JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

Supporting Context Awareness in Highly Dynamic Network Environments

Dissertation zur Erlangung des akademischen Grades *Dr.tech.*
in der Studienrichtung *Informatik*

Angefertigt am Institut für *Systemsoftware*

Von

Dipl.-Ing. Wolfgang Beer

Gutachter

o.Univ.-Prof. Dipl.-Ing. Dr. Hanspeter Mössenböck

o.Univ.-Prof. Mag. Dr. Alois Ferscha

Linz, *01.05.2004*

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst habe. Ich habe keine weiteren als die angeführten Hilfsmittel benutzt und die aus anderen Quellen entnommenen Stellen als solche kenntlich gemacht.

Linz, 2004

Abstract

The aim of context-aware services and applications is to react flexibly on the environmental state. Traditional services and applications are completely independent from their environment and therefore not able to react on state changes in the environment. The introduction of location-based services, which provide location-specific services to the users, represents the first practical appearance of context-aware services. Location-based services take the location information of the service client and return a location-specific service result.

Context awareness is not limited to the location information, but includes all sorts of information which are relevant to classify the situation of a service client.

This PhD thesis introduces a software framework that supports the collection and processing of generic sensor data, in order to provide the information to services and applications. The open framework architecture of the framework enables the use of different transport protocols to deliver the information. The software architecture also enables the integration of new protocol implementations. As an example transport module a web service standard based module was implemented. The integration of web service standards into the transport layer offers language and platform independent service description and delivery for a multitude of different service clients. With the discovery and lookup mechanism of the framework service clients are able to discover service providers even in ad-hoc networks.

One of the main characteristics of the software framework is the possibility to map sensorial state transitions with specific actions. Interpreted rules, which are defined by the application designer or even by a user, realize the mapping between state transitions and the actions. These rules define how the application reacts on a specific state. Through the interpretation of the rules it is possible to change the rule repository or single rules at run time and to change the application's behavior dynamically.

To discover possible communication partners, the framework implements a role-based classification mechanism. The role-based mechanism uses the collection of a service provider's service interfaces to decide in which role the service provider acts in an application. Comparable software frameworks often use static classification hierarchies to classify communication partners which leads to problems in ad-hoc networks.

The framework enables the creation of context-sensitive applications through XML configurations. XML configurations simplify the use of visual development tools for designing context-sensitive applications.

To demonstrate the framework, four example applications were developed and tested in practice.

Zusammenfassung

Das Ziel von Kontext-sensitiven Diensten und Anwendungen ist es, auf den Zustand der Umgebung flexibel reagieren zu können. Traditionelle Dienste und Anwendungen sind von ihrer Umgebung völlig unabhängig und somit auch nicht in der Lage auf Änderungen geeignet zu reagieren. Mit der Einführung von ortsbasierten Diensten, die je nach Position und Ort des Benutzers spezifisch konfigurierte Dienste anbieten, wurde der erste Schritt zu Kontext-sensitiven Diensten realisiert. Ortsbasierte Dienste verwenden nur die Position des Benutzers, um Informationen über die Umgebung des Benutzers zu sammeln und das Ergebnis von Anfragen damit geeignet zu konfigurieren.

Kontext Sensitivität bezieht sich aber nicht nur auf den Ort eines Benutzers, sondern vielmehr auf alle Informationen die Aufschluß über die aktuelle Situation eines Dienstnehmers geben.

Im Zuge dieser Arbeit wurde ein Software Framework erstellt, das es ermöglicht beliebige Sensordaten digital zu erfassen und zu verarbeiten, mit dem Ziel die gewonnenen Informationen Diensten und Anwendungen zur Verfügung zu stellen. Die offene Software Architektur des Frameworks ermöglicht es beliebige Kommunikationsprotokolle für die Übertragung der Informationen zu verwenden und jederzeit neue zu integrieren. Durch die Integration von WebService Standards ist die Beschreibung und die Ausführung der, durch das Framework angebotenen, Dienste für eine Vielzahl von verschiedenen Dienstnehmern gewährleistet. Der *Discovery* und *Lookup* Mechanismus des Frameworks ermöglicht es, dass sich Dienstanbieter und Dienstnehmer auch in einem dezentral verwalteten Netzwerk finden können.

Zu den wichtigsten Eigenschaften des Frameworks zählt die Möglichkeit Zustandsübergänge von Sensoren mit Aktionen zu verknüpfen. Diese Verknüpfung wird in Form von interpretierten Regeln angegeben, die der Anwendungsentwickler definiert oder auch vom Benutzer selbst angelegt werden können. Diese Regeln bestimmen wie sich eine Anwendung zur Laufzeit in einem bestimmten Zustand verhält. Durch die Interpredation der Regeln ist es möglich den Regelsatz oder einzelne Regeln zur Laufzeit zu verändern um die Reaktion der Anwendung dynamisch zu ändern.

Um mögliche Kommunikationspartner zu finden wurde im Framework ein rollenbasierter Ansatz zur Klassifikation gewählt. Dieser rollenbasierte Ansatz verwendet die Summe der Dienstschnittstellen eines Dienstanbieters um über dessen Rolle in einer Anwendung zu entscheiden. Vergleichbare Software Frameworks verwenden statische Klassenhierarchien, die aber in in dezentral verwalteten Netzwerken zu Problemen führen.

Das Framework ermöglicht die Erstellung von Kontext-sensitiven Anwendungen durch XML Konfigurationen. Die Erstellung von graphischer Entwicklungssoftware wird dadurch wesentlich vereinfacht. Zur Demonstration der Anwendung des Frameworks wurden 4 Beispielanwendungen erstellt und in der Praxis erprobt.

Danksagung

In erster Linie möche ich meiner Familie danken, die mir das Studium ermöglicht hat und mich immer unterstützte.

Natürlich möchte ich hier besonders meinen beiden Betreuern, Prof. Mössenböck und Prof. Ferscha, für ihre hingebungsvolle und aufopfernde Arbeit danken. Auch möchte ich an dieser Stelle meinem Projektpartner Volker Christian sowie den Siemens Projektpartnern, vertreten durch Lars Mehrmann, für die erfolgreiche und angenehme Zusammenarbeit danken.

Danken möchte ich auch meinen Kollegen Dietrich Birngruber und Albrecht Wöß, sowie der gesamten Mannschaft des Instituts für Systemsoftware, für eine sehr kreative und freundschaftliche Zeit an der Johannes Kepler Universität.

Ebenso möchte ich meinen Studienkollegen sowie Freunden innherhalb und außerhalb der Universität danken.

Linz, im Mai 2004

Contents

1 Introduction

1.1 Motivation

Ubiquitous and pervasive computing [Wei93] are the vogue terms of the early twenty first century. This sort of techniques stand for massive use of embedded systems, that should act as smart as possible to fulfil specific tasks for the users [Wei91]. Environments that are stuffed with digital equipment would not attract any user, except it is possible to invisibly integrate all the digital technologies into the users natural environment [Wa02]. A short historic overview about digital technology development explains the motivation to move forward to pervasive computing systems and environments.

Digital technology has changed very much in the last decades. At the beginning of the digital age mainframe computing was the state of the art. After the hardware prizes decreased dramatically computer scientists were able to buy computers for their own: the personal computing area began. At this time the relation changed from many users sharing one digital device to one user per digital device. As the size of the personal computers decreased, the mobile computing area started. In the current decade, a user owns a collection of different digital devices, which are specialized for certain tasks. A mobile phone is used to communicate globally while an organizer is used to store contacts or meeting information and to take small notes. Additionally, a user often owns a desktop computer for tasks that require more processor power or hardware support (e.g. a printer or a scanner) as well as a laptop device for mobile work. To support mobility in combination with global availability radio based wireless network technology emerged. Laptops, PDAs, mobile phones and a multitude of specialized mobile devices replaced the typical workstation computer. If we summarize the development we can observe that the digital devices became more and more personalized to solve problems for the user. At the beginning many users shared a common mainframe with probably different personal configuration profiles. Now, every person owns a specific set of digital tools that are exactly personalized for the user.

The next step ahead will improve the personalisation of the user's digital environment to act some sort of smart according to the user's demands. Some people are even speaking about environments where thousands of smart dust devices work together to achieve a common task [Wa00]. On the other side, it seems to be hard to enable traditional applications to run in such insecure, dynamic and short living network environments. One of the major problems is to define an abstract application layer on top of these dynamic network environments. It is impossible for a programmer to design a static application that is able to react on every situation that may occur within an environment. It is getting even worse when the application is supposed to travel through unknown environments.

Another important aspect of smart environments is targeting the human computer interaction (HCI). At the moment, a user, who is using a mobile device, is very much limited ac-

cording to his input and output possibilities. Every digital device tries to offer as much input and output interfaces as possible. As the number of embedded devices increase the need for input and output device sharing grows. The user's attention is a limited resource that does not scale in proportion to the number of devices. The massive use of digital technology should not set the user under physiological pressure [Ar99].

As a matter of fact, the technology has to disappear in our environments and has to react implicitly to the user's personal demands. Pervasive and ubiquitous computing try to solve some of these problems.

This work focuses on context-aware computing [Sch94] in combination with ad-hoc wireless networks, which should be understood as a logical consequence of pervasive environments. In order to act implicitly on existing problems it is necessary to obtain some background information about involved entities. It is necessary to identify relevant entities and their properties as well as the services they provide for solving a given problem. The identification process is not necessarily bound to a global network access, as modern wireless technologies provide communication between single peers. This communication method, called ad-hoc communication, enables the creation of locally relevant networks that are not necessarily a part of a global network. One example for such a sort of networks is a personal area network (PAN), where all personal digital devices build a local ad-hoc network. This thesis mainly focuses on the dynamic delivery of context information in wireless ad-hoc network environments and proposes new methods for modelling context-sensitive interaction scenarios. A developer or a user, in this work also called scenario designer, should be able to easily create context-aware scenarios, where a group of digital devices cooperate to solve a given problem. A software middleware should provide an abstract representation of digital and non-digital devices, as well as sensor and actuator abstractions. A scenario designer is able to combine these abstractions to create new context-aware applications.

1.2 Goals

The purpose of this work is to investigate the implementation of a software middleware that supports the delivery of context information and therefore the modelling of context-sensitive applications and scenarios. The delivery of context information has to work in traditional networks with centralized organisation, as well as in unmanaged ad-hoc networks. In order to handle the two basic problems of pervasive environments [He01] the context framework has to solve the following basic requirements:

- The integration of new hardware sensors and actuators, which are necessary to gather context information, has to be simple. Furthermore, the reuse, extension and combination of already existing sensors and actuators has to be supported to enable the convenient and rapid development of context-aware applications.
- The delivery of context information should be completely event-based.
- The dynamic exchange of context information between different entities should be possible and changeable at runtime. In order to provide more flexibility in the interaction of entities, an *ECA* (Event Condition Action) rule interpreter should be able to intercept context events and react on defined conditions.

- The identification of appearing entities inside local environments should be as flexible as possible. Such entities should not be classified by a central hierarchical class model, but by identifying certain roles in which they act at the moment of classification. By adding or removing properties of an entity it should be possible to change the entity's role at runtime.
- Most of the time, ad-hoc networks are heterogeneous sets of devices with many different physical communication mechanisms and protocols. To enable a broad spectrum of communication possibilities it is necessary to implement flexible and plugeable transport protocol modules. These transport modules should be able to coexist and it should be possible to integrate new transport modules. To support different encodings for different platforms it is necessary to specify an encoding module for every transport module.
- To enable the use of context information on different platforms and different software environments a context scenario should be configurable in a platform and programming language independent format.
- Standard applications such as chat and network conferencing clients, workflow management systems, entertainment environments, or even health and security systems can benefit from context information gathering. Since modern applications usually do not maintain context information the context middleware has to provide a common platform-independent interface to make this information accessible for many different applications. The requirement is to provide a WSDL description for enabling access to context information through SOAP-encoded messages sent over a HTTP transport module.
- Context information processing on mobile devices is limited in terms of processing power and storage. Furthermore, it is nearly impossible to develop an application directly on the mobile device. The development process has to be performed on a device with richer input and output possibilities and then the application has to be sent remotely to the mobile device. To cope with such limitations the context framework has to perform all tasks, which are running on the mobile device, without the use of heavy-weight libraries and performance-critical operations.

1.3 Contribution of this Thesis

The contribution of this thesis is the implementation of a new software framework architecture which manages the context information life cycle. This new architecture hides the complexity of gathering, processing and transporting of context information for context-aware applications and services. The architecture is designed to be open for any new module implementations concerning the transport protocol, the event encoding, and the lookup and discovery mechanism. The framework architecture allows users to plug in or change these modules at runtime. The architecture is designed to operate in ad-hoc networks as well as in traditional networks. The exchange of context information is therefore realized through a peer-to-peer mechanism which allows the direct communication of two devices without any central infrastructure such as a central server.

Another major contribution of this thesis is the use of interpreted ECA (Event Condition Action) rules to realize state transitions of entities. These rules allow programmers to define arbitrary entity interaction scenarios. Since ECA rules are interpreted, a scenario designer is

able to deploy and to change the rule repository at runtime. Changing the rule repository means that the behavior of a scenario changes at runtime. It is even possible to change the rule base as a side effect of a state transition.

The third contribution of this thesis is the use of a role-based classification mechanism to identify entities that appear in an environment. This classification mechanism allows context-sensitive applications to define new roles and to use them to specify new ECA rules. The role-based classification mechanism was specifically designed to work in highly distributed ad-hoc environments where a hierarchy-based classification mechanism would face significant consistency problems.

Furthermore, this thesis shows that the integration of web service standards into classic context awareness software frameworks has significant advantages.

1.4 Outline

Chapter 1 introduces the area of context-aware computing as well as the challenges, goals and requirements of our research.

In Chapter 2 the terms context, context awareness, smart environments, ubiquitous and pervasive computing are introduced and discussed in detail. This chapter also explains how context information is gathered and transformed to fit into a specific context model. The second part of Chapter 2 introduces basic technologies, that are relevant for our research, e.g. wireless communication, wireless object identification, ad-hoc networks, P2P communication and Sun's Jini technology.

In Chapter 3 the state of the art in context-aware computing is presented. Four different research projects are reviewed in detail: Xerox PARC's PARCTAB project, GATech's Context Toolkit, AT&T's Sentient Computing project and HP's CoolTown project.

Chapter 4 gives a detailed view on our SiLiCon framework, its architecture and its implementation. This chapter described the main parts of our research.

Chapter 5 compares the SiLiCon framework with the research projects that were already introduced in Chapter 3, according to some interesting aspects.

In Chapter 6 a collection of context-aware demonstration scenarios are presented and it is shown how a scenario designer is able to configure a new context scenario.

Chapter 7 contains concluding remarks as well as some hints how future innovations in the area of context-aware computing could look like.

2 Definition of Terms

This chapter gives a short overview about the terms and definitions that are used in this thesis to explain the area of context and context awareness.

2.1 Context and Context Awareness

The term *context* has various meanings in different research areas. In this work it addresses the information that could possibly be relevant for an object that performs a certain task. Most of the time, a task depends on context information, which has to be collected from other objects. In smart environments this means that this information has to be passed between different embedded devices over a network. The term *context-aware software* was first used in the Xerox PARC research project PARCTAB in 1994 [Schilit94]. In this work the term was defined and used for software that is able to adapt according to its actual location, the collection of nearby people, hosts and accessible devices. Also the possibility to track the changes of context information over time, in other words to store historic context information, was mentioned. Over the years, different research groups enriched this basic definition of context and context-aware software. Brown et al. [Brown], for example, widened the scope of context information to temperature, time, season and many other factors. Due to the fact that the number of context information factors is nearly unlimited, the definition of context by Anhind K. Dey is one of the most commonly used:

> *"Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves." [Dey]*

This definition of context specifies that context contains any kind of information about an entity in order to understand its situation. So context information is not limited to location information, but could also mean information about the social situation of a person or the person's mood. Usually, such a sort of context information is hard to collect, but there are a reasonable number of research projects that try to collect even this kind of information. An interesting fact about the above definition of context is that Dey identifies three base classes with which all objects can be classified: *person*, *place* and *object*. This kind of classification has practical reasons but is also fixed to a location-dependent view of context information. For simple scenarios this classification is easy to implement and performant, but complex scenarios cannot be created. What if an object that can be located in a certain place is itself a place, such as a bag or a car? Are animals identified as objects?
For this work we can reduce Dey's definition to the following part:

> *"Context is any information that can be used to characterize the situation of an entity."*

Additionally, context information should be viewed in a completely application-independent way. It should be the application's responsibility to select relevant context information and to interpret it according to the task that the application has to perform. The concrete classification of objects should also be the application's responsibility, because different applications could have a different view on the same object.

Context-sensitive applications differ from traditional applications according to their new kind of life cycle. Schilit identified a context computing cycle that contains three basic steps.

- ***Discovery***: This involves the identification of entities that are relevant for the application's tasks. In the first step, a context-aware application has to discover and to explore its environment in order to get information to work with. According to a human viewpoint, the discovery is mostly focused on the local environment, because information around the actual physical location is considered more important than anywhere else.
- ***Selection***: A context-aware application has to filter the information that was discovered according to its specific needs. The selection process is the most important and most problematic part within the context computing cycle. It is no problem to receive a multitude of different sensor data, but the identification of specific information, that has the required semantic value, is only solvable within specific constraints.
- ***Use***: If an application identified a relevant entity and was able to select a specific information, the application is able to use this information to change its configuration.

In this thesis all three steps of the context computing cycle are described in detail and a system architecture is proposed which supports the discovery, selection, and use of context information.

2.2 Representation Models for Context Information

In this section, it is shown how the representation model of context information influences the design of the software middleware framework. In order to find general terms, describing different world models, it is necessary to highlight existing notations and their affinities to the approach in this work.

The representation model of context information is called the *world model*, because it describes entities and their interaction. The definition of world model is:

> *"A world model defines how the description of entities and their relation can be represented in machine readable and changeable form."*

World models are defined to provide machine readable information about whole application environments in order support context-aware self configuring applications. Fig. 1 describes the role of a world model within a software framework that supports the development of context-aware applications. The real world contains a collection of objects (*objects 1-n*). A small spectrum of aspects of these real world objects can be gathered with the use of sensors. Information about these objects has to be stored in machine readable form. Therefore, the

world model defines structures to create a representation of these objects which can be sensed or about which we have some sort of description. The context framework middleware is responsible to retrieve the sensed information about the objects and to store it into a data structure the world model defines. A good representation of a collection of objects also contains information about the relation between the different objects. A relation is a state transition that influences the state of another object. These relations are shown between object 1 and object 2 in Fig. 1. An example for a relation between two objects could be a person that changes its location. When the location of the person changes all related objects have to be informed about the state change. The context framework middleware uses the machine readable description of objects in order to trigger actions through actuators or to deliver parts of the representation to a multitude of different context-aware applications. The framework middleware, generally, is responsible for the gathering, mapping, representation and the transport of the sensed data into its world model. A context middleware tries to remove the complexity of writing context-sensitive applications through wrapping and hiding the context information life cycle.

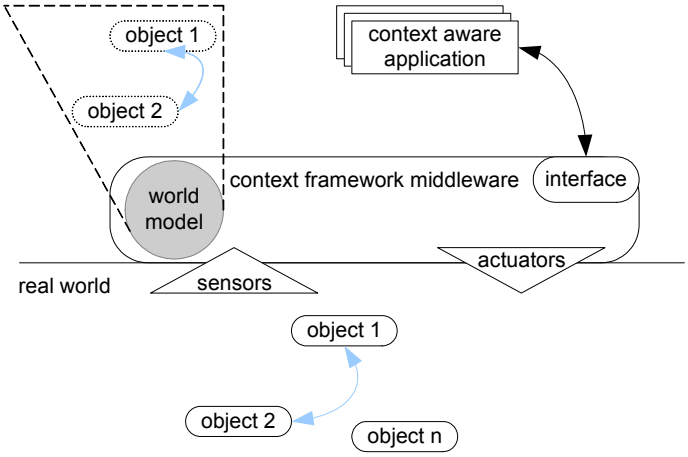

Fig. 1 World models represent a small view on object relations in the real world

The world model contains every information a context-sensitive application is able to retrieve. Different research groups, like the SemanticWeb group or various Ontology research projects, try to find general structures and types of objects in order to create representations of real world objects [Onto][SeWeb]. Such general structures are not only necessary to create a consitant representation of a real world situation but enable the interoperability of different context frameworks. Structured, machine readable representations of digital and real world scenarios are the first step towards intelligent web search algorithms and context-aware applications and services.

The implemented middleware is able to manipulate a scenario represenation at runtime. This basic relation is shown in Fig. 2. Sensors gather information from the real world which are mapped into the world model through the use of the middleware framework. On the other side, the middleware framework is able to change the state of the real world objects with actuators. Context rules are able to express the relation between the different objects. The context world model always represents a more or less small view on the real world.

Fig. 2 shows a context information life cycle that represents the information flow inside the project's middleware. The context information life cycle was defined in [Fer03]:

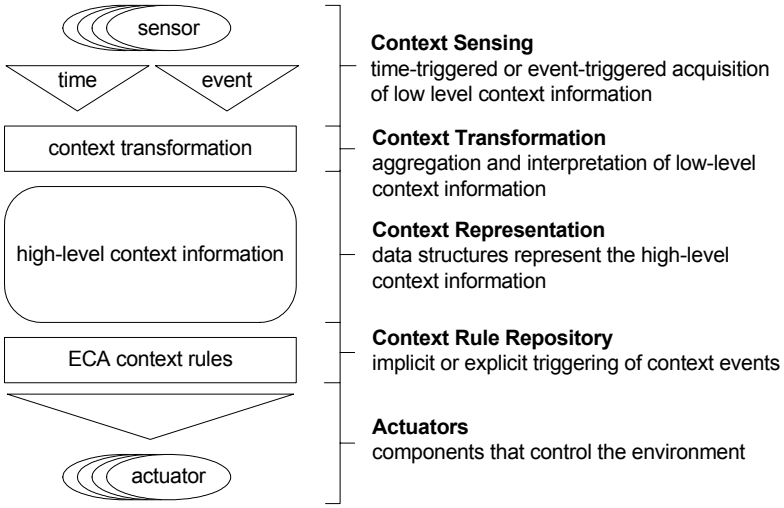

Fig. 2 Context information life cycle

Due to the different requirements of context-related research projects, different world models resulted. While the *HP Cooltown* [HP01] project developed a world model that is based on three kinds of objects (Person, Thing and Place), the *Context Toolkit* [Dey01] does not classify its information within such a basic hierarchy.

One of the reasons why so many different world models have been devised is that the projects focus on specific areas in the context information life cycle. The CoolTown project focusses on the context information's web representation. In contrast, the Context Toolkit tries to support the context transformation process for the programmers.

2.3 Smart Environments

Smart environments, also called *intelligent spaces*, appear to be the next step towards a natural interaction between users and digital devices [Oxy]. Places or environments invisibly filled with a multitude of different *sensors* and *actuators* such as cameras, location trackers, identification transponders, surround sound systems, displays, or public media walls [WebWall] and many more, try to help the user to solve certain tasks. The goal of this technology is to improve the interaction between the user and its digital devices to happen implicitly and unconsciously, so that the user does not realize how complex the interaction scenarios in the background are. The human-computer interaction could be performed by natural input technologies like gesture recognition, speech recognition, human emotion recognition, human action sequence recognition, or even through tangible interfaces and artifacts [Ishii97]. The vision is that an optimal interaction between an embedded system and a human user would demand no extra information displayed or typed into a static interface, which would need the user's full concentration. A human user could interact with his digital environment as he would do with a human being. Unobstrusively, the user should be able to control a collection of embedded devices such as sensors and actuators.

Applications of smart environments are not limited to office rooms but unfold their immense possibilities also in areas like transportation, automotive, logistics, industry, home and even in personal area networks, which could be embedded in smart clothing. One of the reasons why the pervasive technology emerges to improve personal computing, is the incredible progress in chip design. Today, single chips can host entire digital systems that are easy to embed because of their tiny sizes and low costs. The era of *Pervasive Computing* was born. IBM's definition of Pervasive Computing is:

> *'Convenient access, through a new class of appliances, to relevant information with the ability to easily take action on it when and where you need it'*

Pervasive Computing is based on four fundamental characteristics, which are [Per01]:

- ***Decentralization***: The change from a centralized digital system like a mainframe, to a strongly decentralized computing environment. Today, the development goes from a Personal Computer to a multitude of tiny or embedded devices, which interact to solve the users' problems (like a headset, cell phone and a PDA).
- ***Diversification***: Diversification means the shift from a single universal device, which is able to perform a large variety of tasks from entertainment to professional computing, to many specialized devices.
- ***Connectivity***: Pervasive and Ubiquitous Computing environments have a strong demand for connectivity and communication. Connectivity between traditional computing systems requires a wired network between static digital nodes. Connectivity in pervasive environments means to connect a multitude of heterogeneous mobile devices, operating systems and system architectures with a wireless network.
- ***Simplicity***: Decentralization, diversification, and connectivity provide a lot more possibilities than traditional paradigms but on the other side these characteristics also complicate the use of such devices for human users. The need for simplicity is a direct consequence of the increasing complexity of pervasive systems. Convenience, ubiquity, and intuitiveness are the requirements modern environments have to fulfil.

The area of *grid computing* is often mentioned in the same context as pervasive computing. Grid computing tries to answer the question how an application can be distributed on a grid of processors. The grid computing aspects are considered important for pervasive computing, because a distributed sensor and actuator network poses similar distribution and synchronisation problems. In smart environments different embedded devices could solve specific parts of a problem. In fact, every embedded device has its own specialized task. A GPRS mobile phone could solve the global communication task, while an embedded location tracker with six degrees of freedom could provide exact location information. Other examples of grid computing problems are the transfer of performance-critical calculations from PDAs to more powerful devices or the distribution of calculations among different embedded devices.

2.4 Pervasive Computing

The term *Pervasive Computing* stands for the philosophy to embed limited intelligence into objects that surround us [Ar99]. *Ubiquitous Computing* on the other hand means that digital services and applications are mobile and can be consumed everywhere. Often it is not really clear how in which aspects the research areas of Pervasive Computing, Ubiquitous Computing and *Mobile Computing* differ. Pervasive means that digital technology diffuses through every part which implies high embeddedness. Mobile Computing describes environments in which the user is able to use mobile devices and wireless networks but does not imply any use of embedded devices. Most of the modern applications that are running in smart environments include aspects of all three computing philosophies. Therefore, it is hard to identify which part of the hardware or software is associated to one philosophy. Fig. 3 shows how these computing philosophies can be distinguised according to embeddedness and mobility.

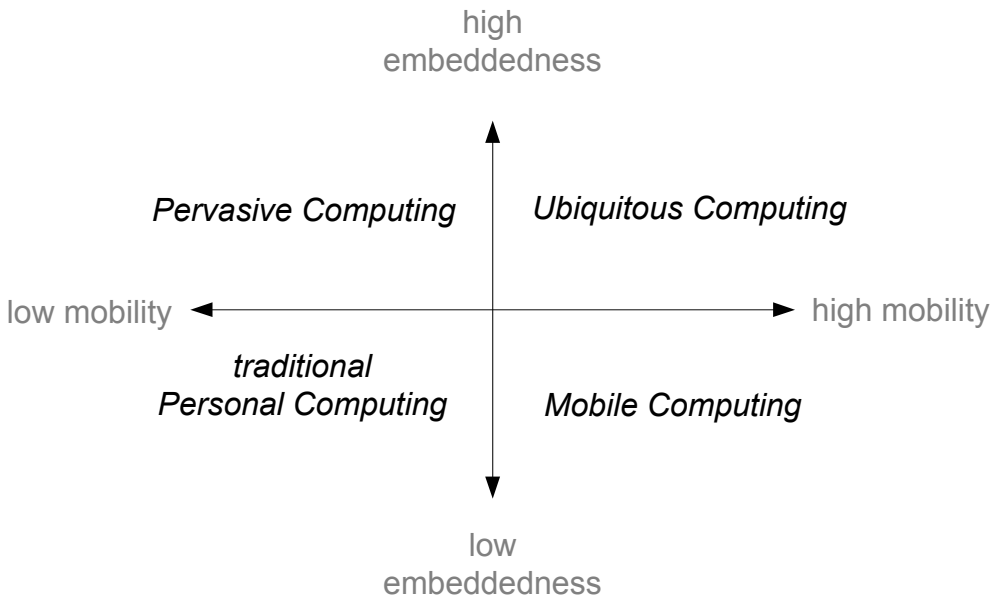

Fig. 3 Differences between Pervasive, Ubiquitous, and Mobile Computing

Pervasive Computing implies that everyday objects can get the possibility to communicate and to discover the environment. The idea is, that the technology should not visibly change an environment but should improve common objects below the visible surface. Examples for such changes can already be found in our daily life. One of the first applications of wireless object identification was implemented for supermarkets to prevent the customers to take products without paying them. A primitive mechanism changes the state of the product from *not paid* to *paid* when the customer visits the cash box. For the customer it seems as if the system did not change at all.

Another example targets the automotive industry where smart components should improve the safety of vehicle drivers. Limited intelligence is put into some parts of vehicles in order to warn the driver when certain measurements are not within specified limits. One example is the use of embedded chips inside the tires to measure the pressure and other critical values. All the tires communicate their measurements to the car which decides how and when

to contact the driver or the mechanic. Different automotive companies are already using these smart tires [SmarTire].

Pervasive systems differ from traditional systems by the fact that the user does not have to be in front of an input/output interface and does not have to focus his attention. Pervasive systems are meant to function with a minimum of supervising by the user, which means that these systems demand less concentration from the users. To solve problems without the user's attention such systems need information about their environment (context information) and the possibility to communicate and to share this information. Many of these ideas were already mentioned by *Mark Weiser* in his article "*The Computer of the 21st Century*" [Wei91].

One of the most important aspects of pervasive computing was not mentioned so far. The fact that many objects of our daily life get some sort of specific intelligence to perform operational tasks leads to the question of who controls an environment, or more general, how the need for security and safety should be solved in pervasive environments. As security targets the issue of not sharing critical information with the wrong people or devices, safety asks the question how such systems may change the user's situation. How smart devices and environments will change the human's safety is hard to discover, due to the fact that the systems are steadily growing and are already taking control over some areas of our daily life. Most people already depend on smart devices embedded in their cars that take control when the car is in a critical situation (ABS, ESP, air bags). Other components support the user to control the vehicle (drive by wire). So it can be realized that already today smart components have taken over the control in every days life and that embeded intelligence is already reality. The question how security issues could be handled in such environments is treated in Chapter 7.3.

2.5 Wireless Communication

As mentioned before, communication technology and network protocols as well as information-encoding mechanisms are the basic principles on which context-aware computing is built. In order to understand the problems of working in heterogeneous network environments, this section gives a short overview about the latest developments in wireless network technology as well as in wireless identification technologies.

Traditional applications were based on wired networks connecting static base stations with fixed network addresses. Due to a shift from static workstations to mobile computing wireless connections became more and more important. Wireless networks reach from infrared connections, also known as *IrDA* (*Infrared Data Association*) connections which operate in direct line of sight around 1,5 meters, to globally available radio-based networks like the *GSM* (*Groupe Special Mobile*), *GPRS* (*General Packet Radio Service*) and *UMTS* (*Universal Mobile Telecommunications System*) standards. The most popular wireless radio-based data communication technology today is the *IEEE 802.11* standard family (802.11a-g) for radio-based local area networks and the IEEE 802.15.1 standard, also known as *Bluetooth*, for personal area networks. Other radio-based local area network standards are IEEE 802.15.4, called *ZigBee*, for extremely simple hardware environments, IEEE 802.15.3a and

DECT (*Digital Enhanced Cordless Telecommunications*). All these radio-based communication standards differ according to their application, communication range, energy consumption, network topology, bandwidth and latency, connection setup time, and scalability.

Another important aspect, is that wide area radio networks like GSM, GPRS and UMTS communicate in frequency bands that are licensed by the government. Local and personal area radio networks like the 802.11 family and Bluetooth use the *ISM* (Industrial, Scientific and Medical) frequency band (2,4 GHz) which can be used without any restriction. Bluetooth was originally not designed as a networking technology, but as a cable replacement technology. At the moment Bluetooth is used for nearly every purpose, also for personal area networking. The fact that the ISM band can be used without having to obtain a license lead to a massive use of ISM band devices in recent years.

A *Frost&Sullivan* marketing study [Frost] showed that in 2003 over 70 million digital devices with integrated Bluetooth support existed. In 2004 this number will grow to about 120 million devices.

The following subsections describe the most important radio-based wireless network technologies in more detail now.

2.5.1 IEEE 802.11 (WLAN)

The IEEE 802.11 standards, often called *wireless Ethernet* or *WLAN*, have evolved into a wireless replacement of the typical wired Ethernet scenario. The first specification in 1997 defined a maximum data rate of 2 MBit/s. Today, the 802.11a standards achieve a data rate of 54 MBit/s and the 802.11b standards about 11 MBit/s. The IEEE 802.11 standard family is one of many IEEE 802 network specifications that share the same layered architecture [MoCo]. The network layer of the ISO/OSI seven layer architecture can therefore always use the same interface irrespective of which underlying protocol is used (e.g. Ethernet, WLAN, Token Ring). Fig. 4 shows the protocol architecture layers of the WLAN standard.

<table>
<tr><td rowspan="2">ISO/OSI
Data Link Layer</td><td colspan="3">802.2 Logical Link Control (LLC)</td></tr>
<tr><td colspan="3">802.11 Media Access Control (MAC)</td></tr>
<tr><td rowspan="2">ISO/OSI
Physical Layer
(PHY)</td><td colspan="3">802.11 Physical Layer Convergence Protocol (PLCP)</td></tr>
<tr><td>PMD 802.11
Infrared</td><td>PMD 802.11 FHSS
Frequency Hopping
Spread Spectrum</td><td>PMD 802.11 DSSS
Direct Sequence
Spread Spectrum</td></tr>
</table>

Fig. 4 IEEE 802.11 protocol architecture

Fig. 4 shows that the physical layer of 802.11 is divided into two different layers: *PMD* (*Physical Medium Dependent*) and *PLCP* (*Physical Layer Convergence Protocol*). The PMD layer offers physical medium-dependent access for infrared, FHSS (*Frequency Hopping Spread Spectrum*) and DSSS (*Direct Sequence Spread Spectrum*) communication, while PLCP provides a medium-independent interface for the MAC (*Medium Access Control*) lay-

er, which manages the package transport from one network interface to another through a shared transmission channel.

- ***FHSS*** uses the frequency hopping mechanism to avoid collisions with other WLAN devices. The baseband is divided into 79 channels, which are changed in a random order.
- ***DSSS*** uses the *CDMA* (*Code Division Multiple Access*) mechanism, which enables multiple transmissions on the same frequency channel for more than one transmitting device. The different signals are multiplexed with the help of device-unique codes and are demultiplexed at the receiver's side. The DSSS mechanism is more stable with respect to collisions than the FHSS method and it allows more than one transmission per frequency channel. In modern WLAN devices the DSSS method succeeded the FHSS method.

Another important aspect of WLAN devices are the different operation modes that are defined by the IEEE 802.11 standards:

- ***Infrastructure mode***: The infrastructure mode allows the association of WLAN client devices (called *Access Points*) to a central base station. Access Points are wireless routers that connect the wireless client devices to a wired network. The communication of two wireless client devices, which are located in the same wireless area (hot-spot), is also managed via the Access Point. The wired network between the Access Points is also used to deliver roaming information about mobile WLAN clients.
- ***Ad-hoc mode***: The ad-hoc mode of WLAN devices allows the connection of devices which are in communication range. If no higher level packet routing protocol is used the stations can only communicate with other stations that are in communication range.

2.5.2 IEEE 802.15.1 (Bluetooth)

The IEEE 802.15.1 standard was originally designed as a cable replacement between digital devices. There are three main application scenarios for Bluetooth connectivity:

- Bluetooth access points operate as bridges between wired and wireless networks.
- Bluetooth is used to build spontaneous ad-hoc networks, called *Piconets*, to communicate without central control.
- Bluetooth is used as a cable replacement between digital devices.

Bluetooth operates in the same ISM frequency band as IEEE 802.11 devices and microwave ovens. Therefore these devices interfere with each other. To solve this problem, Bluetooth uses *frequency hopping* to avoid transmission collisions. The available frequency band (83,5 MHz) is divided into 79 channels, each of which having 1 MHz bandwidth. The frequency hopping procedure randomly changes the transmission channel 1600 times per second (*fast frequency hopping*). With the *Gaussian Frequency Shift Keying* (GFSK) mechanism Bluetooth offers a maximum data rate of 1 Mbit/s [Br01].

Because Bluetooth was designed as a cable replacement technology the connection range was originally defined to be less than 10 meters. Today, many manufacturers offer Bluetooth devices with higher transmission power in order to reach distances around 50 to 100 meters.

Bluetooth distinguishes between two kinds of connections: *Synchronous Connection-Oriented Links* (SCO) and *Asynchronous Connection-Less Links* (ACL). SCO connections are primarily used for audio connections, which need a full duplex connection with fixed-size data packages that are transmitted synchronously. SCO links are limited to a maximum of three full duplex voice links per Bluetooth device.

ACL connections are used for data transmissions with variable-length data packets that are sent asynchronously. Fig. 5 shows the Bluetooth protocol stack.

The Bluetooth protocol stack contains the *TCS* layer (*Telephony Control Protocol Specification*) for telephone-related services. The *SDP* layer (*Service Discovery Protocol*) enables the discovery of services which are offered by other Bluetooth devices. The *RFCOMM* layer offers standard serial communication emulation for higher-level protocols. A layer that is able to access the functionality of the baseband layer directly is called *Audio.* This layer manages the SCO connections for direct audio transmissions.

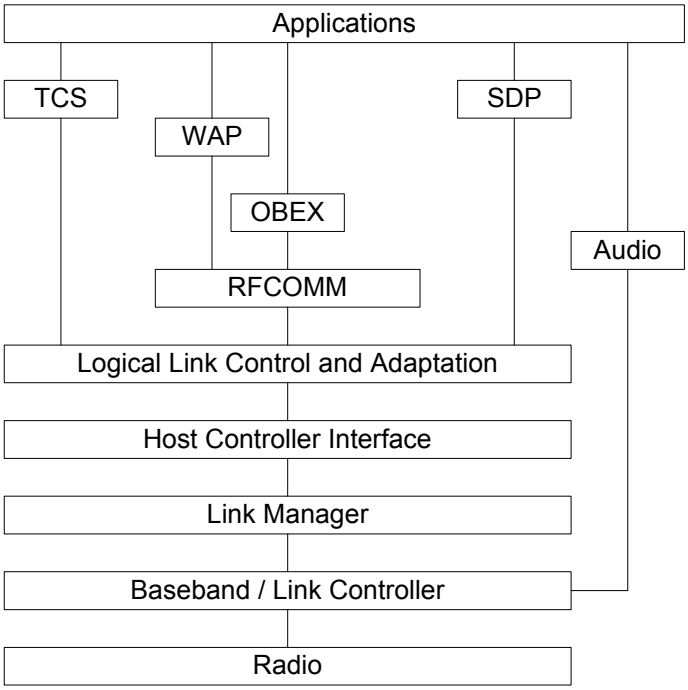

Fig. 5 Bluetooth protocol stack

Bluetooth devices can operate in two modes: *master* and *slave*. The master sets the frequency hopping sequence and the slaves are following this sequence. Every Bluetooth device has a unique Bluetooth device address and a clock value. When a slave connects to a master it gets the master's address and clock value with which it is possible to calculate the frequency hopping sequence. The number of slave devices that are managed by a master is limited to seven. A network that consists of one master device with a maximum of seven slave devices is called *piconet*. Inside a piconet all transmissions are managed by the master without any direct connections between the slaves (see Fig. 6).

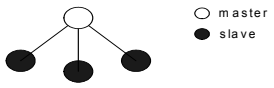

Fig. 6 Bluetooth piconet

When more than one piconets are connected the resulting network is called a *scatternet*. In a scatternet one device is either a member of two piconets, or one device is acting both as a master and as a slave, as it is shown in Fig. 7. Scatternets allow the ad-hoc connection of more than seven Bluetooth devices.

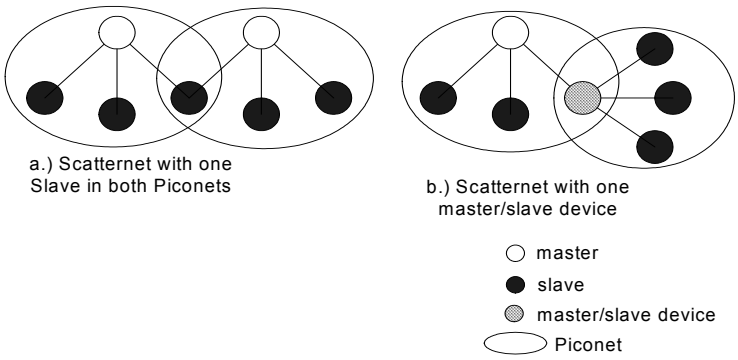

Fig. 7 Two different kinds of scatternets

2.5.3 IrDA

In 1993 the *Infrared Data Association* (IrDA) was founded to establish a common standard for infrared data communication. In 1994 the IrDA 1.0 standard was published which allowed a maximum data communication rate of 115 kBit/s. Because of this low data rate, the IrDA group announced IrDA 1.1 (*Fast Infrared*) in 1995 and VFIR (*Very Fast Infrared*) in 1999. IrDA 1.1 offers a data rate of 4 MBit/s and VFIR even of 16 MBit.

Infrared (IR) communication is a popular and cheap way to transmit data without cables and wires. However, there is quite a difference between IR communication and radio-based communication. IR communication is based on infrared light, which needs a direct line of sight between the sender and the receiver. Due to the fact that the daylight contains parts of the infrared spectrum IR communication can be interrupted or blocked. While radio-based transmissions can permeate objects like walls, doors or clothes, IR transmissions are entirely blocked by such objects. The IR communication range is limited to a few meters whereas the radio-based communication ranges, generally, are higher (e.g. radio based WLAN with 100mW transmission power is limited to 100 meters).

The limited communication range and the need for a direct line of sight between sender and receiver offers more privacy than radio-based networks. IR-based communication that is performed within a few meters is hard to intercept from outside.

All modern operating systems support the IrDA standard and many mobile devices offer infrared ports. The IrDA standard is based on two substandards:

- ***IrDA Data***: This substandard is responsible for data transmissions over infrared connections.
- ***IrDA Control***: This substandard defines how input devices like keyboards, mice or joysticks can send control information over an infrared connection.

Fig. 8 shows the IrDA protocol stack. At the bottom of the stack there is the infrared bit transport layer, which manages the encoding of data bits in infrared signals. The *IrLAP* layer (*Infrared Link Access Protocol*) is responsible for a reliable connection between sender and receiver. While the IrLAP layer supports only a single reliable channel, the *IrLMP* layer (*Infrared Link Management Protocol*) can manage multiple logical channels on a single physi-

cal connection. The *IAS* layer (*Information Access Service*) allows the discovery of services that are offered by other devices.

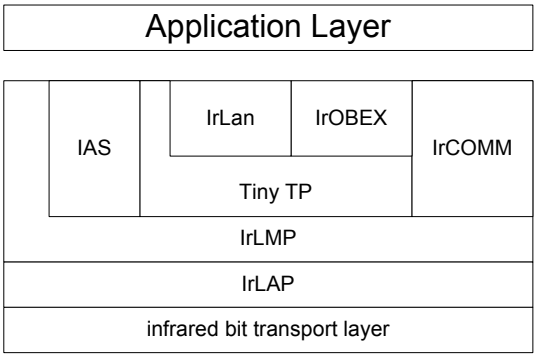

Fig. 8 IrDA protocol stack

The other protocol layers are optional and not necessarily implemented within every IrDA device. The *Tiny TP* layer (*Tiny Transport Protocol*) provides the possibility to transmit bigger messages through segmentation. *IrLAN* layer (Infrared Local Area Network) offers a bridge for connecting to a LAN. *IrOBEX* (*Infrared Object Exchange Protocol*) enables the exchange of complex messages such as v-cards, which is a protocol for the exchange of business cards. *IrCOMM* emulates a standard serial communication, which enables applications to communicate through a serial port.

2.6 Wireless Object Identification

One of the most important aspects of context-aware computing is the ability to identify objects. The identification of unknown objects, no matter if they are digital or non-digital, allow devices to discover their environment and to reason about the actual context. A multitude of identification technologies are already common in our daily life: barcode scanners in supermarkets identify products and their prices, chip cards or ID cards identify their owners, or RFID transponders identify customers in skiing areas or wellness temples.

For digital systems it is hard to identify unknown objects. To simplify the identification process, various identification technologies have been developed. Each technology has its advantages and disadvantages. Since the identification of objects is an essential aspect of context-aware computing, this chapter gives a detailed overview about some of the most popular identification methods.

2.6.1 Radio Frequency Identification (RFID)

For the work in this thesis, the *Radio Frequency Identification* (*RFID*) technology was one of the most important identification mechanisms. The massive use of RFID technology in our project is explained by the advantages that RFID offers over alternative wireless identification systems.

One of the most important advantages is that RFID identification tags, also called *transponders*, are passive devices. RFID transponders are designed to receive energy from an active reader device without any physical contact and to communicate with the reader in

wireless mode. The transponder does not need to have its own energy supply. It is possible to tag nearly every object with an RFID tag without worrying about energy supplies. RFID tags can be produced in tiny sizes because they are only composed of an integrated chip and an antenna. They can be quite expensive, hoewever, depending on their form and the kind of system in which they are used. Optical tags are much cheaper but RFID transponders allow the identification of objects that are behind solid obstacles; it is not necessary to have a direct line of sight.

RFID systems also provide information about the proximity of a RFID-tagged object relative to the position of the reader [Fer02].

In recent years many different RFID systems appeared, which differ according to their operating range, their frequency, and the kind of communication the transponders support. Generally, it is possible to distinguish between three basic classes of RFID systems [Fi00]. Some of them are already defined in standards of the *ISO* (*International Organisation of Standardization*) and *IEC* (*International Electrotechnical Commission*) and some standards are still in progress:

- ***close coupling***: An RFID system is called *closely coupled* (*ISO/IEC Standard 10536*) when its communication range is below 1 cm. This means that the transponder has to be placed directly on top of a reading device. Due to the small reading distance the inductive energy transfer is better than in remotely coupled systems and the RFID chip is able to transfer complex data to the reader. It is even possible for the RFID chip on the transponder to encrypt the transfered data and to allow write operations on RAM, EEPROM or FERAM memory.
- ***remote coupling***: Remotely-coupled RFID systems provide a reading and writing range of up to 1 meter. Around 90% of all RFID systems that are used in industrial, medical, and commercial systems are remotely coupled. Remotely-coupled systems are classified into *proximity coupling* (ISO/IEC 14443) and *vicinity coupling* (ISO/IEC 15693). Proximity-coupled systems are used for high-speed data transfer over a small distance. Remotely-coupled systems use frequencies less than 135 kHz. There exist also remotely-coupled solutions which use the following frequencies: 6,75 MHz, 13,56 MHz or 27,125 MHz.
- ***long range***: Long-range RFID systems use active transponder devices to achieve sending ranges between 1 and 10 meters. Because inductive energy transmission is not possible over such large distances, active transponders include an energy supply. Long-range RFID systems operate in the microwave frequency band around 2,4 GHz as it is shown in Fig. 9.

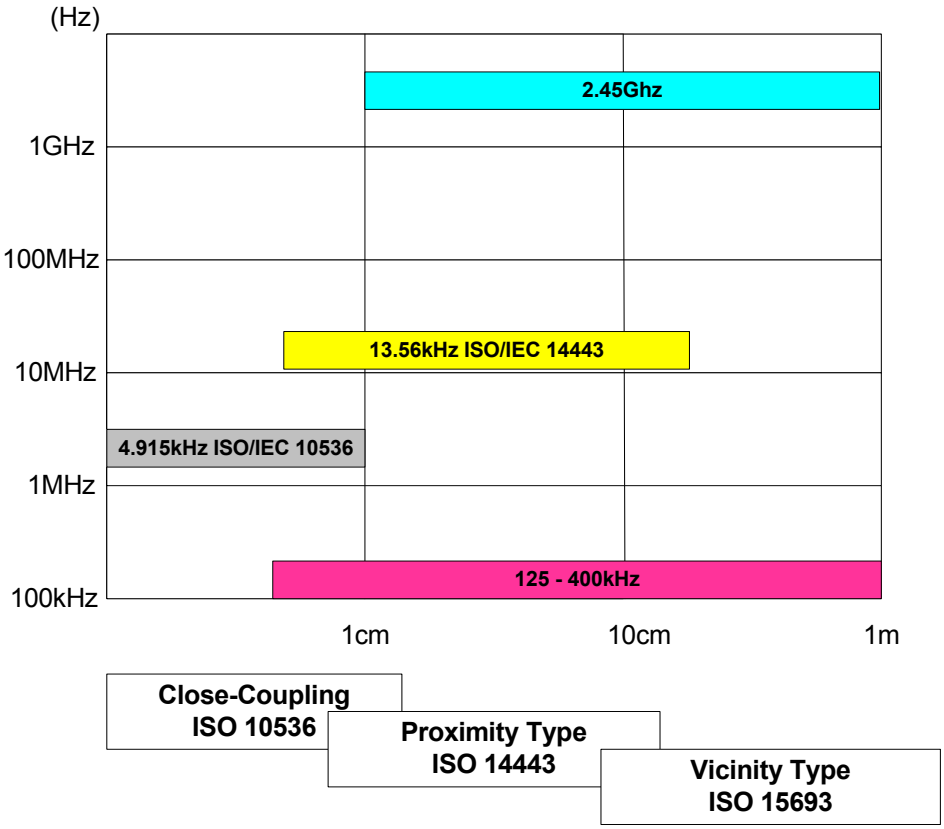

Fig. 9 Frequency and range of actual RFID systems

To distinguish between different RFID systems, it is necessary to know which functionality those systems offer. The functional range of RFID devices starts with *low-end* systems, which provide *read-only* transponders, and goes to high-end systems, which can even have an operating system running on the transponders. RFID systems can be categorized into the following functional classes:

- ***read-only***: Read-only RFID systems permanently transmit a small amount of data (e.g. the transponder ID) when an electromagnetic field of an active reader is close enough. It is not possible to read more than one transponder ID at a time, so collision detection is not supported. It is not possible for the reader to write data to the transponder. Low-end read-only RFID systems are often used to replace optical barcode systems.
- ***anti-collision***: Anti-collision detection enables the identification of more than one RFID transponders within the reading range. It can be implemented with a *Time Division Multiple Access* (*TDMA)* ALOHA protocol. Anti-collision detection RFID systems are getting more and more important due to the increased usage of modern appliances in commercial environments (e.g. product tags in supermarkets).
- ***read-write***: Read-write RFID systems offer the possibility to store small amounts of data on the passive transponder devices (between 16 Bytes and 16 kBytes on EEPROM or SRAM).
- ***authentication and cryptography***: RFID transponders which are based on a microcontroller chip can offer authentication and cryptography mechanisms. Such trans-

ponders are not based on a static state machine but can even host an operating system that provides complex functionality. Such high-end RFID systems are similar to microprocessor chip cards.

Basic architecture of inductive coupled RFID systems. As already mentioned above, most RFID systems are using inductive coupling to access the passive RFID transponders. Inductively-coupled passive RFID transponders consist of an integrated chip and a coil that represents the antenna of the passive transponder. To read data from an inductively-coupled RFID transponder the active reading device generates a electromagnetic field that penetrates the transponder's coil and creates a voltage at the passive transponder. Fig. 10 shows the transmission of energy in inductively-coupled RFID systems.

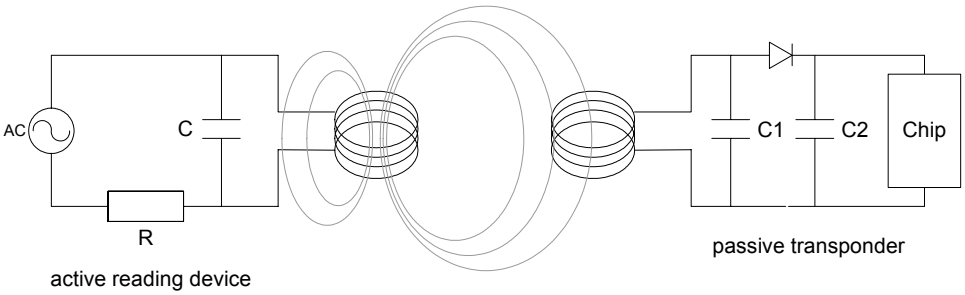

Fig. 10 Energy transmission in inductively-coupled RFID systems

Form factors of RFID transponders . Today there exists a wide range of form factors for building RFID transponders. Depending on the application area where the RFID transponders are used to identify objects or people, the range of form factors varies from the *smart label* to the *disk transponder* form [Tex]. Fig. 11 shows some popular inductively-coupled RFID transponder forms. Smart labels are already very popular for logistic systems and warehousing applications, where it is important to track single objects through their whole production cycle.

RFID transponders in credit card sizes are often used for personal identification as it is shown in Fig. 11, where a student card with an RFID transponder is displayed. This sort of transponders are often used for contactless access controls.

Glass transponders are extremely small and can be combined with biological material. Therefore they are used for animal identification and tracking. Often the glass transponders are delivered in combination with injection devices to place the transponder under the skin of an animal.

The recent years showed that RFID technology is one of the most promising wireless identification technologies.

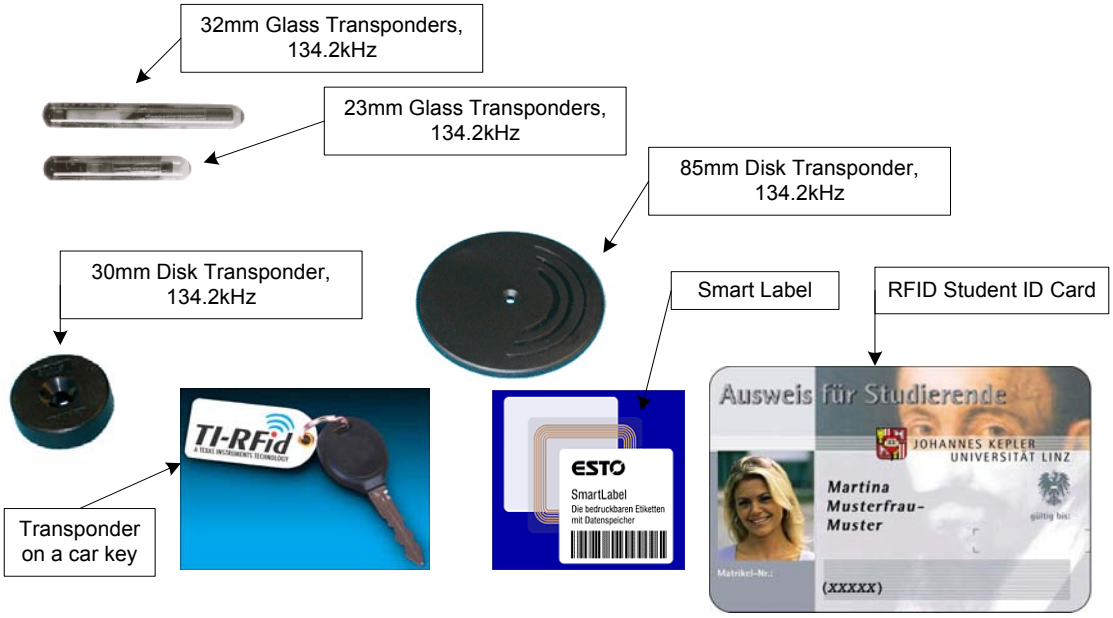

Fig. 11 Popular form factors of inductively-coupled RFID transponders

2.6.2 Ultrasonic Identification

The identification of objects which are equipped with active ultrasonic senders is used in many research [Sen] and industrial projects. The active ultrasonic sending device, often called *bat*, emits a short pulse of ultrasound that is received by statically installed ultrasound receivers. The receivers are able to identify an object by specific ultrasound pulse times and lengths. One of the most important aspects of ultrasonic identification of objects is that the receivers are able to calculate an object's fine-grained position by trilateration. This means, that the system is able to calculate the three-dimensional position of an object by using the time the signal needs to travel from the active *bat* to the receivers that are in range.

ORL system (*Olivetti and Oracle Research Laboratory*). The *ORL* identification system [ORL] is based on active ultrasonic sending devices, which are equipped with a 418 MHz radio transceiver for network communication. Each device has a 16-bit unique ID. To enable an environment to identify the positions of the *bats,* it is necessary to mount a matrix of receivers, which are connected to a controlling PC. The PC periodically broadcasts one of the unique IDs in the 418 MHz band. All the *bats* receive the message but only the one that has this ID is allowed to respond with an ultrasonic pulse. So the PC is able to identify the specific device and to determine its position in the environment by measuring the signal travel times. It is even possible to obtain information about the object's orientation.

An ORL system that mounts 16 ultrasonic receivers on the ceiling is able to cover an environment of about 75m^3 and offers a location accuracy of about 14 cm around the real position of a device.

Fig. 12 shows a prototype of an active ultrasonic device.

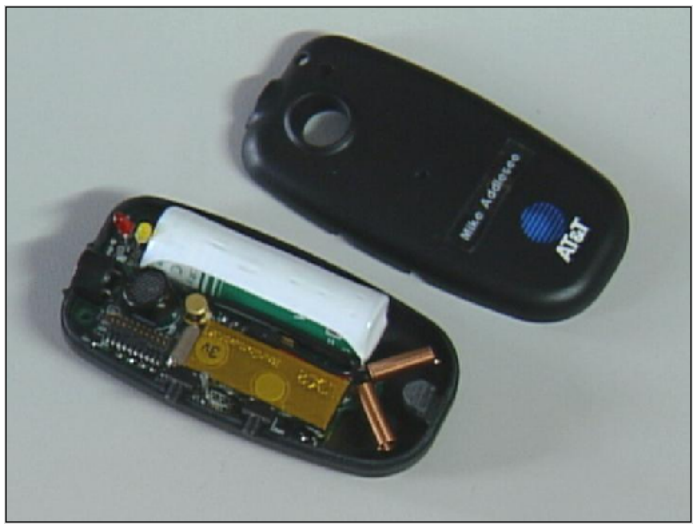

Fig. 12 Active ultrasonic identification device, called *bat*

The ultrasonic location identification mechanism is one of the most useful and cheapest possibilities for locating the position of objects within buildings. Radio positioning systems are successful mechanisms for outdoor location tracking. Within buildings, however, radio-based location tracking is vulnerable to signal reflections and therefore not useful.

2.6.3 Infrared Identification

Infrared object and location identification is a widely-used technology. It uses cheap hardware and most of the modern mobile devices are already equipped with infrared interfaces. Another advantage of this technology is that it can be used for both communication and object identification. Whereas ultrasound is just an object identification and not a communication technology, infrared interfaces can be used for both purposes. However, infrared light is blocked by solid obstacles and interfered by glaring sunlight. That means that the location granularity depends on the environment (e.g. rooms) so that infrared communication and identification is usually an indoor technology.

The first ubiquitous computing projects that used infrared object and location identification were Xerox PARC's *PARCTAB* environment (described in Chapter 3.1) and Olivetti's *Active Badge* system [ABa].

The Active Badge environment was one of the pioneer projects that established active infrared identification badges. People in the Olivetti Laboratory are using infrared-emitting Badges for identifying themselves and for determining their location. In order to save energy the badges send a unique ID only every 15 seconds. Sometimes they offer also limited user input facilities through a set of buttons. The main advantage of Active Badges compared to other identification technologies is that the devices are cheap and simple to build. The energy consumption of Active Badges is much lower than that of ultrasonic bats. An active badge can operate for more than a year. Fig. 13 shows an Active Badge as it is used in the Olivetti Laboratory.

Fig. 13 Olivetti Laboratory's Active Badge

2.6.4 Vision-Based Systems

Vision-based systems use visual input from digital cameras to identify objects and locations in an environment. In order to identify specific visual characteristics it is necessary to have a profound knowledge about the visual representation of an environment. In general, there are two methods for vision-based object identification [Ipi]:

- ***Untagged vision-based systems*** try to recognize an object according to its visual representation. In a simple environment these systems work fine, but as the complexity of objects and their environment increases the identification process gets extremely complex. Untagged vision-based systems require much CPU processing power and for complex object identification, such as human face recognition, these systems often fail completely.
- ***Tagged vision-based systems*** simplify the identification of an object by attaching a unique visual tag to an object. Successors of this technology are *barcode* systems, which are used in every supermarket. Visual tags are one of the cheapest identification mechanisms, because the tags can be printed with a standard printer. Due to their low costs visual tags are the most popular identification technology until now.

The main disadvantage of visual tags is that the scanner has to directly face the tag in order to identify it. Any obstacle that comes between the scanner and the tag prevents the identification process. The use of the cheap visual tags such as *barcodes* has in many ways revolutionized contactless identification in large-scaled logistic systems.

TRIP. The *TRIP* (*Target Recognition using Image Processing*) system was developed by the Laboratory for Communications Engineering at the University of Cambridge. It uses circular two-dimensional barcode tags, also called *ringcodes*, to identify objects by image processing. The TRIP system uses simple CCD or CCTV cameras to identify the ringcode tags within the camera's field of view.

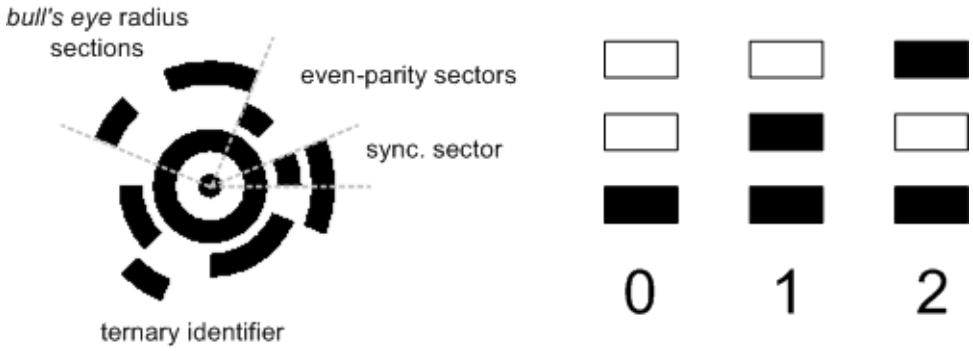

Fig. 14 A TRIPtag with its different encoding sections

A TRIPtag consists of two concentric black-colored rings in the middle, which are also called the bull's eye. A typical TRIPtag is shown on the left side of Fig. 14. Around the bull's eye two concentric rings, which are divided into 16 sections, are used to encode the TRIPtag's information. The first section of these rings is always colored black in order to provide a synchronisation sector. The TRIP system uses a ternary encoding, which is shown on the right side of Fig. 14. The synchronization section is the only place where all the ring sectors are black. The next two sections specify an even-parity check and the following four sections specify the radius of the bull's eye. The remaining 9 sections are used to encode the unique ID of the tag. TRIPtags can represent IDs between 1 and 19.683 (3^9 - 1).

The reason why concentric rings were chosen for identifying objects and their locations is that round shapes are not as common in man-made environments as square and rectangular shapes. Furthermore, the identification of round shapes is easier and less CPU-intensive than the identification of rectangular shapes.

To extract the 3D position of a tagged object and its orientation it is necessary to use the known size of the tag and to calculate its perspective projection. The size of TRIPtags is encoded within the sectors 3-7, so it is possible to use variable-sized tags.

Visual marker systems, such as the TRIP system, are flexible and inexpensive according to the fact that every web cam can be used to identify tagged objects.

2.7 Ad-Hoc Networks

Mobile devices, in combination with wireless networks, require network protocols that allow the dynamic creation of network topologies. Ad-hoc networks can be established between a group of devices that are able to communicate with each other with an automatically created routing table. Ad-hoc networks operate without a connection to a central server that has to be available globally. They do not rely on any infrastructure or already established central administration [Toh].

Networks, such as ad-hoc networks, that do not rely on any infrastructure are called *infrastructureless*. Infrastructureless networks offer great advantages, but also have some disadvantages compared to static networks with fixed topologies and a central administration. Ad-hoc networks can be used in any situation between any group of nodes in order to access services, exchange data, or forward requests to other nodes. In environments where no network

infrastructure is available (e.g. outdoors), or in environments where a static network topology is not useful (e.g. PANs between MP3 player, mobile phone and headset), ad-hoc networks provide a convenient communication solution.

The main disadvantages of ad-hoc networks are their lack of a central security administration and the additional network administration overhead for the individual nodes.

Every node in a infrastructureless network has to perform some additional tasks to establish network communication to other nodes. A node has to serve as a router, in order to forward packages to the next hop on the route to the package's destination node. Due to the fact that ad-hoc networks are highly dynamic (e.g. it is normal that nodes can disappear without notification) the routing information often changes. In static networks, where millions of devices are connected by a central administration, the network address of a device is often mapped directly to its location in the network topology. Therefore it is easier to calculate the network route in static networks.

In ad-hoc networks the network address is completely independent of the device's location in the network. Most of the time ad-hoc network nodes are mobile devices which change their location. Due to this fact, the network topology changes dynamically and the calculation of a route is much more complex. Routing algorithms can be classified into adaptive and non-adaptive routing mechanisms, where adaptive mechanisms are able to react automatically on changes in the network topology. For ad-hoc networks only adaptive routing mechanisms can be used due to the highly dynamic nature of such network topologies. Adaptive routing mechanisms can be categorized into the following groups:

- **Table-driven** routing mechanisms (*proactive algorithms*) use routing tables to find a route to a destination. Such mechanisms update their tables periodically and also gather routing information about hosts that were not demanded before. Examples for table-driven routing mechanisms are *WRP* [WRP] and *DSDV* [DSDV].
- **On-demand** mechanisms collect routing information to a destination address when a node has to forward a package to this address, but they do not store this information for later use as table driven mechanisms do. Examples for such mechanisms are *DSV* [DSV] and *ABR* (*Associative-Based Routing*) [ABR].
- **Hybrid** routing mechanisms combine aspects of table-driven routing protocols (caching already known routes) and on-demand routing protocols (collecting routing information only when required). The *Zone Routing Protocol* [ZRP] is one example for a hybrid mechanism.

An important characteristic of routing protocols in mobile computing environments is the amount of energy that is necessary to transmit a package on a calculated route. Changing the power adjustment of a mobile device (e.g. into power saving mode) can therefore change the network topology and the routes.

Ad-hoc networks, most of the time, consist of a multitude of heterogenous devices with different hardware capabilities. A device which has a permanent power supply, a powerful CPU and a high amount of memory is a better alternative than a mobile device with very limited resources.

Bandwidth constraints are also a critical aspect of ad-hoc routing protocols, because wireless networks often offer significantly lower bandwidth than wired networks. Wireless con-

nections are also likely to change their bandwidth according to signal strength or latency measurements.

For connecting ad-hoc networks to traditional infrastructure networks, such as the global Internet, it is necessary to specify how ad-hoc connections can be routed into a static wired network. The resulting mobile Internet can be divided into two layers: the *mobile host* and *mobile router* layer [CO99], as it is shown in Fig. 15. The mobile host layer consists of several mobile hosts that are temporarily connected to fixed routers, which are directly connected to a wired network. The mobile host layer is supported by the standards *MobileIP* [RFC2290] and *DHCP* [RFC2131]. In this layer the communication between mobile hosts is only possible though the infrastructure.

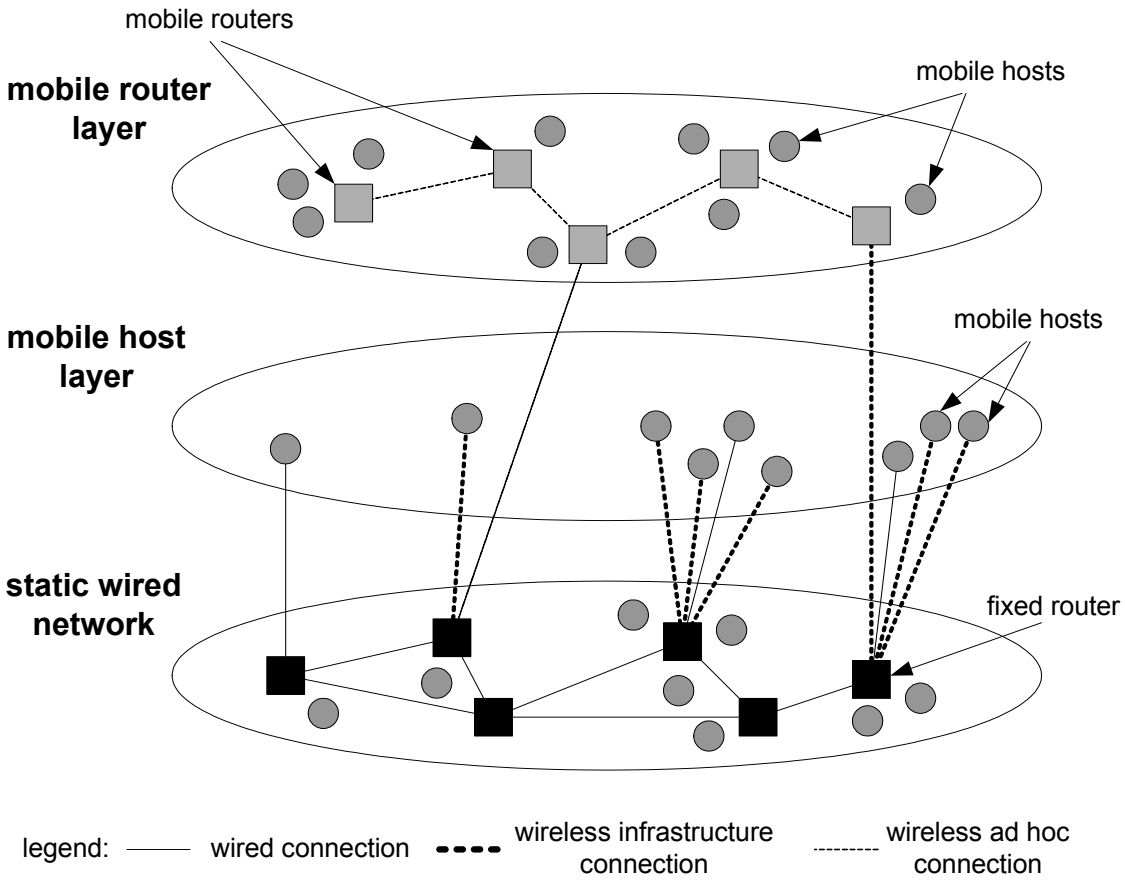

Fig. 15 Mobile host and mobile router layers of a mobile ad-hoc network, connected with a wired network

The mobile router layer consists of mobile hosts and mobile routers. Each mobile host in the mobile router layer is associated with a mobile router through a wireless ad-hoc connection. A mobile router routes between other mobile hosts or into a traditional static network through a wireless infrastructure coonection. The mobile network layer does not need any infrastructural support from the traditional static network. The mobile router layer (the ad-hoc network) forms a parallel network to the static network.

In recent years, research on ad-hoc networks has focused on military scenarios where many heterogeneous devices have to communicate in unknown environments without any

infrastructure. With the emergence of *Peer-To-Peer* file- and resource-sharing frameworks, as well as with personal area networks, ad-hoc networks gain more and more importance.

2.8 Peer-To-Peer Computing

P2P describes systems and applications that share resources, such as files or services, without the use of any central authority, like a server. P2P systems are comparable with ad-hoc networks where every node is able to communicate and to forward service requests without any central infrastructure. While ad-hoc networks provide basic networking protocols to enable communication between devices without a central router, P2P computing provides higher level services on top of any networking protocols. The difference between ad-hoc network protocols and P2P systems is that ad-hoc network protocols are located in the network layer and tranport layer of the ISO/OSI seven layer architecture [OSI] and P2P systems are operating in the presentation and application layer.

The term *Peer-To-Peer Computing* (*P2P*) refers to a research area which gained a lot of popularity in the last years. P2P computing is a controversial topic, according to the fact that many of the technologies and mechanisms used by P2P frameworks are already known from other research areas (e.g. grid computing, parallel computing, network communication).

P2P systems often rely on an arbitrary network structure (ad-hoc or managed, wireless or wired) and realize a higher level decentralized organisation of resources. In fact P2P applications are typical designed to run on ad-hoc networks, even if most of the actual P2P applications are running on traditional networks. One of the most popular showcases for a P2P application is *SETI@home* (*Search for Extraterrestrial Intelligence*), which distributes small amounts of signal recognition calculations among millions of private PCs. The SETI@home scenario shows that the P2P technology can offer great advantages compared to traditional mechanisms. Some companies have already started to establish software frameworks to support the development of P2P applications, such as Sun's Java based *JXTA* framework [JXTA], or MIT's *IRIS* framework [IRIS]. There is even a framework which is able to test P2P protocol implementations, called *p2psim* [P2PSIM]. Other well-known examples are file sharing P2P applications like *Napster* [NAP] and *Gnutella* [GNUT], which came to questionable fame in the last years, by sharing copyrighted resources.

A typical P2P node, which is also called *peer*, is designed according to a hybrid client-server model. This means that a peer may act as a server for some peers as well as a client for others. The high autonomy of peers leads to the same problems as in mobile hosts scenarios. One negative aspect is the lack of trustworthyness, because there is no possibility to contact a trusted central server. Other negative aspects are the high redundancy of information that may travel through a P2P network as well as the limited scalability of such networks because they require higher management effort.

The design of P2P applications is an alternative to the classic client-server model, but many P2P frameworks use a hybrid approach with some central peers in order to reduce the security and redundancy problems. Fig. 16 shows the taxonomy of computing systems, including P2P, taken from [P2PHP]. As the right taxonomy in Fig. 16 shows, P2P systems can be used in four different contexts: *distributed computing*, *file* or *resource sharing*, *collabo-*

ration, and *platform infrastructure*. *SETI@home* represents a typical instance of distributed computing. *Napster*, *Gnutella* and *Kazaa* are classified as file or resource sharing systems. The messenger tool *Jabber* is a collaboration system, *JXTA* and *IRIS* form P2P platform infrastructures.

The SiLiCon framework, that is presented in this thesis, is a combination of all four of these aspects.

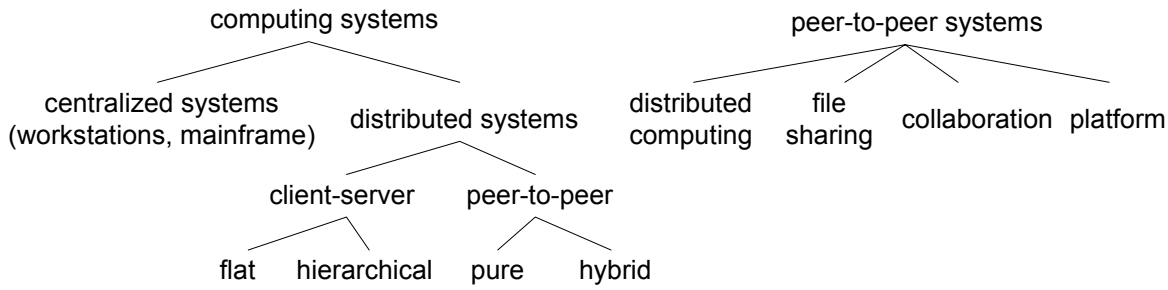

Fig. 16 Taxonomy of computing systems and P2P systems

2.8.1 P2P Discovery Algorithms

One of the most important parts of a P2P system is the discovery of other peers and the lookup of services. P2P architectures try to work as decentralized as possible. Therefore, it is necessary to provide a powerful discovery mechanism for finding other peers. There are several such algorithms ranging from completely decentralized to centralized discovery.

Centralized discovery. Centralized discovery algorithms use a central repository to store the contact information of all other peers. When a peer tries to find another peer it just has to get its contact information from the repository in order to be able to communicate with it. This mechanism offers at least a small amount of security and privacy compared to broadcast discovery mechanisms, because the centralized discovery mechanism depends on a central trusted repository. Also a big scalability factor is assured through the use of this central peer. The problem of this approach is that the central peer is a bottleneck. Furthermore, there has to be a connection between every peer and the central repository. A centralized discovery mechanism is completely useless within an ad-hoc infrastructure.

Broadcast or flooding discovery. Broadcast discovery mechanisms are pure P2P solutions, where the peers have no shared information. They are based on completely decentralized algorithms where all peers have to announce their appearance by a broadcast or multicast message. When two peers meet they exchange their contact information as well as the information that they have collected from other peers before. Broadcast mechanisms provide a good solution for peer discovery in local environments. In large-scale environments with a high number of peers, however, they produce too much network traffic and are not scalable. Due to scalability problems broadcast discovery algorithms often limit the number of hops a discovery package is able to travel. Therefore search requests may return without a result, even if the desired peer is running and able to communicate. Broadcast discovery mechanisms have a non-deterministic behavior and a search request without a limited number of hops might take an indefinite amount of time.

Hybrid discovery solutions. Modern P2P frameworks use hybrid discovery mechanisms in order to reduce the scalability problems in large-scale environments. One possible solution is the definition of *superpeers* which are peers with a higher reliability and more CPU power than an average node. A superpeer collects information about a dedicated group of standard peers in order to speed up the discovery process. Every discovery request is first sent to the nearest known superpeer. The file sharing application *KaZaa* uses such a hybrid mechanism based on superpeers.

It is expected that P2P applications and platforms will appear also in areas other than file sharing. Decentralized systems offer the possibility to operate in centralized networks (such as the Internet) as well as in ad-hoc networks. There are even some sorts of P2P systems that are designed to run on mobile devices such as smart cell phones or PDAs, which offer some higher-level operating system (e.g. Symbian OS, WindowsCE or Linux).

2.9 Jini

Jini (*Java Intelligent Network Infrastructure*) [Jini] was introduced by Sun Microsystems to provide a reliable software infrastructure for ad-hoc connectivity between heterogeneous digital devices. It was presented to the public in 1999 and is based on the well-established Java environment. Jini should widen the spectrum of Java-enabled devices to mobile and embedded systems. It should be usable, for example, to spontaneously connect even smart household appliances such as refrigerators, microwaves or dishwashers.

Traditional Java applications are running on top of a virtual machine, called the Java Virtual Machine (JVM). Jini is designed to build a spontaneous grid of JVMs in order to distribute services between heterogeneous devices. Service platforms like Jini facilitate the rapid creation and deployment of services, as well as the provision of dynamic service discovery mechanisms. The Jini service communication is based on *RMI* (*Remote Method Invocation*), and requires clients to be implemented in Java. RMI is an extension of the traditional *Remote Procedure Call* (*RPC*) mechanism, which allows a completely transparent call of methods over a network. In Java source code an RMI call cannot be distinguished from a local method call. The use of underlying Java technologies offers great advantages for developers who can rely on already established Java technologies and libraries. Java *Object Serialization* enables the transport of complex RMI parameters over the network and even to move entire objects including their code. To access low-level device capabilities it is often necessary to implement a bridge between Java and C++ using *JNI* (*Java Native Interface*).

Fig. 17 shows the layered architecture of a Jini service provider and a Jini client application.

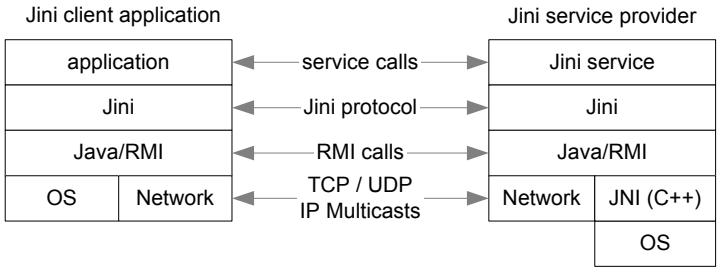

Fig. 17 Jini protocol stack

At the lowest level of the Jini protocol stack the network layer is responsible for the transport of byte arrays. The Java network package allows Jini to use *TCP/IP* sockets, which provide a reliable connection between two network endpoints, as well as *UDP* (*User Datagram Protocol*), which is an unreliable package-oriented network protocol. *IP Multicast* is used to send packages to IP multicast groups in order to discover a Jini lookup service provider.

2.9.1 Jini discovery and lookup

The kernel of the Jini service platform is based on three protocols: *discovery*, *join* and *lookup*. When a Jini device appears, it tries to find the next lookup services provider in the current environment; this is called *discovery*. Once a lookup service provider is found, the device registers its own services there; this is called *join*. A service provider which joins a lookup service provider has to register a *service object*. A service object represents a proxy object which is able to call the service at the service providers host. It is up to the service object how the service is called at the remote host but most of the time RMI (Remote Method Invocation) is used. The service object is registered at the service provider through the Java serializing mechanism which allows the transmission of Java objects. The discovery process is performed with the use of IP multicast packages, which means that the Jini device has to know the IP multicast group address where a possible lookup service provider listens.

Services are described by Java interfaces. Therefore, a Jini client asks a lookup service provider for a specific interface. If a service with this interface has been registered there the client receives the service object from the lookup service provider which is able to call the service.

Fig. 18 shows how two Jini devices, a digital camera and a photo printer, send IP multicast discovery packages in order to find a lookup service provider. When a lookup service provider receives a discovery package from a device it responds with a multicast response package containing it’s address. The requesting device, the camera and the printer, get the response package from the lookup service provider. The devices register their services at the service provider. Jini services are specified through a Java interface definition. For registration the Jini service provider has to send sends the 128 Bit UUID (Universal Unique Identifier), the service interface and a collection of attributes to the lookup service provider. The attributes provide optional metainformation about the service.

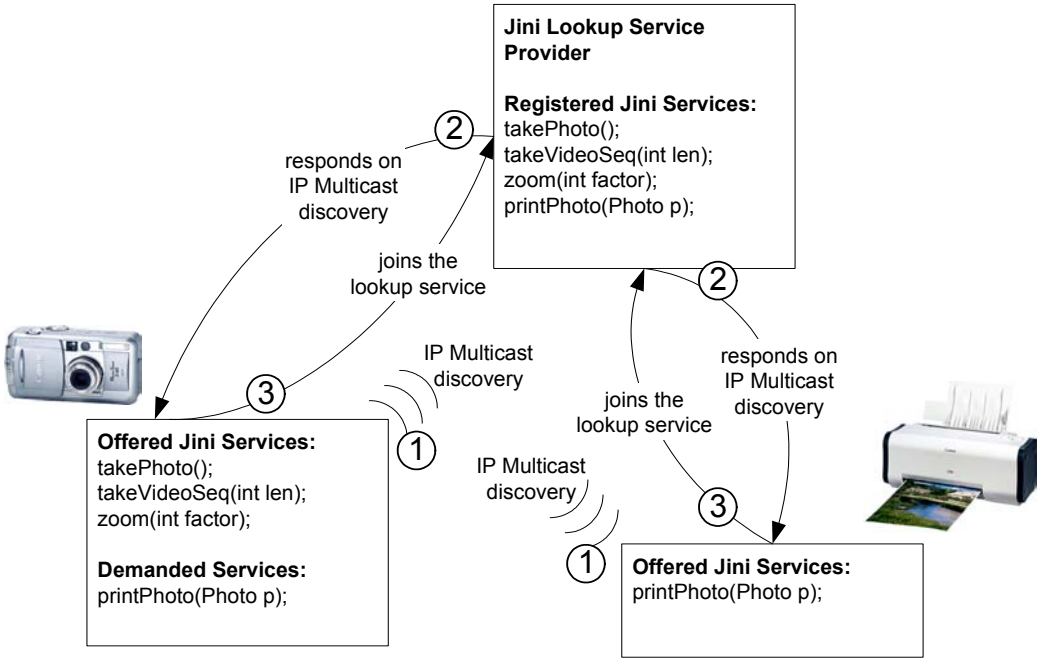

Fig. 18 Jini discovery and join process

In a TCP/IP environment, all multicast IP packages are sent via UDP, which is a unreliable package-oriented protocol. With the Java Serialization mechanism it is possible to encode the request and response into a byte array, which can be transmitted in a UDP datagram package. Because a UDP datagram package is limited to 512 bytes the discovery requests are also limited to this size. The Java serialization mechanism guarantees also platform independence. Fig. 19 shows how the digital camera does a lookup for a service with the specific Java interface, which offers the method `printPhoto(Photo p)`. Jini devices use the Java type system to determine whether an interface matchs a service lookup. Java interfaces that offer the same signature but implement different types do not match. After the camera has received the address and the service object of the service provider it is able to call the service with the printers service object.

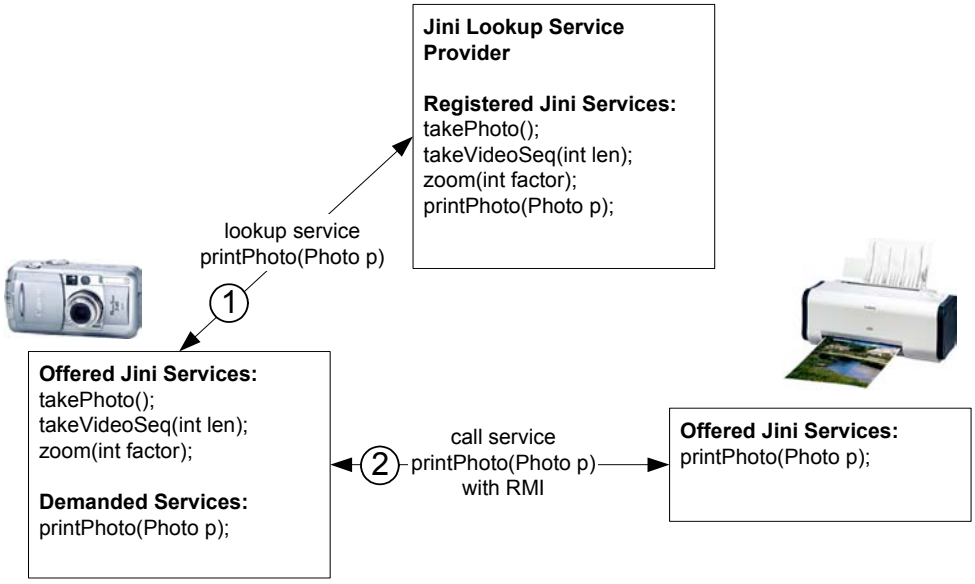

Fig. 19 Lookup process for a `printPhoto(Photo p)` service

2.9.2 Leases

In Jini environments, as in all distributed systems, a client has to know how long a service runs and how reliable it is. These characteristics may differ from device to device. While a printing service may be running for months without interruption a mobile device could power off unexpectedly.

Jini tries to handle this problem with *leases*. A lease is a period of time for which the service provider guarantees that the holder of the lease is able to access a Jini service. Jini works with lease duration rather than with absolute time because of synchronisation problems. A client can request a lease for a certain period of time. It can renew an existing lease or cancel it if the client is not longer interested in the service. If the lease times out without a renew request the lease expires.

2.9.3 Jini Summary

Jini was one of the first service platforms that supported the spontaneous lookup and interaction of distributed services. It is based on the Java environment, which allows Jini developers to use many well-established technologies such as Java *RMI*, Java object *Serialization*, *JavaBeans*, *Enterprise JavaBeans*, and *JavaSpaces*.

One of the major problems of Jini is that it is not possible to run it on the Java Micro Edition (*J2ME*), which is the Java Virtual Machine implementation for embedded systems. Jini needs the full Runtime Environment, which means that a Jini device can not run on most embedded and mobile devices.

Another drawback of Jini is that the service interface has to be specified as a Java interface type which means that a Jini service has to be implemented in Java. Modern service platforms use language- and platform-independent interface definition languages such as *WSDL* (*Web Services Description Language*).

3 State of the Art

In order to show the progress in creating context-sensitive applications and designing middleware solutions that are able to deliver context information it is necessary to describe some existing research projects. Context-awareness and the supporting middleware is an important aspect for many research groups in the area of pervasive computing. This section describes a selection of research projects, which heavily influenced our work. Some projects, such as *PARCTAB* from Xerox PARC or the *Context Toolkit* from the Georgia Institute of Technology exist for quite a while now. They have pioneered the research in this area and have coined the term context-awareness. Around 2002 many companies realized that with the increasing number of mobile and embedded devices the need for a lightweight interconnecting middleware grew. Therefore we will also discuss a few projects from companies such as AT&T and Hewlett Packard. The projects described in this section represent the current state of the art in context-aware computing.

3.1 The PARCTAB Project

In 1994, a Xerox PARC group under *Bill N. Schilit* developed a test environment for ubiquitous computing in an office environment, called *PARCTAB* [ParcTab]. This project was the first that mentioned the idea of using context information to support distributed applications. It also stimulated some interesting thoughts about the area of context information handling.

Schilit used the term "*dynamic environment object*", which should not be confused with objects in the object-oriented sense, but should be seen as a non-specified collection of data that is self describing, because it includes metadata. The data in a dynamic environment object is organized as *Attributes* that are key-value pairs holding pieces of information about the object. Attributes can also contain collections of values.

The following example shows how the attributes of a printing device are encoded in PARCTAB:

```
{ { Name Printer:snoball }
  { Location LID:35-2-1-06 }
  { format { ip ps text } }
  { features { duplex staple highlight } } }
```

The term *context-aware computing* was defined as follows in PARCTAB:

> *"Context-aware computing is the ability of a mobile user's applications to discover and react to changes in the context in which they are situated."*

According to PARCTAB a context-aware system has to perform the following three basic functions:

1. **Discovery**: Learning about entities in the environment and about their characteristics.

2. **Selection**: The ability to decide which resource can be used to achieve a certain task.
3. **Use**: The action of sending work requests to the selected resource.

3.1.1 PARCTAB system architecture

The architecture of PARCTAB is based on *infrared tabs* and corresponding *tab agents* [PTab95] as it is shown in Fig. 20. An infrared tab is a small electronic device with longer battery life time than for example PDAs. Infrared tabs provide operation times of more than a month. An example for an infrared tab is shown in Fig. 13.

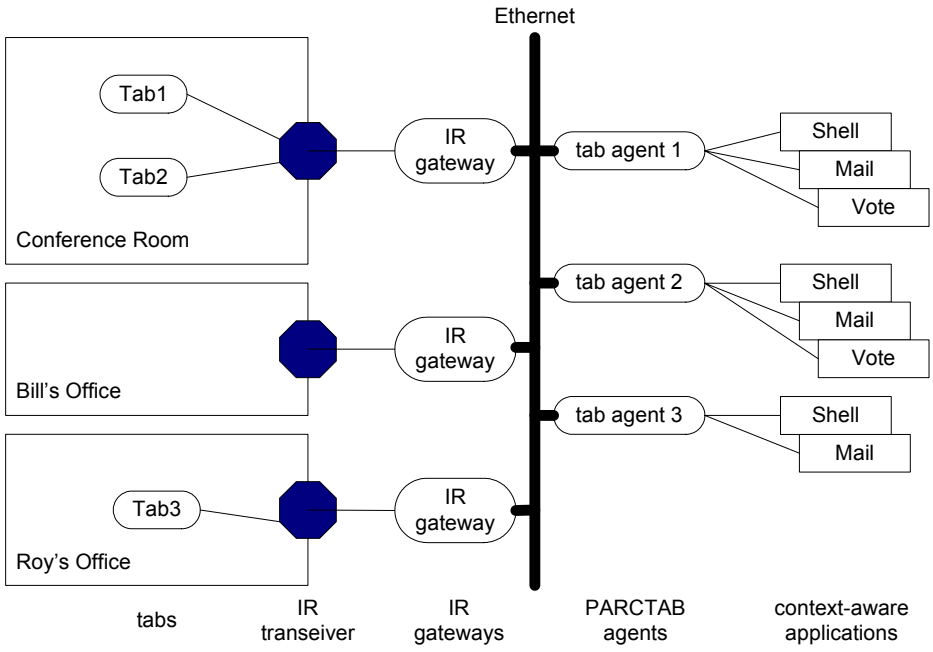

Fig. 20 PARCTAB system architecture overview

Every location in a PARCTAB environment (e.g. every office room) contains one or more IR transceivers which collect IR *beacons* and deliver them to the corresponding IR gateways. Beacons are small data packages that are broadcasted on a specific medium in order to announce the existance of a digital device inside an environment. IR beacons are sent through the infrared medium. Radio based beacons are used by WLAN access points to announce their existance to a mobile device. One or more IR transceivers can be bound to one gateway. The gateway itself is connected to an Ethernet network and is therefore able to send the tab information to the responsible tab agent. The tab agent can now provide context information to context-aware applications. In order to get the location information about a person wearing an IR tab, a containment hierarchy was defined. Fig. 21 shows an example of such a hierarchy, where a region is divided into several buildings, a buildings into several floors and a floor into several rooms.

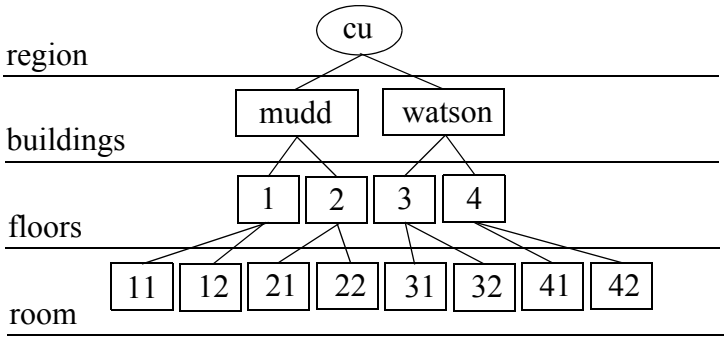

Fig. 21 PARCTAB containment hierarchy

Despite the fact, that the PARCTAB project came to years, it proposes a solid definition of terms and problems in the area of context-awareness. Many modern context research projects began with examining the PARCTAB architecture. For our work, the notion of context as a collection of attributes within an entity was important. In the following chapters we show how the PARCTAB architecture influenced our SiLiCon framework middleware and where we tried to improve it.

3.2 The Context Toolkit

A group at the Georgia Institute of Technology headed by *Anind K. Dey* developed a software development framework, called *Context Toolkit*, which integrates sensor and actuator devices into a context-aware system [Dey01]. Their definition of *context* is:

> *"Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves."*

The definition of context-aware computing was given as follows:

> *"A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task."*

The Context Toolkit focuses on the development process of context-aware applications and on the reuse of different software components. Once a sensor or actuator type has been integrated into the Context Toolkit's library, it is possible to reuse the component in different applications. A *context widget* is a Java object that offers services that are bound to a specific type of sensor or actuator. A context widget hides the complexity of accessing the specific hardware of sensors and actuators. The context widget provides similar functionality as a GUI widget. It is possible to access the context information in a context widget either by polling or by a notification mechanism. A context widget is able to store its actual state within a database, in order to trace the widget's state over a period of time. With a trace of widget states over a period of time it is possible to claculate statistics about possible future states of the widget. Possible future states of the widget could lead to automatic configuration or smart system behavior. The Context Toolkit architecture consists of three building blocks:

1. **Widgets**: A context widget provides attributes and callbacks. Attributes are pieces of context information, which can be accessed via polling or subscribing. The context widget is able to notify interested objects via callbacks. Other objects can query the widgets' attributes and callbacks at runtime. Objects can use the widgets to discover their environment.
2. **Interpreters**: The context interpreters are responsible for an interpretation of available context information at runtime. The interpretation of context information means that available information is used to trigger a state change of one or more context widgets. For example if a location widget collects the information that more that three people are inside the conference room and the beamer is powered on an interpreter could use this information to interpret the situation as a meeting and to change the peoples state to 'in meeting'. Context interpreters are separated from the application layer. Therefore it is simple to reuse them in different applications.
3. **Aggregators**: Aggregators can be used to group several context widgets into a single compound widget, which has the same functionality as a simple context widget. A compound widget is also able to store its history and to provide attribute information and notification.

Abstraction with context widgets. One benefit of context widgets is that they abstract from raw context data and provide high-level context information that is interesting for applications. Furthermore, it is useful to be able to track the history of a context widget in order to calculate statistics or to predict possible future states. These abstraction facilities are modular and thus reusable. As it is shown in Fig. 22 context-sensitive applications can retrieve context information from aggregators, from widgets, or from interpreters.

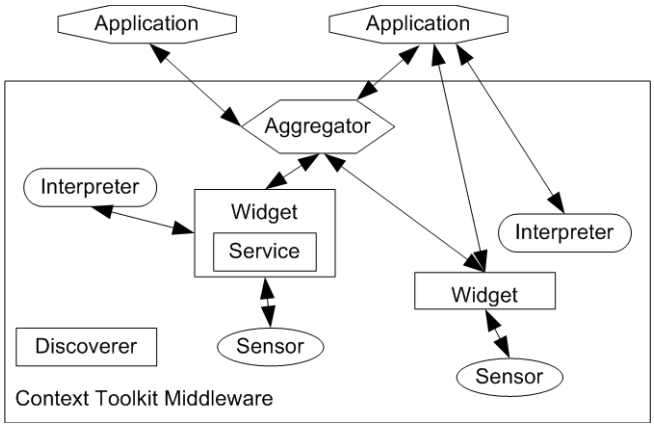

Fig. 22 Context Toolkit sample application

The main focus of the Context Toolkit is to make it easier for application designers to create context-sensitive applications. The PARCTAB project that was discussed in Chapter 3.1 provided a first approach for developing ubiquitous applications, because context-sensitive and ubiquitous applications were not implemented before. The Context Toolkit focuses on rapid application development (RAD) of context-sensitive applicationsand uses components for reusing sensor abstractions and interpretation processes.

Another interesting point in Dey's work is, that he distinguishes between user input and context retrieval in general. The first difference is that user input is usually limited to a single workspace while context information retrieval is a distributed process most of the time. The

second difference is that the retrieval of context information sometimes requires another abstraction layer. Raw context information often has to be transformed into a client-specific format (for humans a location given by latitude and longitude may not be as useful as a point on a map or the name of the street). Therefore context widgets hide the complexities of gathering sensor information and can be reused on a higher level by different context-aware applications. Finally, the third difference is that context-retrieving middleware has to provide the collected information to multiple client applications while user input usually goes to just one specific application.

Context services. Context widgets offer public methods that can manipulate the state of the environment. These methods are called context services. The Context Toolkit offers synchronous service calls, as well as asynchronous service calls to invoke context services. An example could be a context widget that measures the light level of a room and offers context services to change the light level.

Interpreters. Context interpreters take specific context information and combine this information to reason about the state of the environment. As an example, Dey provides a situation where several people are sitting in a room and the noise level increases. In this situation an interpreter could use the noise level information and the number of persons in the room to guess that a meeting is going on. Interpreters can be loaded or unloaded at runtime which offers more flexibility than a static system.

discovery. In contrast to the SiLiCon framework, the discovery mechanism used by the Context Toolkit is based on a single central registry service. This solution was chosen because of simplicity. Besides the fact that a central discovery service represents a bottleneck and a single point of failure, it is not possible to use it in an ad-hoc communication environment. In an ad-hoc communication environment every peer has to offer its services itself and should be able to discover other services. In the Context Toolkit a plugeable discovery mechanism is used, in order to enable the use of different discovery mechanisms or standards such as *SLP* (*Service Location Protocol*) or the discovery mechanism used with Jini.

3.2.1 Context Toolkit Example Applications

In/Out Board. An In/Out Board is a device for tracking the presence of users in a research lab. It displays the presence state on a board in front of the research lab. The user tracking, a so-called *presence widget*, is based on a user-triggered docking mechanism and on a radio-based indoor location system. The users wear pager-sized tags, that can be detected by antennas inside the buildings in order to change the presence state of a user automatically. A visualization of the presence state displays red dots for every person that is not inside the area and green dots for users that are inside the area. An interesting fact is that the meaning of the states *in* and *out* depends on the location of the requesting user. If the requesting user changes the building his in/out state will change, because the user changed his location to outside the first building and inside the second building. So the location state of the user changed to a different building and his In/Out Board configuration changes. Fig. 23 shows a screenshot of the In/Out Board.

Fig. 23 Screenshot of the In/Out Board based on the Context Toolkit

DUMMBO (Dynamic Ubiquitous Mobile Meeting Board). With the DUMMBO project the Context Toolkit team showed how an existing computer-supported cooperative application that did not use context information originally could be improved by using context information. DUMMBO supports the digitising and recording of meetings.

The original application provided an interactive whiteboard which is able to capture notes of spontaneous meetings. The system is able to record the audio signal in a meeting room as well as the notes that are painted on the whiteboard. After a meeting it is possible to store all the captured information.

The original system started to record a meeting when a person changed the content of the whiteboard. The augmented system is now able to start the recording process when the Context Toolkit gathers information about a group of people who are located in front of the whiteboard. The recorded meeting data (e.g. the recording date, time, meeting members, and location) is automatically stored in a repository. A user is now able to search for meeting data by specifying criteria such as the time, date, location and meeting members. Another interesting feature of the augmented DUMMBO system is the possibility to view the timeline of all changes that were made on the whiteboard. That is possible because the context widgets of the Context Toolkit are able to archive their historic states. It is also possible to pick a specific time and to view the whiteboard state at this time. Synchronously, the user can hear the recorded audio stream at this time. It is quite easy to replay a recorded meeting, or to find a specific situation that happened in a meeting.

3.3 The Sentient Information Framework

The *Sentient Information Framework* was developed by AT&T's Sentient Computing Group [Ipi01]. Its goal is to enable software applications to sense their environments and to define a model of their surrounding ambiance. If parts of the environment are mapped into a digital world model, it should be possible to use the environment as an interface to sentient applications. Using the natural environment as an interface to interact with applications is directly related to the research area of *Tangible Interfaces* [Ishii97] and context awareness. Sentient

computing is based on information retrieval with environmental sensors, as it is also described in context-aware computing research.

The sentient computing research group developed different locating technologies, like the *TRIP* (*Target Recognition using Image Processing*) system, which was described in Chapter 2.6.4. TRIP offers a cheap location-sensing mechanism based on visual ringcode markers. The goal is to allow mobile devices to sense their environment and to discover entities around them.

CORBA based software middleware. The *Sentient Information Framework* (SIF) architecture is based on CORBA and on object mobility in CORBA. SIF defines an application construction model for sentient application implementations. It separates the context capture and abstraction layers from the sentient application semantics. Like other context information middleware frameworks (e.g. Context Toolkit and the SiLiCon framework) SIF is sensor-independent. The visual location sensor technology TRIP is one instance of a sensor that is encapsulated in a context generator.
The SIF components are categorized into three types:

- **Context Generators**: A Context Generator (CG) encapsulates sensors (such as the TRIP location sensor) that generate context information. Context-aware applications are also encapsulated as context generators, because often the output of one application is used as input by other applications. Context generators are sources of sensorial events that are fed into the SIF middleware.
- **Context Channels**: Context Channels (CC) are implemented as *CORBA Notification Channels*, which receive and filter events for different consumers. The consumer is able to specify filters in order to receive a subset of all events.
- **Context Abstractors**: Context Abstractors (CA) consume context data, interpret it and provide new combined context information from the input data. A context abstractor is also able to use the output of another CA in combination with other information to generate a required data format that a specific application is able to process. Context Abstractors are comparable to the Context Toolkit's interpreters and to the SiLiCon context rules.

Fig. 24 shows that an example modelled in the SIF architecture looks similar to the Context Toolkit's architecture, except that SIF combines the idea of interpreters with aggregators. According to our experiences in the SiLiCon work, this combination seems to be useful, because most of the functionality of interpreters and aggregators overlaps.

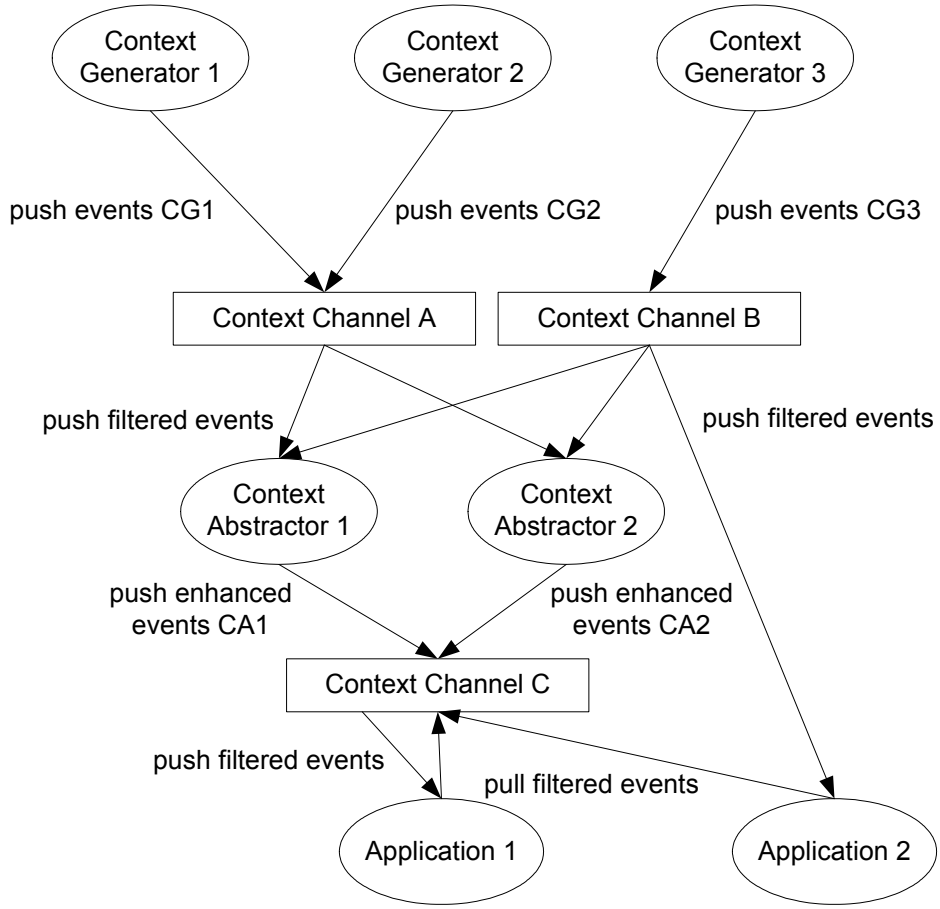

Fig. 24 Sentient applications modelled within the SIF architecture

LocALE. Sentient computing applications are based on a distributed and mobile environment where users move between different environments. When a new sentient application is developed, all dependent components have to be restarted locally. To avoid the local restart of application-dependent components it is necessary to design an environment in which the life cycle of components can be managed automatically over the network.

The *LocALE* (*Location-Aware Life cycle Environment*) is designed to manage the life cycle of CORBA objects that are distributed across a network. Additionally, LocALE provides load-balancing and fault-tolerance features for all CORBA objects whose life cycle it manages. The architecture of the LocALE middleware is shown in Fig. 25.
The LocALE architecture consists of the following components:

- **Life cycle manager**: The life cycle manager offers operations for controlling the life cycle of LocALE-managed CORBA objects. It caches the locations of all CORBA objects in order to provide references to them. Those references are used by clients that request a connection to a LocALE-managed CORBA object.
- **Life cycle server**: Life cycle servers represent containers for LocALE-managed CORBA objects. They follow the operational instructions that are delivered by a Life cycle manager. Life cycle servers also help their hosted objects with the migration process to other containers.
- **Type proxy factories**: A type proxy factory is responsible for the creation of strongly typed objects. A client can therefore use the life cycle manager to call a

strongly typed constructor instead of a generic constructor. When a client uses the strongly-typed creation mechanism of type proxy factories, it will get a compile-time error instead of a runtime error if an argument type does not match. A client could also use the generic object creation interface where any constructor arguments are taken and evaluated at runtime. If an argument mismatch is ecountered at runtime the client gets a runtime exception. For clients it is generally better to use the strongly-typed object creation interface to simplify the creation code and to provide type checks at compile time.

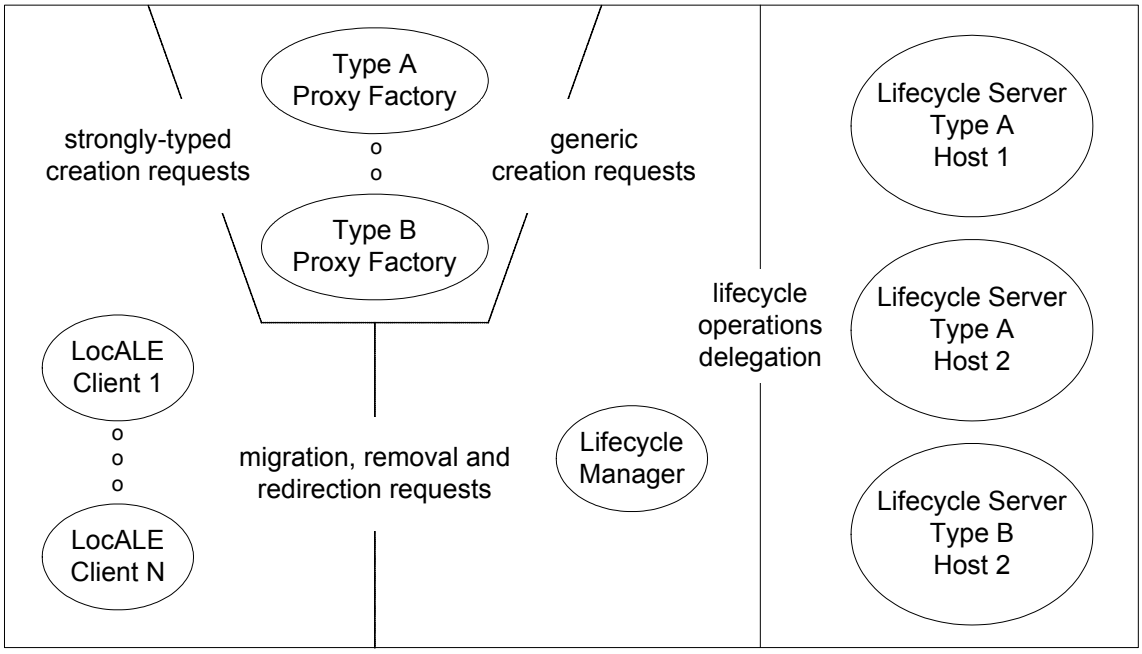

Fig. 25 Three tier architecture of the LocALE middleware

3.4 The Cooltown Project

The Cooltown project from Hewlett Packard is one of the most popular research and demonstration projects in the area of pervasive and mobile computing. Its context-aware environments are heavily based on embedded web servers and a massive use of the HTTP protocol, in order to provide simple access to resources anywhere. The goal is to create a web presence for real-world objects classified as *people*, *things* and *places* [HP01]. Embedded web servers are put into everyday things (e.g. printers, furniture, books or artwork) and are filled with information. Information about everyday objects is addressed by URLs and URL sensing is used to discover object information. URL sensing means to discover the URL of a newly appearing object in order to locate its machine-readable information. URL sensing can be realized with different sensor technology (e.g. visual barcode recognition, RFID, Network discovery).

3.4.1 Pushing Web Technology into Physical Objects

In order to create an environment where nomadic users can find information about everyday objects, the HP Cooltown group adapted traditional web technology to fit their requirements.

They found that the web model, which is built upon a decentralized but standardized interaction, provides a solid basis for environments that support nomadic users. Most of their work focuses on the creation of bridges to map physical objects with digital information. While digital information has naturally many links to the physical world, the physical world lacks links to digital information. Most of the time, people work with physical objects instead of digital devices. Therefore a closer link between virtual web content and physical objects improves the usability of many applications and services. The Cooltown group integrates location specific digital services, which are able to communicate with nomadic users. Things that are physically related are grouped into places, where a visiting user can get information about interaction possibilities with the physical objects he encounters. With places, Cooltown realizes a primitive containment hierarchy, which is used to implement location dependent services and applications.

Fig. 26 shows the different layers of the Cooltown web presence infrastructure. At the bottom of there are mechanisms for gathering the address of the objects. These mechanisms perform URL sensing to receive a reference where the digital description of a discovered object is stored. URL sensing is not specifically bound to a sensing technology, but is implemented with a multitude of different object identification technologies, which were discussed in Chapter 2.6. The middle layer represents the content exchange and coordination layer, which is responsible for the transport and hyperlinking of semantic information over the HTTP protocol. On top of the web presence architecture stand the service-oriented communication with nomadic users. This layer is responsible for delivering information about the objects to the users.

Cooltown distinguishes between three main object classes: *Person*, *Thing* and *Place*. For most of the appliances these three classes provide a pragmatic way without any complexity or performance problems. The main goal of Cooltown is to make these three classes web present. Web presence means that every object owns a digital description that provides meta information about the object and links to related objects. Web presence means that information about an object is accessible through the Web. The information about an object targets different types of consumers. For humans a HTML representation is presented and for machines a machine readable format is used (e.g. RDF, CC/PP). There are several ways to make real world objects web present. One possibility is to embed a web server inside an object in order to enable the object to host its own digital information (for humans HTML and additional machine readable information). A second possibility is to host digital information about an object on an external web server. A person is web present if any web server hosts a directory, which is accessible with the URL of the person, which contains information about the person, like the location of the person. Places are called web present if a so called *PlaceManager*, which is a web server module, keeps track of all objects in a place.

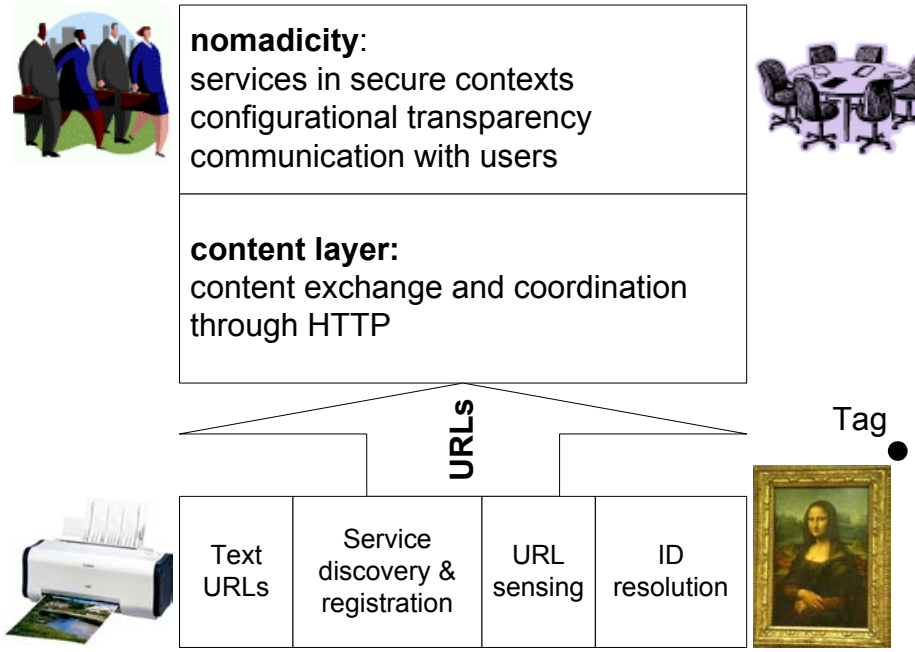

Fig. 26 Cooltown web presence infrastructure layers

In Cooltown the term *context* is defined as follows.

'the context of an object is represented through a virtual collection of related resources'

Relevant pieces of information about an object are called '*resources*' and the relationship between different objects are stored in a so-called '*relationship directory*'. Dynamic parsers use HTML templates to generate a user specific view on machine readable data. Fig. 27 shows how a dynamic parser generates a user-specific view on the object information.

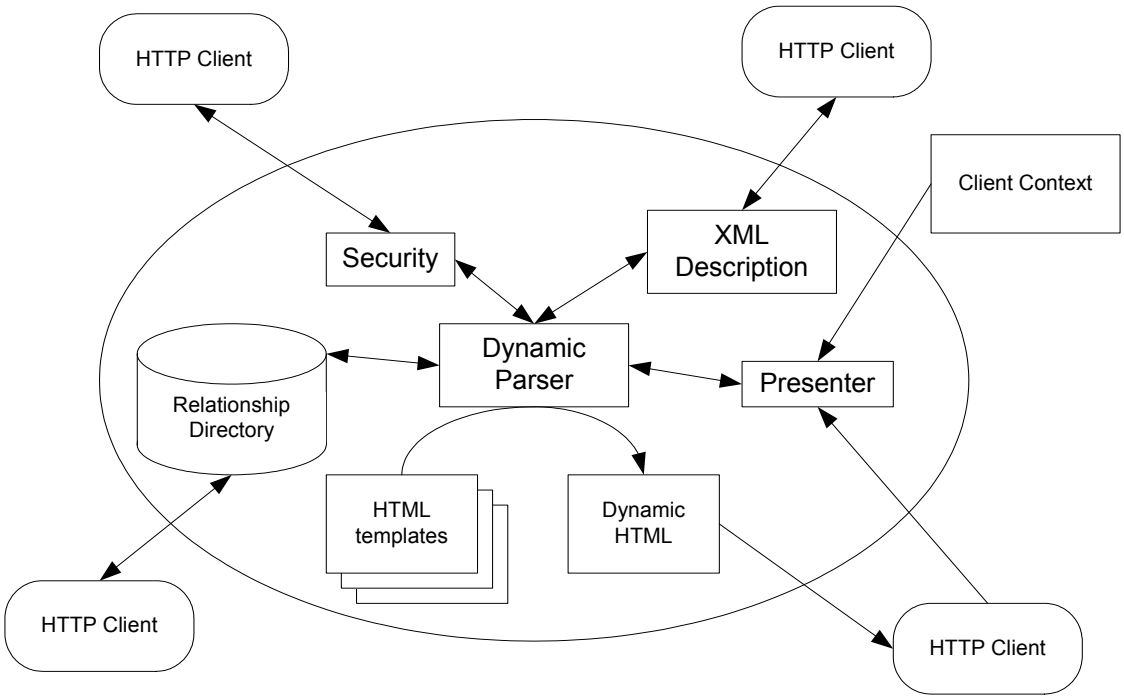

Fig. 27 HP CoolTown's architecture of a web presence server

The purpose of the web presence architecture shown in Fig. 27 is the generation of dynamic content according to the actual state of related web-present objects. The *Presenter* module is responsible for triggering the *Dynamic Parser* to generate the user-requested HTML resources. The *Dynamic Parser* receives information about related objects through the *Relationship Directory*, which automatically stores all semantic object information as well as logic links between them. The *Security* module is responsible for the security policy configuration. The raw content information is generated and processed in a XML-based format.

URL sensing. In order to find the web presence of an object, as it is shown in Fig. 26, *direct* and *indirect sensing* is used. A sensing process is called direct, if a device gets an URL from another URL-emitting device (sent within beacons). This URL leads directly to the requested resource via a PlaceManager, which is a server implementation providing policy-driven views on location-relevant information. The PlaceManager is necessary in order to model physical places, which do often not correspond to the underlying network topologies. Indirect sensing, on the other hand, uses location-specific IDs that are used to look up the mapped resource in a locally available registry.

4 The SiLiCon Context Framework

The SiLiCon (=**Si**emens-**Li**nz-**Con**nection) framework was developed in cooperation with Siemens Munich (CT-SE 2) and implemented between 2002 and 2004. The main focus in this work was to provide a stable and easy to use software middleware in order to integrate context information into existing application areas as well as to design new context-aware demonstration applications. As the market for low-cost embedded sensor and actuator devices grows, the need for an integration middleware software framework increases. As mentioned in Chapter 3, various research groups have already started to design software frameworks, which support the retrieval, processing, and delivery of context information. Many of these frameworks are specialized to a certain task and according to this task the world model and design of those frameworks was chosen. For human-centered and infrastructural scenarios the person, thing, and place classification system emerged. In this work a more general role based classification mechanism is used to describe environments. The requirements resulting from the use of radio based networks demand for a more decentralized view of communication processes. The SiLiCon context framework is based on a peer-to-peer communication architecture with support for different kinds of transport protocols and message encoding mechanisms. This architecture provides a flexible basis for context-aware applications that operate on a multitude of heterogeneous embedded and mobile devices. The next sections show how the SiLiCon framework is able to simplify the context information retrieval, how it interprets state changes in the context information, and how the distributed delivery of context information is managed.

4.1 Concepts

This section gives an overview of the SiLiCon framework's main features by defining its central terms in the following subsections. Every subsection introduces one specific aspect of the SiLiCon framework. The detailed design and implementation of this list of features is given between Chapter 4.2 and Chapter 4.7.

4.1.1 Retrieval of Raw Context Data

One of the most important tasks of a context information management framework is the gathering of raw context data from different types of sensors. These sensors are divided into two kinds, *event-triggered sensors* and *time-triggered sensors*, as it is described in [Fer02]. An example for an event-triggered sensor is an RFID reader which triggers an event when an appropriate RFID transponder enters the communication range. Whenever the state of the RFID reader changes it triggers a context event with detailed information. Time-triggered sensors, on the other hand, are able to measure context states that change continuously over

time. For time-triggered sensors it is necessary to specify a certain reading interval. An example for this kind of sensors is a temperature sensor which is able to measure the environment temperature or a sensor which is able to measure the ambient light level.

In addition to these sensors another important context information source should be mentioned. Any information in a context framework, which does not change its value automatically, is called *passive context information*. For example, the maximum screen resolution of a digital device is a passive context information. The user or the context framework is able to change this information, for example if the device gets a new screen, but the passive information does not change automatically as the temperature information does.

One task of the SiLiCon framework was to simplify the context information retrieval for the application designers. The framework simplifies the retrieval process to a level where there is no difference between the various kinds of context retrieval mechanisms. Fig. 28 shows the SiLiCon framework's classification of context information retrieval mechanisms, with some concrete sensor examples. Every low-level context sensor is derived from a base class that abstracts all available context sensors. In Fig. 28 this base class is represented by the node called *context attribute*. This node is called attribute, because it represents one specific attribute or property of an environment or of an entity in this environment.

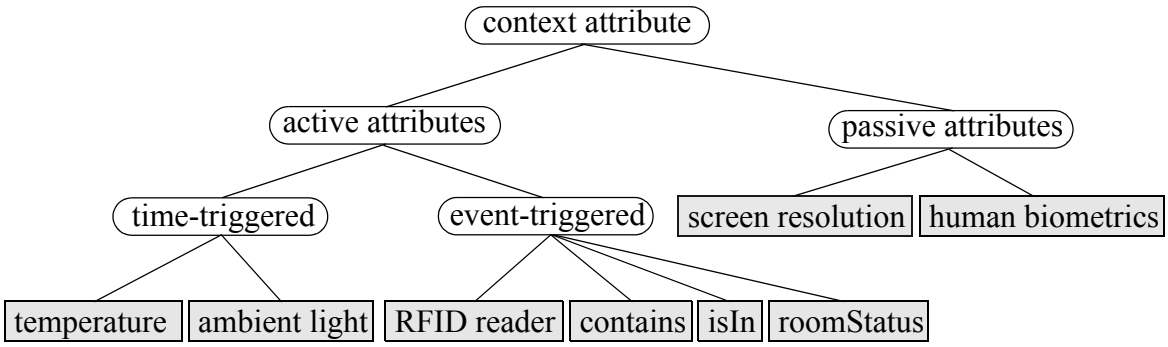

Fig. 28 SiLiCon classification of context retrieval mechanisms

The SiLiCons attributes are similar to the static declarations of the RFD standard. The *RDF* (**R**esource **D**escription **F**ramework) standard has beed defined by cellular phone manufacturers like Nokia to create a description for specific attributes of a mobile phone. The GSM service providers can use this RDF descriptions to improve their service delivery according to the attributes of specific client phones. It is possible to specify attributes like the screen resolution, computational power, audio in/output capabilities, storage possibilities, calendar and mail functionality and many more. RDF is an XML-based representation for static entity properties. It does not contain any information that changes over time unless an external process changes the RDF description.

In relation to the terms used in the RDF standard we define the term *attribute* as follows:

> *'An attribute is a software component that expresses one specific aspect of an entity in order to be able to provide services which are related to this specific aspect.'*

The SiLiCon framework uses a set of attributes to describe objects or entire environments. Attributes can be reused in different environment scenarios and enable the rapid context-

aware application development. All attributes are derived from the class `CFAttribute`, which provides basic methods for initializing and for processing of events. A simplified view on the interface of class `CFAttribute` is given here:

```
public class CFAttribute {
    private String identifier;
    public CFAttribute(XmlTag tag);
    public void init();
    public void CFEventReceived(Object sender, CFEvent ev);
    public Object CFSyncEventReceived(CFEvent ev);
}
```

The field `identifier` represents a unique identifier, with which the attribute can be refered to. The constructor of an attribute takes a `XmlTag` object which contains the initial configuration elements. The methods `CFEventReceived` and `CFSyncEventReceived` are responsible for receiving incoming events.
Attributes that derive from the class `CFAttribute` can perform several different tasks. Attributes can represent key value pairs which are accessible with getter and setter methods. Attributes can wrap sensor and actuator hardware, in order to offer methods to access the specific hardware functionality. A possible attribute could be a display attribute which offers methods to get the display resolution information and to change the resolution information. Every method of an attribute which is publicly accessible and marked with 'CF' is called a *service* or a *context service*. Attributes are able to trigger events in order to inform other objects about information changes.

4.1.2 Object Description with Entities

To describe context-aware scenarios it is necessary to create a representation of all entities that are relevant for this scenario. As Schilit already mentioned in the PARCTAB project (chapter 3.1), a *dynamic environment object* is a non-specified collection of data. In the SiLiCon framework a dynamic environment object is represented by a *context entity*. A context entity, or an *entity* for short, is a non-specified and non-classified collection of attributes. We define the term *entity* as follows:

> *'An entity is a collection of attributes which describe every relevant aspect of an object that is a member in a context-aware scenario.'*

An entity is responsible for the representation of object information in a context scenario. Entities express their information with attributes that can be loaded into an entity. Every entity contains a collection of attributes, that is used by applications for classification purposes. Applications can use these attributes to assign roles to entities in order to classify them. A collection of attributes is called *template*. Fig. 29 shows three entities with their attributes. The first entity represents a PDA device, the second a person and the third a location.

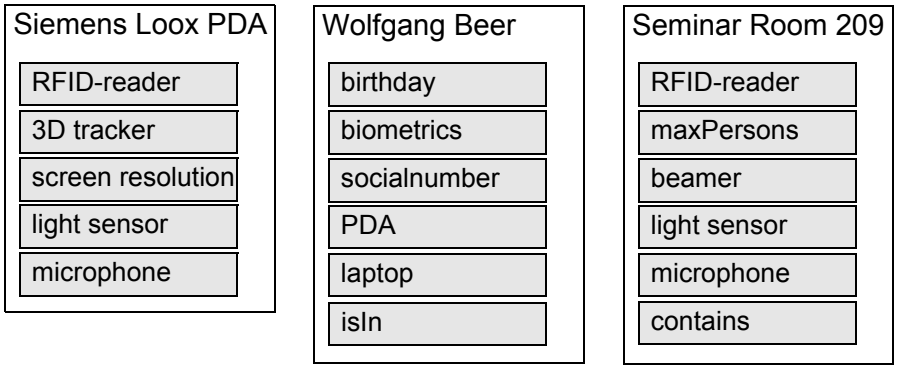

Fig. 29 A group of entities describing a PDA, a human user and a seminar room

Attribute templates fulfil several important tasks in the SiLiCon framework. An attribute template can be used by an application to classify an entity which enters a scenario and to assign an application-specific role to it. An application could for example identify all persons in an environment by finding all objects that have a *Person* template. This is called *role-based classification*. Role-based classification provides a powerful mechanism for middleware context frameworks because the attributes of entities and therefore their roles can change at runtime. Furthermore, attribute templates can be used by visual builder tools to create new instances of entities. So, if a new instance of an entity should describe a person, it is simple to use the *Person* template to create the basic skeleton of this instance. Classification with attribute templates is an important design decision in the SiLiCon framework. It differs completely from static classification approaches where entities are classified at design time.

4.1.3 Event-Based Communication

In the SiLiCon framework any communication between entities and attributes is event-based. When the actual state of the context world model changes, context events are triggered to inform interested communication partners about the state change. Every entity or attribute is able to trigger two types of context events: *addressed events* and *non addressed events*. When an event is triggered it is put into an event queue, which is global for each host. This global event queue is actively delivering context events to the event destinations. If the event is non addressed, the queue delivers it to every entity on the local host. The entities themselves have to check if they are interested in the event or not. If the event is directly addressed the queue delivers it to the specified receiver.

Every context event contains information about the entity and the attribute which triggered the event, the name of the event, and a list of parameters containing information about the event. If it is a directly addressed event the event object also contains information about the receiving entity and attribute, as it is shown in Fig. 30.

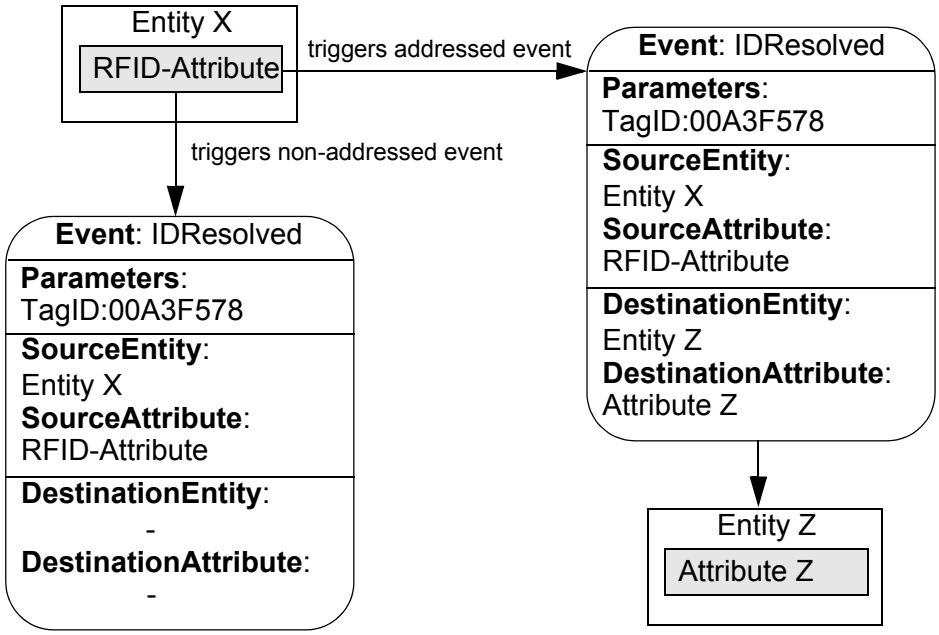

Fig. 30 Trigger of addressed and non-addressed context events

The transport of context events is completely transparent for the sender and the receiver.

4.1.4 Dynamic Definition of Context Scenarios through Rules

One of the most important aspects of our work is the possibility to define distributed interaction scenarios between attributes using *ECA rules* (Event Condition Action Rules). An ECA rule consists of an event, a condition and an action. If an event occurs and the condition yields true the corresponding action is performed. An example ECA rule, which reacts on the appearance of a specific RFID transponder, is shown here:

```
on RFID.TagAppeared ( string id ) {
    if ( id == “000A03CE000” ) {
            // perform any actions
    }
}
```

With ECA-rules it is possible to specify how an environment or a single entity reacts on a specific event. The scenario designer is able to define a set of rules that specify which actions are triggered on a state change. As described before, the SiLiCon framework is event-based. Thus the scenario designer defines rules that describe the reaction to every event in the framework. Rules can be added at runtime to change the behavior of the whole system. It is possible to implement a distributed rule editor that can be used to develop and deliver ECA rules over a network at runtime. ECA context rules provide a powerful and dynamic mechanism, to define the behavior of context systems. The total set of context rules specifies a state machine which controls the distributed system. By adding or deleting rules at runtime the scenario designer is able to change this state machine dynamically. It is even possible for the action part in a rule to trigger the insertion of new rules or the unloading of existing rules at runtime. So the state machine is able to change itself at runtime by managing the rule repository.

4.1.5 Discovery of Entities in Local Environments

Working within a mobile and dynamically changing network environment makes it necessary to provide a mechanism for resource discovery in local environments. Entities which enter an environment must be able to discover possible communication partners. Depending on the running application, the middleware has to discover entities and their attributes in order to determine in which role the entities act at the moment. For a chat application, which sends messages to persons in the environment, the SiLiCon middleware has to discover all entities which act in the role of a person. The application can define specific attribute templates or use already existing attribute templates. The middleware takes an attribute template that describes a certain role and the lookup mechanism discovers all entities that act in this role. The application can then perform application-relevant interactions with these entities such as sending messages to a set of people.

The discovery mechanism, like the transport mechanism, is implemented as a pluggable module that can be exchanged. As an example we implemented a module which sends IP broadcast or IP multicast packages to announce discovery information in the local network environment.

4.1.6 Configuration of Context Applications with XML Scripts

For application designers it should be as simple as possible to define new environments and to design context-aware applications that are running within these environments. The designer starts with the identification of relevant entities. The set of those entities is not fixed, however, because entities can enter or leave the environment at any time. The identified entities only provide an initial set of members that fulfil the requirements. The SiLiCon framework offers the possibility to configure entire environments and interaction scenarios with XML scripts. The application designer configures a *context container* on each digital device, which is able to host and to manage a collection of entities. A context container is an entity that exists only once per host. It provides references to shared resources (lookup, transport and logging modules) and holds a collection of hosted entities. The container configuration contains the specification of different transport modules, lookup modules and a system logging facility, as well as a link to the configurations of the attributes. The configuration of an entity consists of a list of attributes with their initial parameters and a set of context rules which coordinate the interaction between the entity and its communication partners.

The advantage of this hierarchically structured XML configuration of context-aware applications is that even complex scenarios can be created automatically or by a visual builder tool. The application designer does not have to write a single line of code, but designs a complete scenario with any XML editor such as Microsoft Visio or XML Spy.

4.1.7 Resource and Performance Optimization

The SiLiCon framework was written in Java and is based on Sun’s Personal Java Profile [PJSpec] for running on mobile devices. The Personal Java Profile implements the JDK 1.1.8 base class library. This means that most of the modern libraries for parsing XML-coded data (jaxp, xerxes), creating web services (axis, glue), or creating a peer-to-peer framework (jxta) are not supported at the moment.

We decided to use the kXML2 library for parsing XML-coded data and the kSOAP2 library to parse and create SOAP calls [Enhydra]. Both libraries were designed to run on mobile devices and do not need much storage (20KB for kXML2 and 50KB for kSOAP2).

In order to create WSDL descriptions of context services at runtime, a library called kWSDL was implemented. kWSDL takes a Java class and generates the WSDL description of all public methods with their parameters. kWSDL is based on kXML2 and the generation process is more efficient than with comparable libraries. The performance gain was achieved by limiting the possible method parameter types to simple types (long, double, boolean, char and String).

4.2 Framework Architecture

The basic concept of the SiLiCon framework allows the description of digital and non-digital entities with their attributes. In this work digital entities are referred to as entities which are able to store, process, and deliver digital data. They are able to host the SiLiCon framework and to store their own digital description. Examples for digital entities, sorted by their available CPU power, storage and memory amount are: servers, personal computers, laptops, PDAs, industrial PC104s, mobile phones, and embedded processors.

Non-digital entities, on the other hand, represent objects which are not able to process digital data, such as humans, everyday things, or locations. For this kind of entities it is necessary to use an object identification technology in order to sense them within the environment. Non-digital objects are depending on digital devices to host their digital descriptions.

The SiLiCon framework was designed to work in a dynamically changing network environment, also called ad-hoc network. Such networks do not need to have access to a global network. Typical instances of ad-hoc networks are personal area networks that build automatic connections between mobile phones, headsets and other personal digital devices. In this kind of networks it is necessary that every communication partner is able to provide information or services (i.e. act as a server) and to receive information or to use services (i.e. act as a client). An entity must implement both a client and a server part in order to be able to communicate with other entities. The disadvantage of these networks is that they are not stable. Entities are likely to appear or to disappear in the environment at runtime. There is no central authority where the entities can check whether a communication partner is reliable or not.

A context container, which manages the life cycle of a collection of entities, is implemented through the class `CFContainerEntity`. Every digital device in the SiLiCon framework has an instance of class `CFContainerEntity`. `CFContainerEntity` is derived from `CFEntity` and inherits all functionality from a genuine entity. At startup, the container parses a configuration file and loads the specified modules. After the container has successfully loaded all sending and receiving transport modules, the lookup module and the logging module, it starts to load the specified collection of entities. The class `CFEntity` represents an entity. Every entity is itself responsible for loading its attributes from the configuration file. Fig. 31 shows the initializing process at the container startup. When the container initializes, it calls the constructor of its base class `CFEntity` with an `XmlTag` as actual parameter. The `XmlTag`

contains the initial configuration of the container, the configuration of all entities and their attribute collections.

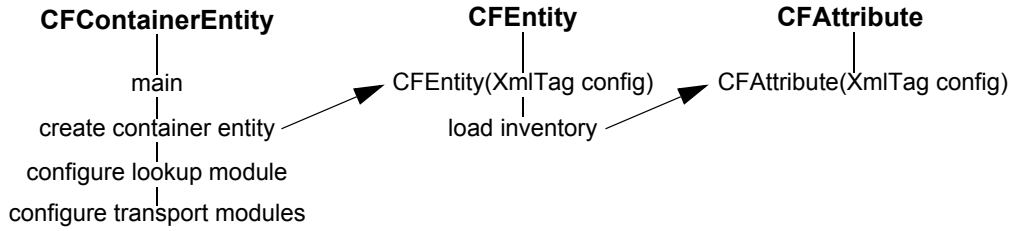

Fig. 31 Simplified container initialization

The constructor of class `CFEntity` loads its inventory as a collection of entities and attributes. The attributes are also initialized through calling their constructor with a configuration `XmlTag`, which contains the attribute-specific XML elements. After the container entity loaded and initialized its lookup module and a collection of transport modules, it loads all contained entities and attributes. Fig. 32 shows the configuration's XML schema. It contains a root element 'Container' which initializes the container entity. The Container element consists of one 'Logging', one 'Lookup', one 'Transport' element, and a collection of entities that are managed by the container. The Transport element always contains two elements; the 'SendModules' element and the 'ReceiveModules' element. This pair of elements contains the configuration of all sending and receiving transport modules and their encoding mechanisms.

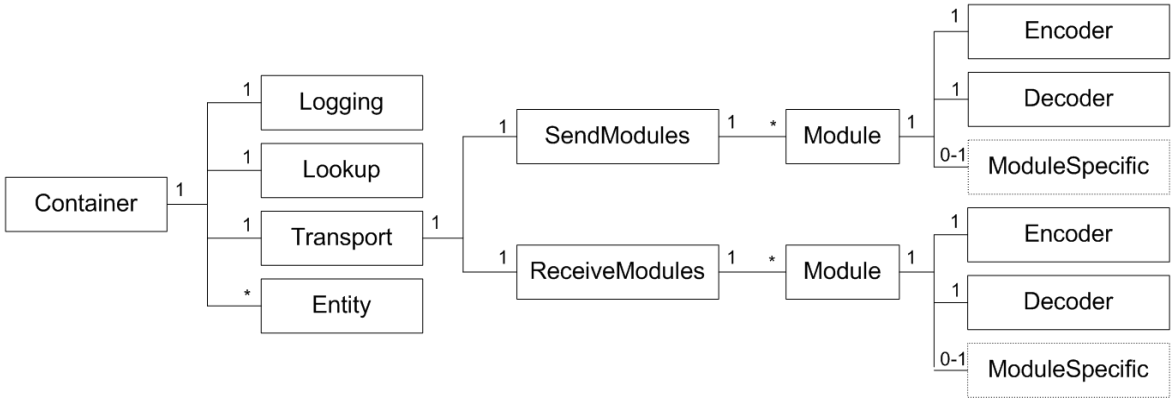

Fig. 32 XML schema for the container configuration

The container entity also initializes its event queue, which is responsible for the delivery of context events (addressed events and non-addressed events). In every container there is exactly one event queue. All the entities hosted in a container refer to this event queue and use it to send their events.

Whenever an entity is created at container startup time it creates and initializes an instance of a context rule interpreter, which is implemented in class `CFInterpreter`. The interpreter is able to load context rules into an entity and to interpret incoming events with these rules. When the initialization process has ended, the container waits for incoming events triggered by entities inside the local container or by remote entities. Fig. 33 shows the structure of a

container entity after the startup process. A container entity, like every entity, also contains an interpreter which is able to react on events.

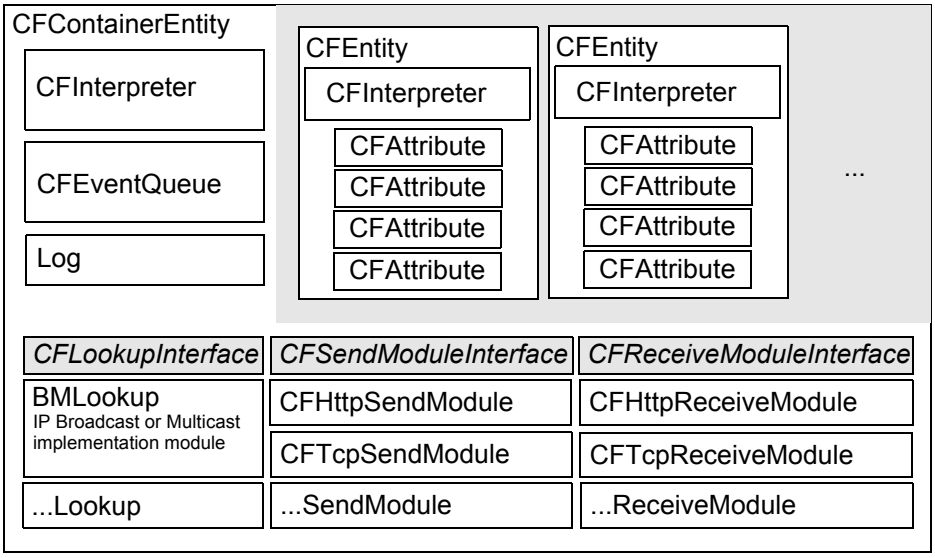

Fig. 33 SiLiCon framework architecture module overview

Events that are triggered by the container entity itselve, could also be interesting for all entities inside the container. Examples for such events are *EntityAppeared* and *EntityDisappeared*, which are triggered by the lookup module of the container when there is a state change in the local environment. The interpreter module of each entity contains a repository of ECA rules, which are able to catch specific events according to specified conditions. Conditions can refer to event parameters or to the global state of the entity and its attributes. The interpreter of an entity defines a state machine which represents the behavior of the owner entity according to its environment.

The lookup module interface is defined by an abstract class called `CFLookupInterface`. The `BMLookup` module implements this lookup interface to provide an IP broadcast and IP multicast lookup mechanism.

4.3 Lookup and Discovery Mechanism

The abstract class `CFLookupInterface` defines methods for the implementation of different lookup mechanisms. The lookup module is an important part inside the framework, because the delivery of events is based on it. The event queue uses the lookup module to resolve the address of the destination entity in order to be able to deliver directly addressed context events. The following lookup mechanisms can be found in the literature:

- ***Request Protocols***: An entity which tries to discover a lookup service sends out a broadcast request and waits for incoming information. When a lookup service gets a request message, it opens a reliable connection to the requesting entity and provides the discovery information (which contains the address of the container and its supported transport protocols). To transport the request messages, a non reliable transmission protocol is used. Request messages should be sent several times,

because it could happen that the messages get lost before they reach a lookup service provider.

- ***Announcement Protocols***: A lookup service provider permanently sends out service announcement messages to inform about the services it provides. Clients that enter the local environment receive these announcement messages and can decide whether they want to contact the service provider or not.

Join and Leases. Once a service provider discovered a lookup service provider, a service can be registered at the lookup service (called *join*) for a certain period of time (called *lease request*). For this period of time the service provider guarantees that the registered service is available and running. The service provider acknowledges that it offers the joined service to other client entities for a period of time that has to be equal or larger than the service lease (called *lease grant*). The use of leases and timeout periods is necessary, because digital devices, such as PDAs, could be turned off without removing the service registration from the lookup service. The lookup service provider is able to remove services after their lease expiration. An example for lease and grant messages of Jini service providers is shown in Fig. 34: the Jini service joins a lookup service and requests a lease time (*Lr*). The lookup service grants a lease time (*Lg*) which has to be greater or equal to the requested lease time (*Lr*). The periodic lease requests have to be sent before the last granted lease time expires. Because the lease request is sent over a reliable network protocol (TCP), the client is able to check if the lookup service is still running and vice versa. The responsiveness of the lookup sequence is measured with the error time (*Terr*), shown on the right-hand side of Fig. 34. The error time is calculated as $Terr = Tg + Lg - Tr$, which means the time that is left before the lease grant expires. The size of *Terr* depends on the network latency. If the responsiveness of the lookup sequence is calculated as being too small, it can happen that the service periodically expires despite of the fact that the lookup service is still on.

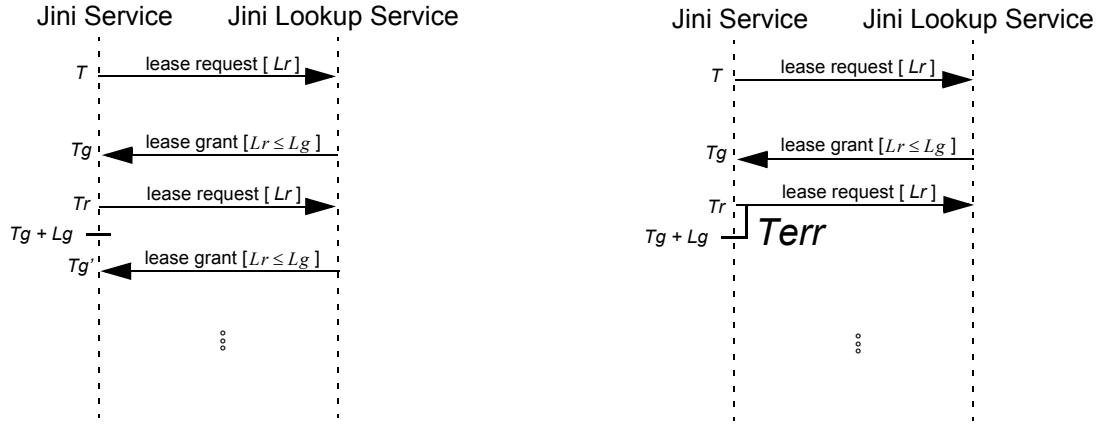

Fig. 34 Lease and grant sequences of the Jini lookup service

The Jini lookup sequence is a typical example for a service lookup process in dynamically changing environments. Discovery and service lookup mechanisms are also an essential part of the "self-healing and self-adapting networks" research area [Da02].

The SiLiCon framework's discovery and lookup interface does not force the application designer to use a specific kind of discovery and lookup mechanism. The framework's lookup

interface defines methods for the registration of local entities and for the lookup of entity addresses and attribute templates. The application designer can use the predefined `BMLookup` module or can implement new lookup and discovery modules. The BMLookup module is able to send IP broadcast packages or IP multicast packages (broadcasting to a IP multicast group). IP broadcast and multicast were combined in one module because both use the same sending mechanism but a different destination address. Other module implementations could include the Jini discovery mechanism or a JXTA discovery mechanism in order to connect to other distributed computing frameworks. Here is a list of methods that the abstract class `CFLookupInterface` defines for implementing lookup modules:

- **public abstract String getLookupIdent();** Returns an identifier specifying which kind of lookup mechanism is implemented (e.g.: IP_MULTICAST).
- **public void addSendModule(CFSendModuleInterface si);** Registers a new transport send module.
- **public void removeSendModule(CFSendModuleInterface si);** Removes a transport send module from the list of available send modules in this context container.
- **public void addReceiveModule(CFReceiveModuleInterface ri);** Registers a transport receive module.
- **public void removeReceiveModule(CFReceiveModuleInterface ri);** Removes a transport receive module from the list of available modules in this context container.
- **CFAddress getCFAddressFor(String entity, String protocolIdent);** Returns the address of a given entity with the specified protocol identifier.
- **public abstract void CFaddCentralLookupAddress(URL adr);** Allows the registration of one or more central lookup services in order to decrease scalability problems by using one or more central lookup services.
- **public abstract void CFremoveCentralLookupAddress(URL adr);** Removes a central lookup service.
- **public void addCFLookupEventListener(CFLookupEventListener l);** Registers a listener, which receives lookup events. These lookup events are delivered directly as Java event objects and are not handled by the context interpreter.
- **public void removeCFLookupEventListener(CFLookupEventListener l);** Removes a lookup event listener.

IP broadcast/multicast lookup module. The `BMLookup` module represents the default lookup module implementation for the SiLiCon framework. It is possible to switch between the IP broadcast mechanism and the IP multicast mechanism to send the lookup packages either as broadcast or as multicast packages. This switch can be set within the context container's configuration file or at runtime through a method call. The `BMLookup` module is based on a broadcast announcement protocol which uses leases. Because the SiLiCon framework is also able to operate in ad-hoc networks, no access to any global lookup service provider is guaranteed. The SiLiCon framework is designed for access of services in infrastructural networks and for direct access of services between two communication partners, which communicate over an ad-hoc connection. To support direct access of services between two or more communication partners, it is necessary that at least one of them provides a lookup service. To guarantee communication with every entity, each context container hosts its own lookup

service module. On every host there exists exactly one lookup service (inside the context container) which provides a description of the context services running on the local host.

The class `BMLookup` contains two members which are responsible for the broadcast announcement and the broadcast receiving. Class CFBroadcastNotifier sends out lookup information of the local context container and about the entities which are available locally. A context container therefore announces its appearance inside an environment with such broadcast packages. The broadcast information is XML-encoded in order to offer platform- and language-independent lookup information.

The class `CFBroadcastReceiver` receives all the broadcast packages that were sent in the local network. It parses the lookup information in the packages, which includes the entity and attribute information and registers discovered entities that appeared. The lookup information package includes a lease value which defines how long the information is valid. The `CFBroadcastReceiver` checks the list of entities for entities which timed out and removes them from the list. One of the advantages of this announcement mechanism is the fact that a container is not responsible for unregistering itself from other lookup services. If a device is powered off or disappears, all lookup services remove their reference to this device after its lease time has expired. PDAs are likely to power off or to disappear unsuspectedly. Every device is able to specify its own lease time according to its type. A server will specify much more lease time than a PDA. Fig. 35 shows the periodic lookup information announcement process of the lookup module `BMLookup_1`. The announcement sequences of module `BMLookup_2` and `BMLookup_N` are the same and are therefore not shown in Fig. 35.

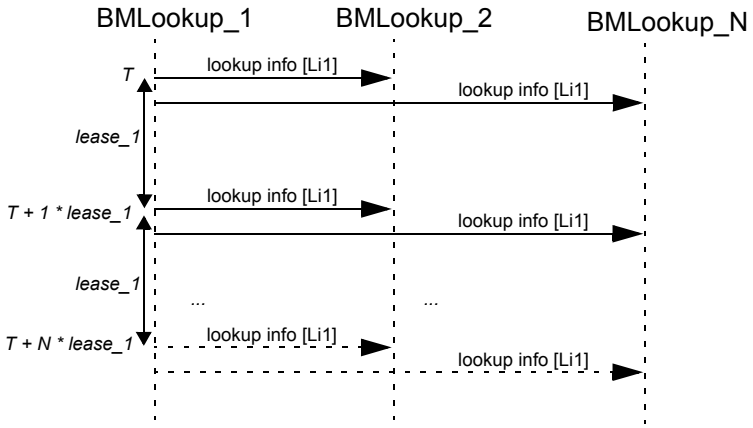

Fig. 35 Announcement sequence of `BMLookup` module number one.

Fig. 36 shows a visualization of the XML Schema definition for the lookup information announcement packages, which are sent by the `BMLookup` module. The lookup XML element contains additional information about the *lease* time, the *computational power* of the host hardware platform (which is important for entity and attribute migration), *synchronization time* and a *command string* which is set to *ANNOUNCE*.

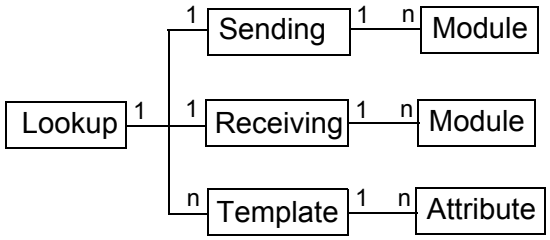

Fig. 36 XML Schema definition of a `BMLookup` announcement package

The following listing shows an example of a lookup information package from a `BMLookup` module. The lookup element contains attributes which specify the lease time, the CPU power level of the sender, a command and the name of the container. The sending and receiving elements contain information about the collection of supported transport protocols and their address. Every entity, which is contained by the announcing container, creates a template with all of its contained attributes and adds this template to the broadcast information. The name of the template is the identifier of the entity:

```
<?xml version="1.0" encoding="utf-8"?>
 <lookup  ident="Artifacts"
          power="1"
          time="..."
          lease="10000"
          cmd="ANNOUNCE"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:cfBMlookup="jku.at/context/Schematas/BMLookup">
  <sending>
   <module name="HTTP_1_1"/>
   <module name="TCP_SOCKET"/>
  </sending>
  <receiving>
   <module name="HTTP_1_1" adr="http://140.78.145.32:8079"/>
   <module name="TCP_SOCKET" adr="http://140.78.145.32:8077"/>
  </receiving>
  <template name="Artifacts">
   <attribute name="ControlRules" interface="..controlrules.ControlRules"/>
   <attribute name="HttpResourceAtr" interface="...tangible.HttpResourceAtr"/>
   <attribute name="SystemCalls" interface="...platform.SystemCalls"/>
   <attribute name="Rfid" interface="uni.linz.context.attributes.rfid.Rfid"/>
   <attribute name="DynamicLoader" interface="...stdattrs.DynamicLoader"/>
   <attribute name="BMLookup" interface="...transport.BMLookup"/>
   <attribute name="MouseDropArea" interface="...swing.MouseDropArea"/>
   <attribute name="Menu" interface="...tangible.Menu"/>
  </template>
  <template name="Thing">
   ...
  </template>
  <template name="Person">
   ...
  </template>
  <template name="Place">
   ...
  </template>
 </lookup>
```

4.4 Pluggeable Transport Modules

As already mentioned in Chapter 4.3, it is possible to implement different transport modules, in order to extend the SiLiCon framework's communication possibilities. A transport module is split into a sending and a receiving module, which can be separately implemented and registered with a lookup module. When a receiving module is installed in a lookup module, the entities which are using this lookup module (i.e. which reside in the same context container) are able to receive context events that are sent over compatible sending modules. As the interface of a lookup module (`CFLookupInterface`) shows, it is possible to register more than one sending or receiving module. That implies, that the lookup has to compare the sending modules of an event's source entity with the receiving modules of the event's destination entity, in order to find a compatible pair. A compatible pair of sending and receiving modules is necessary to deliver an event from the source to the destination entity. A transport module has to provide a unique identifier that consists of the name of the protocol which the module implements (e.g. TCP, HTTP, SMS, or SMTP).

The lookup module uses the first compatible pair of transport modules for sending events, but this mechanism can easily be extended to consider priorities. Priorities could be specified by the event source, according to specific demands that an application has. For example, in order to send large data packages it would be more efficient to use a binary transport protocol and a powerful encoding mechanism. A binary transport protocol can be used only, if the event receiver has registered a compatible receiving transport module.

Another important aspect of the transport modules is the event encoding style. Examples for encoding styles are Java Object Serialization and XML based SOAP. `CFEventEncoderInterface` and `CFEventDecoderInterface` represent interfaces for the implementation of different event encoding or decoding styles. Every transport module has to use instances of both interfaces, to encode outgoing events and to decode incoming events.

While the sending of asynchronous events requires just an encoding module and the receiving requires just a decoding module, synchronous events need both modules. To send or receive synchronous events a module has to receive and to send the response message immediately. Therefore, it is necessary to have a pair of compatible encoding and decoding modules.

The different implementations of the interfaces `CFEventEncoderInterface` and `CFEventDecoderInterface` have to provide a unique identifier which specifies the kind of encoding. In combination with the unique identifier of the transport module a lookup module has all the information it needs to find a compatible transport module. Fig. 37 shows how a lookup module manages a set of sending and receiving transport modules with their respective encoding styles.

Fig. 37 shows an example container which has two different transport protocols. The classes `CFTcpSendModule` and `CFTcpReceiveModule` implement communication modules that use plain TCP sockets to transmit the encoded event data. The TCP module in Fig. 37 uses the two encoding classes `CFXmlEventEncoder` and `CFXmlEventDecoder` to encode and to decode the event data.

The second transport protocol, which is implemented in the example, is represented by the classes `CFHttpSendModule` and `CFHttpReceiveModule`. These modules implement the

HTTP protocol, in order to transmit event data over a high level transport protocol that is often used for SOAP messages and static HTML pages. The HTTP protocol was chosen in order to test to what extent the web service technology can be used to extend the framework's transportation and integration possibilities.

In combination with the HTTP modules, SOAP encoding and decoding modules were implemented, which are also shown in Fig. 37.

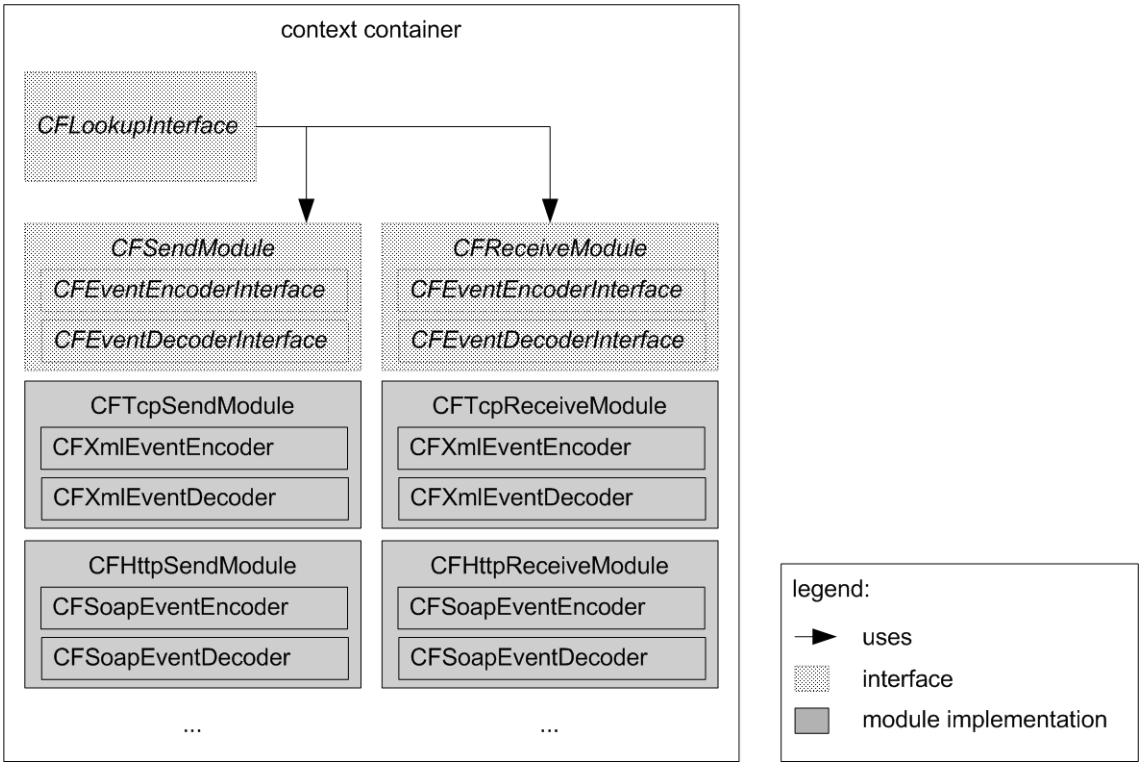

Fig. 37 Set of registered transport modules with their encoding styles

The SiLiCon architecture allows any combination of encoding modules and transport modules. Fig. 37 shows just two possible combinations.

CFTransportModule. The following abstract class `CFTransportModule` is the base class for all sending and receiving transport modules in the SiLiCon framework:

```
public abstract class CFTransportModule {
   public String getProtocolIdent(); // identifies the protocol which is used (e.g.: HTTP or TCP)
   public void setProtocolIdent(String i);
   // returns the encoding style object which is responsible for encoding the payload (CFEvents)
   public CFEventEncoderInterface getEncodingStyle();
   public void setEncodingStyle(CFEventEncoderInterface e);
   // returns the decoding style object which is responsible for decoding the return of sync events
   public CFEventDecoderInterface getDecodingStyle();
   public void setDecodingStyle(CFEventDecoderInterface e);
   public void setLookup(CFLookupInterface l);
   public CFLookupInterface getLookup();
   public abstract URL getAddress(); // returns the URL address for the transport object
   public abstract CFEvent sendSync(CFEvent ev);
   public abstract void sendAsync(CFEvent ev);
   public void addTransportEventListener(CFTransportEventListener l);
   public void removeTransportEventListener(CFTransportEventListener l);
```

```
    // configures the module at startup acording to the xml-configuration
    public abstract boolean configure(XmlTag tag);
    // method which offers the functionality to read data from a input stream
    public static byte[] readData(InputStream in) { ... }
}
```

CFEventEncoderInterface. `CFEventEncoderInterface` represents the interface for all event encoding modules in the SiLiCon framework. The *encode* method gets an event object and returns the encoded byte array:

```
public interface CFEventEncoderInterface extends CFEncoderInterface {
    public byte[] encode(CFEvent evt);
}
```

CFEventDecoderInterface. `CFEventDecoderInterface` represents the interface for the event decoding modules in the SiLiCon framework. The *decode* method gets a byte array and returns an event object:

```
public interface CFEventEncoderInterface extends CFEncoderInterface {
    CFEvent decode(byte[] b) throws IOException;
}
```

TCP protocol modules. Using raw TCP sockets to transmit event data over IP-based networks is one of the most efficient implementations of a transport module. The `CFTcpSendModule` allows one to open a TCP socket to the destination `CFTcpReceiveModule` and to transmit an array of bytes. A TCP connection is a reliable connection, which means that if the destination module is reachable and the connection is successfully established it is guaranteed that the destination gets the context event (except if the connection breaks, but in that case the sending module gets an `IOException` and is able to retransmit the event).

The `CFTcpReceiveModule` starts a thread which is responsible for listening to incoming TCP connections. The listening thread is implemented by the class `CFEventReceiver`. If a new connection is established the thread reads the data from the TCP socket and decodes it using its registered event decoder object. In the decoding was successful, the receive module puts the received context event in the local event queue.

CFXmlEventEncoder and CFXmlEventDecoder. The modules `CFXmlEventEncoder` and `CFXmlEventDecoder` are implementations of the interfaces `CFEventEncoderInterface` and `CFEventDecoderInterface`. These modules use a proprietary XML schema definition to encode and decode context events into XML messages. The schema was designed to contain the information that the class `CFEvent` offers.

The Xml schema definition for the encoding style is shown in Fig. 38. The root element is called *event*. Its subelements *destination entity* and *destination attribute* specify the destination of the event message, while the elements *source entity* and *source attribute* specify the source of a message. The *event command* element contains the name of the event and a list of parameters, which can be found in the *parameter list* element. The parameter list has a set of parameter elements which hold the name, type, and value of an event parameter.

The *synchronous* element specifies if the event message should be delivered synchronously, which means that an immediately response is expected. If the event is delivered asynchronously no immediate response is expected.

The *sequence number* element holds the message sequence number at the source entity's context container. With this number the transport module is able to guarantee the correct ordering of incoming events, even when a unreliable protocol such as UDP is used.

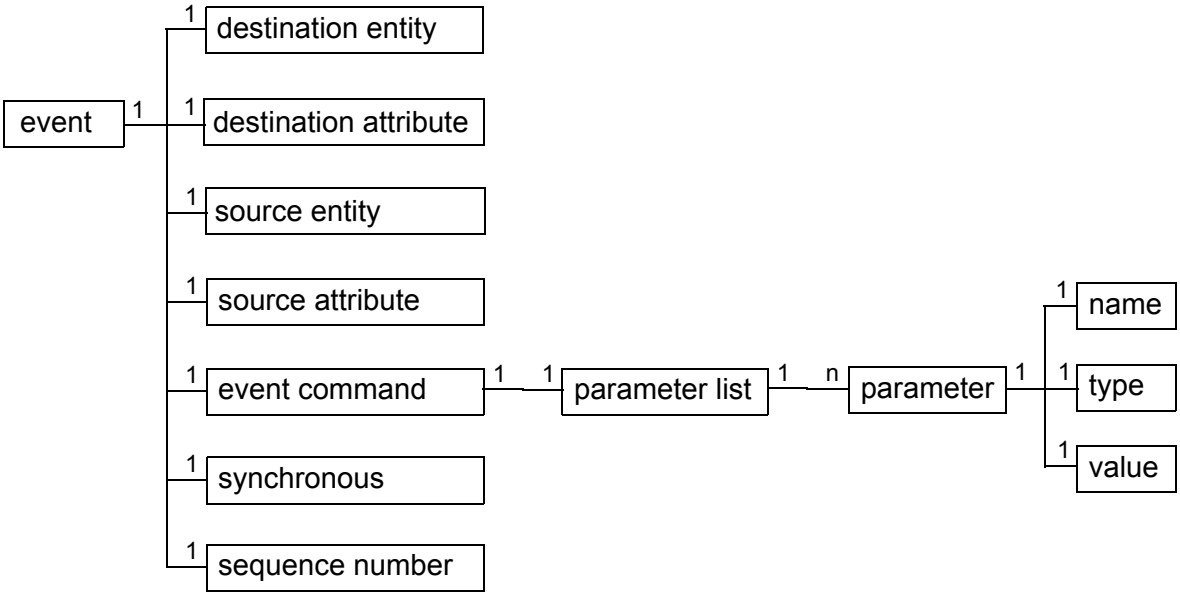

Fig. 38 Proprietary XML encoding scheme for context event objects

The following XML listing shows an example for an XML-encoded context event. The CFEvent instance is directly mapped into an XML schema representation. The XML message contains the sending and receiving entity and attribute as well as the event name (represented through the command element). The parameter list element contains the parameters that are sent with the event:

```
<?xml version='1.0' encoding='UTF-8'?>
<xmlEnc:Event tns='http://www.jku.at/silicon/XmlEncoding'
              xmlns:xmlEnc='http://www.jku.at/silicon/XmlEncoding/schema'>
  <xmlEnc:SourceEntity>
    <xmlEnc:EntityName>PDA_Loox_02</xmlEnc:EntityName>
  </xmlEnc:SourceEntity>
  <xmlEnc:DestinationEntity>
    <xmlEnc:EntityName>PDA_Loox_03</xmlEnc:EntityName>
  </xmlEnc:DestinationEntity>
  <xmlEnc:SourceAttribute>
    <xmlEnc:AttributeName>Time</xmlEnc:AttributeName>
  </xmlEnc:SourceAttribute>
  <xmlEnc:DestinationAttribute>
    <xmlEnc:AttributeName>Notification</xmlEnc:AttributeName>
  </xmlEnc:DestinationAttribute>
  <xmlEnc:SeqNr>0</xmlEnc:SeqNr>
  <xmlEnc:Command>
    <xmlEnc:CommandName>Notify</xmlEnc:CommandName>
    <xmlEnc:ParameterList>
      <xmlEnc:Parameter>
        <xmlEnc:ParameterName>Message</xmlEnc:ParameterName>
        <xmlEnc:ParameterType>string</xmlEnc:ParameterType>
        <xmlEnc:ParameterValue>Hallo</xmlEnc:ParameterValue>
      </xmlEnc:Parameter>
```

```
        <xmlEnc:Parameter>
          <xmlEnc:ParameterName>Kind</xmlEnc:ParameterName>
          <xmlEnc:ParameterType>long</xmlEnc:ParameterType>
          <xmlEnc:ParameterValue>1</xmlEnc:ParameterValue>
        </xmlEnc:Parameter>
        <xmlEnc:Parameter>
          <xmlEnc:ParameterName>BoolParam</xmlEnc:ParameterName>
          <xmlEnc:ParameterType>boolean</xmlEnc:ParameterType>
          <xmlEnc:ParameterValue>true</xmlEnc:ParameterValue>
        </xmlEnc:Parameter>
      </xmlEnc:ParameterList>
    </xmlEnc:Command>
    <xmlEnc:Synchronized>false</xmlEnc:Synchronized>
  </xmlEnc:Event>
```

HTTP protocol module . The HTTP [HTTP] protocol implementation consists of a module `CFHttpReceiveModule`, which is as simple HTTP server, and a module `CFHttpSendModule`, which is responsible for sending data over the HTTP protocol. Together with the SOAP encoding module the HTTP transport protocol offers an interface to the web service world, which is described in Chapter 4.4.1.

The HTTP transport module allows also the delivery of an automatically generated HTML representation of the context container structure. A representation like this is generated by many web service development environments to simplify the development process. With this HTML representation the scenario designer is able to navigate through the contents of a container and to get a link to the WSDL description of an attribute's interface. The integration of web services into the SiLiCon framework is discussed in the next section.

4.4.1 Integration of Web Service Mechanisms

In the last few years, the term *web services* was established by a group of international companies to push the development of distributed services. *Web service* is a term that implies the use of a specific family of standards. These standards define how distributed services have to describe their service interfaces (*WSDL = Web Services Description Language*) [WSDL], the kind of interaction (*SOAP = Simple Object Access Protocol*) [SOAP] and even how these services can be found in a global network (*UDDI = Universal Description, Discovery and Integration*) [UDDI]. All these standards are based on XML.

Motivation for the web service integration. The basic motivation for integrating web services into the SiLiCon framework was the appearance of many web service tools and services that could be used in our framework. Actually, the web service technology is not a monolithic middleware solution like CORBA [CORBA] or Java RMI, but it offers some interesting extensibility aspects, even though it is still under development. One of these aspects is the use of an XML-based standard to describe the service interfaces, called *Web Services Description Language* (WSDL). WSDL is independent of any specific platform, language or transport protocol. If a service is able to describe its interface through a WSDL description it is possible to create remote service proxies automatically. Various software companies, which implement software development environments, are working on solutions which allow the easy development, deployment, and use of web services. Even companies such as Macromedia, which develop software for the visual creation of web animations, rec-

ognize the power of web services and include them into their frameworks. The most important advantage of web services is the fact that all developers share the same description, deployment, and execution standards. Therefore, it is possible to share a multitude of different services between heterogeneous platforms and frameworks.

The integration of web services into the SiLiCon framework targets following aspects:

- **Context Service Description**: The description of context services using WSDL, which allows clients to read and process context service interfaces. For development environments such as Visual Studio .NET the integration of WSDL descriptions of context attributes means that remote service proxies can be created automatically.
- **SOAP encoding of context events**: The encoding and transport plugin architecture of the SiLiCon framework allows the registration of new modules. Therefore a SOAP encoding and decoding module could be integrated into the framework without having to change the framework's architecture. The SOAP encoding module delivers context events as SOAP calls and thus offers context services to the web service world.
- **HTTP protocol integration**: The integration of an HTTP protocol implementation into the SiLiCon transport layer allows SOAP-encoded calls as well as WSDL descriptions to be delivered over HTTP. SOAP calls can be bound to nearly every high-level communication protocol such as TCP or SMTP. They can even be bound to protocols that are offered by cell phones such as SMS or GPRS. However, most of the SOAP services are bound to HTTP.
- **Mapping of third party web services into the SiLiCon framework**: The last aspect targets the use of third party web services within the SiLiCon framework. Those web services can simulate context information sources that trigger context events. To integrate third party web services into the framework it is necessary to use a wrapper object as an attribute, that polls the web services for new values. If a new value is detected the wrapper attribute triggers a context event.

The following sections describe the implementation of these four aspects.

Context Service Description. The dynamic context service description with WSDL is mainly solved in the class `WSDLBuilder`, which does reflection on other classes, in order to generate a WSDL description of their service interfaces. All of these classes are derived from class `CFAttribute`. The following section describes the generated WSDL information:

Web Services Description Language. A WSDL description separates the abstract functionality of a service from its concrete implementation such as the transport protocols and the location where that functionality is offered. The WSDL describes a service as a set of messages that are transmitted between the service client and the service provider. These messages are not specifically bound to a concrete protocol, but are defined in a protocol-independent format (most of the time XML schema definitions are used).

A WSDL description is divided into an abstract and a concrete part. The abstract part describes the service and its messages and the concrete part binds the abstract service to concrete protocol and encoding mechanisms. WSDL describes services as a set of network end points, also called *ports*. The definition of ports and the definition of *messages* is separated

to allow the reuse of messages in different services. A *Port type* represents an abstract collection of *operations*. A *binding* maps concrete transport protocols and encodings to a particular port type.
The following list shows which kind of elements a WSDL description contains:

- **Type element**: contains data type definitions specified in a specific type system such as XSD [XSD].
- **Message element**: holds the abstract definition of the messages which are sent and received by a web service.
- **Operation element**: contains the abstract definition of an operation which the service offers to a client.
- **Port type element**: contains an abstract set of operations supported by one or more end points.
- **Binding element**: specifies a transport protocol and data format for a particular port type.
- **Port element**: defines a single end point as a combination of a binding and a network address.
- **Service element**: describes a web service as a collection of related ports.

The following sections describe these WSDL elements in more detail using a *Hello World* web service as an example. The Hello World web service was implemented in *C#* and the source looks like as follows:

```
public class ExampleService : System.Web.Services.WebService {
    [WebMethod]
    public string HelloWorld ( string name ) {
            return "Hello, " + name;
    }
}
```

Service element. This element specifies a set of ports that are grouped together by a service name that has to be unique within the WSDL document. For the Hello World example it could look as follows:

```
<service name="ExampleService">
   <port name="ExampleServiceSoap" binding="s0:ExampleServiceSoap">
      <soap:address location="http://localhost/ExampleWebService/ExampleService.asmx" />
   </port>
</service>
```

The port element *ExampleServiceSoap* contains a binding extensibility element, which adds protocol binding specific information to each WSDL element. The extensibility element `soap:address` specifies the SOAP protocol address for this port.

Extensibility elements contain additional information that are specific for certain protocols and are not defined within the WSDL schema. In the example the protocol binding refers to the SOAP protocol.

It is possible to specify more than one service element in a WSDL document as long as the names of the services are unique.

Binding element. A binding element assigns a transport protocol and a message format to a port type element. Since a WSDL description does not restrict the number and kinds of pro-

tocol bindings it can contain a variable number of binding elements. The binding element also contains a collection of operation elements, which specify operation names and their messages. The mapping between operation and message is realized with the portType element. The SOAP extensibility element specifies how the parameters of a SOAP call are encoded.

The *Hello World* service contains the following SOAP binding section:

```
<binding name="ExampleServiceSoap" type="s0:ExampleServiceSoap">
   <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
   <operation name="HelloWorld">
      <soap:operation soapAction="http://tempuri.org/HelloWorld" style="document"/>
      <input>
         <soap:body use="literal"/>
      </input>
      <output>
         <soap:body use="literal"/>
      </output>
   </operation>
</binding>
```

Port type element. The port type element describes for every operation which messages are received by the service provider and which messages are sent back in response. The port type element uses references to message element definitions, in order to specify the type of the messages. For each operation, which the service offers, the port type element has to specify the type of the *input* and *output* messages. For the *Hello World* example the port type definition looks as follows:

```
<portType name="ExampleServiceSoap">
   <operation name="HelloWorld">
      <input message="s0:HelloWorldSoapIn" />
      <output message="s0:HelloWorldSoapOut" />
   </operation>
 </portType>
```

Message element. The message elements define the types of the input or output messages. A message consists of one or more parts that describe the contents of a message according to its binding. A message part has an associated type that is specified by a so-called message typing attribute. There are two kinds of typing attributes: *type* and *element*. A *type* attribute refers to a simple or complex XSD type and an *element* attribute refers to an XSD element. The following example shows the message elements for the *HelloWorld* service:

```
<message name="HelloWorldSoapIn">
   <part name="parameters" element="s0:HelloWorld" />
</message>
<message name="HelloWorldSoapOut">
   <part name="parameters" element="s0:HelloWorldResponse" />
</message>
```

This example shows that the SOAP-bound messages have an attribute called type which specifies the type of the message. It refers to an XSD-defined element inside the types of this service. The portType element specifies which messages are sent and received by an operation.

Types element. The types element is used for defining all complex types that are used in the WSDL description. If complex types are used as service parameters, this section is the most complex part to generate and to parse. Since WSDL wants to be platform-independent, it uses XSD as the type system standard. For the *HelloWorld* example two element types have to be defined in the types element: `HelloWorld` and `HelloWorldResponse`. The message `HelloWorld` represents the input message and `HelloWorldResponse` defines the message that is returned by the service:

```
<types>
  <s:schema elementFormDefault="qualified" targetNamespace="http://tempuri.org/">

    <s:element name="HelloWorld">
      <s:complexType>
        <s:sequence>
          <s:element minOccurs="0" maxOccurs="1" name="name" type="s:string" />
        </s:sequence>
      </s:complexType>
    </s:element>

    <s:element name="HelloWorldResponse">
      <s:complexType>
        <s:sequence>
          <s:element minOccurs="0"
                     maxOccurs="1"
                     name="HelloWorldResult"
                     type="s:string" />
        </s:sequence>
      </s:complexType>
    </s:element>
  </s:schema>
</types>
```

Generation of WSDL descriptions on mobile devices. The SiLiCon framework was designed to run mainly on mobile platforms. Therefore it was necessary to use libraries that do not use a lot of system resources and are able to run on a *Java Virtual Machine* which supports only the *Personal Profile* (which is comparable to the libraries of JDK 1.1.8). Common Java web service development libraries such as *Axis* [AXIS] or *glue* [GLUE] are based on XML parsing libraries that run on the Java Personal Profile but use a lot of system resources.

Therefore we decided to implement our own library for generating WSDL descriptions from Java classes at run time. This library should be based on a fast and slim XML processing library such as kXML [kXML].

The class `WsdlBuilder` represents the main service description. The client is able to create a new `WsdlBuilder` object and to add new services to this object. The interface of the class `WsdlBuilder` is defined as follows:

```
public class WsdlBuilder {
  public WsdlBuilder(URL tns, Protocol[] ps); // sets the target namespace and a set of protocols
  public WsdlBuilder(); // initializes with a SOAP protocol binding and a default namespace
  public void addProtocol(Protocol p); // adds a protocol binding
  public void insertNs(String nic, URL nsu); // inserts a namespace with a dedicated short name
  public void addService(String sn, URL sadr, Class c, Method[] mds); // adds a new service
  public XmlNode getWsdl(); // returns the WSDL description as an XmlNode object
  public String toString(); // returns the WSDL description as a String object
```

```
}
```

The `WsdlBuilder` constructor initializes the target namespace and a set of protocol bindings. The empty constructor initializes the target namespace with a default value and with a SOAP protocol binding. The interface `Protocol` defines methods for accessing the protocol information that has to be written into extensibility elements of this protocol. A client is able to inherit from the interface `Protocol` in order to create a new protocol binding. A new protocol binding is registered at a `WsdlBuilder` object by calling the `addProtocol` method. After the registration, the `WsdlBuilder` generates additional binding extensibility elements (in addition to <soap:...> e.g. <smtp:...>, <http:...>) for the new protocol.

To add a new service to the `WsdlBuilder` object it is necessary to call the `addService` method, which takes parameters that contain the service name, the address, the service class, and an array of methods. The array of methods specifies which methods should be described as publicly available services in the WSDL description.

The SiLiCon framework automatically exports all public methods whose names begin with the string "`CF`" (e.g. `CFshowMessage`). The delivery of WSDL descriptions is implemented as an HTTP-GET response mechanism from the HTTP transport protocol module. The worker process of the HTTP module is responsible for handling client requests and to send back the result values over HTTP. According to some special features of the web service technology, it was decided to enlarge the standard functionality of the HTTP module to fulfil the following tasks:

- Receive and deliver context events according to the encoding module that is specified (which is a standard functionality of all transport modules). Context event data can be received through the HTTP-POST or HTTP-GET method.
- Receive and deliver resource requests over HTTP-GET, which means that files (e.g. images, XML or HTML) can be sent to a client. The client could be a web browser that wants to receive documentation or pictures about the context entities in human-readable form.
- Automatic generation of an HTML representation showing the structure of a context container. The worker process is able to deliver an HTML representation of an entity with its set of attributes.
- Automatic generation of a WSDL description which represents the public services of a context attribute.

In the next sections these tasks are described by some concrete examples.

Delivery of resources over HTTP. If an HTTP request refers to a resource that can be found in a directory of the context application and does not refer to an entity or an attribute path, the HTTP worker process returns the resource to the client. This feature is used by the generated HTML description, in order to display images. The resource delivery is also useful for offering information and documentation about the applications running on the device and about the different context services that are offered. HTML pages offer the client the possibility to refer to information which is written in a human-readable form, in contrast to a WSDL description which is intended for being processed by a machine. For security reasons this feature can be disabled or restricted to specific resources for single HTTP modules. Fig. 39 shows a web browser client which requests the resource `about.html` from the registered

HTTP receive module. The receive module accesses the context application directory, where all startup configuration scripts are located and tries to find the resource `about.html`. As we can see, the resource `about.html` is located in the specified application path and the HTTP receive module delivers the resource to the client. The client displays the received resource in its browser window.

Automatic generation of an HTML representation. When the client requests an HTTP resource path that leads directly to a hosted entity or attribute, the HTTP receive module shows detailed information about the requested object. The HTTP receive module was extended to automatically generate an HTML representation of entities and attributes. The HTML representation includes a link to the WSDL description of the attribute and links to navigate to the parent entity and to child attributes and entities. This navigation functionality is useful for scenario designers to get an overview of the context container's contents and structure as well as a list of available services. To demonstrate the navigation through a context container using the HTML representation, the sample SiLiCon application 'VRML-Browser' is used, which is described in Chapter 6.1.

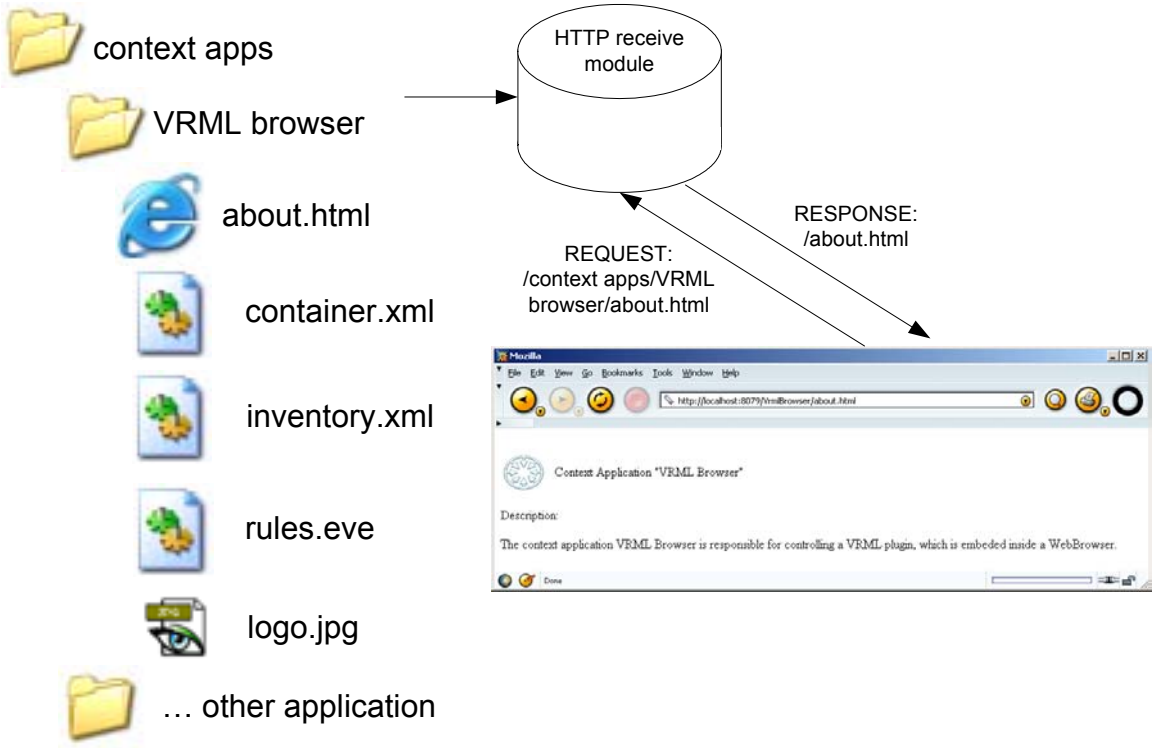

Fig. 39 Web browser client requests resource *about.html*

When the client requests the resource "http://*address*:*port*/VRMLBrowser", the HTTP receive module realizes that this path refers to an entity and generates an HTML representation that is shown on the left in Fig. 40. The entity `VRMLBrower` has a list of attributes on which the scenario designer may click in order to navigate to the WSDL information of the selected attribute. The HTTP receive module configuration needs a specific extensibility element, called *RootEntity*, which holds the identifier of the entity that represents the root object for the navigation tree.

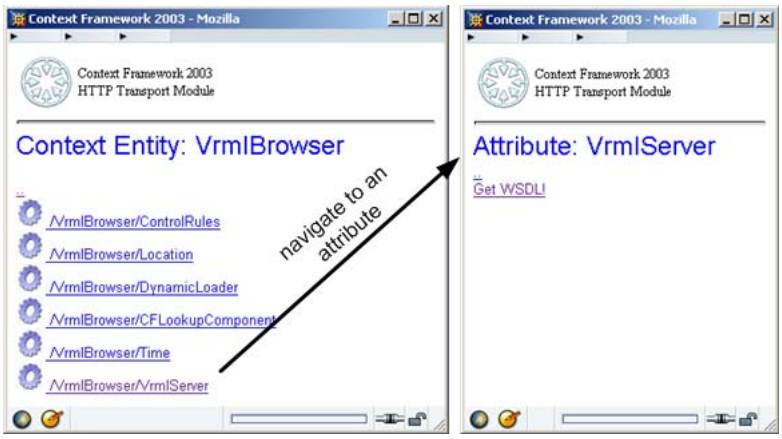

Fig. 40 Generated HTML representation of entity `VRMLBrowser`

Automatic generation of WSDL descriptions. Many web servers offer the feature to return the WSDL description of a web service over HTTP. To request such a description, one has to specify the path of the web service and an empty *WSDL* parameter, for example:

"http://www.sampleurl.com/WebService.asmx**?WSDL**"

In the SiLiCon framework it is also possible to refer to the WSDL description of a context attributes with the attribute's URL. In order to specify that the HTTP receive module should generate a WSDL description instead of showing the HTML representation of the attribute, it is necessary to add the empty *WSDL* parameter. The HTTP receive module calls the method `generateWSDLfor`, which uses the class `WsdlBuilder` to translate all public methods that begin with the string `'CF'` into WSDL. Here is the implementation of `generateWSDLfor`:

```
protected String generateWSDLfor(CFAttribute a, String path) {
   String adrStr = CFHttpReceiveModule.address+path.substring(0, path.indexOf("?"));
   org.kwsdl.generate.WsdlBuilder wb = new org.kwsdl.generate.WsdlBuilder();
   String wsdl = "";
   // only describe methods that start with the "CF" context event endpoint mark
   Method[] mds = a.getClass().getMethods();
   Vector v = new Vector();
   for(int i=0;i<mds.length;i++)
      if(mds[i].getName().startsWith("CF")) v.addElement(mds[i]);
   Method[] tm = new Method[v.size()];
   for(int i=0;i<v.size();i++)
      tm[i] = (Method)v.elementAt(i);
   try {
      wb.addService(a.getIdentifier(), new URL(adrStr), a.getClass(), tm);
      wsdl = wb.toString();
   } catch(MalformedURLException e) {
      e.printStackTrace();
   }
   return wsdl;
}
```

The method `generateWSDLfor` takes the attribute that should be described as well as its resource path, in order to select all methods of the attribute's class that begin with `'CF'`. After the vector of methods has been copied into an array, the `WsdlBuilder`'s method `addService`

is called to add the methods to the description. Finally, the method `toString()` is called, which returns the WSDL description of the attribute as a string.

As an example, let us look at how `WsdlBuilder` generates a WSDL description for an attribute of class `Time`. The `Time` attribute offers two service methods, which trigger events depending on the local time. The method `CFperiodicTrigger` takes a delay value and triggers events periodically. The method `CFtimeTrigger` takes an hour, minute, and second value and triggers an event at the given time.

```
public class Time extends CFAttribute implements Runnable {
   public void CFperiodicTrigger(long delay)
   public void CFtimeTrigger(long hour, long min, long sec)
   ...
}
```

Both methods begin with the string `'CF'` and are declared as public. Therefore only these two methods are extracted by the `generateWSDLfor` method. To request the automatic description of the attribute `Time` in the entity `VRMLBrowser` one has to specify the following ULR:

```
http://www.sampleurl.com:port/VrmlBrowser/Time?WSDL
```

The generated WSDL description is visualized in Fig. 41. The interface of class Time offers two public services periodicTrigger and timeTrigger. These services are descibed via WSDL. The WSDLBuilder automatically generates a service namespace s0 in which the services elements are defined. According to the fact that a service method inside a class can be overloaded, the WSDLBuilder has to coose a unique name for each method. That is the reason why the WSDLBuilder adds the class name and a unique number in front of the operations input and output names. In the example the resulting name is 0TimeCFperiodicTrigger:

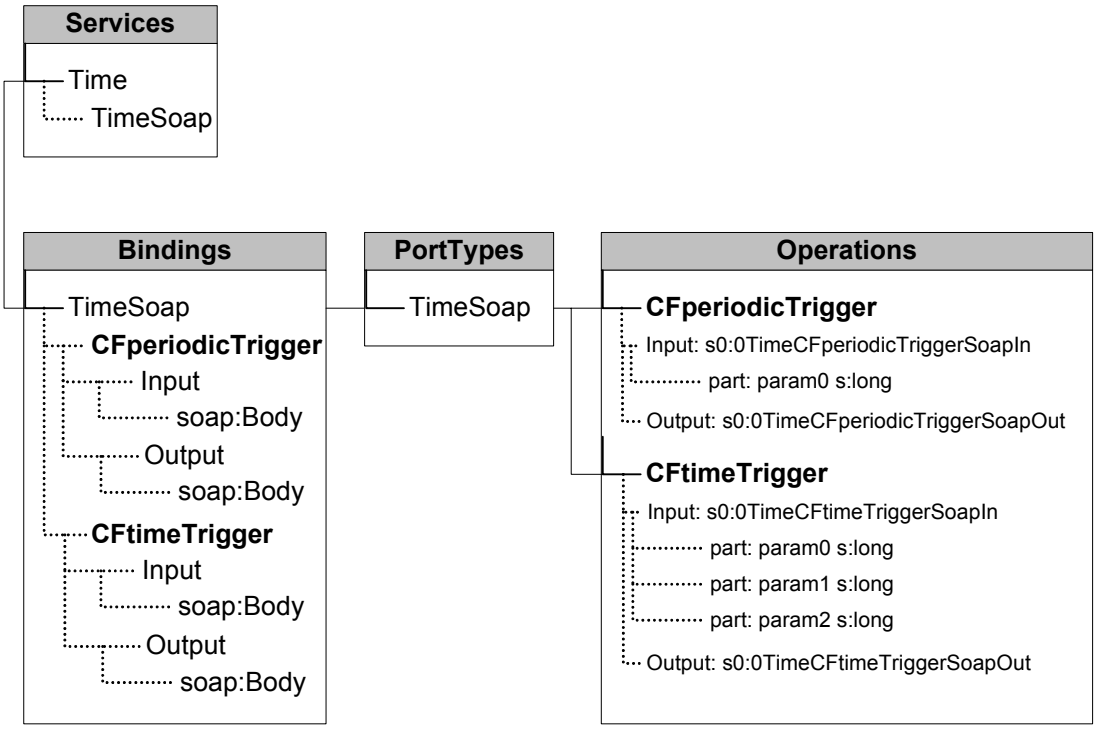

Fig. 41 Visualized WSDL description of the attribute *Time*

4.5 Role-Based Classification with Attribute Templates

The classification of entities that suddenly appear in an environment is a major aspect of context frameworks that are operating in highly dynamic networks. To identify an appearing entity it is necessary to use a discovery mechanism, which was already described in Chapter 4.3. There are different ways how appearing entities can be identified as possible interaction partners. A popular mechanism is the *Person*, *Thing* and *Place* classification model. This kind of classification was first mentioned in the Cooltown project (see Chapter 3.4). A static classification hierarchy, which has to be the same on all distributed devices, distinguishes only between the three basic classes: *Person*, *Thing* and *Place*. The reason for this classification schema was that it covers many possible real world situations and that it is simple. Context information is traditionally bound to location information so this was the main reason to distinguish between places and objects that are located in places. Humans are the users of pervasive and context-aware scenarios. Therefore Cooltown distinguishes also between objects and human users in form of the classes *Thing* and *Person*. For the sample scenarios in Cooltown this classification schema is sufficient and easy to implement.

Other classification mechanisms, such as the one in Jini, use Java interface types to identify possible communication partners. The disadvantage of this technique is that only communication partners can be considered, which are implemented in Java. For example, it is not possible to implement a Jini service in Pascal, C, or C++. The Jini service provider must run a Java Virtual Machine. This restriction reduces the number of possible communication partners.

Role-based classification mechanism. The main problem in the classification of appearing entities is to find out which attributes an object has. Describing objects and their relations to each other is one of the major aspects of the RDF [RDF] standard. The RDF group and the Semantic Web [seWeb] group was founded by the World Wide Web Consortium in cooperation with mobile phone manufacturers like Nokia. The RDF standard was established to describe the attributes of mobile phones in order to give providers of mobile services the possibility to customize their content automatically. The service provider checks the attributes of a device and sends only such content that the device is able to understand. It does not make sense to send color photos to mobile devices which can only display monochrome text. Our research-like the RDF approach- shows that a universal and simple entity classification mechanism improves the flexibility of mobile services and interaction scenarios. In the SiLiCon project we distinguish between two object classification models:

1. **Closed World Assumption**: In this classification model the whole class hierarchy is defined statically at design time. That means that all distributed devices have to use the same static class hierarchy. Of course, this can lead to consistency problems. A change in the class hierarchy affects all mobile devices and the applications running on them. If the class hierarchy is simple, as it was the case in the Cooltown project, it is an effective and efficient way to classify objects. However, most programming languages do not support multiple inheritance, which means that an object cannot be classified both as a Person and a Place. Fig. 42 shows how the classification mechanism in a closed world assump-

tion would classify an object according to three classes *Person*, *Thing* and *Place*. An object is classified by choosing a class where the object fits best.

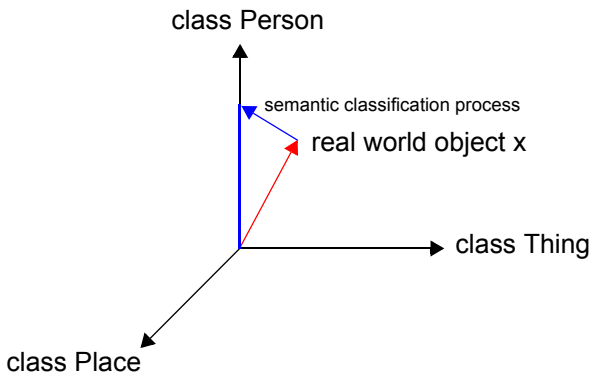

Fig. 42 Semantic classification process in a closed world model

2. **Open World Assumption**: This classification model is completely open for changes in the number of object types (roles). Data and services are not completely known at design time. All the semantic information of the participants in a scenario have to be derived at runtime. As a matter of fact, the classification of an object can even change at runtime. An object is also able to act in more than one role. Fig. 43 shows how a role-based classification according to three attributes could be performed.

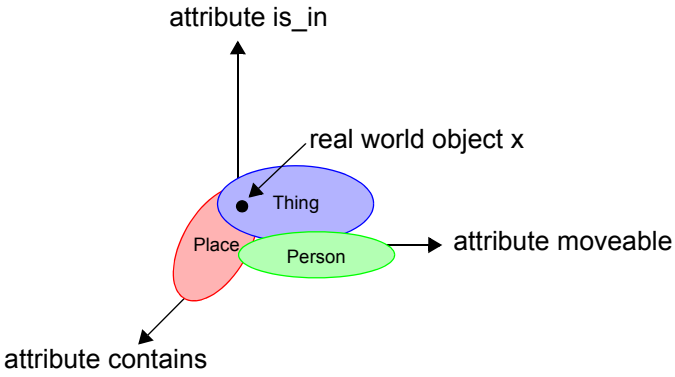

Fig. 43 Role-based classification in an open world model

Instead of classifying objects with a static class hierarchy at design time (as in Cooltown) the SiLiCon framework tries to apply another classification mechanism: all entities are containers that can hold sets of attributes. The framework does not classify the entities by their types but allows context-sensitive applications to identify entities according to the role that they play. Roles are defined by the applications. The same entity could act in more than one role for different applications. Is is possible to model scenarios in which an object acts as a thing for one application and as a place for another application. To classify an entity means to check which attributes the entity implements. It is clear that a classification process has to rely on characteristics that are unique throughout the whole scenario. In the SiLiCon framework these characteristics are not represented by the classes of the entities, as in Cooltown, but by the service interface of their attributes. As it is already known from the web service technology, a unique service namespace identifies the interface of an attribute. It is possible

to exchange attributes at runtime, if they implement the same unique service namespace and therefore the same interface.
The following section shows an example attribute template. A template has a name that has to be unique within the context container. The name is used to refer to the role within a ECA rule. An attribute like Name has an interface XML attribute which refers to a language and platform independent interface description, e.g. a link to a WSDL description.

```
<?xml version="1.0" tns="http://silicon.org/template"?>
<template name="Place">
    <attribute name="Location" interface="silicon...Location"/>
    <attribute name="Owner" interface="silicon...Owner"/>
    <attribute name="Display" interface="silicon...Display"/>
</template>
<template name="Person">
  <attribute name="Name" interface="silicon...Name" />
  <attribute name="Age" interface="silicon...Age" />
  <attribute name="Socialnumber" interface="silicon...SocialNumber" />
</template>
```

Unique service namespaces are used in the SiLiCon framework to identify specific sets of attributes. These sets define in which role the entity acts at the moment of discovery. As it was already mentioned in Chapter 4.3, the lookup module sends out role information, also called *attribute templates*. In the SiLiCon framework the term '*template*' or '*attribute template*' is defined as follows:

> *An attribute template specifies a set of service namespaces which an entity at least has to offer, in order to act in a specific role.*

The template manager module is responsible for loading a collection of attribute templates at startup, but templates can also be registered at runtime. The template manager represents a repository in which all locally known roles are registered. When a new context-sensitive application is installed on a device it registers a set of roles. This set indirectly defines potential interaction partners, which the application would like to include in its scenario.

The class `CFTemplate` implements an attribute template (i.e. a role) and offers basic functionality for working with the template:

- **`CFTemplate(String ident)`**: Constructor of class `CFTemplate` which initializes the template object with a given unique identifier.
- **`String getName()`**: Returns the unique identifier of the template object **(91)**.
- **`setAddress(URL adr)`**: Sets the URL address where the template object is located.
- **`URL getAddress()`**: Returns the URL address where the template object is located.
- **`void addAttribute(String nic, String interface)`**: Adds an attribute to the collection of attributes that define the role.
- **`String[] getAttributeInterfaces()`**: Returns an array of unique identifiers that identify the interfaces of the set of attributes.
- **`boolean contains(CFTemplate t)`**: Checks whether a given template is a subset of this template.

The method `boolean contains(CFTemplate t)` checks whether the given template is a subset of the receiver template. The class `CFEntity` offers a method called `CFTemplate getContextTemplate()`, which returns the set of attributes that the entity offers. This set is wrapped in a `CFTemplate` object, in order to be able to perform set operations with other template objects. Set operations on templates can provide interesting results that can be used in role modeling:

- ***Aggregation*** ($A \cup B$): The aggregation operation combines the attribute sets of two templates. This resulting template acts in the role defined by template A as well as in the role defined by template B. With the aggregation operation it is possible to design multiple inheritance scenarios, for example, where an entity has to act in the role of a *Thing* as well as in the role of a *Place*.
- **Intersection** ($A \cap B$): The intersection operation returns all attributes that are members of template A and of template B. Therefore, the intersection calculates how similar two templates or entities are with respect to their attribute sets.
- **Difference** ($A - B$): The difference operation returns all attributes that belong to template A but do not belong to template B. $B - A = \varnothing$ means that the operation `A.contains(B)` would return `true`. The template A contains all attributes that the template B defines and therefore the template A acts in the role of B.

Fig. 44 shows how an entity *X* is classified in our role-based model. The result of applying the `contains` operation to the templates shows that the entity is acting in the role of a *Projector* and a *Place* but it does not act in the role of a *Person*.

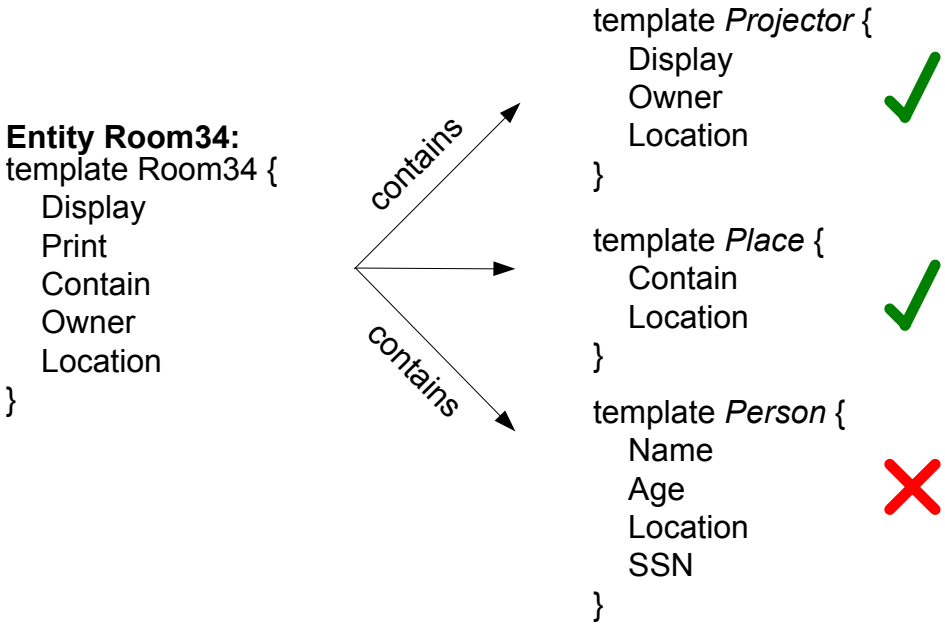

Fig. 44 Contains operations between attribute templates

4.6 Interaction Scenarios Defined by ECA Rules

One of the requirements for the SiLiCon framework was to react on state changes in a flexible way. By specifying how devices should react on environmental events one gets a powerful way to control the behavior of entire scenarios. The idea was to use an abstract layer between the sensor and the actuator modules, which are wrapped in SiLiCon attributes. The scenario designer or even a user is able to react on state transitions that lead to a demanded behavior in the environment. The scenario designer first has to identify all entities and attributes that are relevant for a certain scenario. The digital description of entities and attributes are then running on digital devices. At startup time every entity initializes its attributes, whose initial values represent the initial state of the scenario.

The scenario designer is now able to specify a set of state transitions, in order to define the state machine of a single entity. Fig. 45 shows a simple state machine for an entity which could be running on a PDA. The entity contains attributes that collect information about the weather, the stock values, the owner's time schedule and so on. Some of the attributes offer services to change the state of the environment, for example by sending an E-mail or SMS (Short Message Service on cellular phones) or by attracting the user's attention with popups on a display. Every entity has a repository of ECA rules which have the form:

ON *event* **IF** *condition action;*

If the event occurs and the condition yields true the specified action is performed. An action can read the attributes of other entities and can call service methods on them. It can also trigger new events on which this entity or other entities can react. The rule repository of every entity can be modified at runtime. New rules can be added and existing rules can be deleted or exchanged. Therefore the scenario designer can model the behavior of the entities in a very flexible and dynamic way.

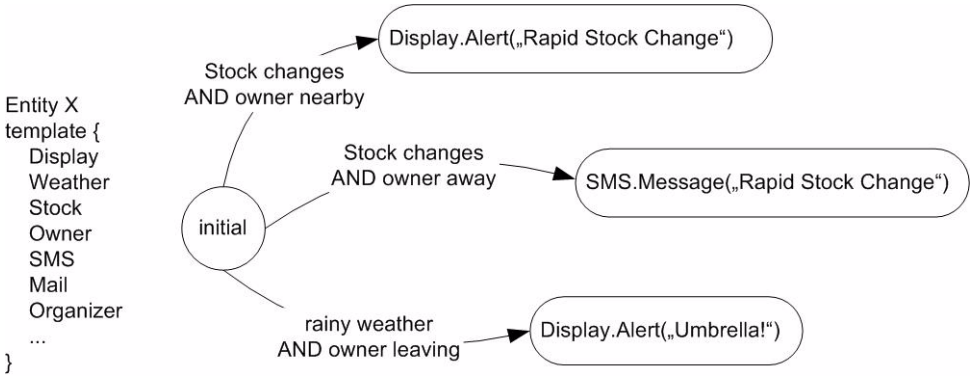

Fig. 45 Simple entity state machine.

In the SiLiCon framework an ECA rule is also called a *context rule*, because the rule is able to react on situational changes. As a matter of fact, events are also called *context events* because they inform about changes in the environment. In this work an application that is able to react on context events is called a context-aware application.

4.6.1 Event Handling

In this section we describe context events and and how they are handled in the SiLiCon framework. We will look at the contents of a context event, who triggers it, how it is transparently sent over the network as well as how the event queue and the interpreter process events.

Content of context events. A context event carries information about state changes in the SiLiCon framework. This information is transferred between distributed attributes and entities and consists of the following parts:

- ***Event name***: It specifies which event was triggered by an attribute. Therefore, the event name has to be unique within a specific type of attribute. Two different attributes can use the same event name with a different semantics.
- ***Event parameters***: The parameters are a list of identifier/value pairs that specify additional event information. Their order is important because they are used to map an event to a certain rule or to a method (if no rule is present).
- ***Source entity***: It specifies the event source object (represented by an object of type `CFEntity`) where the event was originally triggered.
- ***Source attribute***: It specifies which attribute in the source entity triggered the event.
- ***Destination entity***: It specifies to which entity the event should be delivered.
- ***Destination attribute***: It specifies to which attribute in the destination entity the event should be delivered.
- ***IsSync***: This property specifies whether the context event should be delivered synchronously or asynchronously.

The destination entity and the destination attribute are optional due to the fact that it is possible to trigger events that are not directly addressed. By specifying a destination entity and a destination attribute an event is directly sent to a specific event listener. Context events are represented by the class `CFEvent` in the SiLiCon framework.

Context event sources . There are various ways how events can be triggered and how they can be routed to a receiver. One possibility is that an event is triggered by an attribute when the state of this attribute changes. These events are not addressed, that means that the attribute does not fill in the event destination (i.e. the destination entity and the destination attribute). Such events are delivered to the entity that owns the triggering attribute informing it about a state change in its attributes. This entity can then send the event to another destination by firing a context rule. The action in the context rule can find out which entities are interested in the event by finding entities that match a certain attribute template and can delegate the event to them.

On the other hand, the scenario designer is able to directly address events to a destination entity or to a group of destination entities. In this case the destination attribute and destination entity information is filled into the event object and the event is added to the event queue of

the sending entity. The queue is then responsible for delivering the event to the specified destination by using the lookup and transport module. If the destination entity is not known or currently not reachable, the event queue triggers an error event that is processed like any other event. This is descibed in Chapter 4.6.3. It is also possible for attributes to trigger directly adressed events, but this option is quite inflexible and therefore not often used by attributes.

Processing of context events . The framework itself is also able to trigger context events containing information that might be interesting for entities. Generally, the implementation distinguishes between two event sources, which are completely transparent to the context scenario designer:

1. ***Local event sources***: A event source is considered as local when the entity, which triggered the event, is managed by the same container as the destination entity. Therefore, the framework is able to just hand over the event to the entity.
2. ***Remote event sources***: A event source is considered as remote when the entity, which triggered the event, is not managed by the same container as the destination entity. The transport layer receives the event from the network and passes it to the event queue which is registered at the context container.

For the scenario designer delivery of events to remote receivers happens transparently. For a context rule the receiver of an event is transparent. Events are delivered by the event queue of which there is exactly one in every context container. The event queue has a reference to the lookup module on the same host, which allows it to distinguish between local event receivers and remote event receivers. The delivery of events happens in the following steps:

- An attribute of an entity triggers an event and puts it into the entity's event queue.
- The event queue decides whether the receivers of the event are located on the local host or on a remote host.
- If the receiver is on the same host as the sender the event queue passes the event directly to the rule interpreter of the destination entity.
- For remote receivers the event queue passes the event to a suitable transport module, which is the one that has a compatible receiving transport module at the remote host.
- The transport module encodes the event using an encoding mechanism for which there has to be a compatible decoding mechanism at the receiving container. After the event has been encoded, it is sent to the remote container.
- The transport module of the remote container receives the event, decodes it and puts it into its own event queue.
- The event queue of the remote container delivers the received event to the destination entity.

The event queue passes events to the interpreter of the destination entity which is registered as an event listener. If there are one or more context rules for this event, the interpreter executes the rule(s) with the matching event parameters. If there is no context rule for this event,

the interpreter tries to find public service method, where it can pass the event to the destination attribute. The name and the parameter list of the event specify the name and parameter list of the attribute's method. To distinguish between normal methods and methods which are offered as an event end point, it was decided to use *'CF'* as a prefix for the method name. The prefix is not necessary when the event is triggered. When an event is triggered inside an event rule the event name without the 'CF' prefix is used. Every service method which begins with *CF* can receive events. Another possibility to deliver events is to synchronously send them. The difference between synchronous and asynchronous method calls is that a asynchronous call returns the result through triggering a new event. In order to send a synchronous event and to immediately receive the result, it is possible to pass the event to the event queue by calling the method `CFEvent deliverEventSync(CFEvent ev)`. Fig. 46 shows an event queue that delivers an event to an entity. The event is called *TagAppeared* (this is an event which an RFID reader attribute triggers when an RFID transponder comes into sensing range). The event has one parameter of type `String` which identifies the ID of the RFID transponder that appeared. The event is passed to the interpreter of the destination entity by calling its method `CFEvent[] processEvent(CFEvent ev)`. This method tries to process the event and returns a vector of outgoing events that were triggered by processing the incoming event. The outgoing events are also put into the senders event queue and processed as normal events.

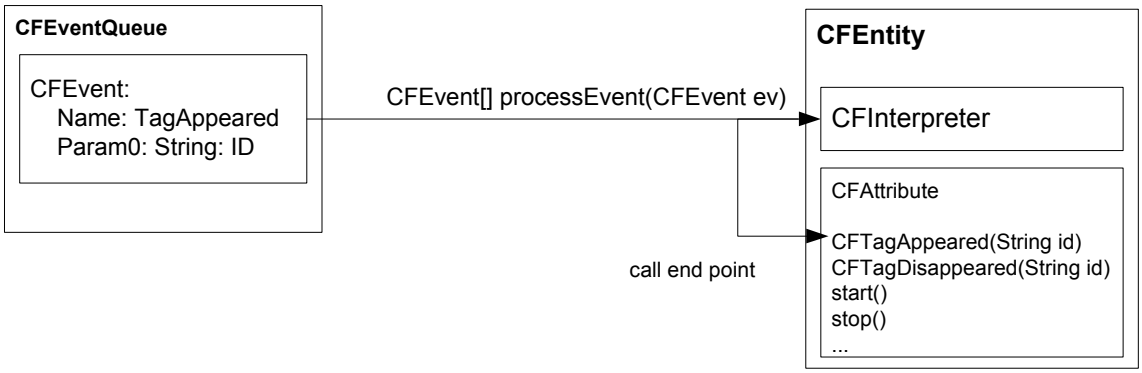

Fig. 46 Processing of events delivered by the event queue

As Fig. 47 shows, the interpreter searches its rule repository for a rule with a matching signature that can handle the incoming event. If such a rule is found, it is processed by the interpreter. Otherwise, the interpreter tries to find a direct end point within the destination attribute. If a direct end point is found (e.g. the method `CFTagAppeared` in Fig. 47), the interpreter calls this method and passes it the parameters of the context event.

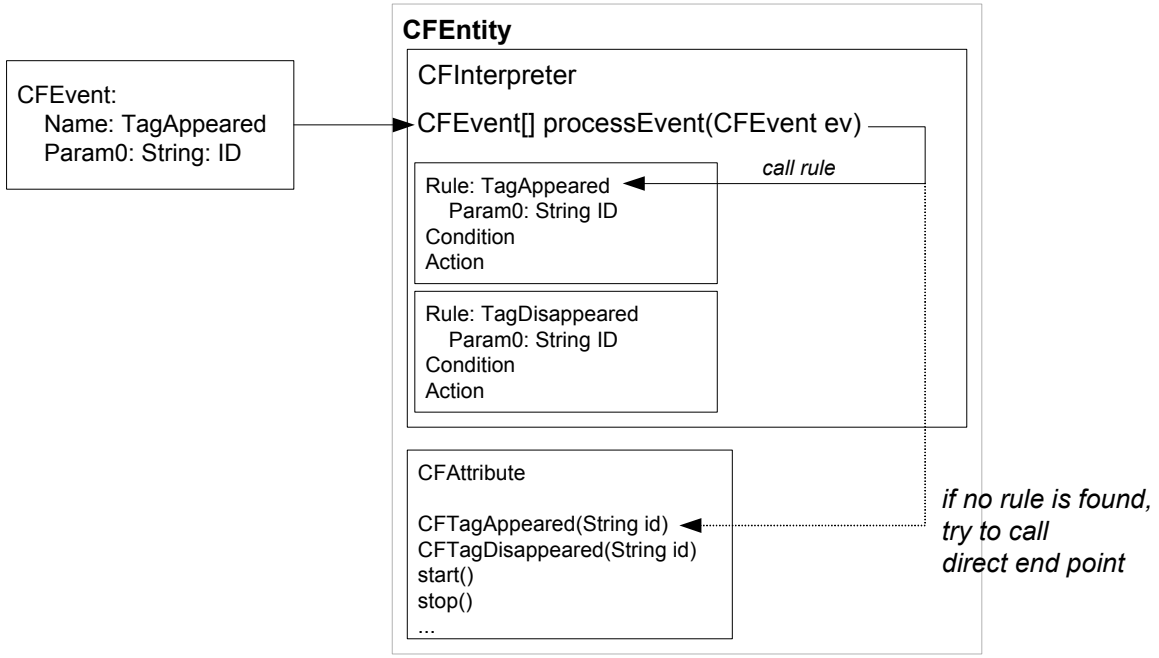

Fig. 47 Interpreter processes incoming context event

4.6.2 Syntax and Semantics of SiLiCon Context Rules

In similar projects ECA rules are often represented in XML. In the SiLiCon framework, however, we decided to define our own rule syntax, in order to create a more compact and readable format for context rules. XML has the disadvantage that it is rather verbous and contains redundant information. Because SiLiCon rules have a compact format, it is possible to send them over the network within small packages. For example, this is important in scenarios where rules are designed and sent from a cellular phone as an SMS (Short Message Service) message.

A typical SiLiCon context rule consists of three blocks which are: *an event-catching declaration block*, *constraints* and *an action block*.

In this section the syntax and semantic of the different parts of context rules are explained in detail. The simplified syntax of the different parts is specified in EBNF [EBNF].

Rule repository. Following grammar defines the syntax of a SiLiCon rule repository. A SiLiCon interpreter is able to parse a rule repository in order to load new rules.

```
RuleRepository = {RuleGroup}.
RuleGroup = "rules" [ForClause] "{" {Rule} "}".
ForClause = "for" Entity {"," Entity}.
Entity = ident | RoleTemplate.
Rule = EventHandler | VarDecl.
RoleTemplate = “<“ ident “>”.
VarDecl = Type ident { “,” ident } “;”
Type = “long” | “double” | “boolean” | “string” “;”.
```

A rule repository contains a variable number of rule groups. A rule group is introduced by the symbol ’rules’ and an optional `ForClause`, which should not be confused with the *for* loop in programming languages. The `ForClause` within a rule group specifies to which entities the grouped rules and global variables should apply to. If no `ForClause` is given the

interpreter identifies all rules and global variables inside the group as relevant for its entity. If a `ForClause` is given, the interpreter starts to identify the list of entities for which the rule group is relevant. An entity is represented through an `ident` or through a `RoleTemplate` which specifies a family of entities. If the entity's name is equals one of the idents in the `ForClause`, or the entity is acting in one of the given roles, the interpreter loads all the rules from the rule group.
In the case that the `ForClause` does not match the entity, the interpreter ignores the rule group and continues to parse the next RuleGroup.
Following example shows a rule repository which is relevant for the entity *Loox* and *iPAQ* only, according to the specified `ForClause`:

```
rules for Loox, iPAQ { ... }
```

The next example shows a rule group which is relevant for entities that act in the role of a *PDA* and for the entity *AcerLaptop* only:

```
rules for AcerLaptop, <PDA> { ... }
```

Inside a rule group it is possible to specify event rule declarations, which are explained in the next section, and global variable declarations. Global variables, specified through the non-terminal symbol `VarDecl`, contain values that are accessible through all rules inside an interpreters rule repository. Following example shows how global variables can be declared in a rule group.

```
rules for Loox {
    long hg, mg, sg;
}
```

Event rules. Event rules are declared inside a RuleGroup. An event rule declaration defines on which event the interpreter should react. Following grammar shows the syntax of an event rule declaration:

```
Rule = "on" AttrName "." EventName "(" [FormalParams] ")" Statement.
AttrName = ident.
EventName = ident.
FormalParams = FormalPar {"," FormalPar}.
FormalPar = Type ident | Expr.
```

An event rule declaration always starts with the symbol 'on' followed by `AttrName` which represents the name of the event receiver attribute and the name of the event: `EventName`. An event rule declaration contains an optional list of formal parameter declarations: `FormalParams`. A formal parameter declaration, which is represented through the non-terminal symbol `FormalPar`, is a variable declaration or an expression. Formal variable declarations are used to access the parameters the incoming event contains. The interpreter activates the rule if the receiver attribute name matches the `AttrName`, the event name matches the `EventName` and the list of event parameter types matches the formal variable types. Otherwise the rule declaration is not activated and the interpreter checks the next rule in its repository. It is important that if more than one event declaration match the incoming event, the interpreter activates all of them.

Here is an example for an event rule declaration with a list of formal variable declarations. The rule is activated if an incoming `Alarm` event has a receiver attribute `Time` and a list of three parameters of type `long`:

```
rules {
    on Time.Alarm(long h, long m, long s);
}
```

As it was already mentioned above, an expression or a constant can replace a formal variable declaration in a rule declaration. An expression or a constant represents a shortcut for a rule contraint. The interpreter takes the constant or evaluates the expression and compares the result with the parameter value of the incoming event. If the values are equal, the interpreter activates the rule otherwise it ignores the rule.
Following example shows an event rule definition which uses a constant and an expression in order to shortcut rule constraints. The interpreter activates the rule only if the first event parameter has the value 24 and the second has the value 6:

```
rules {
    on Time.Alarm(24, 1 + 2 + 3, long s);
}
```

Statements. This section explains which statements can be used within an event declaration block. Following grammar shows the syntax of a statement and statement sequences:

```
Statement = (   ident “=” Expr “;”
                | “if” “(“ Expr “)” Statement [ “else” Statement ]
                | EventOrCall “;”
                | StatSeq
                ).

StatSeq = “{“ { Statement | VarDecl } “}”.
```

A statement can be an assignment, an if statement, an event trigger operation, a method call or a sequence of such statements. The event trigger and method call operation, which is represented by the non-terminal symbol EventOrCall, is explained in a later section.

Assignments are possible to global variables and to local variables, which were defined inside the rule definition. Local variables are used to calculate intermediate result and to pass these values to method calls or to event trigger operations.

If statements use expressions to realize rule constraints. Expressions which are used as constraints in an if statement have to return values of type boolean. Otherwise a syntax error is thrown. Expressions are explained in the next section.

Sequences of statements, represented through the non-terminal symbol StatSeq, contain local variable declarations and statements. Local variable declarations are syntactically equal to global variable declarations.

Expressions. Following grammar shows the syntax of expressions inside event rules:

```
Expr = Term { (“+” | “-” | “||”) Term}.
Term = Factor { (“*” | “/” | “&&“) Factor}.
Factor = [“+” | “-” | “!”] Primary.
Primary = ident | constant | “(“ Expr “)” | EventOrCall.
```

The types of each operator depends on the different operations that are possible. The operations +, -, * and / are possible for operands of all available primitive types. The unary oper-

ation - is valid for the types `long` and `double`. The unary operation ! and the logic operations && and || are only valid for boolean types.

Triggering events and calling methods. The non-terminal symbol EventOrCall, which was introduced in the section dealing with statements, represents the operation for triggering new events and calling the service methods of attributes. Following grammar shows the syntax of the non-terminal symbol EventOrCall:

```
EventOrCall = [Receiver "."] Attribute "." Event "(" [ActualParams] ")".
Receiver = Entity | RoleTemplate.
Entity = ["$"] ident.
Attribute = ["$"] ident.
Event = ["$"] ident.
ActualParams = Expr { "," Expr }.
```

The EventOrCall operation is able either to call a method or to trigger a new context event, depending on the context in which the operation is called. The optional `Receiver` specifies the receiving entity while the Attribute specifies the receiving attributes service namespace in which the event name has to be unique. If no receiver is specified, the operation takes the containing entity as the receiver. The Event specifies the name of the event which is triggered or the name of the method to call, followed by an optional list of parameters.
Following example shows how a new event is triggered inside an event rule. A new event is created and sent to the entity Loox. The receiving attribute is specified through `AttrName` and the event name is `NewEvent`:

```
rules {
    on Time.Alarm(long h, long m, long s) {
            Loox.AttrName.NewEvent();
    }
}
```

At the receiving entity this new event can be handled by defining one or more rules with the specific attribute and event name:

```
rules for Loox { // rule repository of the receiving entity
    on AttrName.NewEvent() {
            // reaction on the incoming event
    }
}
```

If the receiving entity (Loox) does not define any rule to handle the `NewEvent`, its interpreter tries to find an attribute service method which has the same name as the event and where the parameter list matches. If such an attribute service method exists, the interpreter calls the method.
An EventOrCall which is located within the contraint expression of an if statement is always interpreted as a synchronous method call where the result is immediately retruned to the caller. Following example shows an EventOrCall within the contraint of an if statement:

```
rules {
    on Time.Alarm(long h, long m, long s) {
            if(Time.GetSeconds() == s) { ... }
    }
}
```

Also EventOrCall that are part of an assignment have to be interpreted as synchronous method calls with immediate return of the result. Following example shows an EventOrCall in an assignment statement:

```
rules {
    on Time.Alarm(long h, long m, long s) {
            long seconds = Time.GetSeconds();
    }
}
```

If an `ident` is preceded by a "$" this denotes the variable whose name is stored in the variable with the name `ident`. This indirection is necessary for triggering events with variable destinations at runtime. The following example shows how this indirection can be used to specify at runtime where the response event should be sent to. The entity *PDA* listens to events that are triggered by the attribute Lookup in order to find entities that act in the role of a beamer. When such an entity is found, the *PDA* stores the beamer's name in a global variable called `bEntity`. The expression `a actsAs b` is a shortcut for `a.contains(b)` and checks if the entity *a* acts in the role *b*. Every time a user notification event was triggered, the notification message is sent to the most recently found beamer:

```
rules for PDA {
    string bEntity = "";  // next beamer entity which enables a rich output possibility

    // when the lookup attribute triggers an EntityAppeared event we check if a Beamer was found
    on Lookup.EntityAppeared(string entity) {
    if (entity actsAs <Beamer>)
        bEntity = entity;
    }

    // when a user notification event was triggered we send it to a Beamer device
    on Notification.UserNotify(string msg) {
    if (bEntity != "")
        // indirection enables dynamic event destinations at runtime
        $bEntity.Display.ShowMessageBox(msg); // trigger ShowMessageBox event
   }
}
```

Indirection can also be used for specifying the attribute and the event name, as it is shown in the following example:

```
$entityVar.$attrVar.$eventVar(/* parameters */);
```

Triggering multicast events. Instead of an entity the programmer can also specify a *Role-Template* denoting all entities which act in this role. This allows the specification of multicast event destinations. Multicasts are only performed as asynchronous event trigger operations. There exist no multicast method calls. The previous example could be changed so that all available beamers get notification messages:

```
rules for PDA {
    // when a user notification event was triggered we send it to all available Beamer devices
    on Notification.UserNotify(string msg) {
        <Beamer>.Display.ShowMessageBox(msg);
    }
}
```

Fig. 48 shows how the event `UserNotify` is caught by a context rule and how it is sent to a multicast destination. The event is sent to all entities that act in the role of a beamer.

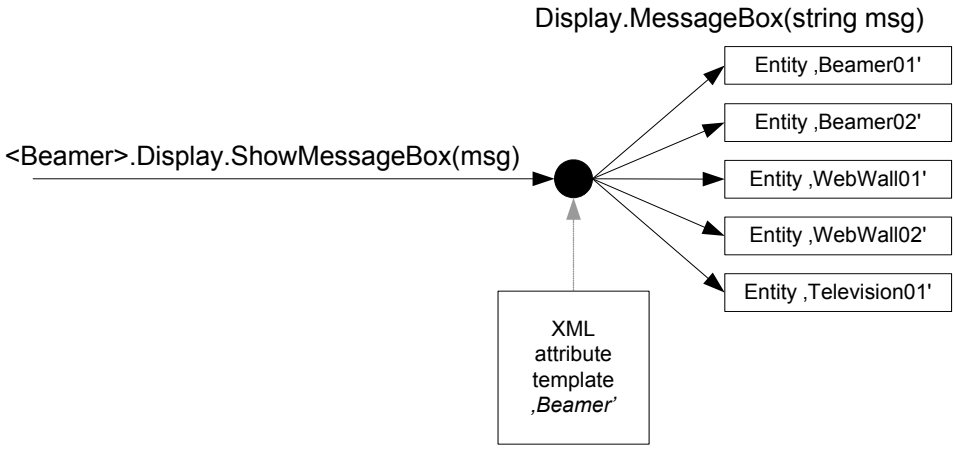

Fig. 48 Specification of multicast event destinations

4.6.3 Error Handling within Context Rules

If context events are processed in a dynamically changing network environment a number of runtime errors can occur. Entities are likely to disappear from a scenario and lookup leases can expire due to wireless network latency. The dynamic change of the network configuration is not the only possible source of errors. Changing attribute templates, dynamic loading or unloading of attributes and context rules, as well as entity migration between context containers could lead to unexpected runtime errors.

To react on known kinds of runtime errors the interpreter of the SiLiCon framework has to offer a mechanism for realizing problems and for defining actions in response to them.

An important aspect of error handling mechanisms is the possibility to separate error handling code from normal code. To meet this requirement and to design an error handling mechanism that fits into the architecture of the SiLiCon framework, an event-based error handling mechanism was defined.

Statements in a context rule can throw an `InterpreterException` when a runtime problem occurs. The `InterpreterException` contains information about the problem, namely an error identifier and a set of parameters. When an `InterpreterException` is thrown, the interpreter stops the execution of the context rule, catches this exception and maps it to a context event. The error identifier is mapped to the event name and the error parameters are copied into the event's parameter list. After the new event was created, the interpreter processes this event before it handles other events that are waiting in the event queue.

A scenario designer can therefore specify error handling code in a context rule that catches the error event. The following code is taken from the Java implementation of the interpreter and shows how an error event is used to catch a runtime error. If the lookup module does not find the event's destination entity it throws an *InterpreterException* which triggers a *DestinationEntityNotFound* event at the current source entity:

```
if (lookup.getEntity(destEntity) != null) {
    if (lookup.isLocalEntity(destEntity)) {
```

```
            ((CFEntity)lookup.getEntity(destEntity)).processEvent(ev);
        } else { // send to remote context container
            CFSendModuleInterface sendMod =
            lookup.chooseSendModule(ev.getDestinationEntity().getIdentifier());
            if (sendMod != null) sendMod.sendAsync(ev);
        }
    } else { // lookup cannot find the destination entity, throw an InterpreterException
        Object[] params = new Object[1];
        params[0] = destEntity;
        throw new InterpreterException(
            ev.getSourceEntity().getIdentifier(),
            ev.getSourceAttribute().getIdentifier(),
            "DestinationEntityNotFound",
            params);
    }
```

The next code fragment shows how an `InterpreterException` is mapped into a new context event. The interpreter catches the `InterpreterException`, creates a new context event and fills it with the error information. The parameters of the `InterpreterException`, which was triggered by a Java program (or a program in some other language), are mapped to SiLiCon data types. The class `CFParameter` offers a static method `wrapObject(object o)` which maps an object from the actual Programming environment (e.g. Java) to a SiLiCon type (if this is possible) and returns an instance of class `CFParameter`. The resulting list of `CFParameters` is forwarded to the constructor of class `CFEvent`. The new `CFEvent` is then sent to the source entity which triggered the error:

```
    try {
        deliverEvent(ev);
    } catch(InterpreterException e) {
        CFCommand cmd = new CFCommand(e.getErrorIdent());
        Object[] params = e.getErrorParams();
        for (int i = 0; i < params.length; i++) {
            CFParameter eP = CFParameter.wrapObject(params[i]);
            cmd.addParameter(eP);
        }
        CFEntity entity = lookup.getEntity(e.getSourceEntity());
        CFEvent eEv = new CFEvent(entity, entity.getAttributeByName(e.getSourceAttribute()), cmd);
        ((CFEntity)lookup.getEntity(eEv.getSourceEntity().getIdentifier())).processEvent(eEv);
    }
```

The scenario designer can now specify the error handling code in a specific context rule that resides in the entity which caused the error. The following rules show how to handle the *DestinationEntityNotFound* event. The first rule triggers the event *EntityX.AWTDialog.Show*. If *EntityX* is not found the interpreter triggers an error event which can then be handled by the original event source (the entity *TestErrorHandling*):

```
    rules for TestErrorHandling {
        on Time.PeriodicTriggerEvent() {
            trigger EntityX.AWTDialog.Show("Hello World");
        }
        // error handling rule
        on Time.DestinationEntityNotFound(string EntityX) {
            // error handling code goes here
            AWTDialog.Show("EntityX was not found!");
        }
    }
```

The advantage of this kind of error handling is the possibility to supply error handling code at runtime over the network using the same event mechanism as for normal rules. It is even possible to insert, remove or disable context rules in response of a runtime error. According to the requirements of the SiLiCon project this mechanism provides a flexible solution to context-aware error handling. The scenario has the possibility to change error handling policies according to situational changes.

The separation of error handling code from the normal code, which implements a specific distributed scenario, is also solved with this mechanism.

4.6.4 Runtime Deployment of Context Rules

Designing context-aware applications or scenarios in highly dynamic network environments requires that the middleware has to support configuration changes at runtime. To react on changes in the network environment, it is necessary that the state transitions—and therefore the rule repository—are changeable. As the SiLiCon framework is based on interpreters that execute context rules, one can define an interface for modifying the rule repository at runtime. This should even be possible using any kind of transport module that is registered at the context container.

To change the rule repository of each entity that is reachable over a network connection means that a scenario designer is able to stop, redesign, deploy and restart a distributed application without any major effort. The remote development and distribution dramatically simplifies the development and testing of distributed context-aware applications.

Controlling the rule repository with an attribute. The remote development and deployment of context rules poses a security problem for context-aware scenarios. For this reason, the SiLiCon framework allows access to the rule repository and the interpreter only via a specific attribute. If an entity does not load this attribute, its rule repository and its interpreter cannot be accessed. By unloading these attributes after the test phase one can disable the modification of the rule repository.

In the SiLiCon framework, the attribute *RuleRepository* provides the public interface for changing the rule repository of an entity. It has all the advantages of standard SiLiCon attributes such as:

- It can be loaded and unloaded at runtime.
- It can be accessed via all transport modules that are registered in the context container.
- It can react to context events, for example, by adding or removing context rules in response to an event.
- Other rules can access it by triggering events that are handled by the *RuleRepository* attribute.

Fig. 49 shows a typical entity with the two attributes, *LookupAttribute* and *RuleRepository*, that are loaded by default.

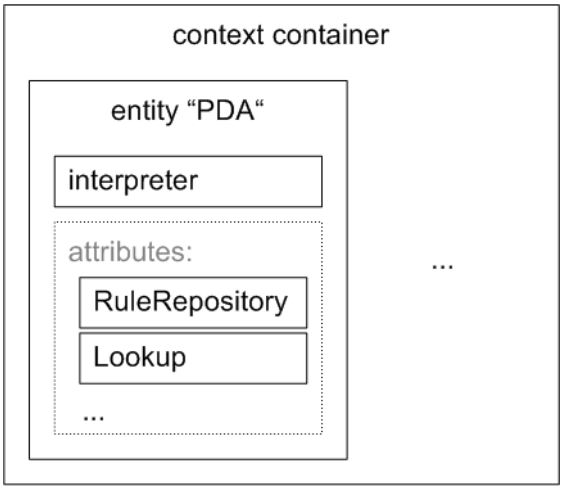

Fig. 49 Entity PDA with the attributes *Lookup* and *RuleRepository*

The *RuleRepository* attribute offers a method `XmlTag CFgetRuleInfo()`, which returns all rules of the repository in XML format. The method `CFsetValid(String event, boolean valid)` enables or disables all rules which handle the event that is described by the first parameter. If the flag *valid* is true the rules are enabled otherwise disabled. This method is useful for debugging and testing, when the scenario designer does not want to delete a rule, but just to disable it. Furthermore, it is can also be useful to disable or enable rules from action blocks of other rules, which is shown in the following example:

```
rules {
    // a rule that reacts on alarm events
    on Time.Alarm(long h, long m, long s) {
        RuleRepository.setValid(“Notification.UserNotify“, false); // disable a event rule
    }

    // a rule that reacts on user notifications
    on Notification.UserNotify(string msg) {
        .. // some notification action
    }
}
```

To count the number of rules in a repository, *RuleRepository* offers the function `CFcountRules()`. The method `CFclearRules()` can be used to reset a rule repository, i.e. to delete all its rules and global variables. There are two possibilities for adding new rules and global variables to a rule repository:

- **Asynchronously**: The inserting entity sends an asynchronous event with the new rule definitions to the *RuleRepository* attribute, which adds them to the repository and sends back a status event. If the rules contain syntax errors they are reported with the status event. The method which adds new rules asynchronously is: `CFaddRulesAsync(String fromEntity, String rules)`.
- **Synchronously**: To add new rule definitions synchronously one has to call the method `String CFaddRulesSync(String fromEntity, String rules)`. This method inserts the rules and returns the status immediately back to the caller. The

contents of the returned status is the same as the event contents that the asynchronous mechanism sends back to the caller.

When *RuleRepository* adds new rules it has to check that they do not corrupt the existing repository (e.g. by syntax errors). Before the repository is changed, the interpreter is suspended and the repository is saved. Then the new rules are parsed and inserted into a copy of the repository. If this succeeds, the interpreter resumes. If it fails, the old repository is restored before the interpreter resumes.

The method `boolean CFremoveRule(String event)` can be used to remove all rules that react on the specified event from the repository.

The SiLiCon visual tool support. The development of distributed scenarios that run on a multitude of platforms and devices is a complex and time-consuming task. The SiLiCon framework offers some basic concepts for the remote development of such scenarios. For example, it supports the dynamic loading of attributes. The scenario designer can deploy new attributes over the network and can thus change the role of an entity. Another important concept is the deployment of rule definitions over the network in order to change the behavior of an entity.

Since all relevant information, such as the lookup information and the attribute template information, is sent in XML form, the framework offers the possibility to implement a visual builder tool for the development of distributed scenarios. Chapter 4.1.6 introduced XML configuration files for defining the startup configuration of context containers. With a visual builder tool one can configure the contents of a context container and its behavior and deploy this configuration over the network to any available device where this container should be running. A visual builder tool could dramatically simplify the development of distributed context-aware scenarios.

4.7 HTTP Logging Module

During the development of the SiLiCon framework we discovered some problems in debugging a distributed application, which is running on embedded and mobile devices. One of the major problems is the lack of output possibilities on embedded and even on mobile devices. Most embedded devices do not offer any displays and the displays on mobile devices are too small to view full exception stack traces. So the idea was to display the error information on a remote device. In order to be able to deliver error information also through firewalls we developed an HTML web logging module that uses port 80 (which is normally free for web surfing). An HTML over HTTP logging module has the following advantages:

- HTML output can beformatted nicely using lists, tables, different fonts, font sizes and colors.
- HTML output can be received and viewed on any device that has a web browser installed.
- Most firewalls permit HTTP transport through port 80, so the error information can be viewed even in a network that is guarded by a firewall.

The logging module in the SiLiCon framework is a static module that can be enabled or disabled in the context container configuration. The following configuration section shows how to do that:

```
...
<Container Title="PDA_Container_01">
<Logging Type="http" Port="8080" LogStdOut="No" LogStdErr="Yes"/>
...
```

The *Port* attribute defines which port should be used to provide the error information at runtime. The *LogStdOut* attribute specifies whether the standard output, which normally is shown on the console, should also be put on the specified medium. Exceptions would therefore also appear on the specified medium. The *LogStdErr* attribute specifies whether system exceptions (such as *NullPointerException*) should also be logged.

The *Type* attribute of the *Logging* element supports the following values, which control the kind of logging activities:

- **HTTP**: Type *HTTP* enables the delivery of logging information over the HTTP protocol using HTML format.
- **Console**: Type *Console* enables the delivery of logging information using the system console.
- **File**: Type *File* enables the delivery of logging information using a file, where all the information is stored.

The class *Log* provides the following three methods for printing out logging information, where the output goes to the medium which was specified in the container's configuration:

```
class Log {
   public static void info(Object src, String msg);
   public static void error(Object src, String msg);
   public static void warning(Object src, String msg);
}
```

The output methods take two parameters: the object that called the method and a debug message that should be printed. Fig. 50 shows a sample output:

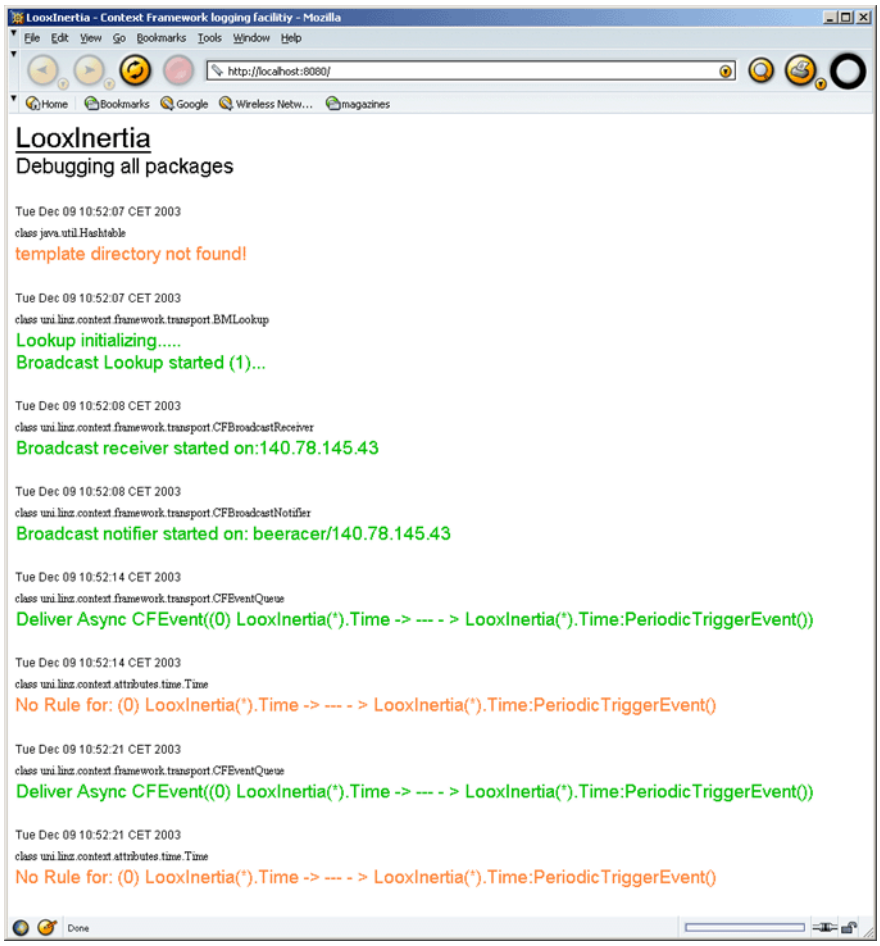

Fig. 50 HTML output generated by the HTTP logging module

By default, error messages of all SiLiCon Java packages are visualized and formatted with HTML. The heading of the logging page shows the name of the context container (*LooxInertia*) and which packages are visualized. In Fig. 50 all packages are visualized (*debugging all packages*), which means that no package filter was set. Every debug message consists of a line with the date and time, a line with the type name of the source object and a line with the actual message. An information message appears in green, a warning in orange and a critical error in red.

The logging module stores a limited number of debug messages. If this number is reached, the oldest message is deleted to create space for the new one. Since there can be lots of debug messages the logging module implements a package filter mechanism. The package filter mechanism is responsible for filtering messages which are specified by the scenario designer. The HTTP parameter '*package*' can be used to specify which debug messages should be visualized. It can be set to a full type name (e.g. `package=uni.linz.context.attributes.time.Time`) or to a substring that has to be part of the class name (e.g. `package=Time`). Fig. 51 shows how the scenario designer is able to filter the debug messages by setting the package parameter to `BMLookup`:

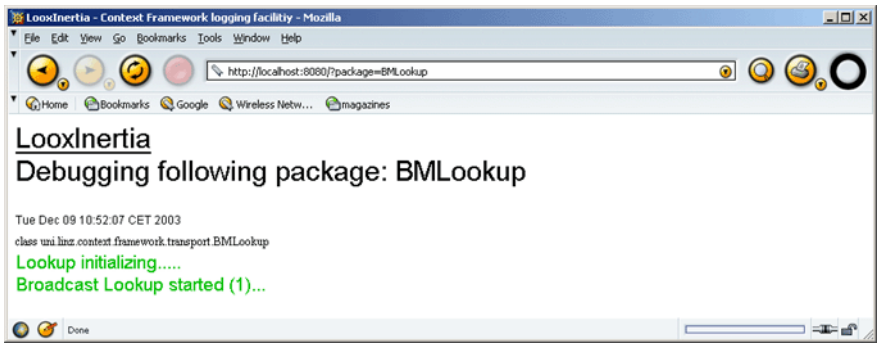

Fig. 51 HTTP parameter '*package*' set to `BMLookup` in order to filter the debug messages

5 Comparison

This chapter gives an overview of the features of various frameworks that support the design of context-aware software. It also points out their advantages and disadvantages in relation to the SiLiCon framework as well as their design goals. Some interesting comparison aspects were taken from [He01]. It should be mentioned, that the application domains of the different frameworks overlap but do not exactly match all aspects. Jini, for example, was designed to operate as a dynamic service platform, whereas the Context Toolkit was designed as a complete pervasive computing environment. The feature sets of both frameworks overlap, but every framework offers more or less specific features according to their original design goals.

5.1 Classification of Context-Aware Software

Context-aware software can be divided into three main classes: stand alone applications, context-aware service platforms and context-aware pervasive environments.

- **Stand alone context-aware applications**: This class of context-aware software is the most commonly used. Such applications perform specific sensorial tasks such as the authentication of users, object tracking in logistic systems, recording of videos according to movement, or online/offline status retrieval in messengers. Since such applications are tailored to their specific needs they usually have a good performance but high development costs.
- **Context-aware service platforms**: Jini is a typical member of the class of context-aware service platforms. In general, service platforms aim at the rapid development and deployment of services. In dynamic environments these service platforms also offer dynamic discovery and service lookup. The development of context-aware applications on top of an existing service platform offers a rapid development process, reliability upon the existing basic layers and interoperability with applications that are also based on the same service platform.
- **Context-aware pervasive environments**: Pervasive environments often require dynamic configuration and a flexible interaction between distributed entities. Therefore frameworks for such environments have to offer advanced functionality. Object identification, object migration, object life cycle management, flexible processing of services, component reuse as well as platform and language independence are only a small number of features that a pervasive environment has to provide.

This chapter gives an overview of the features of current context-awareness frameworks and classifies them according to the three basic classes described above (see Table 9).

Table 9: Overview of context-awareness software and their classification

Product	Class	Language	Platforms
SiLiCon	pervasive environment	Java 1.1.8	all platforms that offer a personal JVM
Jini	service platform	Java 1.2	all platforms that offer a JVM
Web Services	service platform	any	any
ParcTab	pervasive environment	Unix programming environment	Unix, Linux
Context Toolkit	pervasive environment	Java 1.2	all that offer a JVM
CoolTown	service platform	any Web server supported language	all platforms that offer a Web server
Sentient Computing	pervasive environment	any that supports CORBA	all platforms that support CORBA

Table 9 shows that the SiLiCon framework, the ParcTab environment, the Context Toolkit, the CoolTown project and the Sentient Computing environment can be classified as pervasive environments. Service platforms, such as the web service initiative or Sun's Jini framework, provide support for rapid service development and deployment even in dynamic network environments, but they do not provide support for the design of entire pervasive environments. It is, however, possible to design pervasive environments on top of existing and well-established service platforms. The SiLiCon example implementation is running on top of a JDK 1.1.8 environment which means that the SiLiCon framework is runnable on top of Java Virtual Machines which implement the Java Personal Profile. Pervasive environments which need JDK 1.2 are not running on Java Personal Profile which means that they are not running on mobile devices.

5.2 Universal or Practical World Model Assumption

While service platforms limit their world models to service descriptions as well as discovery and lookup, pervasive environments use much more complex world models to identify and to interact with other objects. Pervasive environments have to identify different objects in order to interact with them or to update relations to these objects. In the CoolTown project a *PlaceManager* is able to identify all objects that are in a specific place in order to update the location of the different nomadic objects.

The aspect of how the world model of a pervasive environment is defined plays an important role. Universal world models or practical world models are two antithetic aspects. A world model which is universal can describe everything. On the other hand, it is possible to create a practical world model which is able to model only one specific situation. In this specific situation the practical world model will be intuitive, but it is hard to model other situations with it. Table 10 compares the different pervasive environments according to their world models and classification mechanisms.

Table 10: Universal versus practical world models

Framework	Description of the world model used
SiLiCon framework	Universal world model based on the ParcTab approach with additional role-based classification features.
ParcTab	Universal world model where objects are described as dynamic environment entities with a variable collection of attributes.
Context Toolkit	Universal world model, which is able to model scenarios by reusing context-widgets, aggregators and interpreters. Classification of entities is not supported.
CoolTown	Practical world model where all objects are classified with the three base classes Person, Thing and Place.
Sentient Computing	Practical world model where environment objects are directly mapped into CORBA objects. According to the CORBA object mapping the Sentient Computing project uses a hierarchical classification model.

5.3 Adaptation

Another important aspect is the adaptation effort that is necessary to develop a new context-aware scenario on top of a pervasive environment framework. Stand-alone context-aware applications cannot usually be adapted to other application domains. Pervasive environments, however, offer the possibility to reconfigure them so that they fulfil new tasks or run in new application domains.

Adaptation allows a scenario designer to reconfigure a scenario without much effort. In order to make a framework adaptable there must be a flexible application definition process.

Table 11 compares the adaptation possibilities of the various pervasive environments.

Table 11: Scenario adaptation possibilities of different frameworks

Framework	Adaptation possibilities
SiLiCon framework	Adaptation by dynamic loading of attributes and entities. Scenario definition by XML-based configuration files. The definition of the scenario interaction is based on interpreted rules that can be changed at runtime.
ParcTab	Flexible definition of scenarios by using dynamic environment objects and dynamic attribute collections. It is not possible to change a scenario at runtime.
Context Toolkit	Flexible definition of scenarios by using reusable context-widgets, interpreters and aggregators. Changing a scenario is possible by loading classes dynamically.
CoolTown	The scenario modelling process is more intuitive because the CoolTown project uses only three base classes of objects. On the other hand, the possibilities to change a scenario at runtime are limited and the definition of the interactions tends to be complicated.
Sentient Computing	A sentient computing application is flexible within a CORBA environment. It is hard to change the scenario interaction at runtime. The scenario modelling process is more complicated compared to the other environments.

5.4 Web Compliance

The web compliance aspect evaluates the integration of web technology into the various frameworks. Web protocols and web representation formats are designed to be accessed by web browsers. The main advantage of web technology is that the context information is accessible for a broad spectrum of users. Everybody who uses a web browser is able to access it. In contrast, proprietary protocols and representations are only accessible for a small group of users. On the other hand, web technology (e.g. the HTTP protocol or XML data) is often not as efficient as proprietary binary protocols. Table 12 compares the web compliance of the various pervasive environments.

Table 12: Web compliance of the various pervasive environments

Framework	Web compliance
SiLiCon framework	The SiLiCon framework offers an HTTP transport module which allows sending context events over HTTP. In combination with the SOAP event encoder the SiLiCon framework provides full Web Service compliance.

Table 12: Web compliance of the various pervasive environments

Framework	Web compliance
ParcTab	When ParcTab was designed in 1994, it was not web compliant. Recently, however, web technology has been integrated into the ParcTab framework. **[Bo00]**
Context Toolkit	The Context Toolkit uses the HTTP protocol to transmit context information which is represented in a proprietary XML format.
CoolTown	The CoolTown project is completely web-based. Context information is received, processed and forwarded by Web servers and dynamic web server modules (e.g. the PlaceManager). The CoolTown project uses HTTP as the transport protocol. Complex context information is represented in XML that is transformed into a human-readable HTML format.
Sentient Computing	The original sentient computing project used proprietary protocols for the transport of context information. Some appliances used well-established technologies such as VNC (Virtual Network Client).

5.5 Scalability

The scalability aspect is an important issue for pervasive environments that operate in dynamic network environments. In commercial setups, pervasive environments tend to have masses of devices (e.g. in a wide-area sensor and actuator network). Mobile and embedded devices are getting smaller and smaller while their number increases. With purely decentralized P2P systems it was soon realized that without intelligent organisation of P2P networks the number of peers is limited to a small number. Modern P2P systems use hybrid mechanisms and introduce superpeers that manage parts of the P2P network. To show scalability bottlenecks it is necessary to analyze the discovery, lookup and transport mechanisms of the various pervasive environments in detail. Table 13 shows some scalability aspects of the various frameworks.

Table 13: Analysis of scalability aspects in different frameworks

Framework	Possible scalability issues
SiLiCon framework	SiLiCon devices communicate directly with each other. The discovery mechanism is based on a hybrid model where every device scans the local subnet for other devices and uses registered addresses for global discovery. The SiLiCon framework implements a hybrid P2P system where the major scalability bottleneck is the discovery, which is weakened by the fact that it uses registered addresses for global discovery.

Table 13: Analysis of scalability aspects in different frameworks

Framework	Possible scalability issues
ParcTab	The ParcTab system is a lightweight environment that is operating on top of the TCP/IP protocol. ParcTab implements no P2P functionality and is therefore not working in ad-hoc networks. The ParcTab system has no major scalability bottlenecks, exept when too many Badges are located in the same location.
Context Toolkit	The 1 to n relation between context widget and interpreter poses a scalability problem.
CoolTown	In the CoolTown system , there could be a scalability problem with the relationship directory, which handles many objects that are located in the same place.
Sentient Computing	If the notification channels are not organized hierarchically, the dissemination of the TRIPParser output poses a scalability problem.

5.6 Mobile Device Portability

Pervasive environments include a multitude of different mobile and embedded devices. Therefore it is necessary to design pervasive frameworks such that they can be ported to such devices. The portability of modern P2P and pervasive environments as well as service platforms is an important issue. Jini for example is based on the class library of the JVM 1.2, which makes it difficult to run it on J2ME. The same problem occurs when porting the Java P2P framework JXTA to mobile devices. Most of the modern pervasive environments are designed to run on desktop systems, which offer strong CPU power, large storage and rich input/output facilities. Table 14 lists some problems that occur when the various frameworks are ported to run on mobile devices.

Table 14: Portability aspects of the different frameworks

Framework	Mobile and embedded device portability
SiLiCon framework	The SiLiCon framework was designed to run on Java Personal JVMs, which means that it runs on devices which offer a JVM that supports at least Java 1.1.8. The SiLiCon framework actually runs successfully on PC104 industrial PCs, on PocketPCs as well on embedded Linux devices.
ParcTab	The ParcTab framework typically runs under Linux and in embedded Linux environments.
Context Toolkit	The Context Toolkit is running on devices that support at least Java 1.2 which means that it is currently not possible to run it on PocketPCs.

Table 14: Portability aspects of the different frameworks

Framework	Mobile and embedded device portability
CoolTown	The CoolTown project is based on web technology. Although it is sufficient to have thin web browser clients for receiving context information, the active context processing parts have to run in a web server environment.
Sentient Computing	The sentient computing framework is running on platforms which provide CORBA support.

6 Context Application Scenarios

In this chapter some examples of context application scenarios are presented and discussed. Since one of the major goals of the SiLiCon framework is to support rapid development of context-sensitive applications, it is necessary to show how simple context-aware applications can be created with this framework. Although context-sensitive applications that were implemented from scratch could probably provide the same features as applications implemented on top of the SiLiCon framework, the development process with the SiLiCon framework is much easier, faster and more flexible. It just requires the implementation or reuse of context attributes (i.e. wrappers for sensors or actuators), writing the XML configuration file and designing ECA context rules that control the interaction between the entities and attributes in the scenario.

Our examples range from simple scenarios with two entities and a small number of attributes to complex scenarios with more than 20 entities on different mobile devices.

6.1 A VRML Control Scenario

Our first SiLiCon sample application shows how to model a simple but effective sensor/actuator combination. In this scenario sensoric input is used to control a VRML (*Virtual Reality Modeling Language*) plugin in order to interact with a given 3D scene.

Context-sensitive applications often use 3D scenes to visualize objects or locations. The VRML standard represents a script language for the definition of 3D scenes. It was designed to run as a web browser plugin in order to provide 3D scenes in HTML pages. A VRML plugin (such as *CosmoPlayer* or *WorldView*) is controlled with the EAI (*External Authoring Interface*) which is a Java library. A Java applet that is running in the same HTML page as the VRML plugin can use EAI calls to control the plugin, i.e. to scale, rotate, translate or to change a VRML scene.

We also need a server component, which receives events from the SiLiCon framework and instructs the applet how to change the VRML scene. Due to the security restrictions of Java applets, the server component has to run on the machine from which the applet was loaded, because applets are only allowed to contact servers on this machine. The server component is implemented as a SiLiCon attribute with the name *VRML* that is able to receive and to trigger context events. When an entity owns a VRML attribute it can route any sesoric information through context rules in order to control a VRML scene. Sensoric input could be the movement of a pointing device (e.g. a mouse, trackball, joystick, tablet, touchsceen, etc.) or a location sensor that measures the heading, pitch and roll values of the device (e.g. an InertiaCube). Fig. 52 shows how a VRML scene can be controlled with a multitude of sensoric devices.

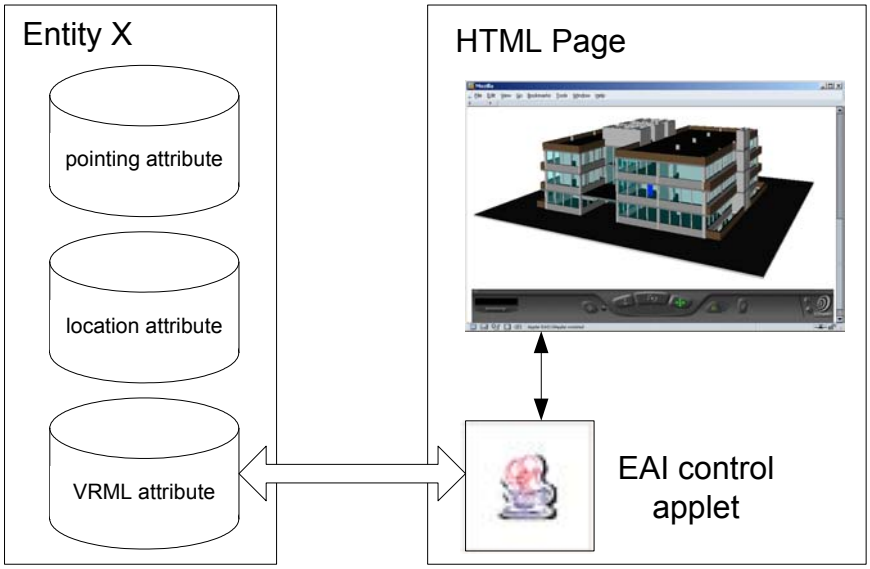

Fig. 52 Control a 3D VRML scene within a SiLiCon scenario

The VRML scenario uses a location sensor, called InertiaCube, to trigger events according to its relative heading, pitch and roll values. The collection of attributes for this demonstration scenario is defined as follows:

```
<Inventory>
    <AttributeSequence>
    <Attribute Name="ControlRules" Host="localhost" Class="...controlrules.ControlRules"/>
    <Attribute Name="Location" Host="localhost" Class="...location.Location"/>
    <Attribute Name="VrmlServer" Host="localhost" Class="...vrml.VrmlEAIServer">
        <AttributeSpecific port="8081"/>
    </Attribute>
    <Attribute Name="Time" Host="localhost" Class="...time.Time">
        <AttributeSpecific delay="6"/>
    </Attribute>
    </AttributeSequence>
</Inventory>
```

The *Location* attribute triggers events describing the heading, pitch and roll values that range from -180 to 180 degrees. The *Vrml* attribute is responsible for controlling the VRML scenario and connects to the applet in the HTML page. To rotate a 3D scene, the EAI library takes the *x*, *y* and *z* rotation value, measured in radiants. So the heading, pitch and roll values have to be converted in order to use them as input values for the rotation. At this point the SiLiCon framework shows its major strenght. It is simple to write a context rule that converts the incoming heading, pitch and roll values into the radiant format that is required by the *Vrml* attribute and to route this converted values to the applet in order to perform a rotation in the VRML scene.

In order to use a second sensoric device such as a mouse pointer, it is necessary to load the pointing device attribute, to convert the sensoric information using a new context rule and to route this information to the *Vrml* in order to perform the operation.

The context rule that performs the heading, pitch and roll conversion as well as the routing is defined as follows:

```
rules for Entity_X {
```

```
        on Location.HPR_Position(double h, double p, double r) {
            h = (h + 180) / 360;
            p = ( (p + 180) / 360) * 6.283;
            r = ( (r + 180) / 360) * 6.283;
            VrmlBrowser.Vrml.setValues( p - 6.283, h * 6.283, r  - 6.283 );
        }
    }
```

It is just a matter of configuration to distribute the scenario across different devices. The *Location* attribute and the *Vrml* attribute can be loaded into separate entities, which are hosted on two different devices.

6.2 A Context-Sensitive Emergency Scenario

The next sample scenario application was designed to provide context awareness in emergency situations. The idea is to collect context information about a patient, who has problems with his health. The patient wears a set of sensors that collect information about his physical health and an actuator that triggers an emergency call when the patient's health situation changes.

Entity and attribute design. The first step in implementing the emergency scenario is to identify all entities and attributes that may be important for this situation. The following entities were identified:

- *HeartPatient*: The entity *HeartPatient* represents the patient with his sensors and actuators. The entity of the patient (HeartPatient) contains an attribute (HeartMonitor) which wraps the services the heart monitor offers. The *HeartPatient*'s heart monitor is responsible for monitoring the health situation of the patient.
- *EmergencyDispatcher*: The *EmergencyDispatcher* entity is responsible for the receiving emergency calls and managing ambulances with their crews. The *EmergencyDispatcher* contains a list of ambulances and mobile doctors in order to react on incoming calls. It also knows where the ambulances and their crews are located and if they are busy or not.
- *AmbulanceCar*: The set of *AmbulanceCar*s is responsible for receiving emergency calls from the emergency dispatcher and for traveling to the location of the patient. Every *AmbulanceCar* entity has a crew which consists of a driver, a nurse and a mobile doctor. Only if the complete crew is in the *AmbulanceCar* entity, it is able to accept an emergency call.
- *MobileDoctor*: The *MobileDoctor* is a member of an emergency team in a specific ambulance car.
- *Hospital*: The *Hospital* entity represents a specific hospital which should be informed about the delivery of the patient who triggered an emergency call.

The emergency scenario is based on RFID technology to locate entities on a map and to update the containment hierarchy. Every mobile container entity has an *RFID* attribute. To resolve the position of an entity on a map it is necessary to contact a *RFIDResolver* attribute, which holds the mappings between RFID IDs and absolute map positions. The emergency scenario maps a real world emergency situation to a local demonstration environment, which

means that indoor communication and location technology has to be used. In a real world situation these indoor solutions would be changed to outdoor sensor and network technology as it is shown in following table:

Table 15: Indoor versus outdoor technology

Information	Indoor	Outdoor
Location	RFID	GSM cell tracking and GPS
Communication	WLAN, Bluetooth	GPRS, UMTS, WLAN
Mapposition	*RFIDResolver* delivers map coordinates	Navigation service delivers map for given longitude and latitude values

Scenario configuration . The emergency demonstration scenario consists of four digital devices (1 Laptop, 2 iPAQs, 1 Loox), each of which hosts a context container. Every context container is able to host a set of entities which are part of the scenario. One of the four digital devices is a Laptop, which executes all the performance-critical tasks. It hosts the *EmergencyDispatcher* entity, the *RFIDResolver* attribute and all the simulated mobile doctors and ambulances. One of the iPAQs hosts the *AmbulanceCar* entity and is directly connected to a physical ambulance car as it is shown in Fig. 53. The ambulance car contains an RFID reader, which updates the containment hierarchy, if a tagged entity enters the car. Thus the ambulance car can sense whether the mobile doctor, the driver, the patient and the nurse are on board.

Fig. 53 iPAQ device hosting the *AmbulanceCar* entity, Loox device hosting the *MobileDoctor* entity

The physical emergency scenario is built upon a floor material which is completely tagged with RFID transponders in order to accomplish location tracking. Furthermore, every person is tagged with an RFID transponder in order to realize when it enters an ambulance car or the hospital. If an ID of a tagged entity or a transponder in the floor material is detected, the

RFID resolver is asked about the identity of the object and its location as it is shown in Fig. 54.

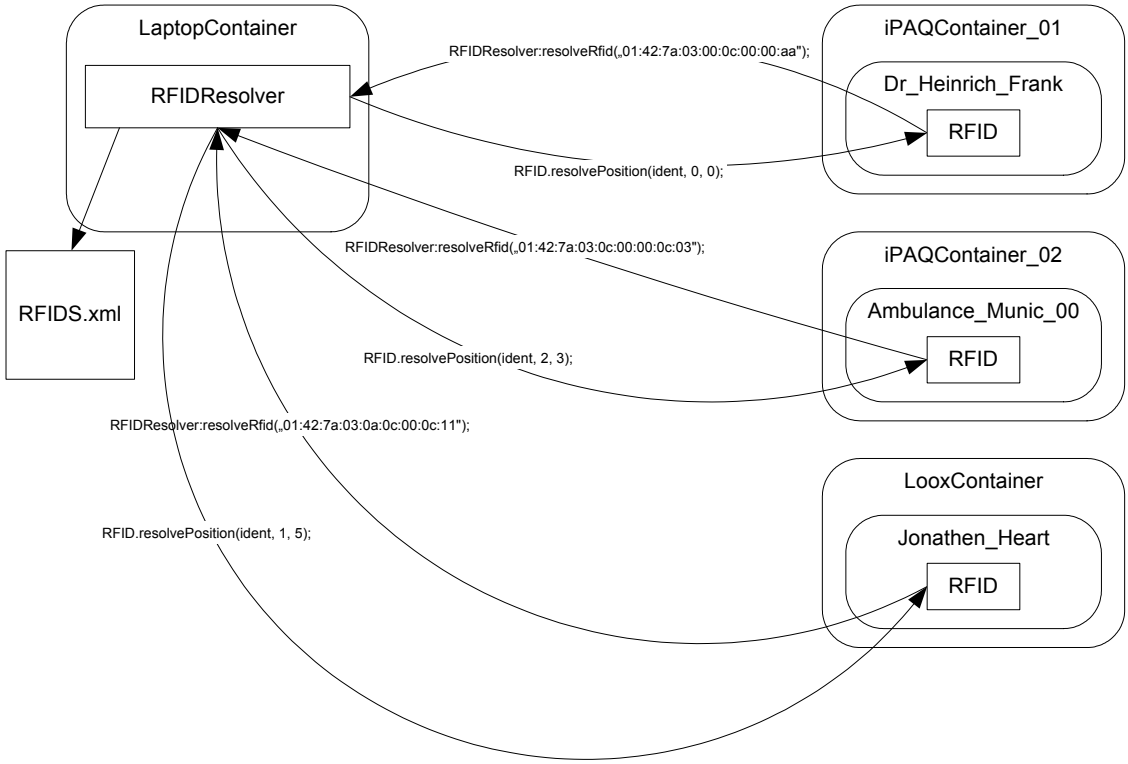

Fig. 54 Requesting location and identification information from the RFIDResolver

After the identity and the location of an RFID transponder ID is resolved, the emergency dispatcher is informed about the location of the identified entity. The emergency dispatcher contains a list of available emergency cars and mobile doctors together with their locations. In case of an emergency call, the location of the patient appears on this map and a context rule calculates the nearest ambulance that is not busy at the moment. The context rule triggers the display of a dialog box, which proposes the human operator the selected ambulance car as an option to select. For an emergency scenario it is important that the operator has the possibility to select between different options which the dispatcher proposes.

The entities update their positions on the map automatically and display this information for the operator. The entities also inform the emergency dispatcher about any changes of their positions or their state. Depending on the tasks an entity has to perform in the scenario, a graphical user interface visualizes the actual state of the entity on the screen.

6.3 A Context-Sensitive Office Scenario

The third context-aware sample application that is build on top of the SiLiCon framework middleware was inspired by the research on *tangible user interfaces (TUIs)*. Tangible user interfaces were first mentioned by Ishii and Ullmer in 1997 [Ishii97]. They defined them as interfaces in which physical objects play a central role as both physical representation of digital information and physical control of a digital scenarios. The word tangible derives from

the latin words “tangibilis“ and “tangere“, which means “to touch”. Ishii and Ullmer realized that the screen as output device is often overconsumed and in many cases not the best way to display complex scenarios, as they showed in an urban planning scenario [Under99]. According to Hiroshi Ishii’s tangible user interface architecture [Ull01], the division between control and presentation should be as it is shown in Fig. 55. In Fig. 55 it is shown that the representation of digital data is divided into two areas: *physical representation* (rep-p) and *digital representation* (rep-d). The rep-p area is the physical representation of digital data and the rep-d is the representation of digital data that is not represented through a specific physical object. Digital data that is represented through a physical object can be controlled physically, while the rep-d cannot be controlled physically.

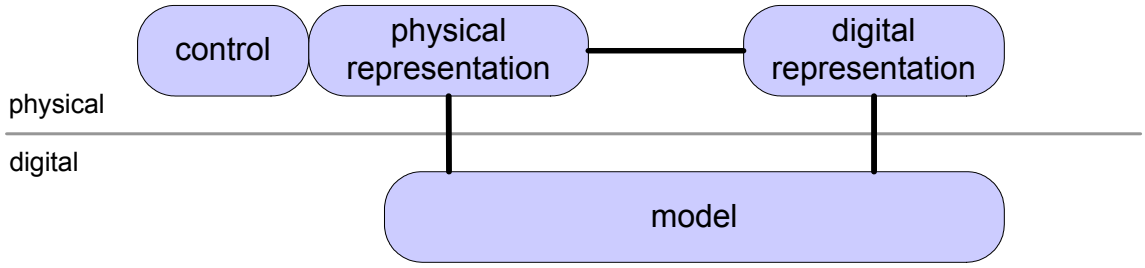

Fig. 55 Ishii’s architecture of tangible user interfaces

Dice example. To understand the physical representation and control of digital information a simple example is given. Three dices are equipped with a sensor that measures their actual state, which is stored in a database. When a human user throws the three dices he is able to control the values of the actual throw in the database. The state of the dices represents, physically, what the database stored for the actual throw. So the result of the actual throw is represented through three physical objects (3 dices), which means that it is a part of the rep-p area. The database also stores the results of the last dice throws. It is useless to represent this kind of information also through physical dices, so this kind of data is represented through the rep-d area. A model of the dice example is shown in Fig. 56.

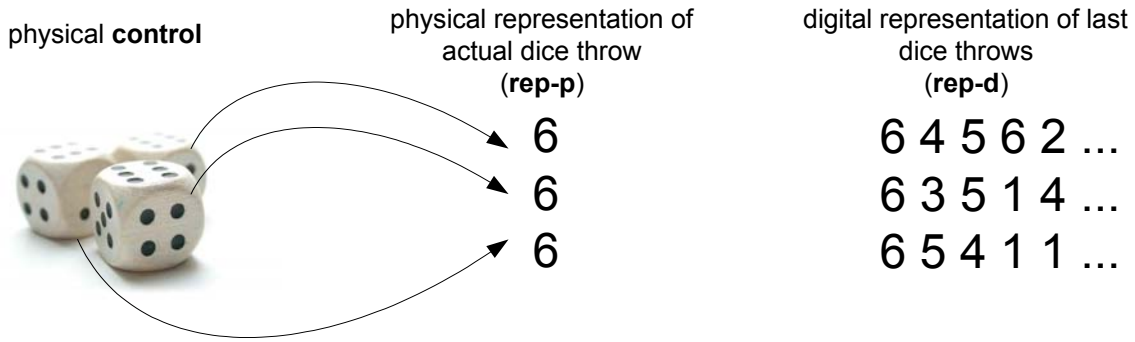

Fig. 56 Example of physical representation and control

The tangible user interface is able to control the digital model, because the physical representation is directly bound to the digital model. If the physical representation changes, the digital model changes as well. Parts of the digital model, which cannot be represented physically or need both a digital and a physical representation, can also be represented digitally

(e.g. on a screen). The physical representation of a digital model has the advantage that its state is persistent and does not disappear if the digital model crashes or the power is switched off. So if the digital model crashes, the physical representation part remains persistent.

Resource control. A major advantage of tangible interfaces is the fact that a human uses more than his eyes to perceive the environment. To manipulate the enviroment with one's hands and one's sense of touch is one of the oldest and most important advantages of humanity.

The idea of our sample context scenario was to combine context awareness and tangible user interfaces, in order to manipulate complex digital environments in a more intuitive way. This combination offers high flexibility, because these two technologies complement each other in many cases. Context awareness provides information that could be important for the interaction with an object and tangible user interfaces represent parts of the state of a digital environment model and, furthermore, allow the user to interact with this model in an intuitive way. With tangible artifacts the user is able to control the environment without deep knowledge of the technology that is embedded in that environment. Context information on the other hand is used to predict the users intention when he uses a tangible artifact in different situations and in different environments.

An example of an tangible artifact would be an object that is able to realize its three-dimensional position. With this information it is possible to control specific states of a digital environment, as it was already shown in the dice example. With a rotation of this artifact it is possible to turn a stereo's volume up or down, or in another context it could be possible to control the mouse cursor on a projector screen. Fig. 57 shows how a tangible artifact with gesture recognition can be used to control various equipment via context rules. The research project *iStuff* gives a good overview about different kinds of sensoric information that could be used to realize specific tangible artifacts [iStuff].

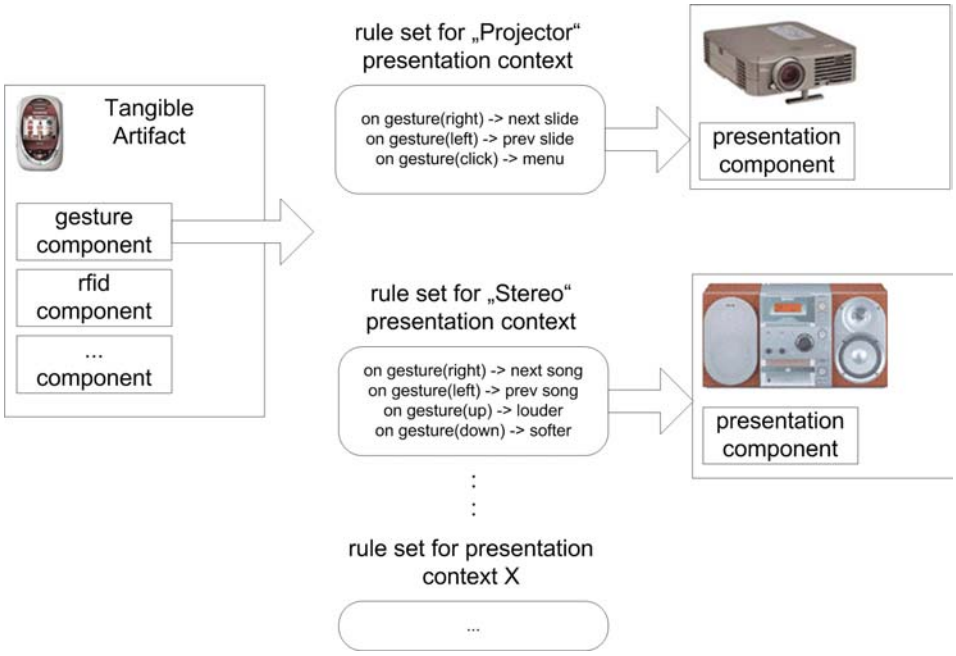

Fig. 57 Dynamic rule sets for different presentation contexts

The context-aware tangible scenario uses RFID tagged objects to represent a digital resource. The user is able to use any object as long as the object is tagged with an RFID transponder (e.g. Cellphones, markers, rubbers, CDs, ...). The user is also able to decide which digital resource he would like to bind to the tagged object. Most of the time the user will bind resources like PowerPoint presentations or video and music files. The goal of the context-aware tangible scenario is to bind such a resource to any tagged real world object and to integrate a position sensor into the real world object in order to control the resource.

Presentation. Every real world object is tagged with an RFID transponder, which enables the contactless identification of objects using the ID of the transponder. The user is now able to take any object and mark it with a transponder, in order to use it as a tangible artifact. The tangible artifact stands then for a digital resource. It represents the virtual location of the digital resource, which is stored on a globally available network drive. The SiLiCon framework determines the context in which the digital resource should be used as a result of the virtual location of its tangible artifact. To bind a tangible artifact to a digital resource, it is necessary to put it in some sort of a loading area. Fig. 58 shows a desktop workplace with a loading area (which is realized through an RFID antenna, which is able to sense RFID transponders). At the loading area two very different RFID tagged objects are used as tangible artifacts: a cellphone and a marker pen. When the RFID antenna senses a tagged object it shows an image of the object on the workplace desktop. The user is now able to drag and drop e.g. an PowerPoint presentation onto the image to bind it to the real world object.

Fig. 58 Loading of digital resources on any tangible artifacts.

6.3.1 Hardware Setup

To combine context sensitivity with tangible user interfaces some standard technologies are used.

Tangible resource binding . To realize the binding of digital resources to real world objects, RFID technology is used. In order to sense objects that contain a transponder, an RFID reader

device is used. The antenna of this reader is able to recognize the ID of a transponder that is in its sensing range.

If the user puts a tagged object (e.g. a pen) next to a reader which is attached to a workstation, the reader entity triggers a context event. The workstation entity retrieves the event and displays the tagged object as an icon on the screen. The user is now able to drag and drop digital resources (e.g. a PowerPoint file) onto the icon and therefore to bind them to the tagged object. The user can then go to a presentation room, where there is also an antenna installed on the table, which triggers a context event when a tagged object is placed next to it. Interested entities can catch this event with a context rule and can handle it accordingy. A beamer entity, for example, could display the mapped digital resource on the wall, as it is shown in Fig. 59.

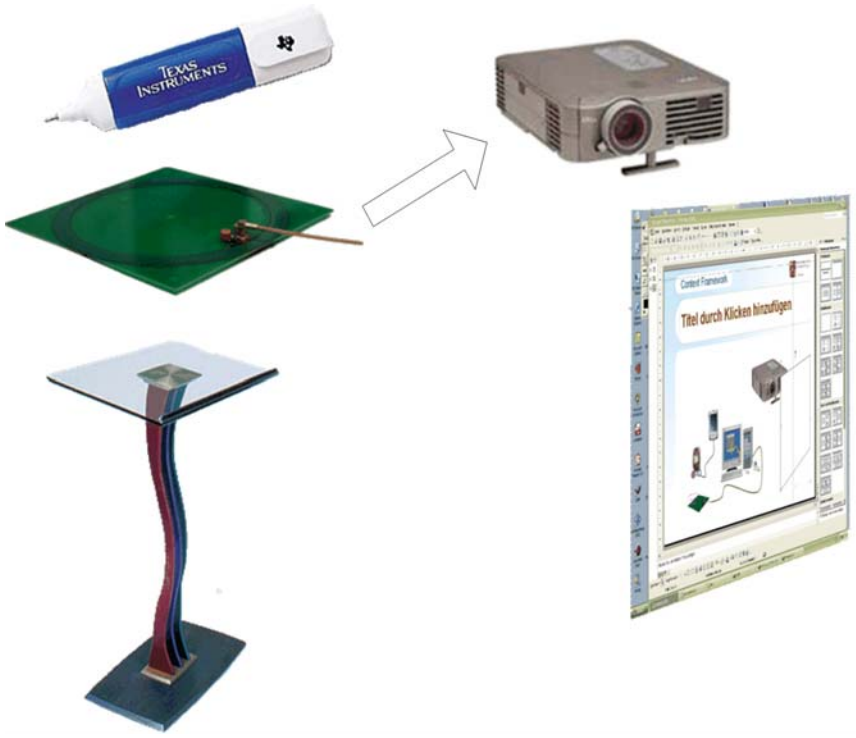

Fig. 59 RFID antenna triggers a context event when the tagged object is put on top of the table.

The SiLiCon framework is responsible for delvering the context event that appears when the user puts the tagged object next to the RFID antenna. In a presentation context, the presentation entity is able to receive this event and to download the attached digital resource from the server. If the presentation entity knows the kind of the downloaded resource, it can deliver the resource to the appropriate application (e.g. a mediaplayer or a presentation program like PowerPoint).

Tangible resource control. While the presentation entity displays the digital resource that is associated with a tagged object, it is possible to control the resource with embedded sensors. To enable an object to act as a tangible artifact that can control associated digital models, it is necessary to embed sensors and actuators into this object.

The intention in our sample application was to control the associated resources as naturally as possible with a tangible artifact. Therefore we decided that the tangible artifact should

be the same physical object that was tagged and associated with a digital resource. In Fig. 58 and Fig. 59 this physical object is presented as a light pen that is tagged with an RFID transponder.

Today, one of the most common personal artifacts, that nearly everybody takes along, is a mobile phone. So a mobile phone is a good choice for a tagged object that acts as a tangible artifact, beside the functionality that such a phone normally provides. An example is shown in Fig. 60, where a location tracker with three degrees of freedom is used to sense user gestures, in order to control the associated digital resource. Tests have shown that for this application no complex gesture recognition is necessary.

Fig. 60 Usage of a mobile phone with transponder and location sensor as a tangible artifact

Heading, pitch, and roll information are used to control, for example, the mouse cursor on a projection screen, as it is shown in Fig. 61. A context rule, that catches the location information from the *InertiaCube* attribute, translates the heading and pitch information into sceen coordinates. The roll location is used to simulate a mouse press, when a certain roll value (~20°) is reached. If the cursor position remains in the same position for a certain amount of time, the DesktopControl attribute sends a mouse press event to the desktop. The following two context rules show how to trigger a mouse press event and how to change the mouse position according to the heading and pitch values. If a HPR_Location event occurs both rules are activated.

```
on Location.HPR_Position(double h, double p, double r) {
  if (r > 20)
      Beamer.Win32DesktopCtl.sendCursorPress();
  }

on Location.HPR_Position(double h, double p, double r) {
  h = (h + 180) / 360;
  p = (p + 180) / 360;
  Beamer.Win32DesktopCtl.sendCursorMove(h, p);
}
```

The performance of this scenario over a WLAN is good enough, so that the control of a desktop over the network is possible without any disturbing delays.

The mouse cursor control on a projector screen is quite intuitive. With a tangible artifact shown in Fig. 60 it is no problem to work on a projected desktop without any prior training or instructions.

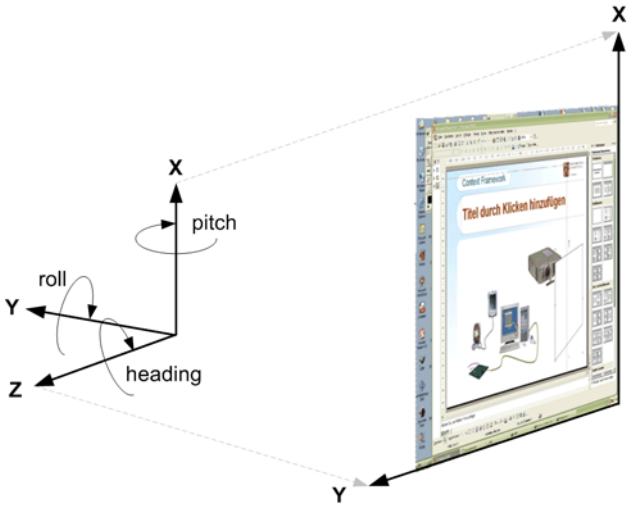

Fig. 61 Translation of heading and pitch values to projector screen coordinates

6.4 An Industrial Maintenance Scenario

The next sample application demonstrates the advantages of the SiLiCon framework in a typical industrial maintenance scenario. The basic idea is, to support service personal by providing detailed machine failure descriptions and possible recovery strategies.

The industrial maintenance scenario consists of a number of machines, which operate as state machines. The machines are simulated by context entities and their state machines are modeled by sets of context rules. To visualize the state changes of a machine we decided to use a robot, which is shown in Fig. 62. The robot is controlled over a serial connection to a PC104 mainboard, which uses a wireless LAN card to provide network access. The PC104 embedded computer hosts a SiLiCon container entity which is able to host a set of context entities, e.g. the *Robot* entity. The robot model supports a set of operations, implemented in a *RobotCtl* attribute. The *RobotCtl* attribute also defines the operating ranges in which the robot is able to move without destructing its hardware.

When the robot, or any other simulated machine produces an error, it triggers a context event which contains the failure number and a short failure description. A context rule routes the failure event to any entity which is acting in the role of a service engineer.

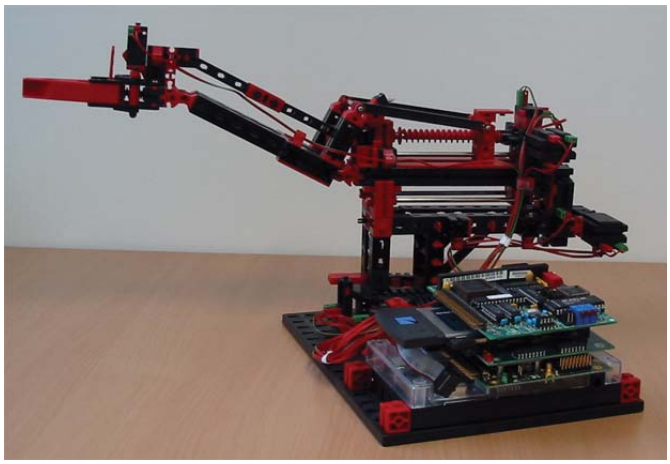

Fig. 62 Model of a SiLiCon-enabled industrial robot device

Service engineers are working on their service terminals and use portable devices, e.g. PDAs, to carry machine information, which is necessary to solve a specific problem. When a machine sends a failure event, an HTML page appears on the service engineer's terminal. This page contains detailed information about the machine, the failure and the possible recovery strategies. According to the fact that a SiLiCon device is able to understand HTTP GET calls by using the SOAP over HTTP transport module, it is possible to embed machine controls directly into the HTML page. The following rule set shows how to define a state machine, which triggers the robot to execute a sequence of defined steps. The event *RobotStop* is triggered when the robot reaches a certain absolute position. The operation `posAbsolut(long a1, long a2, long a3, long a4)` moves all four motors of the robot to a given absolute position. The values a1 to a4 represent the absolute step positions of the four robot motors.

```
rules for Robot {

    on RobotCtl.RobotStop(long a1, long a2, long a3, long a4) {
        Robot.RobotCtl.posAbsolut(100, a2, a3, a4);
    }

    on RobotCtl.RobotStop(100, long a2, long a3, long a4) {
        Robot.RobotCtl.posAbsolut(60, 100, a3, a4);
    }

    on RobotCtl.RobotStop(60, 100, long a3, long a4) {
        Robot.RobotCtl.posAbsolut(0, 0, 0, 0);
    }
    // ...
}
```

Fig. 63 shows the interaction between a defective machine and a discovered service terminal. The entity *Robot* triggers a failure event to any of the discovered entities that act in the role of a service terminal. The robot also triggers its state transition events, in order to perform its normal movements.

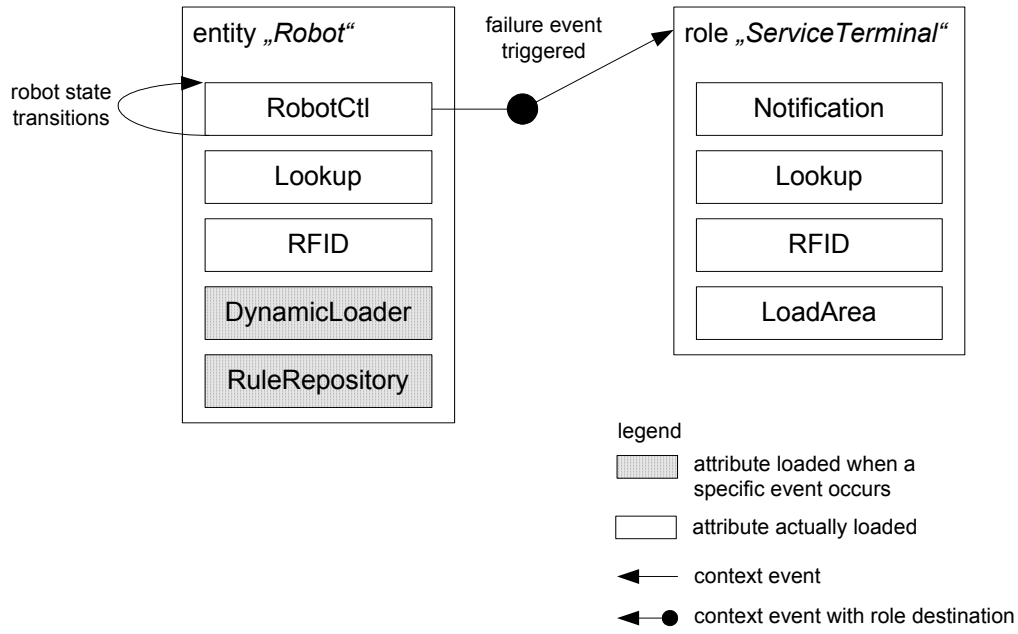

Fig. 63 Industrial machine (robot model) triggers a failure event to any service terminal

The *RuleRepository* attribute, of the entity *Robot*, is loaded automatically, when the *RFID* attribute encounters that service personal is around the machine. To load the *RuleRepository* attribute on demand means to increase the security of the machine according to changes from third persons. The dynamic loading and unloading of attributes can also be specified with context rules, which means that this mechanism is extremely flexible. In the maintenance scenario these attributes are loaded when a registered RFID transponder of a service technician appears in the range of the machine's RFID reader. They are unloaded when all transponders leave the range of the machine's RFID reader.

The reuse of the *LoadArea* attribute, which was used in the office scenario to display RFID tagged objects, adds additional functionality to the maintenance scenario. A service terminal receives failure events from the industrial machines and automatically displays the failure information as well as an HTML page with recovery information. When a service engineer moves his mobile device over the RFID reader of the service terminal a load area icon is displayed. The service engineer is now able to drag the failure description page onto the load area, in order to automatically take the information with him.

Fig. 64 shows a failure description page informing about failure number 3003, which also offers a repair procedure that a service technican can execute. The HTML form has also a text field into which the service engineer can enter a new context rule, which can then be inserted into the machine's rule repository. The machine's behavior can therefore be changed at runtime, which offers a convenient way to repair the machine.

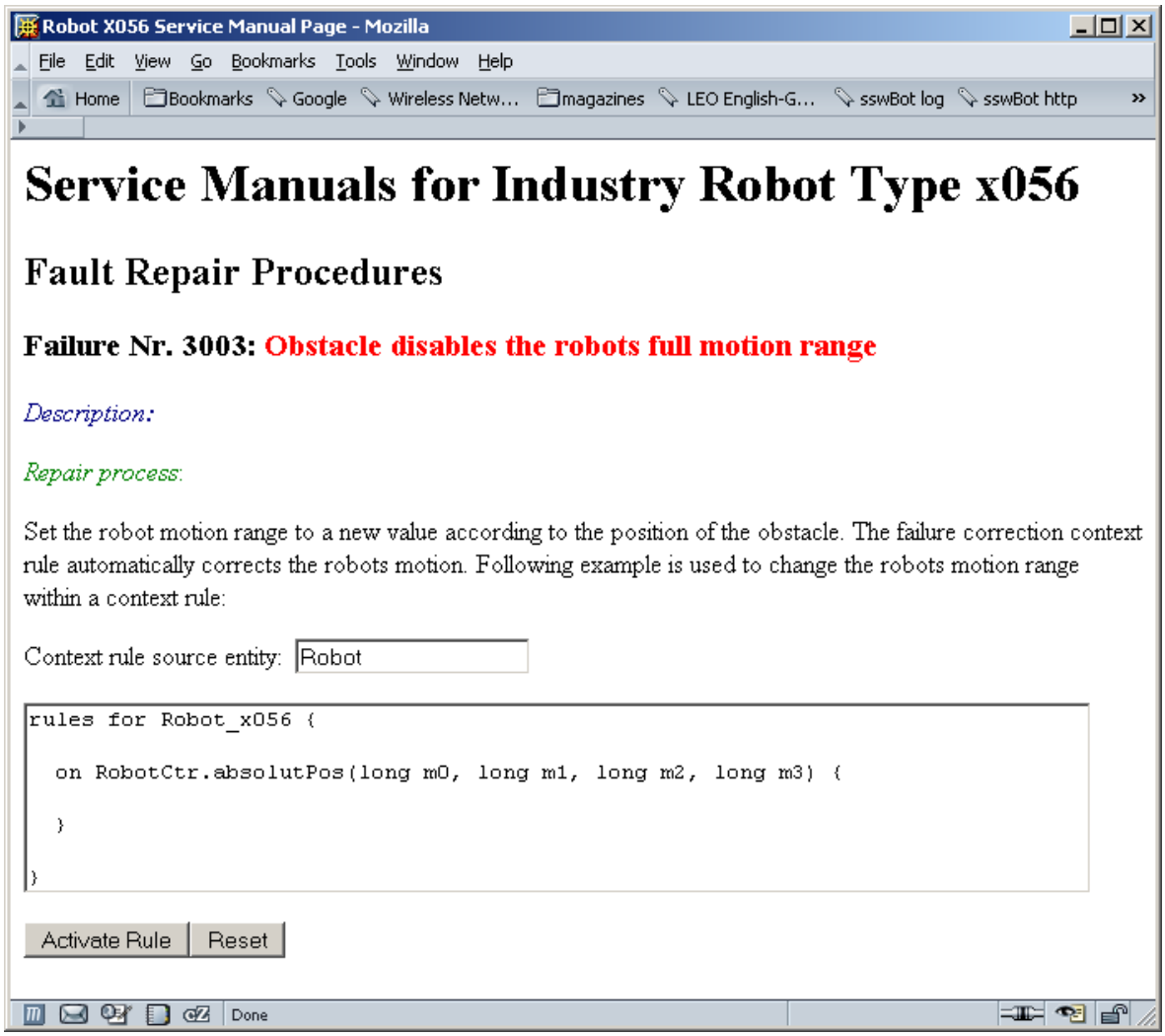

Fig. 64 Sample failure description page with dynamic repair rule

The following context rules enable the appearance of a load area and the mapping of resources with a given RFID ID.

```
rules for ServiceTerminal {
    string idg;

    on HttpResourceAtr.ResolvedResourceEvent(string rname) {
        ServiceTerminal.LoadArea.addResource(rname);
    }

    on MouseDropArea.ResourceDropEvent(string fileName) {
        ServiceTerminal.HttpResourceAtr.uploadResource(idg,fileName);
        ServiceTerminal.LoadArea.setImageURL("http://192.168.25.1/context/"+idg+"/icon.gif");
        ServiceTerminal.LoadArea.show();
        ServiceTerminal.LoadArea.addResource(fileName);
    }

    on MouseDropArea.ContainerClearEvent() {
        ServiceTerminal.HttpResourceAtr.resourceClear(idg);
    }

    on MouseDropArea.MenuChooseEvent(string fileName) {
```

```
        ServiceTerminal.SystemCalls.explore("http://192.168.25.1/context/"+idg+"/"+fileName);
    }

    on Rfid.TagDisappeared(string id) {
        ServiceTerminal.LoadArea.clearResources();
        ServiceTerminal.LoadArea.hide();
    }

    on Rfid.TagAppeared(string id) {
        idg = id;
        if (ServiceTerminal.HttpResourceAtr.CFresourceExists(id)) {
            ServiceTerminal.HttpResourceAtr.resolveID(id);
        } else {
            ServiceTerminal.HttpResourceAtr.registerID(id);
        }
        ServiceTerminal.LoadArea.setImageURL("URL/"+id+"/icon.gif");
        ServiceTerminal.LoadArea.show();
    }
}
```

6.5 A Mobile Robot Control Scenario

The mobile robot control scenario was designed to show how an autonomous, mobile device can be controlled by context rules. The mobile device communicates over a WLAN connection and is able to move around on the university campus. The university campus is completely covered with WLAN access points, so that the mobile robot always has a managed WLAN connection with the Internet. According to a wide range of different sensors that are mounted on the mobile robot, the control of this device is a good example for the power of the SiLiCon framework.

6.5.1 Hardware Setup

The core of the mobile robot scenario is built on an integrated PC104 industrial computer stack that offers a 200 MHz CPU. The PC104 stack that is connected to this CPU module contains a GPS and GSM module, a digital and analog IO board, a magnetic field compass and a pitch sensor. The sensor phalanx, with which the mobile robot discovers its environment, contains 6 ultrasonic distance sensors, 8 infrared short range distance sensors, 6 light sensors, 4 bumpers and one web camera. Fig. 65 shows the assembled mobile robot.

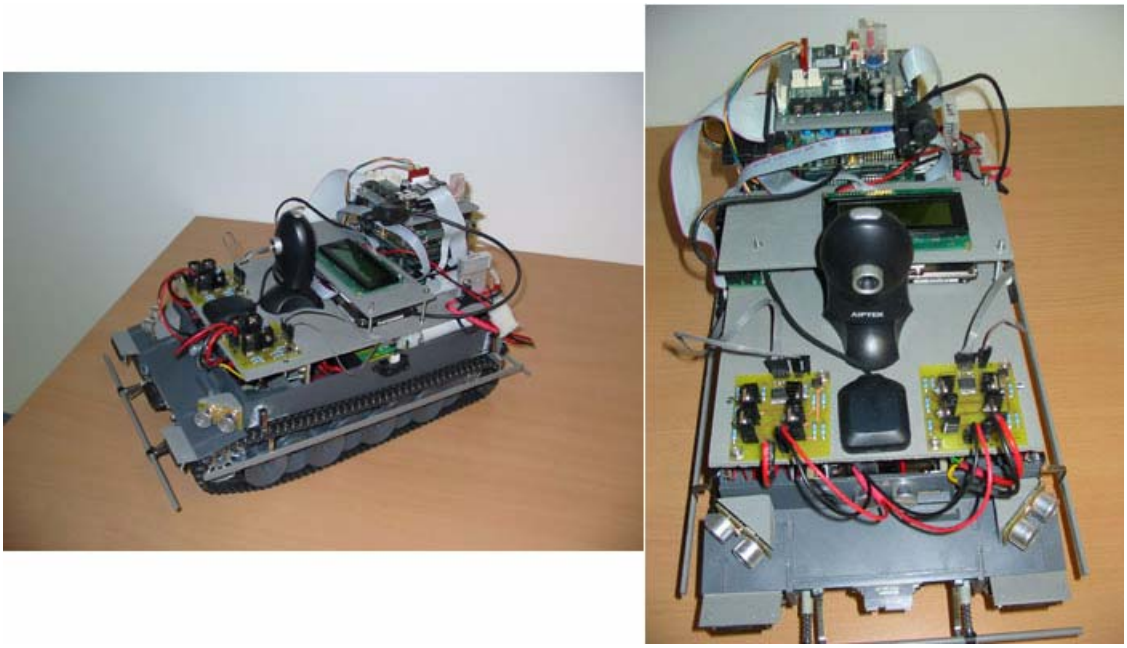

Fig. 65 Pictures showing the assembled mobile robot

7 Conclusions

This chapter gives a summary of the goals that were achieved and an overview about possible future enhancements, that could be integrated into the SiLiCon framework. Due to a lack of manpower some interesting features, such as a *visual context scenario builder*, were only discussed but not implemented until now.

7.1 Summary

In this PhD work a software framework was developed that supports the design and implementation of context-aware applications in pervasive environments. The framework supports a hierarchically organized collection of entities and manages their life cycles as well as the object migration process.

The framework is completely event-based, which means that all entities in a context-aware scenario are communicating via events. A pluggable transport and encoding layer is introduced, that supports the integration of various transport protocols and data encodings. Events are transported transparently with any chosen transport and encoding module. The transparent transport mechanism in combination with the event communication enables a simple and dynamic distribution of complex interaction scenarios. Two example transport and encoding modules were implemented: the HTTP and TCP transport modules and the serialized XML and SOAP encodings. By using the HTTP transport protocol in combination with the SOAP event encoding module the framework offers full web service integration.

The framework offers a hybrid lookup module, which is necessary to discover other lookup modules on distributed devices within the local subnet. The hybrid nature of the lookup also allows the registration of specific central lookup addresses, to decrease the scalability problems of broadcast or multicast discovery.

The framework covers the whole context-awareness life cycle reaching from the gathering of raw context information, over context transformation to actuator triggering. Context attributes wrap sensor and actuator hardware. A rich collection of sensor and actuator abstractions, as well as their reuse, enables rapid application development for context-aware systems.

The major advantage of the framework—compared to other solutions—is that all interactions are modeled by interpreted ECA (Event Condition Action) context rules. Interpreted context rules provide a flexible solution for the definition of complex, distributed scenarios. The framework allows the dynamic definition and deployment of new rules and change of existing rules at run time.

The framework allows the dynamic classification of discovered entities according to a role-based model, which solves many of the problems that appear with the use of a static clas-

sification hierarchy. Entities are able to act in more than one role and applications have the possibility to define new, specific roles at runtime. The framework does not need a central class hierarchy which has to be consistent with respect to all distributed entities.

Last but not least, the framework is designed to run on mobile and embedded devices, which means that it is optimized towards resource-efficient operation.

7.2 Future Work

This section gives an overview about possible future innovations in the SiLiCon framework.

7.2.1 Visual Builder Tool for Context Scenarios

The SiLiCon framework offers rich configuration possibilities for the definition of new context scenarios. A context scenario, which consists of reused sensor and actuator abstractions and context rules, can be designed without compiling a single line of program code. These powerful scenario configuration possibilities call for a visual builder tool. It is even thinkable to define plugins for existing visual building environments in order to design context scenarios with them (e.g. Microsoft Visio).

A visual builder environment could also discover and visualize existing scenarios in a local subnet. Since the SiLiCon framework allows the adition of new context rules to running context-aware systems, it is simple to change the scenario behavior with a visual builder tool. A prototype of a visual context scenario builder was already implemented and successfully used for scenario development and debugging purposes.

A complete implementation of a visual builder tool could manage the design of new entities and attributes, as well as provide an attribute repository of already existing sensor and actuator abstractions. Furthermore, the visual builder tool could support the design of context rules by drag and drop of entities and attributes. A visual builder tool could also manage the deployment of entire scenarios at runtime and the gathering and visualization of already running scenarios.

7.3 Security Considerations

One of the most underestimated issues of pervasive environments and P2P computing in ad-hoc networks concerns the security. Since ad-hoc networks do not offer the possibility to contact a global trusted authority (e.g. a PGP key server), it is hard to verify the identity and the trustworthyness of a communication partner. On the other hand, local ACLs (Access Control Lists) that contain the public key of trusted communication partners could solve this problem. A problem with ACLs that are stored locally on a device is the consistency and that objects that are not in the list are excluded from being communication partners.

Another approach is the use of a bonus and malus system that remembers successful interactions with communication partners and penalizes failed or defective interactions.

For the SiLiCon framework we considered integrating the ACL approach where the user is able to specify or to accept discovered communication partners. Additionally, it was dis-

cussed to implement a policy-based mechanism to control the access of resources on a finer granular level.

The actual implementation of the SiLiCon framework offers interfaces that were designed to integrate ACL and policy-controlled resource access but at the moment there is no concrete implementation of such security modules.

7.4 Rule Consistency Checks and Advanced Reasoning

A problem that occurs with context rules as interpreted state transitions, is to keep the rules consistent. A scenario designer could specify two rules which react on the same event with contradictory actions. It is also possible that rules lead indirectly to the same contradictory action. Generally, contradictory actions are the result of an incorrect scenario modelling, but it is hard for a scenario designer to keep track of all context rules in a large distributed scenario without any tool support.

A possible solution for the consistency problem on local machines would be the use of an existing expert system shell (e.g. the Java Expert System Shell, short JESS [JESS]). A state transition would be directly routed to the expert system shell, where all the ECA rules are declared and checked for consistency. The expert system shell is then responsible for the resulting action.

To use an expert system shell for controlling the ECA state transitions offers another interesting advantage. It is possible to reason about the current state of an entity and about possible future states.

The expert system shell solution for rule consistency checking works only on local devices. Distributed consistency checks are hard to realize and in ad-hoc networks the problem is even worse.

8 References

[ABa] Want R., Hopper A., Falcao V., Gibbons J., The Active Badge Location System, ACM Transactions on Information Systems, Vol. 10, No. 1, January 1992, pp 91-102.

[Ap97] Appel W. Andrew, Modern Compiler Implementation in Java, Camebridge University Press, 1997.

[Ar99] Arnold D. et al, "How and Why You Will Talk to Your Tomatoes", Proceedings of the Embedded Systems Workshop Cambridge, Massachusetts, USA, March 29–31, 1999.

[Be03] W. Beer, V. Christian, A. Ferscha, L. Mehrmann, "Modeling Context-aware Behavior by Interpreted ECA Rules", Euro-Par 2003, Springer Verlag, LNCS 2790, pp. 1064-1073, 2003.

[Bo00] G. Borriello, R. Want: Embedded Computation Meets the World-Wide-Web, Communications of the ACM, May 2000, Vol. 43 No.5. pp59-66.

[Br02] Bray J. and Sturman F. Ch., "BLUETOOTH, Connect Without Cables", Prentice Hall 2001.

[Co99] Corson S., Macker J., Cirincione G., Internet based mobile ad hoc networking, IEEE Internet Computing, IEEE 1999.

[CORBA] Common Object Request Broker Architecture (CORBA), www.corba.org.

[Da02] C. Dabrowski and K. Mills, "Understanding Self-healing in Service Discovery Systems", Published in Proceedings of the First ACM SigSoft Workshop on Self-healing Systems (WOSS '02), November 18-19, 2002, Charleston, South Carolina, ACM Press, pp. 15-20.

[Dey01] Anind K. Dey, Daniel Salber and Gregory D. Abowd, A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications, Human-Computer Interaction (HCI) Journal, Volume 16 (2-4), 2001, pp. 97-166.

[EBNF] International Organization for Standardization, Extended BNF, ISO/IEC 14977, http://www.iso.org/iso.

[Enhydra] Enhydra Open Source Java/XML application server, founder of kXML and kSOAP, www.enhydra.org.

[Fer03] A. Ferscha, S. Vogl, W. Beer, "Context Sensing, Aggregation, Representation and Exploitation in Wireless Networks", Future Generation Computing Systems, North Holland, 2003.

[Fer02] A. Ferscha, S. Vogl, W. Beer, "Ubiquitous Context Sensing in Wireless Environments", 4th DAPSYS (Austrian-Hungarian Workshop on Distributed

and Parallel Systems), Kluwer Academic Publisher, 1-4020-7209-0, 88-106, 2002.

[Fer01] A. Ferscha, W. Beer, W. Narzt, "Location Awareness in Community Wireless LANs", Informatik 2001: Wirtschaft und Wissenschaft in der Network Economy - Visionen und Wirklichkeit, Tagungsband der GI/OCG-Jahrestagung, 25.-28.September 2001, Universität Wien, 3-85403-157-2, 190-195, September 2001.

[Fi00] Finkenzeller Klaus, "RFID-Handbuch, Grundlagen und praktische Anwendungen induktiver Funkanlagen, Transponder und kontaktloser Chipkarten", Hanser Verlag 2000.

[Frost] Frost and Sullivan, Bluetooth Market Competitive Analysis, http://www.wireless.frost.com/.

[Gnut] Matei Ripeanu and Ian Foster, Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design, 1st International Workshop on Peer-to-Peer Systems (IPTPS'02), Cambridge, Massachusetts, March 2002.

[He01] Hendrickson K. et al., "Infrastructure for Pervasive Computing: Challenges", Tagungsband der GI/OCG-Jahrestagung Sep. 2001, Wien.

[Ho02] Hodes, Czerwinski, Zhao, Joseph, and Katz. "An architecture for secure wide-area service discovery", Wireless Networks, March 2002.

[HP01] Tim Kindberg et. al., People, Places, Things: Web Presence for the Real World, HP Laboratories Palo Alto, HPL-2001-279, October 31st , 2001.

[Ipi01] Diego López de Ipiña, Sai-Lai Lo , „Sentient Computing for Everyone" Third IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS, Series ISBN 0-7923-7481-9, Pages 41-54, Kluwer Academic Publishers, September 2001).

[IRIS] IRIS: Infrastructure for Resilient Internet Systems, Massachusetts Institute of Technology, http://iris.lcs.mit.edu/.

[Ishii97] Ishii, H. and Ullmer, B. "Tangible Bits: Towards Seamless Interfaces between People, Bits, and Atoms." In Proceedings of CHI'97, pp. 234-241.

[Jini] Sun Microsystems Jini Technology, http://wwws.sun.com/software/jini/

[JXTA] Sun Microsystem's Java based P2P development framework JXTA, www.jxta.org.

[kXML] XML processing library for mobile devices, http://kxml.enhydra.org/

[MoCo] Jörg Roth, "Mobile Computing", dpunkt.verlag 2002.

[Nap] Napster, P2P resource sharing application, www.napster.com.

[Onto] On To Knowledge Organisation, http://www.ontoknowldge.org

[ORL] Ward A., Jones A., Hopper A.. A New Location Technique for the Active Office. IEEE Personal Communications, Vol. 4, No. 5, October 1997, pp. 42-47.

[OSI] International Standard Organisation ISO/OSI Open System Connection, www.iso.org.

[Oxy] Project Oxygen, Massachusetts Institute of Technology, http://oxygen.lcs.mit.edu.

[P2PHP] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja1, Jim Pruyne, Bruno Richard, Sami Rollins 2 , Zhichen Xu, Peer-to-Peer Computing, HP Laboratories Palo Alto HPL-2002-57 March 8th , 2002.

[P2PSim] A simulator for peer-to-peer protocols, Massachusetts Institute of Technology, http://www.pdos.lcs.mit.edu/p2psim/.

[ParcTab] The Xerox PARCTAB, http://www.ubiq.com/parctab/.

[Per01] Hansmann U., Merk L., Nicklous M., Stober Th., "Pervasive Computing Handbook", Springer-Verlag Berlin 2001.

[PJSpec] Java Community Process, JSR-000062 Personal Profile Specification, http://jcp.org/aboutJava/communityprocess/final/jsr062/.

[PTab95] Roy Want, Bill N. Schilit, Norman I. Adams, Rich Gold, Karin Petersen, David Goldberg, John R. Ellis and Mark Weiser, The PARCTAB Ubiquitous Computing Experiment, Technical Report CSL-95-1, Xerox Palo Alto Research Center, March 1995.

[RDF] World Wide Web Consortium, Resource Description Framework (RDF), http://www.w3c.org/RDF/.

[RFC2131] RFC 2131, Dynamic Host Configuration Protocol. R. Droms. March 1997.

[RFC2290] RFC 2290, Mobile-IPv4 Configuration Option for PPP IPCP. J. Solomon, S. Glass. February 1998.

[Sch94] Schilit, B., Adams, N., and Want, R. Context-Aware Computing Applications.

[Sen] Sentient Computing Project, AT&T and Cambridge University Engineering Department.

[SeWeb] Semantic Web Organisation, http://www.semanticweb.org/resources.html#publications

[SmarTire] SmarTire Systems, company that produces tire monitoring systems for the automotive and transportation industries, www.smartire.com.

[SOAP] World Wide Web Consortium, Simple Object Access Protocol (SOAP), http://www.w3.org/TR/2003/REC-soap12-part1-20030624/.

[Tex] Texas Instruments RFID transponder form factors, http://www.ti.com/tiris/docs/products/transponders/transponders.shtml

[**Toh**] C.-K. Toh, Richard Chen, Minar Delwar, and Donald A l1e n, Experimenting with an Ad-Hoc Wireless Network on Campus: Insights and Experiences, School of Electrical and Computer Engineering.

[UDDI] Universal Description, Discovery and Integration (UDDI), http://www.uddi.org.

[Ull01] Ullmer, B. and Ishii, H. "Emerging Frameworks for Tangible User Interfaces" In "Human-Computer Interaction in the New Millenium" Addison-Wesley, August 2001, pp. 579-601.

[Under99] Underkoffler, J., Ishii, H. “Urp: A Luminous-Tangible Workbench for Urban Planning and Design.” In Proceedings of CHI’99, pp. 386-393.

[Wa00] B. Warneke, B. Atwood, K.S.J. Pister, "Preliminary Smart Dust Mote," Hot Chips 12, Palo Alto, California, August 13-15, 2000.

[Wa02] Want, R., Pering, T., Borriello, G., Farkas, K., "Disappearing Hardware", IEEE, Pervasive Computing Journal, Vol. 1. Issue 1, pp36-47, April 2002.

[WebWall] A. Ferscha, G. Kathan, S. Vogl, WebWall - An Architecture for Public Display WWW Services, WWW2002: Middleware and Applications, Honolulu, Hawaii, USA, May 2002.

[Wei91] Mark Weiser, The Computer for the Twenty-First Century, Scientific American, September 1991.

[Wei93] M. Weiser, Some Computer Science Issues in Ubiquitous Computing. Communications of the ACM, 36(7), 1993, pp. 74-84.

[WSDL] World Wide Web Consortium, Web Service Description Language (WSDL), http://www.w3.org/2002/ws/desc/.

[XSD] World Wide Web Consortium, XML Schema Definition language, http://www.w3.org/XML/Schema.

Lebenslauf

Persönliche Daten

Name	Wolfgang Beer
Geburtsdatum	03.08.1977
Geburtsort	Steyr

Ausbildung

1987 - 1995	Bundesrealgymnasium Kirchdorf/Krems
1995 - 2000	Studium der Informatik an der Johannes Kepler Universität Linz
2000 - 2004	Doktoratsstudium an der Johannes Kepler Universität Linz

Berufslaufbahn

1997	Technologiepraktikant im Technologiezentrum Steyr
1998	Technologiepraktikant im Technologiezentrum Steyr
1999	Java Entwickler im Futurelab des Ars Electronica Centers Linz
1999	Werkstudent bei Siemens München/Perlach, Abteilung Zentrale Technik Software Engineering 1 (CT SE1)
2000	Projektmitarbeiter Ars Elektronica Center (WiFi-Projekt)
2000 - 2004	Assistenzstelle am Institut für Systemsoftware an der Johannes Kepler Universität bei Prof. Mössenböck
2000 - 2004	Mitarbeiter in einem Kooperationsprojekt mit Siemens München (Abteilung CT SE2)