The system can be activated either when the client asks explicitly to check the context and react upon it or a 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 readings and (4) *Sensor Verifiers* asks *Sensor Listeners* for the latest readings if applicable. The request flow is illustrated in Figure 37.

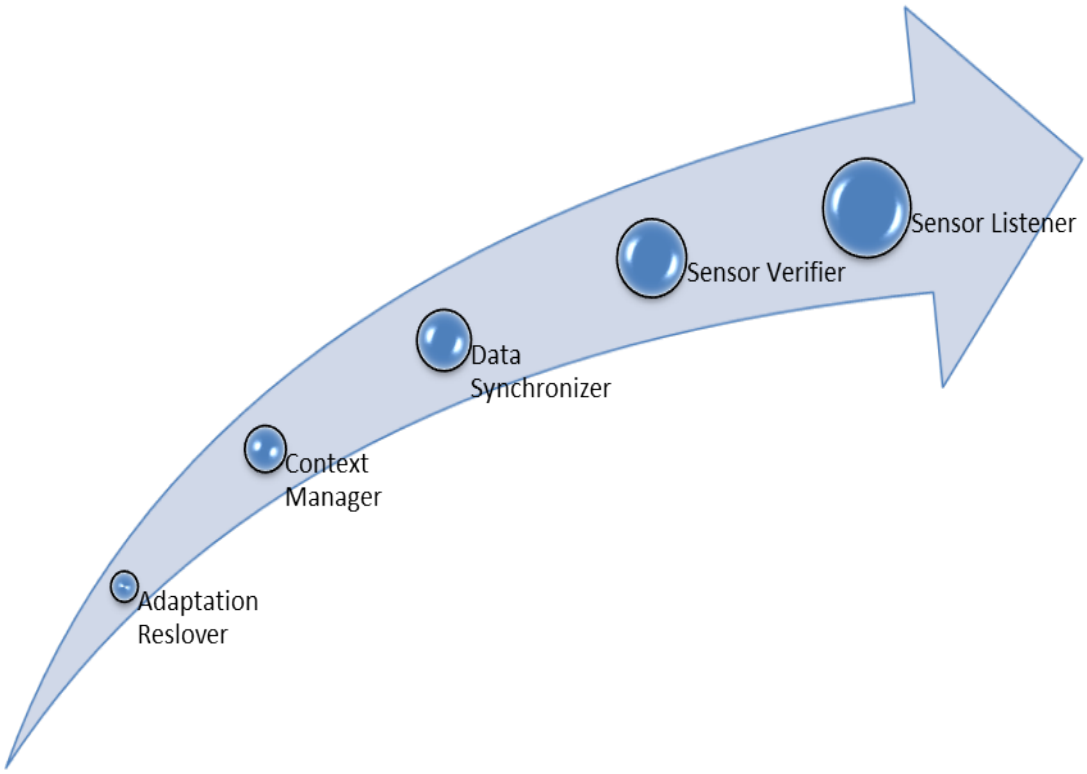

Figure 37: Request Information Chain

Aggregating sensor's information Once the sensor's listeners receive the readings, the verifiers verify the data and then notify the *Data Synchronizer* which waits until all sensor's responded and then notifies the *Context Manager*.

Reasoning over Context Information When the context is ready and synchronized, the *Data Synchronizer* informs the *Context Manager*. Then the *Context Manager* activates *The Reasoner* to discover situations in the current context.

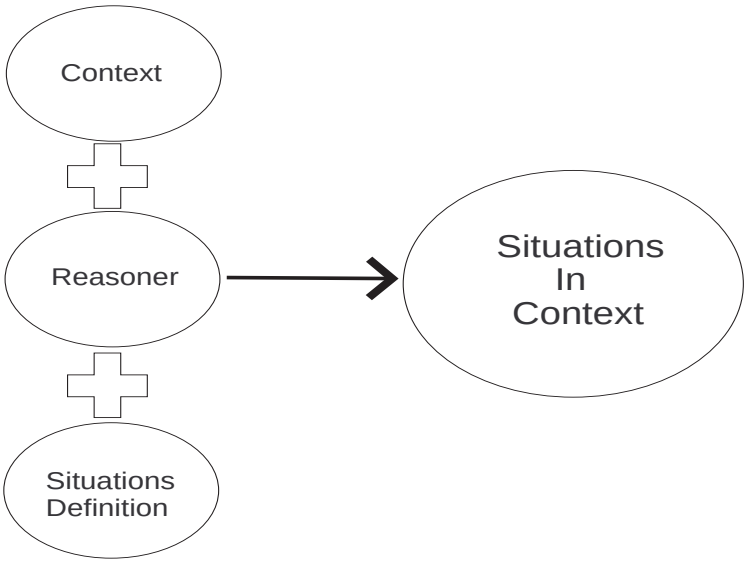

Figure 38: Reasoning Context Information

Resolving Adaptations When the *Context Manager* receives the Situations in Context from the reasoner, it informs the *Adaptation Resolver* with the discovered situations. The *Adaptation Resolver* resolves the appropriate adaptations for the discovered context. Then executes each adaptation. The process is illustrated Figure 39.

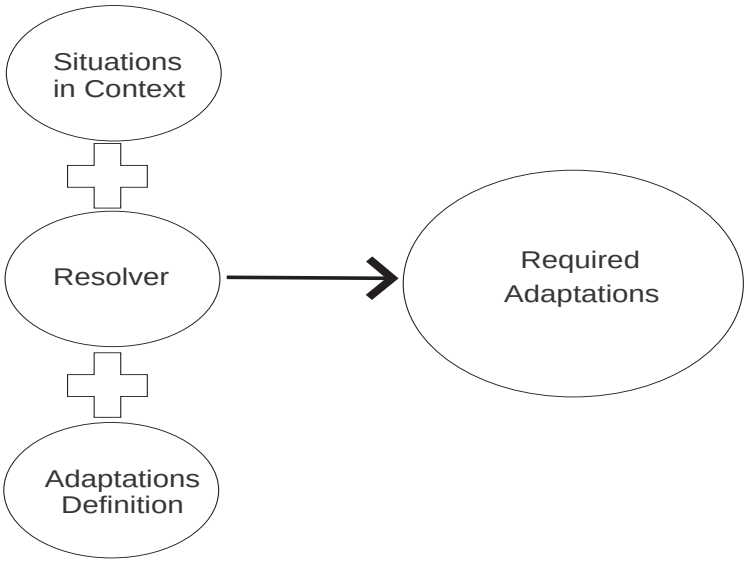

Figure 39: The Adaptation Resolver

Reacting The *Adaptation Resolver* determines appropriate adaptations, and calls the *Workflow Executor* to execute each adaptation. The *Workflow Executor* allocates reactions and activates them based on the execution path. Reactions connect to actuators through *Actuator Controllers* and implement the actions after checking the required policies.

7.3.2 Salesman Case Study

A salesman goes around different cities, visiting customers, collecting information and orders, and distributes products to customers. The products themselves are loaded from a warehouse by the salesman before the tour begins everyday. The tours vary from day to day and are largely driven by context and knowledge extracted from Decision Support Systems. Context information is not only limited to the salesman, the notion of context could be much wider. Statistical data and knowledge extracted from Decision Support Systems (DSS) is a potential candidate of a different source of context information. Such information would help to detect and react upon highly abstracted business situations that affect strategic planning and have long term adaptations. The mobile nature of the salesmen's tour provides a dynamic distributed environment in which contexts change dynamically, which in turn requires real-time reactions and adaptations. Context information also includes constraints in the environment, such us road traffic, and the personal health condition of the salesman. Appropriate reactions, if taken at the right time, could produce a remarkable business value and directly affect the productivity of the salesmen and consequently promote the business. This problem, although not fully stated, was investigated at some depth. Below, a summary of the investigation and how an implementation was arrived at are given. The CAF architecture, being a flexible design, several extensions to the implementation described below are possible. These are outlined at the end.

Review of Context for this Domain

Salesmen start their days at some warehouse, and load goods based on the customers they may visit at the same day. Salesmen then start visiting their customers and supply them

with their needs, collect payments and collect returns. Warehouse management is a key factor for business success. Thus, there is always a need to increase the productivity of the warehouse through optimizing loading, storing and serving customers. Some sensors technologies such as Radio Frequency IDentification (RFID) are used now by sales and distribution companies [Lex07] to help count, check-in and check-out items. The same technology could also be used to identify salesmen when they enter a facility and furthermore to prepare shipment if information is available.

Some context information, such as weather and road conditions, that affects deeply the salesmen work must be tracked. Salesmen schedule could be altered completely due to an accident on a highway. Context awareness could assist salesman in the everyday activity by identifying customers when approaching a neighborhood, viewing information about them, suggesting sales, reminding them with payments, and the sales. At the system level, context information is on the scope of the whole application. Domain knowledge that may be gained through DSS could be used as context information. They could be aggregated and manipulated to construct situations and define adaptation to deal with these situations.

Sensors

This problem requires many types of sensory data to be collected. A sample list of sensor types and the information perceived through them are listed below.

Date & Time Sensor The context variables defined by this sensor type are current *Date* and *Time*.

Salesman GPS Sensor The Global Positioning System (GPS) sensor could be a standalone device or integrated in the salesman tablet or cell phone. This device is responsible for providing updated information about the geographic location of the salesman. The context variables defined by this sensor type is *salesmanGPS*.

RFID Readers Salesman's vehicle contains Radio Frequency IDentification (RFID) sensor that is used to identify objects with special tags. This sensor could be used to discover goods checked in or out from the vehicle and for statistical purposes as well. The context variables defined by this sensor type are *shipmentLoaded* and *shipmentUnloaded*. Warehouses have RFID readers that identify salesmen and goods. The context variables defined by this sensor type is *warehouseRFIDReader*.

Warehouse Loading Manager The context variables defined by this sensor type is *shipmentisReady*. The warehouse loading manager uses this sensor variable to notify the system that items are ready to be loaded. A workstation computer, handheld device or a cell phone could be used for such purpose.

Traffic & Weather Condition Sensors For a specific set of roads and weather conditions a subscription to these types of service deliver information identified through sensor variables. The context variables defined by these sensor types are respectively *roadCondition*, and *weatherCondition*.

Salesman Status Salesman status is a sensor type responsible for determining whether or not the salesman is on duty. The context variables defined by this sensor type are *salesManID*, *callingSick*, *onVacation* or *carIsBroken*.

Business Locator This sensor type is to notify the system with newly opened business nearby a specific location. The context variables defined by this sensor type are *newCustomerinRange* and *newCustomerAddress*.

System Database The sensor type at the system database is associated with context variables *itemsOnSale* and *averageSalesmanSales*. They are used for reasoning and require Reasoner Extenders, as defined in the next section.

Data warehouse This sensor type provides information helpful for identifying system level situations. The context variables defined by this sensor type are *suggestedSale*, *suggestedCut*. They are used for reasoning and require Reasoner Extenders, as defined in the next section.

Reasoner Extenders

Reasoner extenders are user defined functions that provides extension mechanism for reasoning over contexts. Often such functions provides the ability to test customer conditions related to the application domain. Using Reasoner Extenders enriches the Situation Expression Language with dynamic extensions. Following is part of the reasoner extenders used in this case study.

Salesman Sales This function accepts a salesman ID and returns the volume of sales for the identified salesman.

Item Sales This function accepts item ID and returns the number of items sold.

Is On Debt This function accepts Customer ID and returns whether or not the customer is on debt.

Is Good This function accepts Customer ID and returns whether or not this customer is in good standing, as categorized by the decision support system.

Situations

Several situations can be created from sensor readings. These include the situations (1) *Salesman in Warehouse*, (2) *Shipment Ready*, (3) *Shipment Loaded*, (4) *Salesman on Road*, (5) *Environment Changed*, (6) *Salesman plan affected*, (7) *Potential Customer*, (8) *Salesman on Customer*, (9) *Shipment unloaded*, (10) *Items on Sale*, (11) *Good Customer*, (12) *Customer on Debt*, (13) *Good Customer on debt*, (14) *Bad Customer in debt*, (15) *Salesman*

Not Working, (16) *Outstanding Salesman*, (17) *Bad Salesman*, (18) *Good Selling Item*, (19) *Bad Selling Item*, (20) *Suggest Sale* and (21) *Suggest Cuts*. Their semantics are specified as shown below.

Salesman in Warehouse

Salesman in Warehouse = { ($IsSalesman[warehouseRFIDReader]
OR $IsWarehouse[salesmanGPS])}

Shipment Ready *Shipment Ready* = { Salesman in warehouse AND (s*hipmentisReady*) }

Shipment Loaded *Shipment Loaded* = { Salesman in Warehouse AND (*shimpment-Loaded*) }

Salesman on Road

Salesman on Road = {(NOT $IsWarehouse[salesmanGPS] AND
NOT $IsCustomer[salesmanGPS])}

Environment Changed *Environment Changed* = { ((weatherCondition !=Previous Value) OR (roadCondition !=Previous Value)) }

Salesman plan affected *Salesman plan affected* = { *Environment Changed* AND *Salesman on Road* AND ($IsAffected[salesmanID, weatherCondition, roadCondition]) }

Potential Customer

Potential Customer = { (newCustomerInRang AND
$IsNewCustomer[newCustomerAddress]) }

Salesman on Customer *Salesman on Customer* = {($IsCustomer[salesmanGPS])}

Shipment unloaded *Shipment unloaded* = { Salesman in Customer AND (*shimpmentUnloaded*) }

Items on Sale *Items on Sale* = { Salesman in Customer AND (*itemsOnSale != NULL*) }

Good Customer *Good Customer*= { *Salesman on Customer* AND ($IsGood[salesmanGPS]) }

Customer on Debt *Customer on Debt*= { *Salesman on Customer* AND ($IsOnDebt[salesmanGPS]) }

Good Customer on Debt *Good Customer on Debt*= { *Good Customer* AND Customer in debt}

Bad Customer on Debt *Bad Customer on Debt*= { NOT Good Customer in debt}

Salesman Not Working *Salesman Not Working*= { (callingSick OR onVacation OR carIsBroken) }

Outstanding Salesman *Outstanding Salesman* = { ($SalesmanSales[SalesmanID] > 2 * averageSalesmanSales) }

Bad Salesman *Bad Salesman* = { ($SalesmanSales[SalesmanID] < averageSalesmanSales /2) }

Good Selling Item *Good Selling item* = { ($ItemSales[ItemID] > 2* averageItemSales) }

Bad Selling Item *Bad Selling item* = { ($itemSales[ItemID] < averageItemSales / 2) }

Suggest Sale *Suggested Sale* = { (sugesstedSale != NULL) }

Suggest Cuts *Suggested cuts* = { (sugesstedCut !=NULL) }

Reactions

Four basic reactions are implemented, namely *Notify*, *Check in*, *Check out* and *Recalculate list*. These reactions encapsulate the interaction with actuators and are used by the adaptations which will be described later in this chapter.

- *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* which represent the recipient of the message and (2) *Message* which represent the actual message. The Notify reaction can be implemented using SMS Text messages, emails, or method invocation depending on the underlying actuator. The messaging infrastructure is chosen based on the client application. The precondition for this reaction is having a valid connection with the destination, which is related to the underlaying actuator. Say we are using an email client actuator, the precondition is checking if there is a valid Internet connection. The postcondition is either confirming a successful delivery of the message or notifying the sender of the failure to send the message.

- *Check in*: This reaction is responsible for triggering a database transaction that check-in items in a specific account. This reaction accepts two input parameters (1) *Account Name* which represents the account the items are checked into, and (2) the actual *Items* need to be checked into this account. The precondition is having a valid connection with the database. The postcondition is either confirming that the operation was successfully completed or notifying the system of the failure to conduct this operation.

- *Check out*: This reaction is responsible for triggering a database transaction that check-out items from a specific account. This reaction accepts two input parameters (1) *Account Name* which represents the account the items are checked out from, and

(2) the actual *Items* need to be checked out from this account. The precondition is having a valid connection with the database. The postcondition is either confirming that the operation was successfully completed or notifying the system of the failure to conduct this operation.

- *Recalculate list*: This reaction is responsible for recalculating the customers list for a salesman based on specific traffic and weather condition. The reaction accepts three parameters (1) *Salesman ID* which identifies the salesman in need for this service, (2) *Weather* which represents the weather condition and (3) *Traffic* represents the traffic condition. There are no preconditions for this reaction, however the postcondition is either confirming the successful competition of this process or notifying the system of the failure to conduct this operation.

Policies

In this section we present the polices defined for this case study, namely (1) *Salesman Status Policy*, (2) *Customer Financial Standing Policy*, (3) *Customer List Recalculation Policy* and (4) *Salesman Authorization Policy*. These policies are used in the adaptations defined later in this chapter. The policies are defined as follows.

- *Salesman Status Policy*: This policy enforces that reactions should be held against active salesmen only. For example, reactions should not be held against salesmen on vacation, or retired salesmen. This policy is implemented through a policy checker that defines *Is Active* method. The method accepts *Salesman ID* as parameter and returns whether the salesman is currently active or not.

- *Customer Financial Standing Policy*: This policy enforces that certain actions are not executed when the customer has an outstanding balance. For example, a salesman can not check items in a customer account with an outstanding balance. The policy is implemented through a policy checker that defines *Is on Dept* method which accepts *Customer ID* and returns whether the customer is on dept or not.

- *Customer List Recalculation Policy*: This policy enables customers list recalculation only under certain conditions. In order for the recalculation to be worthy the context information change should exceed a certain threshold. Also, recalculation needs an Internet connection which may or may not be available. This policy makes sure that all conditions are met before executing the recalculation. The policy is implemented through a policy checker that defines *Is Necessary* method that accepts *Salesman ID*, *Weather Condition* and *Traffic Condition* as parameters and returns whether a recalculation should be held or not.

- *Salesman Authorization Policy*: This policy enforces that only authorized salesmen can perform certain activities. In some situations special offers can be made to customers in good standing. Such offers may be made only by authorized salesmen. This policy is implemented through a policy checker that defines *Is Authorized* method which accepts a *Salesman ID* as input and returns whether the salesman is authorized or not.

Adaptation

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

Prepare Shipment This adaptation is triggered in a warehouse when a salesman approaches it. The adaptation notifies the system which prepares the shipment either manually or automatically depending on the infrastructure. This adaptation is illustrated in Table 27.

Adaptation Name	Prepare Shipment
Situations	Salesman In Warehouse
Adaptation Workflow	Exec(Notify [WarehouseID, SalesmanID]) ;
Adaptation Policies	N/A
Adaptation Reactions	Notify

Table 27: Prepare-Shipment-Adaptation

Notify Salesman This adaptation is to send a message to a salesman. One instance is to notify the salesman to load the shipment when the shipment is ready in the warehouse. This adaptation is illustrated in Table 28.

Adaptation Name	Notify Salesman
Situations	Shipment Ready
Adaptation Workflow	Exec(Notify [SalesmanID, Message]);
Adaptation Policies	N/A
Adaptation Reactions	Notify

Table 28: Notify Salesman

Transfer from Warehouse This adaptation is triggered once the loading is completed in the warehouse. The items and the number of each item loaded from the warehouse should be checked out from the warehouse account and checked into the salesman account. The adaptation is illustrated in Table 29.

Adaptation Name	Transfer from warehouse
Situations	Shipment Loaded

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

Adaptation Policies	Is Active
Adaptation Reactions	Check in Check out Notify

Table 29: Transfer from Warehouse

Transfer from Salesman This adaptation is triggered once the shipment is unloaded at the customer's place. The items and the number of each item unloaded should be checked out from the salesman's account and checked into the customer account. The adaptation is illustrated in Table 30.

Adaptation Name	Transfer from Salesman
Situations	Shipment Loaded

Adaptation Workflow	If ($IsActive[SalesmanID])
	{
	If (NOT $IsIndebt[CustomerID])
	{
	Exec(Checkin [CustomerID, Goods]);
	Exec(CheckOut [SalesmanID, Goods]);
	}
	}
	
	else
	Exec(Notify[SalesmanID, “Your account is not active”]);
	}
	else
	Exec(Notify[SalesmanID, “The Customer is on debt”]);

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

Table 30: Transfer from Salesman

Recalculate Tour Each salesman has to recalculate the tour map for visiting customers, based on road conditions and/or business to be conducted. This adaptation is illustrated in

Table 31.

Adaptation Name	Recalculate Customer List
Situations	Salesman Plan affected

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

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

Table 31: Recalculate Customer List

Suggest New Visit This adaptation is triggered when new customer is discovered on a nearby location. This adaptation is illustrated in Table 32.

Adaptation Name	Suggest Visit
Situations	Potential Customer

Adaptation Workflow	If ($IsActive[SalesmanID])
	{
	Exec(Notify[SalesmanID, “Visit nearby customer”, NewCustomerAddress]);
	}

Adaptation Policies	Is Active
Adaptation Reactions	Notify

Table 32: Suggest Visit

Suggest Customer Order This adaptation is triggered when the system is aware of the customer currently visited by the salesman. The adaptation suggests to the salesman an offer a specific order to the customer. This adaptation is illustrated in Table 33.

Adaptation Name	Suggest Customer Order
Situations	Items on sale

Adaptation Workflow	If ($IsActive[SalesmanID])
	{
	Exec(Notify[SalesmanID, salesItems]);
	}

Adaptation Policies	Is Active
Adaptation Reactions	Notify

Table 33: Suggest Customer Order

Offer Discount A discount could be offered by a salesman to customers with good standing. This adaptation is illustrated in Table 34.

Adaptation Name	Offer Discount
Situations	Good Customer

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

Adaptation Policies	Is Active Is Authorized
Adaptation Reactions	Notify

Table 34: Offer Discount

Offer Waver This adaptation is in response to the situation when a customer is in a good standing and also on debt. The salesman can offer a waiver to this customer and thus the customer can get more items. This adaptation is illustrated in Table 35.

Adaptation Name	Offer Waver
Situations	Good Customer on debt

Adaptation Workflow	If ($IsActive[SalesmanID] AND $IsAuthorized[SalesmanID])
	{
	Exec(Notify[SalesmanID, "Offer a waver"]);
	}

Adaptation Policies	Is Active Is Authorized
Adaptation Reactions	Notify

Table 35: Offer Waver

Pass This adaptation is triggered in response to a customer whose standing is only average and is on debt. The salesman is instructed by this adaptation to ignore this customer. This adaptation is illustrated in Table 36.

Adaptation Name	Pass
Situations	Bad Customer on debt

Adaptation Workflow	If ($IsActive[SalesmanID])
	{
	Exec(Notify[SalesmanID, "Pass, no offer should be made"]);
	}

Adaptation Policies	Is Active
Adaptation Reactions	Notify

Table 36: Pass

Notify Nearby Salesman This adaptation is triggered by the system to notify a salesman on duty to cover the work of another salesman who is disabled. A nearby salesman is chosen based on location and time parameters. This adaptation is illustrated in Table 37.

Adaptation Name	Notify Nearby salesman
Situations	Salesman not working
Adaptation Workflow	Exec(NotifyNearBySales-man[SalesmanID]);
Adaptation Policies	N/A
Adaptation Reactions	Notify Near By Salesman

Table 37: Notify Nearby Salesman

Notify Manager This adaptation may be triggered for many situations. This adaptation is illustrated in Table 38.

Adaptation Name	Notify Manager
Situations	Item is selling Item is not selling Suggest Sale Suggest Cut
Adaptation Workflow	Exec(Notify [Manager, Message]);
Adaptation Policies	N/A
Adaptation Reactions	Notify

Table 38: Notify Manager

Actuators

The actuators needed for this case study are *Account Actuator*, *System Functions Actuator* and *Messaging Actuator*.

Account Actuator It is responsible for managing account transactions for customers, salesmen and warehouses accounts. This actuator uses a database connector.

System Functions Actuator This 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.

Messaging Actuator This actuator is used by the Notify reaction defined previously. The connector used for this actuator depends on the type of messages, such as email, text message, etc...

Case Study implementation

This case study, with all details given above, was implemented on Windows phone 7 and Silverlight 4 as shown in Figure 40. We omit the specific details on programming.

Some Test Results

In order to verify the correctness of the code a test driven methodology was followed. We approached testing CAF from a white box perspective. The unit test cases for each component were implemented right after the component is implemented and before implementing other parts of the system. For that reason, we created stubs to mock the functionality of other component and that allowed us to focus more on each component functionality as an isolated unit of development. The test cases chosen for each component corresponds to the component responsibilities and failure scenarios.

After implementing all components and their subsequent test cases, we created integration tests to verify that all the components functions properly when interacting with each other and also to verify the bootstrapping operation. During testing phase we discovered

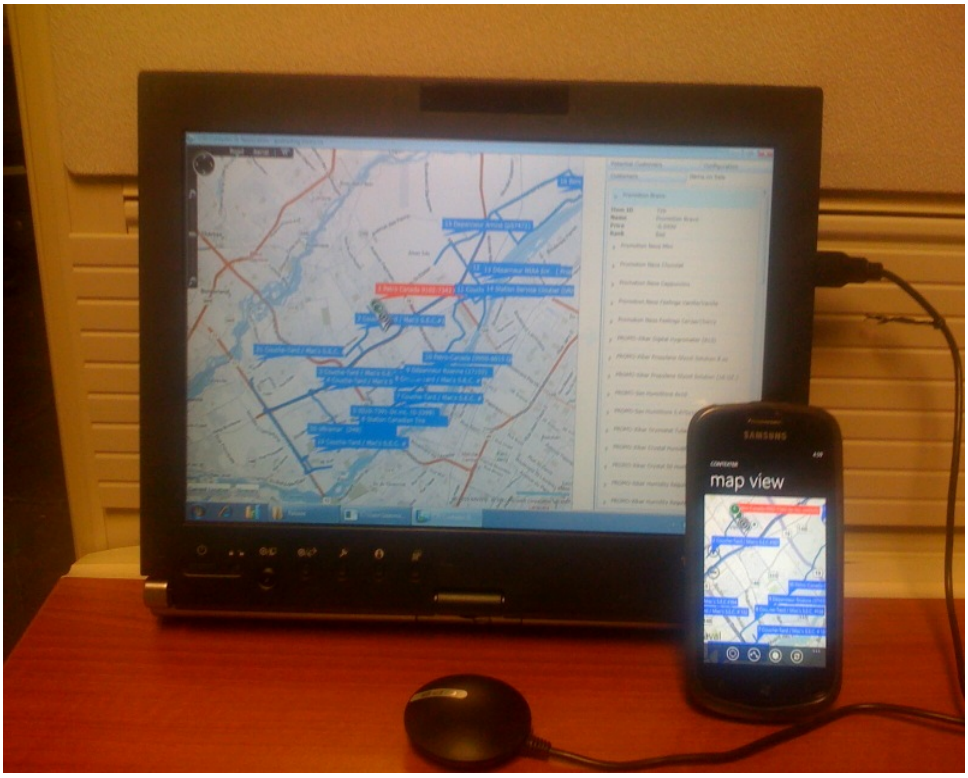

Figure 40: Salesman Case Study Implementation- Phone and Tablet Applications

design problems, thus we iterated back and forth to optimize our design. That highlights the testing role in verifying design in addition to functionality.

In software testing there are multiple measurements to measure the confidence of the software. One important measure is **code coverage** which indicates the percentage of code covered. Table 39, Figure 41 and Figure 42 shows the results of code coverage we achieved. In total we had **88.92%** code coverage for the whole framework which contains more than **7,000** line of code.

However, CAF testing remains limited since testing and verification is outside the scope of this work. Future tests should take into consideration quality and stress tests and also should put more focus on integration tests. As a future work, we are looking forward to conduct a formal design verification on CAF to assure the trustworthiness and dependability of this work.

Project	**Coverage Rate**
Adaptation	91.92%
Aggregation	81.55%
Context	100%
Framework Project	91.79%
Situation Reasoner	86.16%
Workflow	100%
Overall	*88.92%*

Table 39: Framework Test Statistics

Hierarchy	Not Cover...	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
sofian@SOFIAN-PC 2011-02-08 12:07:24	260	11.08 %	2086	88.92 %
CAF.Adaptation.dll	8	8.08 %	91	91.92 %
CAF.Aggregation.dll	31	18.45 %	137	81.55 %
CAF.Context.dll	0	0.00 %	59	100.00 %
CAF.Framework.dll	57	18.21 %	256	81.79 %
CAF.SituationReasoner.dll	164	13.84 %	1021	86.16 %
CAF.Workflow.dll	0	0.00 %	522	100.00 %

Figure 41: The Framework Tests Results

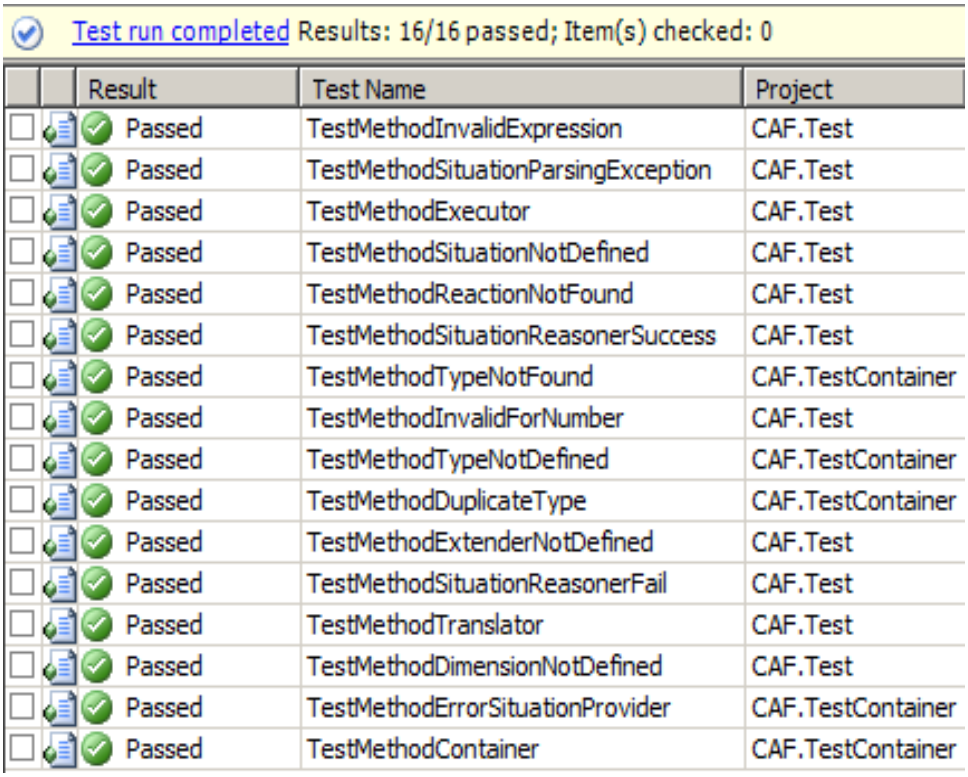

Figure 42: The Framework Unit Tests

Chapter 8

Conclusion And Future Work

The essence of this thesis work is in defining a component based methodology for constructing a *Context-aware Framework* on which any context-aware application can be implemented. This CAF can be used by software developers to empower existing and new applications with awareness capabilities. 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*.

An analysis presented in Chapter 2 has revealed the inadequacies in the existing approaches for constructing context-aware applications. Given the current trend in pervasive and mobile computing applications there is a definite need for a generic architecture for CAF. 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.

The component based architecture for CAF proposed in Chapter 5 is based on the abstract three-tiered architecture of [WAP06]. A detailed design of the proposed architecture has been explained in Chapter 6. A full implementation of the suggested CAF has been done and its expressiveness is illustrated with two case studies in Chapter 7. In particular, the framework implementation includes a full implementation of the *Situation Expression*

Language and the *Workflow Expression Language*, the two key features that distinguish this thesis work from the rest. The implementation of the former is accomplished through a reasoner to construct context situations based on their definitions and context data, and an implementation of the later is achieved through an execution engine for triggering reactions and enforcing polices.

We followed a strict test driven methodology for implementing the framework. In addition, the implementation was tested from a white box perspective.

8.1 Summary

In this section we evaluate the Context-aware Framework CAF with respect to the requirements stated in the summary of Chapter 3.

- *Defining context data model*: In Chapter 4 we provided a formal definition of atomic context information based on [WAN06] Box notation.

- *Defining context situation handler*: In section 4.6 we defined the *Situation Expression Language* to uncover the semantics behind the aggregation of context information.

- *Defining Reasoning Mechanism*: In Chapter 6 we provided the detailed design of the *Situation Reasoner*. This CAF component is responsible for inferring situations that exist in a given context. The situations are defined in the *Situation Expression Language*.

- *Supporting different sensors & Actuators types*: The *Sensor Listener*, *Actuator Controller* and *Translator* defined in Chapter 5 provide CAF with the ability to deal with different types of sensors and actuators, and to translate data between different formats.

- *Verification*: The *Sensor Verifier*, proposed in Chapter 5, provided CAF with the ability to verify sensor's reading. Additionally, the execution policies expressed in the

adaptation's workflow verify that reactions are always taken with respect to certain rules.

- *Context Aggregation*: The *Data Synchronizer*, defined in Chapter 5, empowers CAF with the ability to aggregate sensors readings and to insure that context information are always synchronized and up to date.

- *Communication and Connection*: 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 Chapter 5, to support different methods and protocols of communications.

- *Adaptations and Policy*: In Chapter 4.7 a formal definition of the *Workflow Expression Language* is provided. The language supports the introduction of execution policies as workflow constraints.

8.2 Assessment

In this section, we verify the architecture and design of the proposed Context-aware Framework (CAF) with respect to the three quality attributes *Reusability*, *Testability*, and *Scalability*.

- ***Reusability***: The Component-based Architecture (CBD) chosen for designing CAF allowed us to define each component separately as an autonomous unit of deployment. That privileged us with the ability to reuse CAF component 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 CAF have a well defined input and output, which allows defining independent unit tests for each component with stubs or proxies to simulate the functionality of other components. Integration testing of CAF will require both incremental testing and formal verification.

- ***Scalability***: The design of CAF is scalable in different aspects.
  - CAF design is interface-driven design 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***: The distributed nature of the Context-aware Framework allowed us to address performance issues that raise in environments with restricted resources. For example, due to the limited battery and computing power of handheld devices all compute bound tasks, such as replanning, are done in a dedicated server. The handheld device communicates with the server whenever context changes.

8.3 Future work

Enriching software with context-awareness increases human dependability on computers. Therefore, studying the trustworthiness of context-aware systems is an interesting issue, which could include identifying their unique verifiable attributes and projecting current approaches of building dependable systems on context-aware applications. Incidentally this aspect will require formal analysis at both design and deployment phases.

The issues of presenting the semantics behind context is always a moving target and introducing more effective and expressive means of semantic presentation should bring a remarkable contribution in many domains.

While systems are providing services empowered with context-awareness, context-awareness is never the main motive of building applications. It is always a secondary feature that can be added to enhance the major service provided by any system. Thus, studying *Self-awareness*, which can be defined as the internal monitoring of the system resources and the relation between context-awareness and self-awareness is a challenge to address. An investigation of self-awareness and context-awareness will take us into the

realm of *autonomic systems design* [KC03], a challenging area of research.

Appendix A

Situation Expression Language

This appendix contains the full documentation of the Context-Free Grammars (CFG) of the *Situation Expression Language*. The grammars are provided in the BackusNaur Form (BNF).

```
<Situation>     ::= "{" < SituationRule > "}" |
                        <LiteralSituationRule>

<SituationRule> ::=   <ANDSituationRule>        |
                      <ORSituationRule>         |
                      <NOTSituationRule>        |
                      <LiteralSituationRule>    |
                      <TokenSituation>

<ANDSituationRule> ::= <SituationRule> "AND"
                        <SituationRule>

<ORSituationRule> ::= <Situation>  "OR"
                      <Situation>

<NOTSituationRule> ::="NOT"  <Situation>
```

```
<LiteralSituationRule> ::=  "{" <Dimension> "}"

<Dimension> ::= "("  <DimensionRule> ")"

<DimensionRule> ::=    <BraceDimension>            |
                       <ANDDimensionRule>          |
                       <ORDimensionRule>           |
                       <FUNCDimensionRule>         |
                       <NOTDimensionRule>          |
                       <ADDDimensionRule>          |
                       <DIVDimensionRule>          |
                       <SUBDimensionRule>          |
                       <MULDimensionRule>          |
                       <MULDimensionRule>          |
                       <EqualDimensionRule>        |
                       <NotEqualDimensionRule>     |
                       <BiggerDimensionRule>       |
                       <BiggerOrEqualDimensionRule> |
                       <SmallerDimensionRule>      |
                       <SmallerOrEqualDimensionRule>|
                       <TokenDimensionRule>        |
                       <DimensionValue>

<ParamList> ::= ", " <Param> |
                     <Param>

<Param> ::= <TokenDimension> | <DimensionValue>

<BraceDimension> ::= "(" <DimensionRule> ")"

<ANDDimensionRule> ::= <DimensionRule> "AND"  <DimensionRule>
```

```
<ORDimensionRule> ::= <DimensionRule> "OR" <DimensionRule>

<FUNCDimensionRule> ::= <FunctionName> "[" <ParamList> "]"

<NOTDimensionRule> ::= "NOT" <DimensionRule>

<ADDDimensionRule> ::= <DimensionRule> "+" <DimensionRule>

<DIVDimensionRule> ::= <DimensionRule> "/" <DimensionRule>

<SUBDimensionRule> ::= <DimensionRule> "-" <DimensionRule>

<MULDimensionRule> ::= <DimensionRule> "*" <DimensionRule>

<NotEqualDimensionRule> ::= <DimensionRule> "!=" <DimensionRule>

<BiggerDimensionRule> ::= <DimensionRule> ">" <DimensionRule>

<BiggerOrEqualDimensionRule> ::= <DimensionRule> ">="
                                 <DimensionRule>

<SmallerDimensionRule> ::= <DimensionRule> "<" <DimensionRule>

<SmallerOrEqualDimensionRule> ::= <DimensionRule> "=="
                                  <DimensionRule>

<EqualDimensionRule> ::= <DimensionRule> "==" <DimensionRule>

<DimensionValue> ::= <NUMBER> | <STRING>
```

Operator precedence is in the following order.

*, /, Not,

+, -, AND,

>, <, =>, =<, ==, OR.

Appendix B

Workflow Expression Language

This appendix contains the full documentation of the Context-Free Grammars (CFG) of the *Workflow Expression Language*. The grammars are provided in the BackusNaur Form (BNF).

```
<Workflow> ::= <StatementCollection>

<StatementCollection> ::= <StatementCollection> <Statement> |
                                    <Statement>

<Statement> ::=   <WhileStatement>                    |
                  <ForStatement>                      |
                  <IfStatement>                       |
                  <IfElseStatement>                   |
                  <ExecuteStatement>                  |
                  <BraceStatement>


<BraceStatement> = "{" <Statement> "}"

<WhileStatement> ::= "while" <Condition> <Statement>
```

```
<ForStatement> ::= "for" "(" <NUMBER> ")" <Statement>

<IfElseStatement> ::= <IfStatement> |
                      <IfStatement> "else" <Statement> *prefered

<IfStatement> ::= "if" <Condition> <Statement>

<ParamList> ::= ", "  <Param> |
                      <Param>

<Param> = <TokenDimension> | <DimensionValue>

<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> ")" ";"

Operator precedence is in the following order.

Not AND OR

Bibliography

[ABM10] Richard Han Aaron Beach, Mike Gartrell and Shivakant Mishra. Cawbweb: Towards a standardized programming framework to enable a context-aware web. Technical report, Department of Computer Science, University of Colorado at Boulder, 2010.

[Bar05] Jakob E. Bardram. The java context awareness framework (jcaf) a service infrastructure and programming framework for context-aware applications. In Hans W. Gellersen, Roy Want, and Albrecht Schmidt, editors, *Pervasive Computing*, volume 3468 of *Lecture Notes in Computer Science*, pages 98–115. Springer Berlin / Heidelberg, 2005.

[BBKK01] John Barton, John J. Barton, Tim Kindberg, and Tim Kindberg. The cooltown user experience. Technical report, 2001.

[BC] G. Biegel and V. Cahill. In *Pervasive Computing and Communications, 2004. PerCom 2004. Proceedings of the Second IEEE Annual Conference on, title=A framework for developing mobile, context-aware applications, year=2004, month=march, pages= 361 - 365, doi=10.1109/PERCOM.2004.1276875*.

[BLHL01] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. http:www.scientificamerican.comarticle.cfm?id=the-semantic-web; 20 May 2009, 2001.

[BMK+00] Barry Brumitt, Brian Meyers, John Krumm, Amanda Kern, and Steven Shafer. Easyliving: Technologies for intelligent environments. In Peter Thomas and

Hans-W. Gellersen, editors, *Handheld and Ubiquitous Computing*, volume 1927 of *Lecture Notes in Computer Science*, pages 97–119. Springer Berlin / Heidelberg, 2000.

[BSNc07] Nathalie Bricon-Souf, Conrad R. Newman, and Health care. Context awareness in health care: A review. *International Journal of Medical Informatics*, 76(1):2 – 12, 2007.

[CJP+08] Huanhuan Cao, Daxin Jiang, Jian Pei, Qi He, Zhen Liao, Enhong Chen, and Hang Li. Context-aware query suggestion by mining click-through and session data. In *Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '08, pages 875–883, New York, NY, USA, 2008. ACM.

[CL07] Vlad Coroama and Marc Langheinrich. From sensors to semantics: Intelligent context for situated computing, oct 2007.

[CM04] Grindle Charles and Lewis Michael. Automating terrain analysis: Algorithms for intelligence preparation of the battlefield. *Human Factors and Ergonomics Society Annual Meeting Proceedings*, 48, 2004.

[Dey00] Anind K. Dey. *Providing Architectural Support for Building Context-Aware Applications*. PhD thesis, 2000.

[DLP+10] Mikael Desertot, Sylvain Lecomte, Dana Popovici, Marie Thilliez, and Thierry Delot. A context aware framework for services management in the transportation domain. In *New Technologies of Distributed Systems (NOTERE), 2010 10th Annual International Conference on*, 31 2010.

[GCM+07] E. Goh, D. Chieng, A.K. Mustapha, Y.C. Ngeow, and H.K. Low. A context-aware architecture for smart space environment. In *Multimedia and Ubiquitous Engineering, 2007. MUE '07. International Conference on*, pages 908 –913, 2007.

[GE95] JONSON Ralph VLISSIDES John GAMMA Erich, HELM Richard. *Design Patterns : Elements of Reusable Object Oriented Software*. 1995.

[GFSB11] Ning Gui, Vincenzo De Florio, Hong Sun, and Chris Blondia. Toward architecture-based context-aware deployment and adaptation. *Journal of Systems and Software*, 84(2):185 – 197, 2011.

[GGR+09] Feng Gui, M. Guillen, N. Rishe, A. Barreto, J. Andrian, and M. Adjouadi. A client-server architecture for context-aware search application. In *Network-Based Information Systems, 2009. NBIS '09. International Conference on*, pages 539 –546, Aug. 2009.

[GKJ+10] Diwakar Goel, Eisha Kher, Shriya Joag, Veda Mujumdar, Martin Griss, and Anind K. Dey. Context-aware authentication framework. In Ozgur Akan, Paolo Bellavista, Jiannong Cao, Falko Dressler, Domenico Ferrari, Mario Gerla, Hisashi Kobayashi, Sergio Palazzo, Sartaj Sahni, Xuemin (Sherman) Shen, Mircea Stan, Jia Xiaohua, Albert Zomaya, Geoffrey Coulson, Thomas Phan, Rebecca Montanari, and Petros Zerfos, editors, *Mobile Computing, Applications, and Services*, volume 35 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 26–41. Springer Berlin Heidelberg, 2010.

[Har78] M. A. Harrison. *Introduction to Formal Language Theory*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1978.

[KC03] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41 – 50, January 2003.

[Lex07] Lexar. Lexar gets clear snapshot of warehouse inventory with alien rfid, 2007.

[LOIP10] Tom Lovett, Eamonn O'Neill, James Irwin, and David Pollington. The calendar as a sensor: analysis and improvement using data fusion with social

networks and location. In *Proceedings of the 12th ACM international conference on Ubiquitous computing*, Ubicomp '10, pages 3–12, New York, NY, USA, 2010. ACM.

[MA11] Mubarak Mohammad and Vangalur S. Alagar. A formal approach for the specification and verification of trustworthy component-based systems. *Journal of Systems and Software*, 84(1):77–104, 2011.

[MCC10] MARTIN-COCHER GAELLE SHENFIELD MICHAEL MCCOLGAN, BRIAN. Method and system for a context aware mechanism in an integrated or distributed configuration. Patent, Canadian Intellectual Property Office, 2010.

[OPB+99] E. Ouaviani, A. Pavan, M. Bottazzi, E. Brunelli, F. Caselli, and M. Guerrero. A common image processing framework for 2d barcode reading. In *Image Processing and Its Applications, 1999. Seventh International Conference on (Conf. Publ. No. 465)*, volume 2, pages 652 –655 vol.2, 1999.

[RS06] Anca Rarau and Ioan Salomie. Adding context awareness to c-sharp. In Paul Havinga, Maria Lijding, Nirvana Meratnia, and Maarten Wegdam, editors, *Smart Sensing and Context*, volume 4272 of *Lecture Notes in Computer Science*, pages 98–112. Springer Berlin / Heidelberg, 2006.

[RZPM09] P. Raphiphan, A. Zaslavsky, P. Prathombutr, and P. Meesad. Context aware traffic congestion estimation to compensate intermittently available mobile sensors. In *Mobile Data Management: Systems, Services and Middleware, 2009. MDM '09. Tenth International Conference on*, pages 405 –410, May 2009.

[TL09] David Pollington James Irwin Tom Lovett, Eamonn ONeill. Event-based mobile social network services. In *MobileHCI*, 2009.

[uRS08] Aqeel ur Rehman and Z. A. Shaikh. Towards design of context-aware sensor grid framework for agriculture. In *Fifth International Conference on Information Technology*, pages 244–247, Rome, Italy, 2008. XXVIII-WASET.

[VSL03] Pravin Vajirkar, Sachin Singh, and Yugyung Lee. Context-aware data mining framework for wireless medical application. In *Database and Expert Systems Applications*, volume 2736 of *Lecture Notes in Computer Science*, pages 381–391. Springer Berlin / Heidelberg, 2003.

[vSPK04] Mark van Setten, Stanislav Pokraev, and Johan Koolwaaij. Context-aware recommendations in the mobile tourist application compass. In Paul De Bra and Wolfgang Nejdl, editors, *Adaptive Hypermedia and Adaptive Web-Based Systems*, volume 3137 of *Lecture Notes in Computer Science*, pages 515–548. Springer Berlin / Heidelberg, 2004.

[W3C10] W3C. Ontologies. http://www.w3.org/standards/semanticweb/ontology, 2010.

[WA07] Kaiyu Wan and Vasu Alagar. Security contexts in autonomic systems. In Yuping Wang, Yiu-ming Cheung, and Hailin Liu, editors, *Computational Intelligence and Security*, volume 4456 of *Lecture Notes in Computer Science*, pages 806–816. Springer Berlin / Heidelberg, 2007.

[WAN06] Kaiyu WAN. *LUCX: LUCID ENNRICHED WITH CONTEXT*. PhD thesis, 2006.

[WAP06] Kaiyu Wan, Vasu Alagar, and Joey Paquet. An architecture for developing context-aware systems. In Thomas Roth-Berghofer, Stefan Schulz, and David Leake, editors, *Modeling and Retrieval of Context*, volume 3946 of *Lecture Notes in Computer Science*, pages 48–61. Springer Berlin / Heidelberg, 2006.

[ZLWX08] Yu Zheng, Like Liu, Longhao Wang, and Xing Xie. 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, pages 247–256, New York, NY, USA, 2008. ACM.