WeKnowIt

Emerging, Collective Intelligence for Personal, Organisational and Social Use

FP7-215453

D4.2

Prototype of community management platform

Dissemination level	Confidential
Contractual date of delivery	30.06.09
Actual date of delivery	30.06.09
Work package	WP4 Social Intelligence
Task	T4.2 Community Administration Platform
Type	Prototype
Approval Status	Approved
Version	11
Number of pages	78
Filename	D4.2-man_2009-6-30_v12_emka_deliverable-CAP.odt

Abstract

The Prototype of the Community Administration Platform (CAP) enables the WKI system to use a powerful, flexible access rights engine to define access control to resources under definable conditions. The CAP is represented by a formal language, the Community Design Language (CDL).

co-funded by the European Union

History

Version	Date	Reason	Revised by
1	26.05.09	Creation / Structure	Andreas Sonnenbichler
2	04.06.09	Added section Introduction	Andreas Sonnenbichler
3	10.06.09	Added section CAP, Methodology	Andreas Sonnenbichler
4	23.06.09	Revised all sections	Andreas Sonnenbichler
5	25.06.09	Revised sections 2.2.2, 3, 9	Andreas Sonnenbichler
	25.06.09	Added section 6 (UoKob)	Felix Schwagereit
6	29.06.09	Revised all sections	Andreas Sonnenbichler
7	29.06.09	Revised all sections, changed order	Felix Schwagereit
8	29.06.09	Added section Implementation	Andreas Sonnenbichler
9	29.06.09	Merged Changes in V7 and V8	Andreas Sonnenbichler
10	30.06.09	Revised Sections 1 and 8	Andreas Sonnenbichler Felix Schwagereit
11	30.06.09	Changed CAP Architecture	Andreas Sonnenbichler

Author list

Organization	Name	Contact Information
EMKA	Andreas Sonnenbichler	University of Karlsruhe, Germany
UoKob	Felix Schwagereit	University of Koblenz, Germany

Executive Summary

The Prototype of the Community Administration Platform (CAP) enables the WKI system to use a powerful, flexible access rights engine to define access control to resources under definable conditions. The CAP is represented by a formal language, the Community Design Language (CDL).

The targets we want to reach are:

- Expressive Power: The CAP shall be defined in such a way that different use cases can be supported by the same architecture.
- Flexibility: The CAP shall be able to represent different use case scenarios. Especially the Emergency Use Case Scenario and Consumer Group Scenario of the WeKnowIt Project. Furthermore, for later exploitation, a broad general support for use cases shall be.
- Explicit Definitions: The CAP is represented by a formal language, the CDL. By changing formal definitions in the CDL, the CAP is customizable. By this the access rights engine adapts to changed situations.
- Robustness: The CAP shall be robust in case of user mistakes, system failures or other unforeseen events.
- Performance: Checking and changing access rights shall be fast and with good performance.
- Scalability: The CAP must scale with high object numbers and rule sets.

As a theoretical base for social roles in virtual online communities a Community Membership Life Cycle Model is suggested. This model describes typical roles in a community and links it to authorizational roles. By certain properties, automatic role assignment can take place, enabling community members defined roles.

The technical implementation is done in six layers: Persistence Storage, Persistence Access, Caching, Core Component, Parser and Web Service, completed by an RDF-layer.

Abbreviations and Acronyms

AC	Access Condition
ACL	Access Control List
AJAX	Asynchronous JavaScript and XML
BNF	Backus-Naur-Form
CAP	Community Administration Platform
CAT	Community Analysis Tool
CDL	Community Design Language
CMLM	Community Membership Life Cycle Model
CMS	Content Management System
EC	Emergency Case
ECL	Emergency Case Leader
ECM	Emergency Case Member
iAID	internal Access Condition Identifier
iEID	internal Element Identifier
iRID	internal Relation Identifier
iSID	internal Set Identifier
iTID	internal Test Identifier
RBAC	Role Based Access Control
RDF	Resource Description Framework
SNA	Social Network Analysis
SPARQL	SPARQL Protocol and RDF Query Language
URI	Uniform Resource Identifier
WKI	WeKnowIt
WP	Work Package
XML	eXtensible Markup Language

Table of Contents

Illustration Index

Index of Tables

1. Introduction

This deliverable is part of the "WeKnowIt" (WKI) Project belonging to work package 4 "Social Intelligence", Task 4.2 "Community Administration Platform". In the project's Description of Work (DoW) the task is described as follows

"Technically, this task uses the addition of community design languages as a new representation for Social Intelligence (and the institutional setup) and fine-grained right-models which efficiently describe the lattice structures of users, objects and rights and which support efficient contracting and delegation of rights between users. Rights sets provide the possibility of differentiation of social services like tagging and semantic enrichment along the group structure dimension. These instruments support the self-organisation of groups and communities. UoKob is responsible for the mapping of the community administration to the semantic layer and in cooperation with EM-KA for the development of the community design language. EM-KA is responsible for the design and implementation of the community administration. USFD is responsible for the user interface design of the community administration tool." [*1*]

In this section we will give an overview of the role of the CAP within the WeKnowIt project, introduce important terms like Virtual Community, Services, Authorizations and Roles. In section 2 we will present the idea of a Community Member Life Cycle Model by analysing state-of-the-art work. We continue the chapter by a comparison of the models and suggest an own model. The section is finished with an analysis of current web services like Twitter[1] and Xing[2]. Section 3 will summarize existing approaches to authorization. In section 4 we will introduce the Community Design Language and its methodology. The RDF-layer of the CAP is described in section 5, followed by section 6 describing the technical implementation. Chapter 7 demonstrates the Community Language (CDL) with a use case showing step by step the structure and usage of the CAP. After presenting a conclusion on chapter 8, in section 9 the CDL syntax is described by an extended Backus-Naur-Form (BNF).

1.1. Integration in the WeKnowIt Project

For almost any IT system, access control is an important issue. Especially for modern, state-of-the-art social networking platforms like Yahoo![3],

1 **http://www.twitter.com**, last accessed 28.6.2009

2 **http://www.xing.com**, last accessed 12.6.2009

3 **http://www.yahoo.com**, last accessed 30.6.2009

Facebook[4] and Flickr![5] a flexible, reliable, manageable and easy way of access control is important. Personal data and media has to be protected. User often want to define in detail, who can access their media, e.g. holiday photos.

The same applies to the WeKnowIt project. Two WKI Use Case Scenarios have been defined, an Emergency Response Scenario and a Consumer Group Scenario (see [2]). Both scenarios describe the usage of media objects like images, videos, text documents and audio streams. It is obvious, that an access rights concept (a policy) is necessary to allow or disallow access to these resources.

Besides these media resources, different services will be provided: Tagging services (WP2), image analysis services (WP2), mass data analysis (WP3), social network analysis (WP4) and organizational services (WP5). Not every service will and shall be available for every user under any circumstance. To limit access to these services, again, an access rights concept is necessary.

The definition of the policies are done by the Use Case Scenarios and their users, as here the requirements of the use cases are described. The technical implementation of policies in contrast is describe in this document.

Although focusing in the WeKnowIt project on these two use cases, the Community Administration Platform (CAP) has been designed in a general way, to be able to support use cases which a not in the scope of the project. We therefore developed a new methodology of access rights representation allowing a flexible way of defining and testing access rights.

Two services will be provided for all other WKI work packages: The CAP service is providing access rights definition, testing and querying via the Community Design Language (CDL), a formal language. The second service provided by the RDF-Layer offers semantic queries on the defined access rights.

1.2. Virtual Communities

The phrase **Virtual Community** was first used by Rheingold. *"Virtual communities are social aggregations that emerge from the Net when enough people carry on those public discussions long enough, with*

4 **http://www.facebook.com**, last accessed 30.6.2009

5 **http://www.flickr.com**, last accessed 30.6.2009

sufficient human feeling, to form webs of personal relationships in cyberspace'' (see [3], p. XX[6]).

A virtual community supported by computers and realized in the net does not only consist of people. Of course, people are the heart of any community. On the other hand, what these people can do and how they can do it, is highly influenced by the environment, they are embedded in. In a classic, non-virtual community the possibilities and actions, the members can take, are quite different, if the community is a stone-aged hunters' tribe or a today's New York's party scene. The same applies to virtual communities, at least partly embedded, limited and even driven by computer systems. The technical environment is determined by at least two axiom sets: The implicit axiom set of the system designers. The people, who designed and implemented the system a virtual community uses, define in many ways the possibilities and limits, community members find. Secondly an explicit axiom set is defined by the **Community Administration Platform**.

A Community Administration Platform reflects the ability to customize, to adapt the supporting system to the wishes of its users. The more flexible a community administration platform is, the less implicit axioms have to be assumed by the system designers. The formal representation of "the design" of a community administration platform we will call the **Community Design Language**.

As an example, a very simple Community Administration Platform concerning access rights allows only a group of administrators to assign access rights to a user. Any time, a new user enters the system, an administrator has to assign him access rights that will allow him to use services offered by the system. This fact "all access rights are assigned by administrators" can be modelled in a formal language, the Community Design Language.

In a different environment, administrators grant community moderators the right, to assign access rights to new users. For this, administrators give a "grant right" to those moderators. Again, this fact can be modelled in the Community Design Language. In this second case, the moderators are able to assign users access rights, but can not delegate the right to administrate users' access rights to users.

In a third example, administrators grant moderators the right to administrate user rights including the right, to enable those users themselves to administrate other users concerning their access rights.

Our approach therefore is, to describe all this different behaviours by a formal Community Design Language.

6 This is not a typo. The page number is indeed "XX".

1.3. Services

A Community Administration Platform consists of several components. We want to define **Services** as functions that are directly supported within the community. A community member should be able to do something within the community. Let us assume, a user wants to create a new community and send an email to some friends in order to invite them. This service can be either performed outside the community system, e.g. by his favourite email application, or - integrated - inside the community system. In the latter case the community system must provide the service "email friends about a community".

Services can be atomic or can be a composition of other services. For example, the service "Create a new community" can use the service "email friends about a community" as one functional step.

1.4. Authorative and Social Roles

Another component, besides services are access rights. They decide if a user is allowed to execute a service. A very common approach to realize access rights are authorative roles. These are collections of authorizations, which can be assigned to a user (see for example [4] and [5]). As an example, an authorative role "moderator" includes the access rights in a community platform, to moderate discussions. If a user A is assigned this role "moderator", the system allows A to close any discussion thread.

Obviously this authorative role corresponds to a social role: User A should be given this authorative role only, if and when he is or should be a moderator in discussions.

We can see very quickly, that social roles are closely related to authorative roles and thereby with authorizations: If we can measure the neutrality, fairness and the knowledge about the rules of a user in a discussion, we expect him to be a good moderator. So we assign him the authorative role of a moderator for this specific discussion. More general, we define preconditions for the social role "moderator". If we observe this preconditions in an automatic way, we can infer, a user is a social moderator. Then the system can assign the user the authorative role moderator. But only as long, as the preconditions can be observed. If the user looses neutrality, he is not a good moderator anymore and therefore he should not be one, thus he should not have the access rights to be one. One important issue we have to take into account is, how we can define measurements to find out, if a user is a good or bad moderator. We will come back to this later.

We have to distinguish between different definitions of a role. The first perception defines a social role. A **social role** we define as a quantity of

expectations of and towards a member of a community. These expectations include duties and rights of the role. A social role is seldom explicitly assigned to anybody, but derived from group processes.

Let us consider the role "boss". An expectation towards a boss is, that he tells his subordinates, what work to do. Together with this expectation, a duty is, to take care of the work done, control the progress and give feedback to the executive. His right therefore is, to be informed about any detail concerning the work to be done.

The second perception of a role is as a collection of authorizations. Technical access rights (e.g. to view specific data) is bundled to a role, which is assign to a user.

Both definitions we expect to be linked. We expect the authorative role can be inferred from the social role. This link between social and authorative roles is the base of our effort to identify all relevant social roles in an online community: If we can describe all relevant social roles in a community, we are able to deduce authorative roles. Innovative approaches, how access rights can be granted, can then be easily reached: Let us assume, we identify by some predefined measure that a user in the social role of a regular, normal user has gained a social leadership role in a specific community. Then the community administration platform can assign this user automatically the access rights of a community leader. These can be for example the right to moderate discussions, to ban users or to change attributes of the community. If the system identifies a user by his behaviour as a trouble maker (social role: trouble maker), the system can assign him the authorative role "trouble maker" forbidding him to access mostly all services of the system.

1.5. Cross Usage of (Social) Intelligence

Our idea of a major difference between a social and an authorative role is, that social roles can be observed by user behaviour. If we link both roles together, innovative approaches in authorizations management occur.

One of our research questions therefore is, to suggest a set of social roles. For every role we suggest measures from the Social Network Analysis and User Profiles to identify users which have this role. This will enabling the system to assign corresponding authorative roles. Measures from the Social Network Analysis can be provided by WeKnowIt's work package 4, Task 4.1 "Community Analysis Tool." User profile measures can be defined and provided by WP1 User Profiles. This cross-usage of intelligence – User Profiling, Social Network Analysis and the Community Administration Platform - identifying social user roles can be used to assign authorative roles.

2. A Community Member Life Cycle Model

We can reformulate this idea in other words: What social roles can a member potentially take during his participation in a community? Roles describe positions of members in a community and they do and what actions they take. If we succeed in identifying such roles, our target is furthermore to find a time-depending sequence. This sequence we call the "Community Member Life Cycle". This life cycle describes, what roles a fictive, ideal community member holds over time. Of course holding such a role, influences the services a member usually uses - and here the access rights come into play again: To be able to use such a service, the user must be authorized to do so. This idea is depicted in Illustration 1.

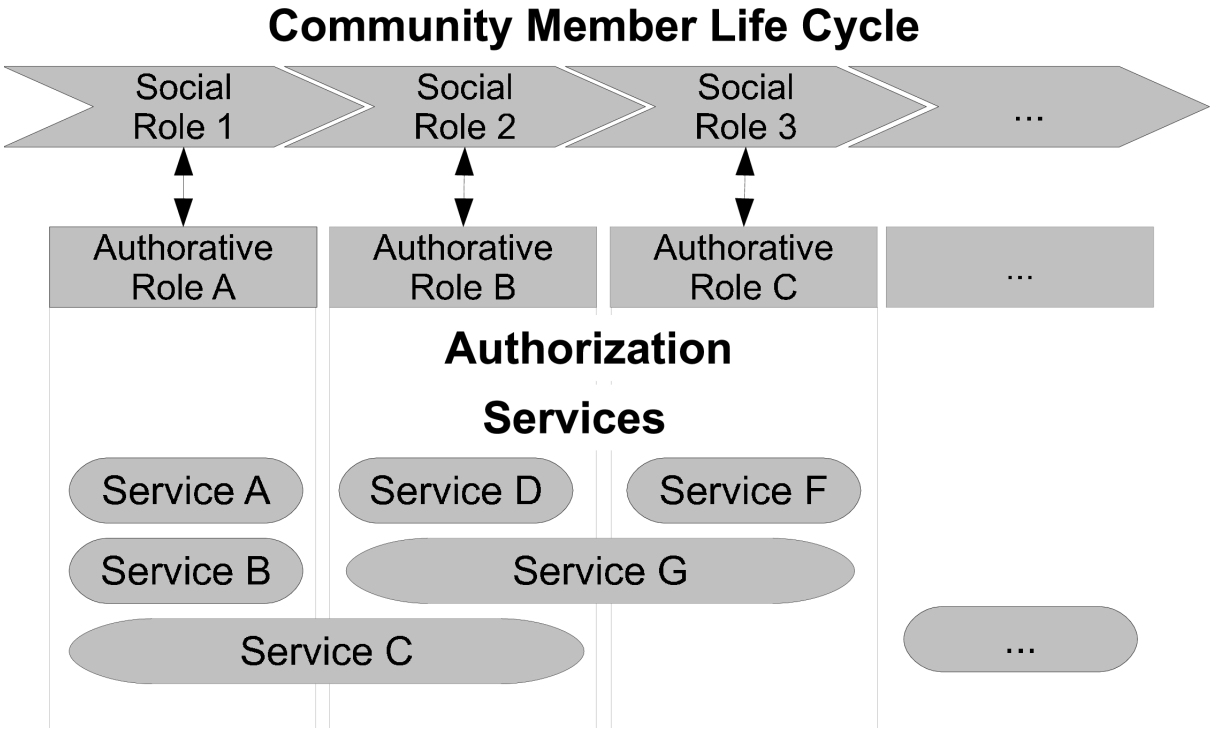

Illustration 1: A Community Member Life Cycle consists of a series of social roles. These are correlated to authorative roles, giving access rights to services of a Community Administration Platform.

A Community Member Life Cycle provides another advantage: All services a system supports can be called unstructured feature list. To identify services in a more structured procedure and to learn, for what social role which service is important (and must be executable) we use the Community Member Life Cycle.

In the following we give an overview to related work that has been done concerning Community Life Cycles. We then compare those models and

introduce our own approach. After that we apply our model to state of the art services like Twitter or Xing.

2.1. Related Work about Community Life Cycles

2.1.1. Small Groups Dynamics

One of the most cited works has been published by Tuckman [6]. His research is about stages in group development. He suggested the four stages forming, storming, norming and performing. Later in 1977 Jensen and Tuckman [7] suggested a fifth stage called adjourning.

In the **forming phase**, the group members orientate themselves and test the reactions of others. The **storming phase** is full of hidden or open conflicts partly with resistance in the group. This phase is followed by the **norming stage**, where each group member finds his place and norms for behaviour are determined and agreed on. After this a constructive **performing phase** follows, succeeded by the **adjourning stage** with anxiousness about leaving the group and feelings toward leaders and group members.

Tuckman did his research for this model on small groups and on group dynamics. The analysed groups consisted of roughly a dozen or less member. Of course, in the 60s and 70s virtual communities were not analysed.

2.1.2. Remote Masters Program Community

A mostly virtual community development is described by Haythornthwaite, Kazmer, Robins and Shoemaker [8].

A remote masters program offered by the University of Illinois consists of a boot camp, where students physically meet on the campus at the beginning of the program. From then on, the classes meet only virtually, coming together physically only once per year for a day. The virtual community formed by each cohort uses tools like Powerpoint for lectures, Internet Relay Chat (IRC) for questions and web boards for discussions and exercises. Clearly this is a closed community as only students of one season, a cohort, belong to it.

The initial phase at the boot camp can be seen as an **initial bonding** phase. Here, in a traditional way, contacts are established, group processes take place. After this come together a **Maintaining Presence** phase is observed. In this phase, *"maintaining ties and community at a distance [...] is perceived by students to require more effort than in a face-to-face community"* (see [8], p.17). The third phase described is the

Disengaging from the community. Here, as the community members "*progress through the program, the desperate need to make contact diminishes. They become familiar with [...] routines, the technologies and norms for their use, and their distanced companions and fellow travelers" (see [8],* p.23).

Other terms for the three phases can be found, Johnson [9] calls these phases **initial bonding**, **early membership** and **late membership**.

We will work with the initial notion used by Haythornthwaite et al.

This research was done on medium group sizes (about 20 members), in comparison to Tuckman, who did research on smaller groups. Haythornthwaite's model represents medium sized groups with a common target to receive the masters degree.

2.1.3. Communities as Products

Owyang did research on successful commercial online communities. So he "interviewed 17 people (many community leaders that you know) to find out the commonalities between successful communities" (see [10]). He determined a 7-phases life process, a successful community passes through (see Illustration 2).

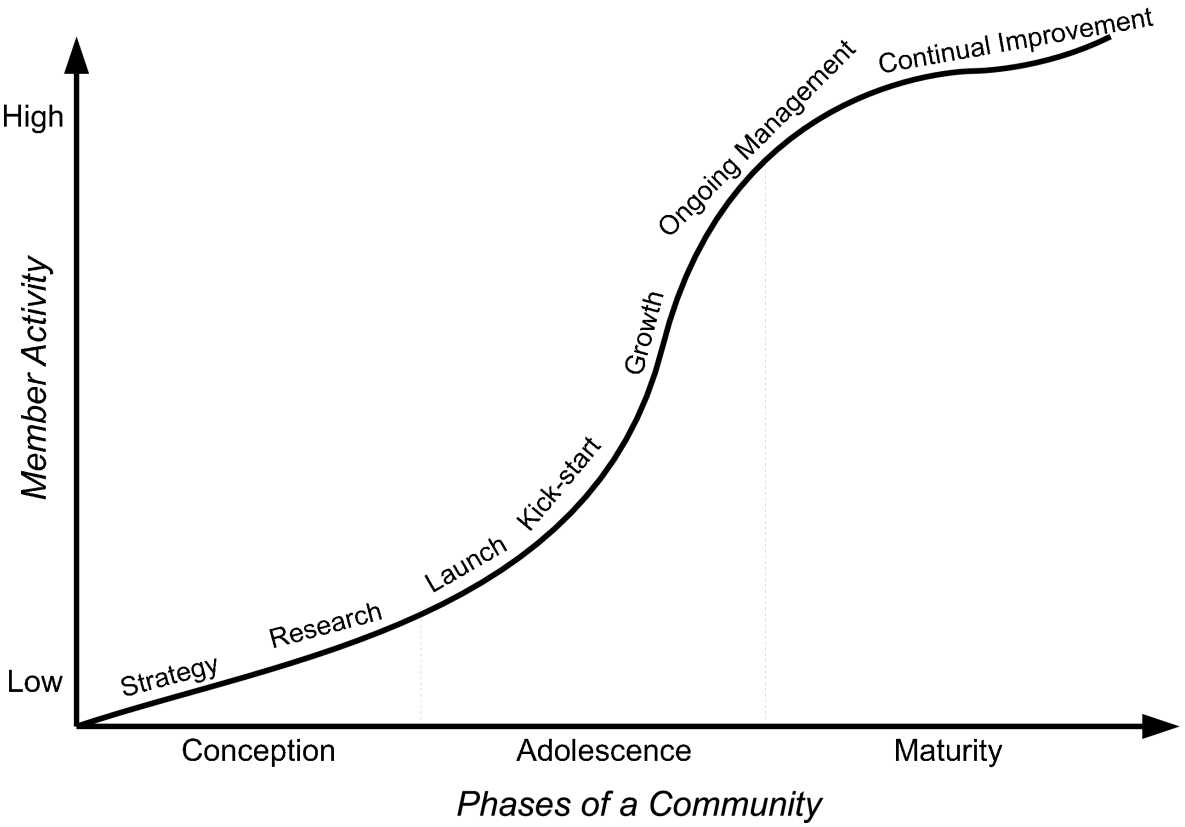

Illustration 2: Life process model of a successful online community by Owyang.

As one can see in Illustration 2, three major phases are distinguished, conception, adolescence and maturity. In the **conception phase** the stages strategy and research are performed, analysing market situation, target for the community and usage for the stakeholders. During launch and kick-start, as stages of the **adolescence phase**, the successful community gains most of its users which is critical for its success. At the end of this phase and also at the beginning of the **maturity phase**, ongoing management and continual improvements need to be done to keep the community successful.

Member Activity mapped on the y-axis is a relative measure, not explained by Owyang, we suggest the numbers of services (functions/activities) performed in a specific time slot in the community.

Owyang's research is mainly on large communities, which potentially have hundreds of members. A main focus of his work is about, how to steer processes and instruments necessary to gain a critical mass of community members, to have a commercially successful community.

2.1.4. Building Online Communities

A community-specific membership life cycle is described in detail by Kim (see [11]). She distinguishes five stages in three main life cycle steps (see Illustration 3).

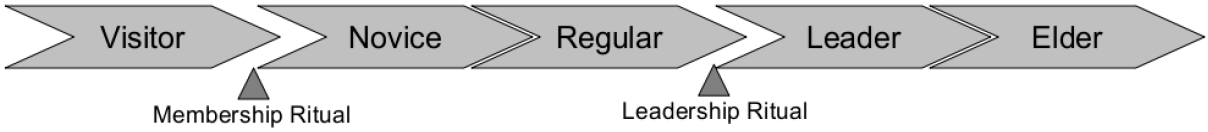

Illustration 3: Membership Life Cycles with five key stages by Kim.

Visitors are new, have not signed up for an account and do not have an identity in the community (yet). They are unfamiliar with local customs, techniques, behaviour and have many unanswered questions. *"For a first-time visitor, a thriving web community can seem confusing and overwhelming"* (see [11], p.120). Kim recommends a visitor centre, *"that helps visitors learn what they need to know and find what they're looking for."*

The next step in the membership life cycle are **Novices**. They do have an identity - have signed up for an account - in the community, but still have to learn the ropes and be introduced to the community life. Novices *"play a special role in the community with needs that are fundamentally different from those of more experienced members. Novices need to learn what they can do, whom they can do it with, where they can do with, and how they're expected to behave"* (see [11], p.133).

Becoming more experienced in the community, the member becomes a **Regular**. Regulars are established members, the mainstays of a community. They are building the base. Regulars know the environment and opportunities, know, how to find what they are looking for, how to personalize their interface and how to communicate efficiently with other members of the community. *"To grow a dynamic and successful community, [one] must continually convert novices into regulars. [One] can help this process along by rewarding members for continued involvement and by offering new opportunities to keep [the] members challenged and interested"* (see [11], p.140).

The fourth step of the membership life cycle are the **Leaders**. They are the ones, who *"help newcomers get settled in, operate the community shops and taverns, volunteer for charities and committees, and run for mayor"* (see [11], p.119). They answer questions and help members to solve problems with the system. Leaders plan, coordinate and run events in the community and might provide special resources or services to members. To do so, the Leaders need to be empowered by the community and the system to fulfil their role.

The last step in the membership life cycle are the **Elders**. *"Over time, some leaders will tire of their day-to-day activities and step down from their official roles. Because they're familiar with the history and inner workings of the community, they're now elders - respected sources of cultural knowledge and insider lore. Along with other long-time residents, they're the teachers and storytellers of the community, the people who give the place a sense of history, depth and soul"* (see [11], p.119). Elders are the community storytellers, the ones that have seen it before, know about backgrounds and reasons and can't wait to tell about it.

Kim gives us with the 5-stage life cycle a reasonable model to distinguish different user roles in already established communities of large size. Owyang's model was also about large-scale communities but with focus on how to make new communities successful. Kim describes user roles in established, large communities.

2.1.5. Online Learning Communities

Another description of roles in virtual communities is given by Palloff and Pratt (see [12]). They investigate inner mechanisms in online learning communities. Especially in discussions about new lectures' content three roles for the students are unfold: Knowledge Generation, Collaboration and Process Management.

Knowledge Generators are people, that actively assimilate knowledge by constructing new forms of knowledge or meaning. They even combine different knowledge together and present new results through these combinations. This is something, which clearly is not limited to learning communities. People, who know about other facts currently discussed or available, bring in new sources, new ideas and knowledge in the community. Therefore Knowledge Generators are not only a role in learning communities, but a very general role.

Collaborators are the second role, Palloff and Pratt suggest. Collaborators assist the community in making sure, that all voices are heard and all members are participating. They will allow a group not to forward until a consensus has been achieved and might use for this tools like web surveys and ratings. On the other hand collaborators do not work only within a community but also between them: So they connect two or more groups helping all of them through information and knowledge exchange. As we see again, collaborators are again not limited to learning communities. Every community might have or need people caring about that nobody is left behind or mediating knowledge between different groups.

Palloff and Pratt's third type are **Process Managers.** They are helping to maintain the process, slowing down discussions or progress if they feel, they or somebody else is lost, holding the direction, if discussions tend to get of the path, feeling generally responsible for the group moving in the right direction.

Palloff and Pratt give us three roles for learning communities, that can easily be adapted to general communities. Knowledge Generators, Collaborators and Process Managers can be found probably in any virtual community: People, that create and have new ideas, and make new suggestions; Collaborators, that take care about the group processes; Process Managers or moderators acting as facilitators to help a community to reach an explicit or implicit goal.

Palloff's research is a special case, as it is about learning communities, probably of small or medium size. In contrast to Tuckman and Haythornthwaite not the development of the group or the group processes are in the focus, but different roles of students in learning groups.

2.2. Comparison And Composition Of A Life Cycle Model

2.2.1. Comparison of Life Cycle Models

As we have seen in the last section, several descriptions of life cycle models for (virtual online) communities have been suggested. If we review these models closely, we notice that three different perceptions of the term "life cycle" exist:

- The first meaning focuses on the development of a single community member and the roles he takes over time. This applies to the model of Kim (see section 2.1.4.). For a clear definition, we want to call this perception **Community Member Life Cycle**.
- The second understanding focuses on group processes, a community undergoes during time. Here not so much the individual development is in a spotlight, but the group processes. This we will reference as **Group Life Cycle**. Examples for this perception are given by Tuckman (see section 2.1.1.) and Palloff (see section 2.1.5.).
- The third interpretation sees the community as a product, evolving over time, hopefully gaining users; a **Community Product Life**

Cycle. Owyang is a representative for this approach (see section 2.1.3.).

A comparison of different community life cycle models

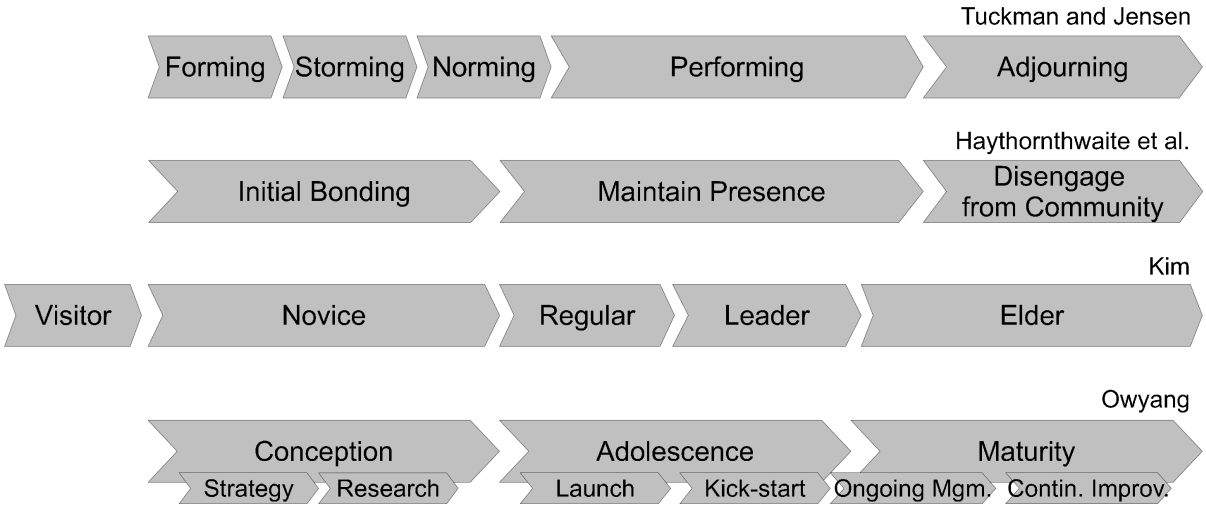

Illustration 4: This figure depicts a comparison of different life cycle models.

A comparison of the life cycles can be found in Illustration 4. As depicted the "roughest" model is given by Haythornthwaite et al. with three main phases "initial bonding", "maintaining presence" and "disengaging". These stages do not clearly define, which perception of the three interpretations concerning life cycles given above it belongs to. Actually Haythornthwaite et al. describe needs, community members have during the development of communities. So we can deduct, that there are three life cycle phases, that we could call "starting", "main phase" and "end", related to the three needs of the participants.

Tuckman and Jensen's model is finer grained, especially in the first stages. In comparison to Haythornthwaite's initial bonding phase, Tuckman ad Jensen split this phase into "forming", "storming" and also parts of the "norming" phase. "The need to present oneself to the group and define its own place", which Haythornthwaite describes her first phase corresponds pretty well with Tuckman's initial three stages. The "norming phase" is not included completely in the "initial bonding" phase. This might be due to the fact, that as in the latter phase, the need to find one's place starts to diminish but does not vanish completely. So we see some parts of this phase in the "maintaining presence" phase, too. In both models, the last phases correlate almost perfectly, describing the same situation in a

community, namely the disengagement, the dissociation of one or more individuals of the group. As Tuckman and Jensen could not take into account *virtual online* communities in the 60s and 70s, a kind of uncertainty remains, if and how these results apply to virtual worlds: communication, physical presence, body language, community size, ... at least *can* differ from physical present group members.

Although based on a different definition, Kim's life cycle model can be compared quite well to Tuckman, Jensen and Haythornthwaite et al. Kim starts with an earlier phase, which can not be found studying the other models: A visitor, who is not (yet) part of the community, has not signed up and thereof not even an identity, as Kim describes it. Her "novice" stage then corresponds with Haythornthwaite's "initial bonding phase". In the proximate phase, Kim differentiates between "regulars" and "leaders", both fully partaking roles in the community, but at distinct activity levels. Leaders are fully committed to the community, identify themselves a lot with it and try to push the community forward very actively. Partly, also the "elder" belong to Haythornthwaite's "maintaining presence" phase, as Kim describes them quite participating and active in communities as storytellers and the soul of a community. On the other hand, the relation of the "elders" to the community diminishes, correlating positive to Tuckman's "adjourning phase".

A completely different view is given by Owyang, who sees a community as a commercial product, which can be managed. Because of this Owyang's model does not correlate with the previously mentioned life cycle models, but relates to traditional product life cycle models commonly used.

Not represented in the figure is Palloff and Pratt's model. They observe three different roles of students in learning communities: Knowledge Generators, Collaborators and Process Managers. As these roles are relatively steady over time and explain more the behaviour of a student and how they contribute to the group, there is of course no sense in interpreting these findings as a life cycle model.

2.2.2. Composition of a Community Membership Life Cycle Model

Our approach is to identify a community member life cycle giving us a collection and description of community roles. So we chose to follow a modified model of Kim, as it represents in our point of view the model closely related to our definition of a Community Membership Life Cycle.

We modified Kim's model in some ways, we will discuss in the next paragraphs. A graphical representation is depicted in Illustration 5.

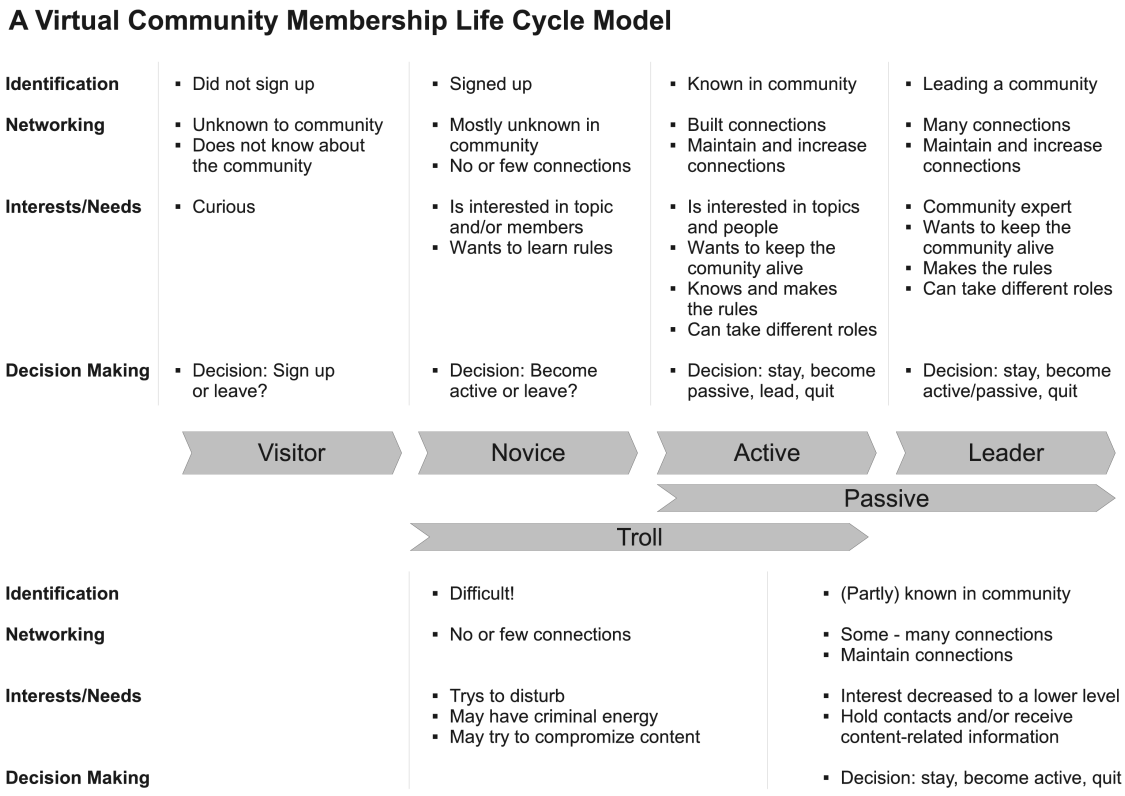

Illustration 5: A Virtual Community Membership Life Cycle Model

For each stage it is described, how it can be identified, how a typical personal relationship to other members looks like, what interests and needs the member has and what successor roles are usually taken.

Identifying A Role

To identify roles, we suggest measurements from the Social Network Analysis (SNA) and activity measures. Typical SNA measures are degree, closeness, betweenness and Eigenvector centrality. For a good introduction we refer to [13]. SNA measures can be provided by WeKnowIt Task T4.3 "Community Analysis Tool".

Besides SNA we suggest activity measures for role identification. Examples are the time since the last login or clicks/operations performed during a sliding time period. This measures can be calculated e.g. by WP1 in the User Profile.

All absolute measures will have to be measured relatively regarding the general "level" of the community: In a very active community, for example, five discussion contributions can be very low compared to some opinion leaders, posting 50 posts a day. In a less active community, five contributions can indicate a leadership position.

Visitors

The first contact somebody has with a community takes place as a Visitor. He has not signed up, has not got an account and therefore has no identification, no name in the community. He does not know much about the community, maybe a little about the topics or some members, depending on how he learned about the community. If somebody recommended the community to him, it is likely, that he knows at least one community member. In case he found the community by searching for a topic, he probably knows about some discussions in the community.

His interest is to find out more about the community, who are the members and what are the topics. If this would not be the case he would not have "pay" the transaction costs for at least finding out about the community and visiting it. Transaction costs are costs necessary to be able to perform operations on a market, here the transaction costs are the efforts needed to visit the URL, such e.g. searching it, typing in the URL, taking the time to read the websites etc. (see [14]). Therefore, we argue, that he is interested in the community. After gathering some information about the site, he has to make a decision: Sign up and become a member, stay as a Visitor (continue being a Visitor) or leave.

Novices

As soon as somebody signed up he becomes a Novice**.** Now he can be identified within the community by his nickname. He still does not know much about the community. Compared to the visitor he knows more about topics and members. He is willing to get introduced to people and issues. He wants to build up a network. For that he is willing to learn about explicit and implicit rules, behaviour and important people.

A novice can be identified reliable by the time, since he signed up. On the "SNA side" a novice probably has a low degree centrality, low betweenness and closeness centrality. For the sake of shortness we can give only examples here, which SNA measures might be usable to identify the roles. Concrete measurements, implementations and thresholds must be determined closely separately in later work.

A Novice can make the following decisions: become an active member, or become a passive member (see later) or leave.

Actives

To find about the decision made to become an Active is not as clear as the decision to become a novice. But there are some indicators: Does the user log in regularly? Did he build friendships/relationships to other users? Does he participate in the community life? If none of these indications can be observed at least at a very low level, this user probably decided to be passive or quit. So, in contrast, an active member builds himself a network, takes part in discussions and the community life and shows up regularly. He is interested in the community life, topics and discussions. He wants to keep the community alive and knows about (some) of the "unwritten community rules".

Actives can be identified by the short time since the last login and an average to high number of operations performed during a sliding time period. His social network should have a average degree centrality.

An Active can decide to stay as an Active, become even more involved as a Leader, retire as a passive member or even quit.

Leaders

Leader are people, that run communities, plan events, moderate discussions and even administrate communities technically. They contribute actively in content, discussions and media. Due to their activity level, they have strong networks which they maintain and increase. They are the community experts, who know members, content and techniques.

Leaders have many and close relationships. So we expect a leader to have a high degree centrality. As Leaders of opinions we expect them to have high betweenness, closeness or Eigenvector centrality. To identify a leader this measurements must be observed with a minimal threshold of activity index (see Actives).

As any other role, a leader can take the decision to reduce his activity level to an active or passive member. Of course, they can also quit.

Kim uses another stage called the "Elders". Elder is a social role, with normally high reputation and long community association. Interestingly enough we could not find any processional or authorative representation of the elders in state-of-the-art applications. This is an interesting finding, which could be investigated in future work. For now we decided to leave out this role of our model. Elders and leaders are closely related

concerning their authorative power and we will treat them both as Leaders.

Passives

A role with a lower activity level than leaders, active members or novices are the Passive members. We introduce this additional role, not included in Kim's model, as we expect differences in services being used compared to Actives. Passive members reduced their activity in the community and can be observed by a low activity level. Their network can vary, depending on the past within the community. If the member is a former leader or very active member, his network might be huge. If a novice decided shortly after his registration to become passive, the network might be very small.

For a passive member, therefore network size is not an indicator. Instead they can be identified by a low activity index, e.g. long periods between logins.

A passive member has the options to become active again - stay passive - or quit.

Trolls

Another additional role we want to introduce, we call Trolls. The role describes a negative, disturbing trouble maker in a community. The word refers to mythological trolls, which are said to be crabby and bad-tempered. We chose to use this term, as it is quite common in network culture (see for example [15]). Trolls are users, who at least disturb other members by posting offending, improper content like flame messages, forum posts, pictures or other media. Spamming is a typical troll activity.

We expect Trolls to have either not many connections to other community members, as their account is usually made just for this negative purpose. Or they have a very high out degree of relations, which are not accepted by others. The activity level of trolls is usually very high in a short time period and reduced to a very low level. We expect a troll trying to cause as much furore as possible and disappear afterwards not to be traced.

This is the reason, why we arranged Trolls below Novices and Active members in Illustration 5: They develop parallel to Novices and Actives. We see a troll's interest in disturbing, provoking or misusing the community.

2.3. Services' State-of-the-Art Analysis

In this chapter we will analyse some well known social networking platforms of today's internet. We will use the member life cycle model introduced in the last chapter to structure our analysis. We limit our analysis to Twitter, as a shooting star of the last months concerning communication. We chose Ning as a social platform for anything, Xing as business network. Sahana was chosen because of its relevance for the Emergency Use Case Scenario.

2.3.1. Twitter

"Twitter is a service for friends, family, and co-workers to communicate and stay connected through the exchange of quick, frequent answers to one simple question: What are you doing?" (see [7]). Twitter lets a user post a short notice about what he is currently doing. This information is stored on his twitter page, keeping all past postings as a kind of web log. A user's twitter messages are shared with his friends (called "followers"), letting them know, what the user is about to do. Vice versa, a member sees what his friends are doing at the moment ("following"). Twitter texts can be updated not only via a web interface but also via SMS and a generic Twitter API.

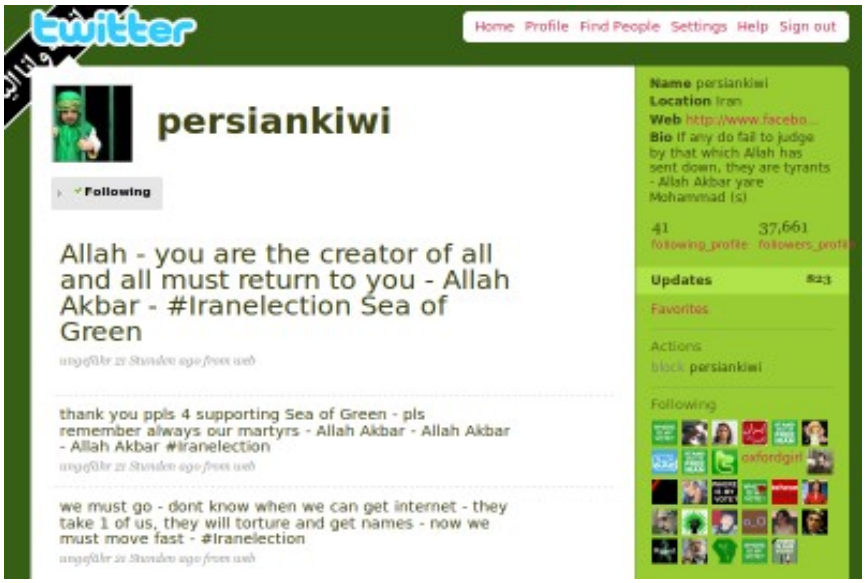

Illustration 6: Example Twitter Page. Here The user "persiankiwi" is proving news from the demonstration in Iran in June 2009.

7 http://www.twitter.com, accessed 25.6.2009

For a Visitor Twitter provides only some basic tour, showing what Twitter is about. Although it is possible to access the page of every twitter user via a generic URL (http://www.twitter.com/username), this service is not published on Twitter's site.

A Novice receives a "what to do now" message, giving the user hints, what Twitter services he might be interested in: First, to fill in the first twitter message; second, find friends and read their twitter texts; third, set up the users mobile phone to update twitter texts on the go. Friends can be searched by existing networks (e.g. yahoo.com, hotmail.com), by email address or by name and geographical location.

Actives can follow other Twitter users and use tag search.

As twitter is not a complex application concerning functionality, no further services for Passives and Leaders are offered.

2.3.2. Ning

Ning[8] allows a user to "create a network for anything". Ning follows a community-centred approach, so that almost everything a user does is in the context of a community. Each community is almost a closed world offering a set out of a common base of services to their members.

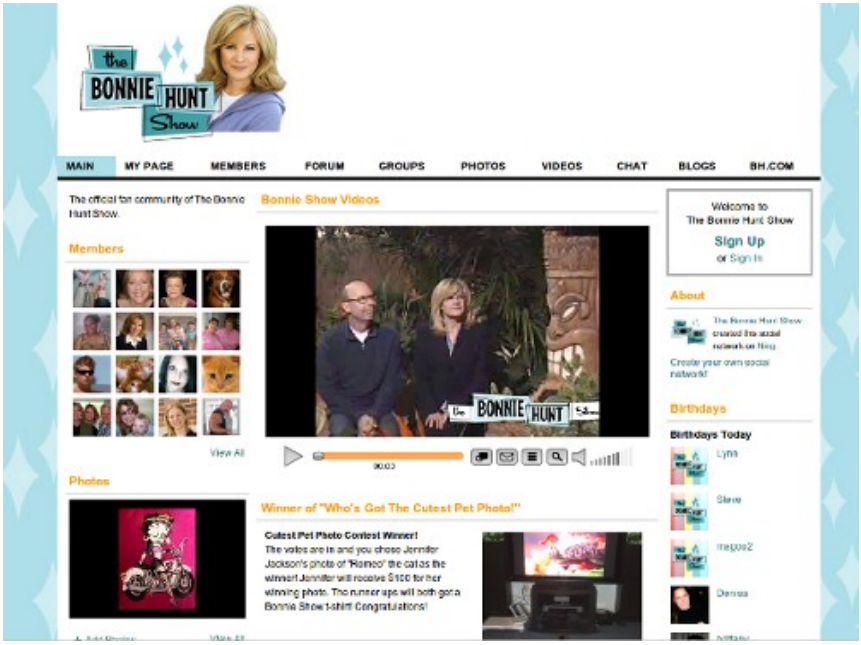

Illustration 7: Example Page from Ning. Here "The Bonnie Hunt Show", a fan site.

8 http://www.ning.com, accessed 25.6.2009

A Visitor is allowed to search for networks and offers a list of so-called "Ning spotlight" networks. Public communities can be accessed without signing up, read access to the community's homepage is granted. Resources as pictures, videos, discussions and blogs can be accessed.

To become a Novice, it is mandatory, to enter community-specific data to your profile data. Novices can access all content related to the community. These are photos, videos, discussions, events and blogs. They can as well invite friends via email or online/offline address books, edit their profile and send and receive messages.

Actives (members who signed up) can add context mentioned before, rate content and comment on it.

For Leaders Ning offers the opportunity to set up a community. Every community receives an unique sub domain name (e.g. mynewcommunity.ning.com). Communities can be open to anybody or by invitation only. They can be tagged, described and annotated with keywords. Some special tools are offered for leaders: They can create HTML-badges, a member can link to another site in the web, e.g. his homepage, showing, that he is part of the community. The layout of the community start page can be changed via a drag and drop technology, allowing to arrange the content in different ways on the page. Of course, also the layout style, as colours, fonts and other details can be adapted. Furthermore sub pages can be created or linked in from external resources. Concerning these services, Ning offers functionality normally included in content management systems[9]. The user management differentiates between regular users and community administrators. The authorizations can be granted at a very high level, allowing or disallowing guests the access to the community.

2.3.3. Xing

Xing[10] describes itself as "global networking for professionals". Its main interest is in connecting professionals and presenting themselves with a short CV summary. Contacts can be initiated by one side or introduced by a third partner - but have to be accepted by the other. Discussions forums and groups are supported as well as newsletters. Xing was founded as openBC, being an acronym for open Business Club, in 2003 by the German OPEN Business Club AG. Xing is free of charge for a number of basic services and offers a subscription for more advanced features.

9 A Content Management System (CMS) allows to create web pages and web sites through a editorial system. General layout and (corporate) design are automatically created. The author of a page / page part only edits the content.

10 http://www.xing.com

Illustration 8: Start page of a Xing User Profile

Xing allows Visitors to search for members, reporting only the number of found results, but no member details. A feature overview and a guided tour are offered.

Novices are encouraged by Xing first of all to fill in their professional profile. Via this short CV other members can be found. Colleagues, friends and business contacts can be found by a huge variety of search terms, e.g. people, who have worked in the same company, people who are interested in some special field, people, that visited my profile. Search agents can be set in place to report new members or changes in member profiles on specific target masks. Messages can be exchanged internally, not seldom used by head hunters to offer jobs or contact candidates.

Xing enables their Actives to manage their address book contacts, invite new users, accept or reject contact offers. A user can import address books (e.g. Microsoft Outlook) and invite users, not signed-up to Xing with automatic invitation emails. As an innovate feature, Xing allows to view ones contacts on a map, highlighting the contact addresses members entered into their profile. Xing offers groups, one can join mainly founded on topic-specific issues. E.g. “Alumni of University of Karlsruhe”, “CIO Forum” or “Marketing Experts”. Within a group, a newsletter can be subscribed to, discussions are supported and appointments can be planned. In a market place, job offers are accessible. Members decide themselves which other members they want to show what contact details: Every detail can be marked as accessible or hidden per contact.

Leaders (in Xing this a very active members) are able to create their own groups, create events and newsletters and send this information as well internally as via email to their subscribers.

2.3.4. Sahana

Sahana[11] is an open source Web 2.0 platform for managing emergencies. We chose Sahana because of its close relationship to the WeKnowIt Emergence use Case Scenario.

It contains modules allowing to share and organize information crucial for emergency response scenarios. This software was used for supporting the emergency response in several incidents in Sri Lanka, Pakistan and Indonesia. Sahana includes models for a registry of missing people, or support for managing available resources like machines, or shelter. Networks at the top level are organized in form of "Disaster Levels" which comprise the nested hierarchy "Disaster", "Event" and "Incident". For the management of volunteers special "Projects" (i.e. groups) can be created, which allow for organizing tasks. In Sahana access rights on certain modules can be defined very fine grained based on roles. The following roles are available: Anonymous User, Registered User, Super User (Head of Operations), Organization Admin, Volunteer Coordinator Camp Admin, Field Officer, Administrator (Admin).

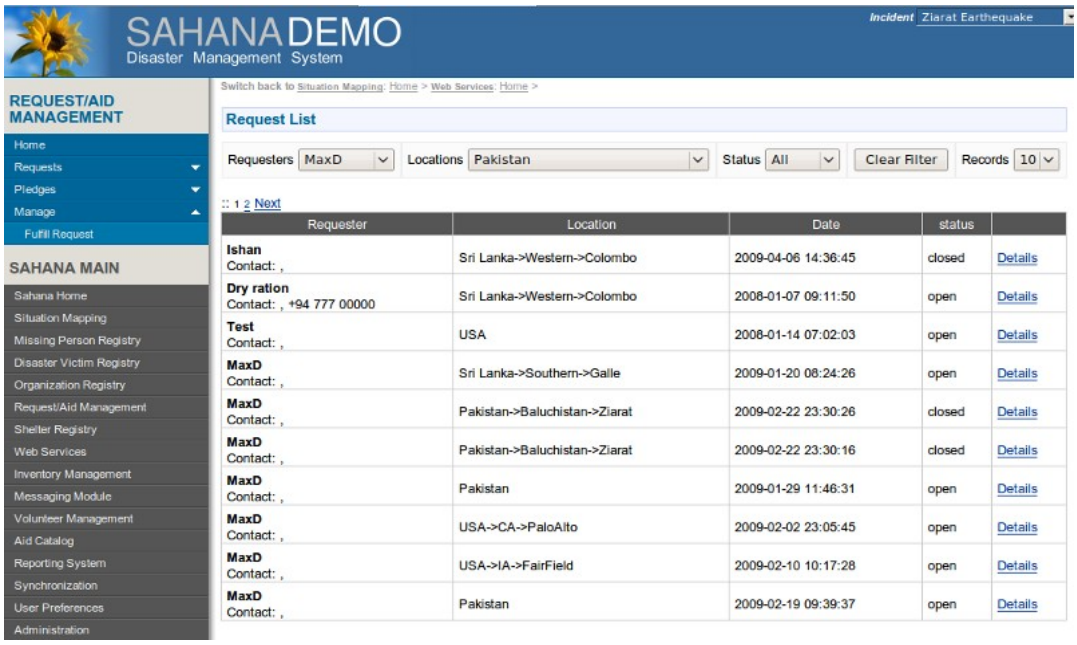

Illustration 9: Example Page of Sahana sumulating an Earthquake

If a person uses the Sahana system anonymously as a Visitor she cannot see much information. Typically only an overview of the incident is given without personal information about other users.

Novices map best to registered users. They might given the right to scan the registry of missing persons and to provide hints, if the current location is known.

11 http://www.sahana.lk/

The Actives of such a system is typically in charge of one ore more specific tasks. So can e.g. an admin of a organization making requests on behalf of her organisation and see contact details and skills of persons who volunteered for specific tasks.

A Leader (called admin in Sahana) is finally the person who can control the whole system and read all data, even sensitive personal data. The admin can influence the community using the Sahana platform manly in such a way, as to define what modules are currently used.

2.4. Summary

In this chapter we introduced a Community Membership Life Cycle Model suggesting systematic social roles being expected in online communities. We gave role descriptions what interests and needs the role has, what decisions are typically made and how the role can be identified. For identification we used SNA measurements and activity indices which can be provided by WP1 an WP4 later in the project.

We analysed four state-of-the-art online communities: What services do these communities offer for the roles.

In the next chapter we will focus on traditional authorization approaches to be able to link the social to the authorizational roles.

3. Traditional Authorization Approaches

In traditional approaches, access rights are mainly modelled by 3-tuples of the form (User, Permission, Object). Access is granted, if the current system state matches one of such 3-tuples in the access rights facts ("facts"). Giving an example, the 3-tuple fact (Alice,read,Pic1.jpg) allows a user with an account named Alice to access a file Pic1.jpg with read access. To be specific, in case the current system state consists of the user Alice wanting to read the file Pic1.jpg and the access rights facts base has the entry, then access is granted.

More generally, the 3-tuple (U,P,O) is a element in the space *UxPxO*.

U hereby is the set of users or subjects, sometimes also called principals. Subjects are entities, typically users or programs, executing in a system on behalf of a real world user (see [16]).

P is the set of permissions. Typical elements of this sets are read, write, execute, delete and so on. The semantic of an access right is usually given to it by the developers of the application or software system.

O represents the set of objects the access right applies to. Typical objects can be files and devices. It is important to notice, that user accounts can be subjects as well as objects, depending whether they act active (do something) or passive (something is done with them).

We call this model the atomic triple model, as each triple element can not be subdivided into finer-grained elements.

Two conclusions are obvious: By this design, a very fine-grained facts modelling is possible. Secondly, it is obvious, that it is practically impossible to manage a more complex system with several thousand objects and lots of users by defining these triples. The access rights facts would be huge, and any change has a quite high probability to affect several access rights which all must be changed manually. This leads to a high error rate by access rights not correctly set.

To avoid this, two major approaches have been introduced. First, several elements are grouped in sets. For example, all user accounts of a department are grouped assigned to one group. The same can be applied to groups of access rights and objects. This grouping then allows to define facts on them and no longer on individual elements. Many approaches using this technique can be found realized (for example see [17] and [18]).

It is important to notice, that these approaches do not extend the functional expressive power of an access rights model. The same result could still be retrieved by the original, atomic triples. Of course, the manageability has increased.

The second approach to improve the atomic triples was the introduction of authorative roles (“roles”) and Access Control Lists (ACL). Roles combine permissions and objects (e.g. “read file A”, “write all files in dir B”). Roles become an intermediate layer, quasi a container holding objects and permission combinations. Roles are assigned then to users, allowing them to perform operations described by their assigned roles. Usually the role approach is combined with grouping. Again, expression power has not increased: To combine permissions and objects into roles and then assign those to users can still be modelled by atomic 3-tuples. Roles on the other hand allow to abstract access rights from individuals by modelling rights in abstract roles. If a role of a user changes (because he changed the department he works in), his new access rights can be modified easily by changing his roles.

ACLs do work similar to roles, but use a different perspective: ACL combine users and access rights to access control lists, which are then assigned to objects. We see immediately, that this is the same approach as roles, applied this time on objects and not users.

The common RBAC model (Role-based Access Model, see [19]), for example, is an example of the first extension. In RBAC access rights - the two latter elements of the triple - are collected in so called roles. In our example a very simplistic role could be defined by role1, allowing to read Pic1.jpg. Roles are then assigned to users. Obviously, the RBAC and the triple model have the same expressive power. RBAC collects permissions and objects in roles and assigns them to users, while the basic triple model directly models triples.

4. The Community Design Language

4.1. Flexible Tests

We want to suggest a different, more general model. In our point of view each of the elements of an access right triple is an authorization condition. The triple, e.g. (Alice, read, Pic1.jpg), can also be seen as three tests, that must be fulfilled. First, the user must be "Alice". Second, the permission must be "read". Third, the object must be "Pic1.jpg". If and only if all of this three tests are true, access is granted.

If we perceive an access system as comparison operations, we are able to describe access rights in a more general way: We want to call each comparison a **test**. A test is a boolean operation.

Normally, only one test (e.g. User = Alice?) is not enough to decide, if access is granted. So, a combination of tests is necessary, to grant access. A combination of tests, which is sufficient to grant access, we call **access condition**. By this, an access condition itself consists of different tests. In the above example, the access condition has three assigned tests.

A test is of higher expressive power than a plain comparison: Instead of only comparing two operators by an equality sign, we can now formulate more complex tests.

4.2. n-tuples instead of triples

Actually there is no reason to limit an access right to a triple of user, permission and object, e.g. (Alice, read, Pic1.jpg). We can think of use cases, where three tests are not enough to fully describe the conditions under which a resource can be accessed.

Let us assume, Alice should be able to read Pic1.jpg only on weekdays. Then we have the time frame as a fourth condition: We would need not a triple, but a quadruple (Alice, read, Pic1.jpg, weekdays).

We can extend this to five or even more tests, e.g. including the device, Alice is allowed to read the Pic1.jpg with a cell phone only. Then we would need to extend the tuple by the test (device=cell phone?). Or, we want to limit the picture quality to a certain amount and so on.

So we suggest, not to speak of triples but of a collection of tests, the access condition, not necessarily limited to three but any number from zero to n. To let an access right be granted, all tests must be true.

4.3. CDL Goals

The Community Design Language (CDL) is a formal language for storing and checking access rights in/against a facts base. The CDL has several goals:

- Let the access rights be modelled in a very general way.
- Be manageable for human beings responsible for managing the access rights.
- Offer delegation of authority - allowing to delegate access rights management to responsible persons.
- Enable a high performance. This applies especially to access right requests, which is usually the most often performed operation of the CDL.

In the following section we will introduce all components needed for the CDL.

Formally the CDL consists of elements, sets, relations, tests and access conditions. We will explain these properties in the following.

4.4. Elements

Everything in the CDL is defined as an element. For example, the user "Alice" is an element. A file "pic1.jpg" is an element. Also permissions, for example the "read" permission is an element. We hereby separate the semantic interpretation of an element of its syntactic meaning: "Alice" is semantically a user. Syntactically it is an element. "Pic1.jpg" is semantically an object and by this not an user. Syntactically it is, again, an element.

To be able to distinguish different elements, we identify them by a unique internal identifier (called iEID, internal element identifier). This identifier is unique and managed by the implementation of the CDL itself. The iEID is used internally only.

All elements are furthermore identified by an unique external identifier, called eEID, external element identifier. While the iEID is an integer, the eEID can be any unique String and therefore any URI[12]. By this, all modules / services / applications using the CAP can continue identifying elements in their way and need to to adapt to the CDL. The identifier mapping will be done within the CAP.

12 This becomes relevant for mapping of CAP to RDF in Chapter 5.

4.5. Sets

Sets are used to organize elements in logical, semantic constructions. By this, sets are containers for elements. A set itself is also always an element, too.

An example for a set are users. The set "users" may consist of the elements "Alice", "Bob" and "Charly". The elements "read" and "write" can be assigned to the set "Permissions". The elements "pic1.jpg" and "doc2.txt" to the set "objects".

Two things are important to mention: A set is an element on its own. The set "Users" therefore is ambivalent: On the one side it is a set of the two elements "pic1.jpg" and "doc2.txt". On the other side it is itself an element called "objects". This is illustrated in Illustration 10.

We will see later, that access rights are modelled on elements only. By defining sets (and other "higher" expressions) not only as containers but also as elements we do not need to change the syntax of the CDL.

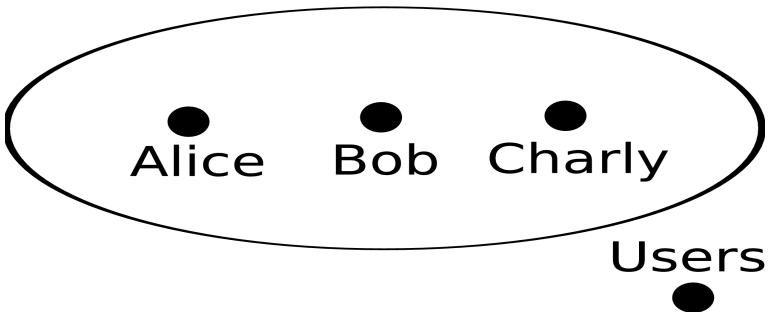

Illustration 10: Sets are containers and elements

Sets have unique internal identifiers (of course, as sets are elements!). If we want to stress the meaning of a sets in its grouping role, the identifier is called iEID (internal element identifier). In case we want to emphasize the sets meaning as a container, we call the set id iSID (internal set identifier). Physically, iEID and iSID are the same identifier.

From a purely syntactical point of view, sets can be containers for every element. Sets can contain them even themselves. From a semantic point of view, in the most cases only "similar" elements will be grouped together. Please be aware, that this semantic interpretation and constraint will have to be done by the administrator configuring the CAP and is not immanent in the CAP system itself! The CAP will not check, if a set assignment "makes sense" in a semantic meaning.

It is important to notice, that there is no syntactical need to assign elements to sets. The question, which elements are assigned to a set can only be answered on a semantic level. The CDL methodology does not

know about different syntactic element types like data types “users” and “permission”. To the CDL both are plain sets and all elements of these sets do not differ (in syntactical ways) from each other.

Sets can - in their role as elements - be assigned to other sets. This assignment links only the set itself to the set, but makes no semantic statement about the elements of the assigned set. E.g., assigning the set “objects” to a set “world” will not assign “pic1.jpg” (in objects) to “world.”

An element can be assigned to multiple sets.

The CAP knows different types of sets:

4.5.1. Named Basic Sets

Named base sets are sets the have an identifier, the set elements have been explicitly assigned to the set. By this, the set is in a way “static”, as it is explicitly defined, has an identifier (a “name”) and elements assigned to it.

E.g. the set Users = {Alice, Bob, Charly} defines a set named “Users” with the three elements Alice, Bob and Charly. Assignments of elements to Named Basic Sets can be changed later. This means, elements can be added to the set or can be removed from it.

4.5.2. Anonymous Basic Sets

Anonymous sets do not have an assigned identifier (although they have an internal set id iSID). The set is defined by enumerating its elements.

E.g. the set {Alice, Bob, Charly} is an anonymous basic set. It has no identifier itself and is defined by the enumeration of its elements Alice, Bob and Charly.

Set assignments to anonymous basic sets can not be changed later.

4.5.3. Named Meta Sets

Named Meta Sets are implicit sets, which can be calculated by applying one (or more relations) to a Named Basic Set. We call the result of this operation also “Result Set”.

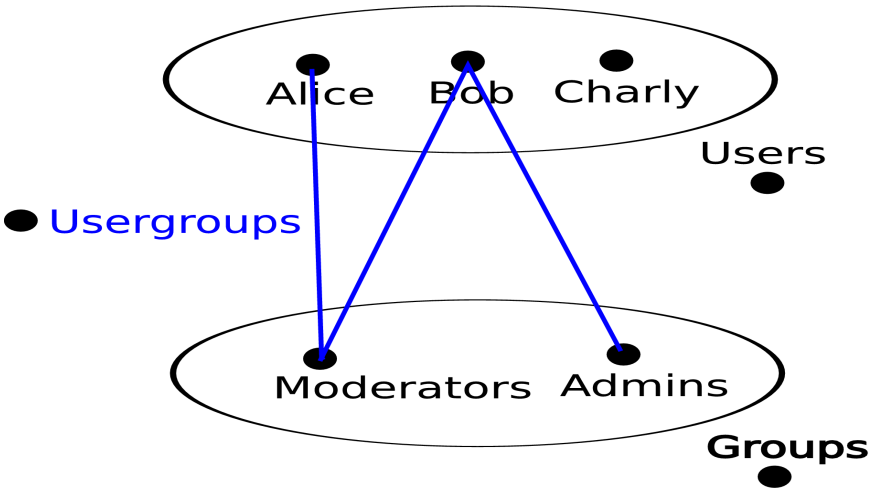

Illustration 11: Relations and Meta Sets

An example for a named meta set is depicted in Illustration 11. We will introduce relations in section 4.7. in detail and mention them here only briefly.

In the illustration two named explicit sets are defined, the set Users and Groups. Both sets are elements as well, which are in this example not assigned to a set themselves. A relation named "Usergroups" is defined on the sets Users and Groups, linking "Alice" with "Moderators" and "Bob" with "Moderators" and "Admins". Charly is not linked.

The Meta Set "Users.Usergroups" - we apply a relation to a named basic set – result in the Result Set {Moderators, Admins} as this are the elements in the set "Groups" which are linked by Alice, Bob and Charly, such all elements in Users.

4.5.4. Anonymous Meta Sets

Anonymous Meta Sets are calculated in the same way as Named Meta Sets. In contrast to them, the original Basic Set is anonymous.

E.g. is the set {Alice}.Usergroups an anonymous meta set. The basic set is anonymous: {Alice} is an anonymous basic set. The Result Set of this Meta Set is {Moderators}.

4.5.5. Notation

Setid	Symbol for all elements in the set "setid". Set as a collection of elements.
{element1}	An anonymous basic set with one element "element1".

{element1, element2}	An anonymous basic set with two elements.
{Setid}	Symbol for the set as an element itself. This includes only the set as an element itself and does not refer to the elements within the set!
Setid.relation	Named Meta Set calculated by applying the relation "relation" on all elements of the set "Setid".
{element1}.relation	Anonymous Meta Set calculated by applying the relation "relation" on all elements of the anonymous set – here only one element.

4.6. Set Allocation and Access Allocation

At a given system-state (situation) the Community Administration Platform must decide if access is granted or not. From a set of elements (e.g. users) one element is chosen or applied to the current system state. For example, if "Alice" is the current user from the user set {Alice, Bob, Charly, Dave}. In this case we call the element "Alice" the set allocation of the set "users".

This applies as well to other sets, e.g. the permission (read permission, if Alice wants to read some object) or the objects.

Each allocation of one (or more) current set elements of one (named basic) set, we call set allocation. The set of all set allocations itself we define as access allocation.

4.6.1. Example

Alice wants to read the file file1.doc. In this case, the element "Alice" is the set allocation of a set "users", "read" the set allocation of the set "permissions" and "file1.doc" the set allocation of a set "objects". The access allocation therefore is {[User]=Alice, [Permission]=read, [Object]=File1.doc}.

4.6.2. Notation

[Setid]	Set allocation of the set "setid".

[Setid]=Alice	Statement, that the current set allocation of the set "setid" is "Alice". "Alice" hereby is an element of the set "setid".
{[Setid1]=Alice, [Setid2]=read}	Access allocation of the set allocations setid1 and setid2.

4.7. Relations

We define a relation on two sets, the source set and the target set. Both sets may be identical. Within a relation, links between elements of the source and the target set may be established ("linked"). A relation itself is an element, too.

Let us once again go back to Illustration 11: A relation "usergroup" is defined on the sets "Users" and "Groups". The elements Alice, Bob and Charly belong to the set "Users", "Moderators" and "Admins" are elements of the set "Groups". By this, obviously we have 7 elements: Alice, Bob, Charly, Admins, Moderators, Users, Groups and Usergroup – as sets and relations are elements, too.

This relation can link elements of the two sets: Alice is linked to Moderators, Bob to Moderators and Admins, Charly is not linked. Semantically, we assign the users to user groups. Syntactically we define a relation and links on this relation.

Relations seem, at the first glance quite similar to sets. The difference is, that relations define constraints, which elements can be assigned to each other: As a relation is defined on a source and target set, only elements of these two sets can be linked. In contrast, a set can include any element. Secondly, relations are directed links allowing an element Alice to be assigned to the element G1, but not vice versa. Sets include an element, or do not, but can not distinguish directions.

Relations can be defined reflexive, symmetric and/or transitive.

Reflexive relations can only be defined, when source and target set are identical. Then, reflexive relations always link elements with themselves.

Symmetric relations are undirected, so a link between elements A and B always will lead to element B related to A.

In transitive relations A will be linked to C, if A and B are linked, and B with C.

4.7.1. Notation

{element}.relation	Application of the relation "relation" on the anonymous basic set with the element "element". The result set is an anonymous set, that is those elements linked by the relation sourcing element.
{element1, element2}.relation	Application of a relation on a anonymous, explicit two-elements set. The result is an anonymous set.
[Set].relation	Application of the relation "relation" on the current allocation of the set "Set". The result is an anonymous set.
Set.relation	Application of the relation on all elements of the set. The result is an anonymous set.
{element}.rel1.rel2	Application of relation "rel2" on the set of elements which are retrieved by application the relation "rel1" on the element. Concatenation of relations. The result is an anonymous set.

4.8. Tests

We define tests as a methodology to check, whether a defined set of elements has a non empty intersection with a second set. We call the first set the "test set" and the second set the "comparative set".

By this definition, obviously we define a boolean function. The parameters of the function are two sets and the fixed operator "non-empty intersection".

Formally: f(S1, S2, op) → boolean

The operator can be any boolean bi-parametric operator. For this prototype, though, the operator is fixed to θ, defined as **θ ::= "intersection of both sets is not empty".**

Of course, in later versions of the CAP more operators can be implemented and supported. We decided to use the operator θ as for many access conditions this operator deals very well with human

expectations and intuitive approaches. We will see this later in the examples (see chapter 7).

4.8.1. Test Algorithm

To execute a test and receive a test result two steps have to be done.

First, the two sets have to be calculated. This means, that the elements belonging to the test set and the comparative set have to be calculated. As we defined in chapter 4.5, sets can be of different types. For all basic sets, this task is very easy, as resulting sets are by definition an enumeration of their elements. For meta sets calculations have to be done to receive the result set.

Second, the two result sets have to be tested by the boolean operator.

4.8.2. Computational Complexity and Test Pre-Calculation

The test algorithm described previously needs to be performed every time, the test is executed. As checking access rights is the main task for a access rights system, the computational time for this tests should be very low. Otherwise scalability and response times might be in danger.

CAP tests correspond to tests of access rights. Therefore, strategies are needed to reduce the computational of tests.

If both sets of a test are basic sets, the calculation time is quite low, as the boolean test can be performed in linear time of the smaller set: Enumerate all elements of the smaller set and check, if this element is found in the second (bigger) set, too. If one match has been found, we did find a non-empty intersection. We just have to make sure, that finding a set element in one set can be done in linear time. In this case, the complete operation can be done in O(n) in the worst case, with n := #elements of the smaller set.

In the case that at least one of the sets in the test is a basic or meta set, execution time of the test can increase significantly: Before the comparison can be performed, the result set has to be calculated. This can be a complex and time consuming process, depending on the definition of the meta set. If e.g. r relations are concatenated on sets of the size s1, s2, … sN the calculation complexity of the result set is O(s1 x s2 x … x sN).

A strategy to reduce this result set calculation time is, to pre-calculate the result sets. This strategy makes use of the fact, that result sets of meta sets can easily be calculated at the time, the relation and sets have been **defined**. This means, we can pre-calculate the result sets at any convenient time after the set and relation definition, but before a test occurs. Obviously, the computational complexity of tests on meta sets can then be reduced to O(n), which is the complexity of basic sets.

As we did not implement test pre-calculation in the prototype, we only want to mention this possibility here. Advantages and disadvantages can be discussed in detail in future work.

4.8.3. Tests as Elements

Again, a test itself can be seen as an element in the CAP system. By this, every test gets assigned an unique element ID. To differentiate between the meaning of a test as an element or an test as a boolean operation, we identify the unique id in the first case as EID (element id), in the latter case as TID (test id).

By assigning a unique identifier to tests, they can be reused in different Access Conditions (see below). The CAP also supports anonymous tests which avoid assigning external ids.

4.9. Access Conditions

We define Access Conditions (AC) as a collection of tests. To make an access condition true, all assigned tests have to become true with the current system allocation.

Semantically an Access Condition is and AND-concatenation of tests. It is AND, as all tests have to be true to become the Access Condition true.

An Access Condition, as well as every other component of the CAP, is an element and identified by an ID. The ID is called AID (Access Condition ID), if we want to stress the perception of a collection of tests; the id is denoted as EID if we point out the meaning of an access condition as element.

All defined Access Conditions (such, the “set of all access conditions”) constitute the access base or the facts base. The CAP tests against this access base, whether access should be granted. Please distinguish an access condition as set of tests, and the set of all access conditions, we refer to now.

The algorithm, to decide, whether access is granted (or not) depending on a given allocation is straight forward: Check all Access Conditions until one Access Condition becomes true. If so, grant access. If all Access Conditions resulted in “false”, access is denied.

By this, the access test is an OR-concatenation of all Access Conditions.

5. RDF Layer

This section describes an additional component of the CAP that allows for exporting the access rights as RDF graphs.

The WeKnowIt System consists of different subsystems that use data from each other. As basis for data exchange ontologies will be defined whose data structures will be used by the subsystems. An example for such ontology is the Event Model F developed in WP5.

As Web standards like OWL for ontologies and RDF for data exchange are employed in the WKI system an additional layer is needed to make information processed in the CAP available to other WKI services.

Therefore we developed a mapping to express access situations. We mapped the concept of access rights and additional information to the semantic layer by allowing for querying this information in the standard language used for querying RDF triples, SPARQL.

In the following we consider the CAP as an inference engine, whose main purpose it is to infer, if access can be given in a specific situation for specific circumstances. According to this interpretation the CAP consists of a T-Box and an A-Box.

The T-Box comprises the schema information and rules. In case of the CAP these are "sets", "relations" between sets, and "access conditions" (consisting of tests).

The A-Box comprises information about specific entities, their properties and relationships. In case of the CAP the concepts "element", "belonging of an element to a set", and "link" belongs to the A-Box.

Based on T-Box and A-Box the CAP can infer by a test of all access conditions if access is granted for a specific situation.

5.1. RDF and SPARQL

RDF[13] is one of the core formats defined by the W3C for the Semantic Web. With RDF facts can be expressed in a way, that includes pointers to definitions (schema, ontology). This eases interpretation of these facts.

Facts which are expressed in RDF consist of a triple of subject, predicate, and object. While subject and predicate always consist of a URI, the object can be a URI or a literal. To illustrate an example we consider the two facts (1) Alice is of type person, and (2) Alice knows Bob. These facts could be expressed in RDF as (1) `wki:Alice rdf:type wki:Person`, (2) `wki:Alice foaf:knows wki:Bob`. By interpreting these triples as nodes

13 http://www.w3.org/RDF/

and arcs `wki:Alice` from triple (1) and triple (2) are unified to one node. So more complex propositions can be made consisting of different related triples.

When many triples are created and used to express facts, there arises the need to extract a subset of triples that contain facts relevant for a specific application. SPARQL[14] is the most prominent standardized query language allowing for querying sets of RDF triples (typically stored in a "triple store).

E.g. the SPARQL query to retrieve a node that is of type Person and knows Bob could be:

```
select ?node where
{?node rdf:type wki:Person.
?situation foaf:knows wki:Bob.
```

The result of this query to the triple store consisting of the above mentioned triples would be `wki:Alice`.

5.2. Mapping of CDL concepts to RDF

We decided to make at first only a subset of CDL features available through a SPARQL endpoint. If in the integration of different services more services are needed the functionality will be further extended.

Two types of mappings are supported: (1) inference query, a query that infers not explicitly stated information and (2) queries that provide facts, explicitly stated in the CAP

5.2.1. Access Decisions

As explained above (cf. Chapter 4.6) access rights in the CAP depend on the configuration of entities, which participate in an access situation, the so called access allocation. The access allocation consists of pairs of sets and elements (e.g. [User]=Alice). For such an access allocation, the CAP can infer based on the access conditions, whether access will be granted.

In RDF access allocations are not natural to represent. Therefore the pairs of sets and elements are mapped to predicates and objects describing the access situation of interest. In this the sets are interpreted as roles.

In an access situation different elements play a role, some of these roles are typically needed (actor, action, resource) to define the access situation precisely. But any other access situation with arbitrary set-element-pairs can be mapped to RDF.

The small ontology, describing the schema of an access situation is depicted in Illustration 12.

14 http://www.w3.org/TR/rdf-sparql-query/

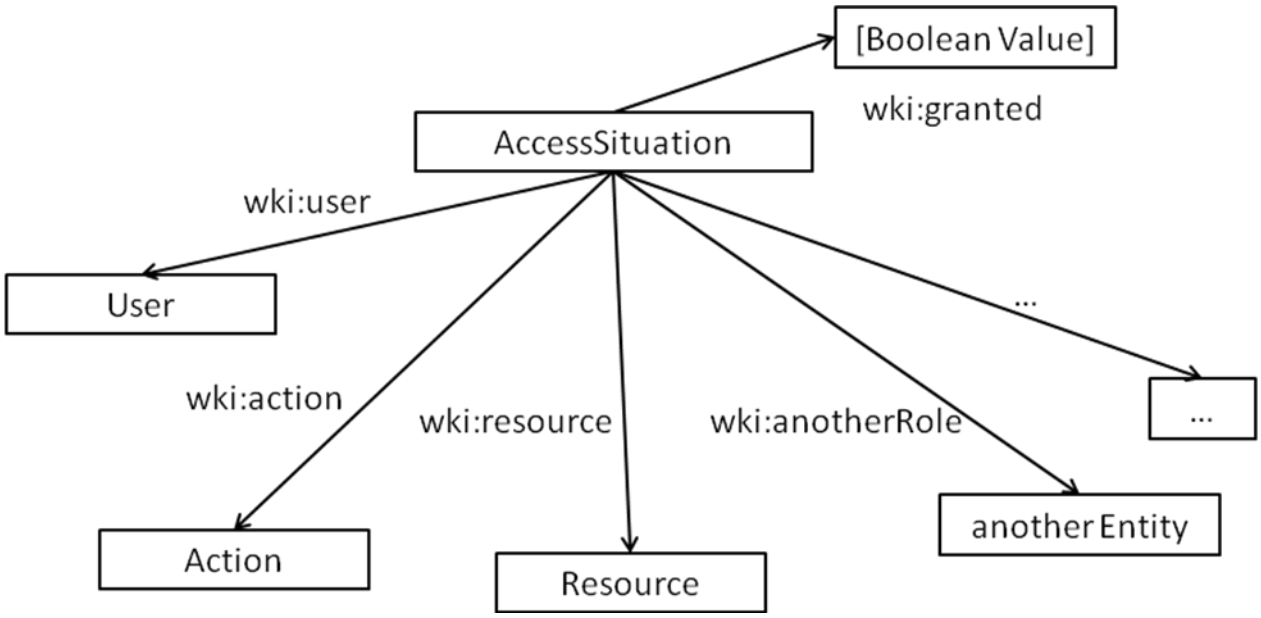

Illustration 12: Schema of an Access Situation

The class AccessSituation combines all properties, which represent the access allocation. In this example an element of type User is referenced by the access situation through the property `wki:user`. The same holds for Action, Resource, and any additional Element. The result of the access check, the access decision as "grant" or "deny" for the described access allocation, is represented in the property `wki:granted`, that consists of a Boolean value ("true" or "false").

While in practise these access situations are not coded as facts in the CAP (neither in RDF nor in CDL), we assume as an abstraction that this is the case. This virtual knowledge base can be then queried via SPARQL.

5.2.2. Elements and Sets

Elements are the most basic elements in the CAP, everything is an element. So elements map naturally to the notion of node in RDF. These nodes are used as subject or object in RDF-triples.

Elements and sets can have a specific relationship in the CAP, indicating that an element is included in a set. In the CAP this relationship will be used to indicate the type of an element by including it to the set defining its type. Therefore it is natural to map this relationship to the standardized predicate `rdf:type`, which exactly expresses this relationship. E.g. the element wki:Alice is in the set [wki:User], then this fact is expressed as the RDF-triple "wki:Alice rdf:type wki:User", indicating that Alice is of the type wki:User.

5.2.3. Links between elements

Relationships between elements in the CAP can be expressed as typed directed links. One link can only be stated between two elements. These links are mapped to RDF-triples (consisting of subject, predicate, and object) indicating the directed link as predicate between the elements indicated by subject and object. E.g. if a link of the type "wki:proxyFor" would stated from wki:Alice to wki:Bob, this would be mapped to the RDF-triple "`wki:Alice wki:proxyFor wki:Bob`"

5.3. Supported SPARQL Queries

In this section we present SPARQL queries which make the information, which was mapped to RDF exploitable to the user. We do this by presenting corresponding SPARQL queries for the fragments of CDL supported by the RDF layer.

5.3.1. Access Decisions

The result of access decisions regarding a specific access allocation is expressed in CDL as follows:

```
CHECK ACCESS: <<allocation>>
     Allocation ::= <eSID1>=<eSID1Value> ...
     <eSIDn>=<eSIDnValue>
```

An access allocation could be e.g. [User]="PM2", [Plant]="P4", [Permission]="write".

As explained above the allocation and the result is represented in RDF around a central node of the type `wki:AccessSituation`. Therefore a SPARQL query to retrieve access decision looks as follows:

```
select ?grant where
{?situation rdf:type wki:AccessSituation.
?situation wki:Permission wki:Write.
?situation wki:Resource wki:P4.
?situation wki:User wki:PM2.
?situation wki:isGranted ?grant.}
```

To define that we are not interested in arbitrary relationships but only in access situations we need to indicate this in the SPARQL query by requiring a "`?situation rdf:type wki:AccessSituation`". Besides the fragment `?situation wki:isGranted ?grant.` the other parts of such a query consist of the access allocation.

The result of this query would be a Boolean value of the variable `?grant` that indicates whether access is given for the specified situation or not.

5.3.2. Elements and Sets

We provide two possible SPARQL queries for retrieving information about elements and sets from the CAP. That is for retrieving all sets containing a specific element or for verifying whether a specific element is in a specific set.

Get all Sets of One Element

In CDL it is possible to retrieve all sets that contain a specific element EID:

```
LIST SETS OF ELEMENTS eEID;
```

Since the element-set relationship is mapped to the `rdf:type` relationship the corresponding SPARQL query asks for all types of one element via:

```
SELECT ?setIds WHERE
{EID rdf:type ?setIds}
```

Set Membership of One Element

In CDL it is possible to verify whether a set eSID contains a specific element eEID:

```
CHECK SETASSIGNMENTS OF ELEMENTS eEID IN SETS eSID;
```

Again this relationship is mapped to `rdf:type`, so the SPARQL query verifying a set membership is:

```
ASK
{eEID rdf:type wki:eSID}
```

5.3.3. Links between elements

The RDF layer of CAP allows for querying the relationships between elements, called links. Two similar types of SPARQL queries are implemented for retrieving all links of a specified element and to to get specific links of one element.

Get Links of One Element

CDL allows for retrieving all links for one element eEID with the following expression:

```
LIST LINKS OF ELEMENTS <eEID>;
```

As explained above links are mapped to predicates in RDF triples. Consequently the corresponding SPARQL query to retrieve these links is:

```
SELECT ?relation WHERE
{eEID ?relation ?element}
```

Get Specific Links of One Element

Finally it is possible with CDL to retrieve the destinations of links that have a specific type eRID for an element eEID:

```
LIST LINKS OF ELEMENTS <eEID> ON RELATIONS <eRID>;
```

In this module such queries can be made with the following SPARQL query:

```
SELECT ?element WHERE
{eEID eRID ?element}
```

6. Implementation

The technical implementation of the Community Administration Platform is done in pure Java. The code abstraction is organized in 6 layers which are depicted in Illustration 13.

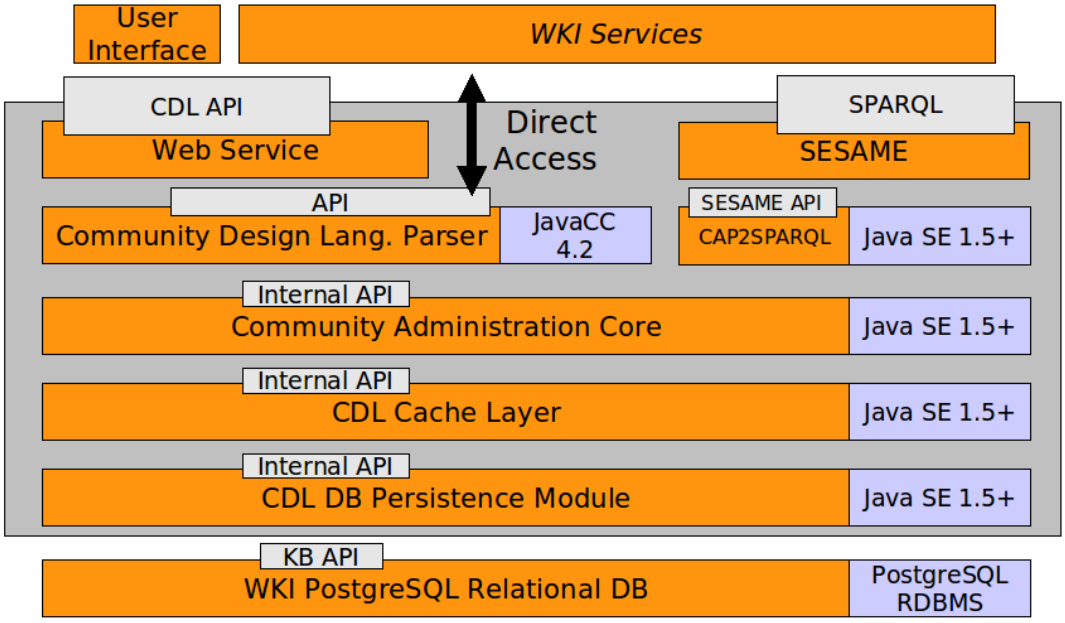

Illustration 13: Technical Layers of the Community Administration Platform

6.1. Persistence Storage

As persistence storage we use a relational database system. We chose PostgreSQL as database managements system as it supports transactions since quite a long time. MySQL started to support transactions recently, but we do not expect it to be so reliably as PostgreSQL. As Oracle is taking efforts to buy MySQL, MySQL's future is not really clear (see [20], p.7). Further more we have good experiences using PostgreSQL with huge tables concerning access times and scalability.

As access times and scalability are important key success factors for any access control system we decided not to use semantic databases. Although having advantages concerning modelling and querying facts in

semantic manors, we see access time and speed and the most important issues for an access control system. Semantic Databases have made big efforts to improve here. Still, in our experience, relational databases are superior in reference to this properties.

The CAP is designed to support any relational database system using standard SQL syntax. By this, CAP is also able to support MySQL. The driver support has already been implemented.

Nevertheless, for the CAP prototype we chose to use PostgreSQL.

6.2. CAP DB Persistence Module

The Persistence Module was developed to access the Persistence Layer (the PostgreSQL Database Management Engine). The targets for this layer are:

- Make database access transparent for the above's layers
- Automatically manage database access
- Support the usage of several, distributed databases by vertical division
- Support access via different database user
- Support SSL and non-SSL connections

State-of-the-art solutions for Java connections as Hibernate have been analysed. We found it quite difficult to realize especially a vertical division of the data in distributed databases with Hibernate. As well, Hibernate would have made it quite difficult to support the goals and requirements of the Cache Layer (see below). Due to these facts, we chose to implement our own Persistence Module.

The CAP Persistence Module is organized in the following way: Two entities are structurally separated. A relational database connection (RelationalDBConnection) and a relation database entity (MapRelationalDB). Both are realized in the Java package "eu.weknowit.social.common.persistence".

The **Database Connection** is managing a physical connection to a database. An initial connection is provided by an XML properties file. When connecting to this first database, further connections defined in this database are imported and used to establish further connections to different databases. SSL connections are supported. Different user accounts for every connections can be used.

The **Relational Mapping** module binds any class through the interface "Persistent" to a MapRelationalDB instance. An initial mapping is defined via a XML property file. Through this first initial mapping, every other mapping is imported by connecting to this initial database/mapping. By this, further mappings are supported simply by adding another entry to the mappings database.

A example of a mapping definition table can be found below.

Mapping	Table	Connection	Sequence
weknowit.eu.social.auth.admin.Element	element	1	seq_element
weknowit.eu.social.auth.admin.Set	set	3	
weknowit.eu.social.auth.admin.Relation	relation	1	
...	...	...	...

Table 1: Example table of MapRelationalDB definitions

In Table 1 the class "weknowit.eu.social.auth.admin.Element" is mapped to the table "element" in the connection referenced by the id 1. The properties of this connection are defined by the definitions of the database connection. The sequence adds an automatic number range to assign ids for new elements.

The internal structure of the entities (e.g. an Element) needs NOT to be described here. It is describe in the implementation of the load() and save() method of the classes implementing the interface Persistence.

Standard methods for SELECT, UPDATE, DELETE and INSERT statements are provided by the Persistence Module to allow the implementing classes of the interface Persistence make use of this standard methods (and make database access transparent).

By this methodology we enable our Persistence Module to define connections and mappings consistently in the relational database itself. No extra definition in XML files are necessary. By this, we avoid a different technology for the code developer. Further more the object to be stored can encapsulate its inner logic completely from the outside by implementing the interface Persistent. Helper methods support standard SQL statements, making the connection completely transparent form SQL. Nevertheless, if judged useful by the developer, the helper methods can be bypassed, allowing direct SQL statements. Still, not knowledge about the database location, access parameters or tables are necessary.

6.3. CAP Cache Layer

The target of the CAP Cache layer is to support scalability and minimise execution time of CDL statements by avoiding database accesses. As memory access is by dimensions faster than access to persistent storage, it is a good strategy to have data necessary for calculations available in memory. On the other hand, volatile memory has much higher costs than persistent storage, making it preferable to use as less amount of memory as necessary. At least, this is the situation today.

A cache generally has the task to store the data probably necessary for the next operations in memory to minimize calculation time.

The requirements for the CAP cache are:

- Support positive and negative caching. Positive caching is the traditional approach by caching existing objects. Negative caching does store the result, if a object miss occurs, such an object is requested NOT existing in the persistence storage. By this, future requests for the same id can be caught by the cache.
- Support immediate flushes (auto-flushing)
- Support key caching different from object caching (see later).
- Support primary and alternative key identification.

Although doing some research on existing solutions (e.g. Hibernate etc.) we were not successful in finding our requirements met. Therefore we decided to implement our own caching layer.

The CDL cache is realized in the package "weknowit.eu.social.auth.admin.cache". It is organized in two sub-layers.

6.3.1. Key Cache

The first layer is the key cache. It is caching information that a certain object id (identified by a database key) is present in the persistent store or not. The key cache supports as well primary keys as alternative (secondary) keys. Each entity can be identified by one primary and n secondary keys. Search and identification are supported on all keys, primary or secondary.

Internally hash maps are used to store the keys.

As a key can be of different types (e.g. integer, long, String, tuples, triples, n-tuples, …) the concept of inheritance is used. The class "DBKey" implements abstract methods to create new keys (e.g. by making use of defined sequences in the Persistence Module), compare keys and hash

keys. Concrete implementations have realized for the key types depicted in the table below. Support for additional key types can be added easily .

Type	Example(s)	Comment
Integer	1, 2, 3	Simple integer key
String	Key1, key2, key3	Keys consisting of Strings
Integer,Integer	(1,1), (2,3), (4,5)	2-tuples of integer values (two-field key)

Table 2: Key Types currently supported by the CAP

6.3.2. Object Cache

An entity in the cache can be assigned a defined cache status, depicted in the table below.

STATUS	Name	Comments
0x1	NONREGISTERED	Key uncached, Object uncached
0x2	REGISTERED	Key present and cached, Object uncached
0x3	CACHED	Key and Object cached
0x4	DIRTY	Key and Object cached. Has been modified in memory.
0x5	NEW	Key and object are only present in memory.
0x6	DELETED	Key and object are marked deleted in memory. Still existing in database.
0x7	NONEXISTENT	The key does not exist in the database

Table 3: Status supported by the CAP Cache

The status 0x7 is only necessary (and used) in the case, negative caching is enabled. Negative caching does cache information, that a key is not present in the persistence storage. Future requests for this key can be caught by the cache then. This avoids costly repeated requests to the persistence storage.

Possible status transitions are depicted in Table 2.

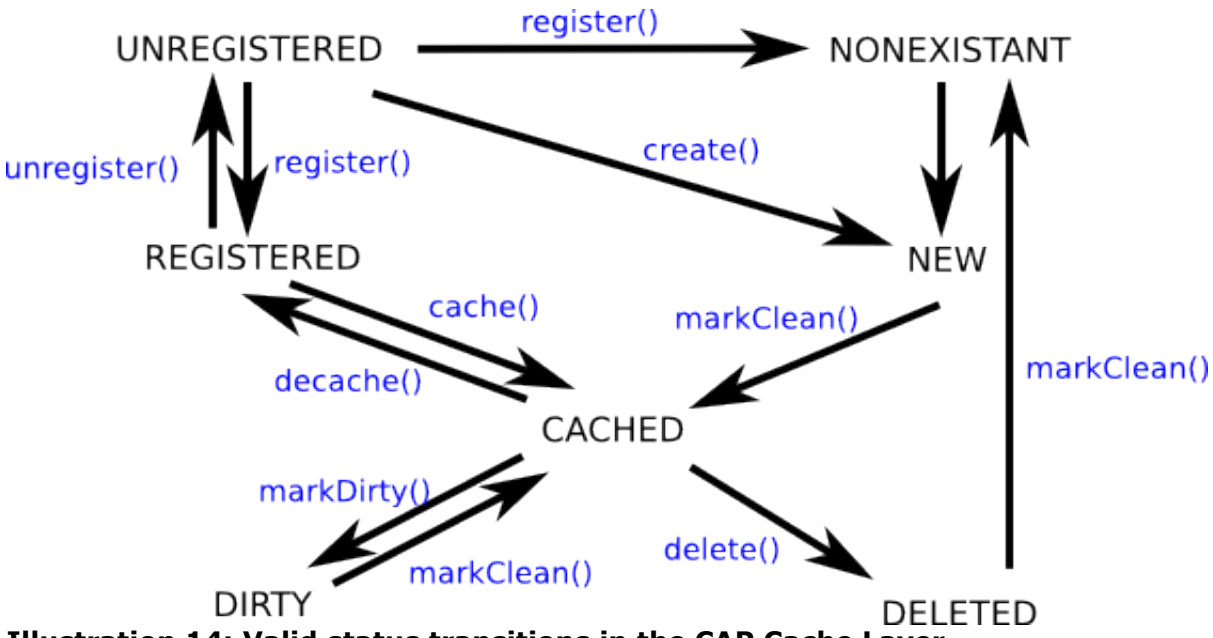

Illustration 14: Valid status transitions in the CAP Cache Layer

As we can see in Illustration 14 a valid status change is from status CACHED to status REGISTERED through calling the method decache(). In other words, by this method the cached object is removed from the cache. The key still is present in the cache. All other transitions can be interpreted in the same way.

Currently we implemented positive and negative caching for keys and objects. No automatic decaching strategies have been implemented yet, as this was not in the focus of our task T4.2. For the prototype therefore the cache will never decrease. Nevertheless, a corresponding strategy can be implemented later easily.

6.4. CAP Core

The CAP Core is extensively described in the chapter 4. We refer therefore to this section for in-detail explanations of the CAPs Core Functionality.

Technically the Core has been implemented in the package "eu.weknowit.social.admin.auth".

Below we will give a brief explanation for each Java Class in the Core. For extensive descriptions and explanations of the code we refer to the JavaDoc documentation generated directly from the code. The JavaDoc documentation is included in the deliverable.

6.4.1. Element

This class represents an element of the CAP methodology. Elements are the most basic structural component of the CAP. Every "higher" component is a element, too.

This class manages the the elements, take care about their persistent storage, the caching, retrieving and updating. Elements can be created or deleted.

6.4.2. Set

This class represents a non-persistent set of elements of the CAP. Sets are collections of elements.

Two different set types are represented by this class:

- Basic Sets: This set type includes are elements, that have been explicitly assigned to it.
- Meta Sets: This set type refers to another set, the basic set.

  A concatenation of relations (a so-called relation chain) is assigned to the Meta Set. The resulting elements of the Meta Set are calculated by applying the relations in the relation chain subsequently to the referenced basic set. Please note, that the basic set is only a reference, such elements can NOT BE DIRECTLY assigned to Meta Sets!

  The referenced basic set can be either a Meta Set or a Basic Set. The relation chain assigned must have the basic set as a source set.

We use the term Basic Set, if we want to refer to either the elements directly assigned to Set, or the elements directly assigned to the referenced (basic) set, in case of a Meta Set.

We use the term Result Set, if we want to refer to those elements, which are calculated by applying the relation chain to the basic set (in case of a meta set). The Result Set of a explicit set is the same as the Basic Set.

THIS SETS ARE NON-PERSISTENT! Use a PersistentSet, if use need persistence instead!

6.4.3. PersistentSet

A PersistentSet is the persistent extension of a Set. Please refer to the documentation of the Set class first!

A PersistentSet has as an internal CAP id. When a PersistentSet is created, a element of the type SET is created first, then the set as a collection is created.

By this, a set is on the one hand a element (of the type set) and on the other a collection of elements. Note, that a set might contain itself as a element!

Please refer to the description of the class set to understand the difference between Basic Sets and Meta Sets. The same differentiation applies to PersistentSet.

6.4.4. Relation

Represents a Relation in the CAP. A relation is defined on two sets, linking elements of the source set to elements of the target set. Source and target set may be the same set.

A relation is always also an element of the type Relation, as well as a link structure between sets.

6.4.5. RelationLink

This class represents a link in a relation on two sets. A link can be established only, if the two set elements to be linked are assigned to the sets defined by the relation (such source and target set of the relation). Otherwise a IdentifierException will be thrown.

6.4.6. RelationChain

A relation chain is a concatenation of Relations. A relation chain is a persistent object. It is also an element at gets assigned an element id.

6.4.7. Test

A Test defines two sets and an boolean operator. The Test compares both sets through using the operator and returns true or false. A Test is always atomic in the sense as only one boolean operator is used. For concatenation of tests please use AccessCondition.

A CAP Test currently only supports the boolean operator "has a non-empty intersection with”. By comparing two sets, the test becomes true, if at least one element is in both sets.

Instead of using a whole set as one parameter of the test, an allocation of the set can be used for the test. An allocation is a subset of elements of the test sets. If the boolean flag is set true, instead of using all elements of the test set only the allocation of this set is used.

A test is a element.

6.4.8. AccessCondition

An AccessCondition is a collection of tests which becomes true, if and only if all tests are true. An AccessCondition therefore is a logical conjunction of tests.

The entirety of all AccessConditions build the Access Base of a system, defining the rules, under which conditions access is granted or not.

Access Conditions are also elements with element ids.

6.4.9. Allocation

An Allocation is a hash table holding all current system states. More specific, in each set, the current valid element is assigned to this set.

Each set is identified by its internal id, building the hash keys. The current valid element of this set is identified by its internal id and stored in the corresponding value.

E.g. the current user is "Alice", the current object is "file1" then Allocation will hold the internal ids of user and object as keys and the internal ids of "Alice" and "file1" as corresponding values.

6.5. CAP CDL Parser

The CDL Parser is responsible for parsing text input from a InputStream. Valid tokens and expressions are defined and interpreted here.

We chose to use the JavaCC implementation as the Parser Generator for the CAP. JavaCC[15] seems to be a quasi-standard in the Java world for Compiler Generators as e.g. YaCC is for the C world.

The implementation has been done corresponding to the definition of the extended Backus-Naur-Form in chapter 9.1. All classes of the parser belong to the package "eu.weknowit.social.admin.parser".

15 **https://javacc.dev.java.net/**, last accessed at 30.06.2009

To separate the token registration and language detection from the Core Logic, a semi-layer has been introduced. This "Controller" class takes care about correct execution of CDL Parser Commands in the CAP Core.

6.6. CAP Web Service

The last layer of the CAP is the Web Service layer. For the CAP Prototype in this delivery only a simplistic and straight forward implementation has been realized for this module.

The implementation has been done in the package "eu.weknowit.social.admin.ui".

HTML is combined with AJAX technology to enable the user to create CDL statements in his web browser. A Java Servlet (implemented for Tomcat) runs several threads to communicate with the AJAX implementation. The Servlet holds an connections to the Parser transmitting CDL commands from the user to the parser and vice versa.

Exceptions and Errors are displayed separately from normal acknowledgements. A screen shot can be found below.

Illustration 15: Screenshot of the Web Service/User Interface of the CAP prototype

6.7. RDF-Layer

The implementation of this module relies on the SESAME, a framework for storing and querying RDF data. The functionality of the SESAME API used for CAP is the library for processing SPARQL queries.

The architecture of the RDF layer of the CAP is depicted in Illustration Illustration 16.

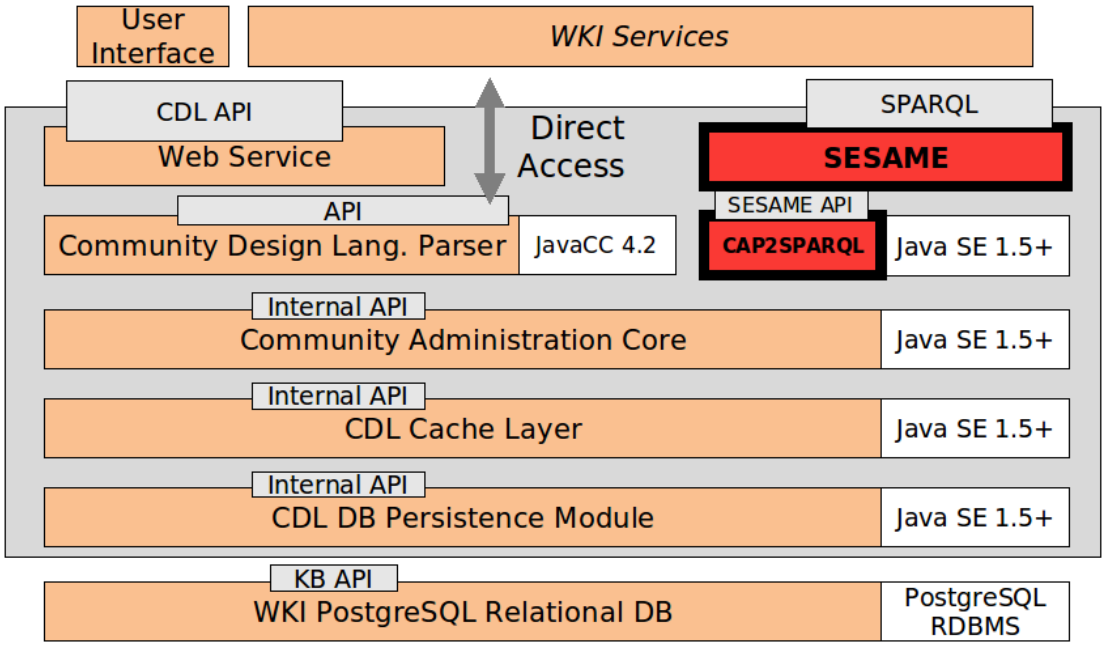

Illustration 16: Architecture of the RDF Layer

The CAP2SPARQL library is the core component of the RDF layer for CAP. CAP2SPARQL uses functionality provided by the Community Administration Core as Java methods. The SPARQL queries are pre-processed by the sesame API. So the CAP2SPARQL library implements the mapping functionality used for transformation of CAP concepts to RDF concepts.

The SESAME API can be queried with SPARQL via a Java interface. It is also possible to connect the CAP2SPARQL library via the SESAME API with a SESAME component that implements a SPARQL endpoint.

7. Use Case Application

7.1. Use Case Description

The Use Case described below will be used to demonstrate the feasibility and efficiency of the Community Administration Platform. The Use Case has been developed on the basis of the Emergency Case Scenario, please see D7.1 "Consumer and Emergency Response Use Case Initial Requirements".

As the scenario, of course, can not in detail describe access rights or policies, we will define a policy in the following. We took into account the scenario where possible.

Emergency Case (EC) Scenario Description CAP:

General Setup

- The Emergency Response Team (ERT) consists of 5 members M1, M2, ..., M5.
- For every Emergency Case (EC) a team of some of the above members is set up. A team can include from 1 to n (here n is maximal 5) members.
- Every Emergency Case has got one leader (ECL) and several members (ECM).
- Resources (files etc.) can be assigned to an EC.
- The permissions "read" and "write" are distinguished. Write permission shall always include read permission as well.
- An ECL can "write" any resource of "his" EC.
- Additionally an ECL of any EC can "read" all resources of any other existing EC. This applies to any future EC which might be created.
- An ECM is granted read access to the resources of his assigned EC.
- Every Emergency Response Team member has a proxy. This proxy can take over the role, the member has. (For example: If M1 is leader of EC1 and M2 is the general proxy for M1, then M2 gets the same rights concerning EC1 as M1, independent from M2's original rights.)

Situational data

- Currently 6 historic emergency cases are stored in the system, identified by EC1, ..., EC6.

- The leaders of EC1 ... EC6 are, respectively: M1, M1, M2, M3, M1, M3.
- The members of the ECs are:
  - EC1: M1, M2
  - EC2: M1, M2, M4
  - EC3: M2, M3, M4, M5
  - EC4: M1, M2, M3, M4, M5
  - EC5: M1, M4, M5
  - EC6: M3, M5
- Proxies:
  - M1's proxy: M3
  - M2's proxy: M3 and M4
  - M3's proxy: M1
  - M4 and M5 do not have proxies.
- Future Emergency Cases can be added at any time. By this, a ECL and members are assigned. The above policies must apply.

7.2. Use Case Implementation

We will now apply the methodology of the Community Design Language to the defined Use Case Policy of chapter 7.1.

7.2.1. Sets

Set	**Elements in Set**
User	M1, M2, M3, M4, M5
EC	EC1, EC2, EC3, EC4, EC5, EC6
Permission	read, write
Relations	perm_super, leader, member, proxy
Sets w/o Set	User, EC, Relations

Table 4: Sets present in the CAP ER Use Case Scenario

The sets User, EC and Permissions will be created. The set elements are depicted in the table above. To model the policies, four relations are

necessary. Each relation is ambivalent: A relation on the one side – which is described in the table below – and an element in the set “Relations”, as shown in the table above.

Three elements exist without an assignment to a set: The elements that are the sets User, EC and Relation.

Of course, all depicted sets are explicit, named sets.

7.2.2. Relations / Links

Relation	Source Set / Element	Target Set / Element
perm_super (r)	**Permission**	**Permission**
	write	read
leader (-)	**EC**	**User**
	EC1	M1
	EC2	M1
	EC3	M2
	EC4	M3
	EC5	M1
	EC6	M3
member (-)	**EC**	**User**
	EC1	M1, M2
	EC2	M1, M2, M4
	EC3	M2, M3, M4, M5
	EC4	M1, M2, M3, M4, M5
	EC5	M1, M4, M5
	EC6	M3, M5
proxy (r)	**User**	**User**
	M1	M3
	M2	M3, M4
	M3	M1

Table 5: Relations and Links present in the CAP ER Use Case Scenario

(r) := reflexive relation,
(s) := symmetric relation,
(t) := transitive relation,
(-) := relation is not reflexive, asymmetric and intransitive.

The table above depicts the definitions of the four relations. E.g. the relation perm_super is defined on the sets Permission and Permission. The relation is a reflexive relation (r). Within this relation, the element "write" is linked to the element "read".

Please be aware, that links in a relation are directed and not necessarily symmetric.

7.2.3. Graphical Illustration

The following illustration depicts the elements, sets and relations defined so far. Relations are colour-coded arrows, elements dots and sets circles.

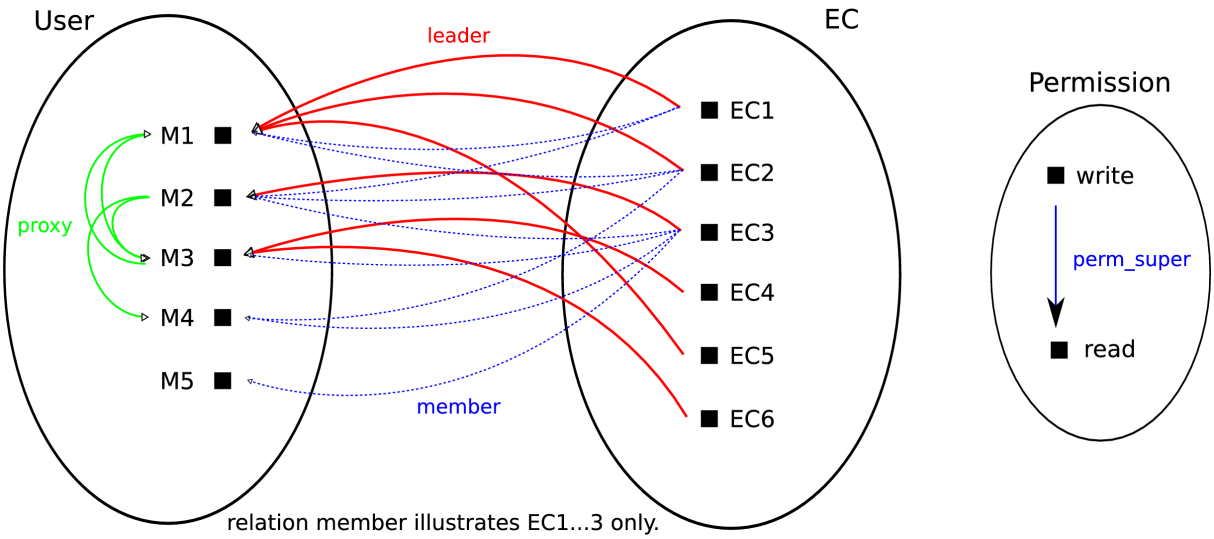

Illustration 17: Graphical representation of the CAP ER Use Case Scenario

On the left side of the illustration the set "User" is depicted having five set elements M1 … M5. The set EC in the middle has 6 elements EC1 … EC6.

The relation "leader" is defined on the set EC and User. The element EC1 is connected to M1. Semantically this makes user M1 the leader of the Emergency case 1. Please be aware that we chose to define the relation leader between EC to User and not User to EC. By this, the semantic interpretation is "a Emergency case EC has a leader" and not "Mx of the set User is the leader of the Emergency Case ECy". Of course, both definitions could be used, leading to different Access Conditions.

The relation "member" is only shown for EC1...3 for the sake of clarity. We can see, that M1 and M2 are both members of EC1.

7.2.4. Tests and Access Conditions

The following tests and access conditions will be necessary to model the use case. We will just list the conditions here and explain them in detail later.

1. [USER] θ [EC].member AND
   [PERMISSION] θ {read}
2. [USER] θ [EC].leader.proxy AND
   [PERMISSION] θ {write}.perm_super
3. [USER] θ EC.leader.proxy AND
   [PERMISSION] θ {read}

θ := boolean operator “intersection is not empty”

Access Condition 1

[USER] θ [EC].member AND
[PERMISSION] θ {read}

If the current allocation of the set “User” has a non-empty intersection with the current allocation of the set “EC”, to which the relation member is applied AND if the current allocation of the set permission has a non-empty intersection with the element “read” THEN grant access.

To explain this in greater detail, the access condition 1 consists of two tests logically connected by the boolean AND-Operator. The first test consists of the part

[USER] θ [EC].member

Tests consist always of a first set, the **test set**, a **boolean operator** and a second set, the **comparative set**. In this prototype the operator is limited to the operation θ := “has a non-empty intersection”.

[USER] denotes the **current allocation** of the set user, which is the current user, who wants to perform an operation in the system. E.g. M1 wants to access a file of resource EC5 by read. In this example, [USER] is transformed to {M1}.

This test set is compared to the expression [EC].member by the boolean operator θ. [EC], again, is the current allocation of the set EC. Continuing our above example, the allocation is {EC5}.

To this one-element set, the relation “member” is applied. The resulting set is {M1, M2, M5}, as the set relation member links the element EC5 to the elements M1, M2 and M5.

The test set [USER], which results in {M1} is then compared by the operator “θ” with the second set [EC].member, which is {M1, M2, M5}. θ is defined as “has a non-empty intersection”, which is obviously true.

The second test of the access condition is

[PERMISSION] θ {read}

Applying the same syntax interpretation, the current allocation of the set “permission” must have a non-empty intersection (θ) with the one-element set “read”. In our example, the allocation for the permission set is “read”. Therefore, this test is true, too.

As both tests have the result true, access is granted.

Semantically this tests grants read access to all members of a emergency case.

Access Condition 2:

[USER] θ [EC].leader.proxy AND
[PERMISSION] θ {write}.perm_super

Again, this access condition consists of two tests. The test set [USER] is the current allocation of the set user. Let us continue the example of the first access condition, where user “M1” wants to access a file of the resource “EC5” by read. [USER] then results in the one-element set {M1}.

This test set is compared to the comparative set [EC].leader.proxy. [EC] is, again, the current allocation of the set EC, here {EC5}. To this set two relations are applied consecutively. {EC5}.leader results in {M1}. {M1}.proxy in {M1, M3} as the relation “proxy” is defined as reflexive.

The comparison of the test set {M1} and the comparative set {M1, M3} is true.

The second test “[PERMISSION] θ {write}.perm_super” results in:

- test set: {read}
- comparative set: {write, read}

Therefore the result of the second test is true, as well. Obviously, the access condition is true then, too.

Semantically this access condition grants read and write access to all EC leaders and proxies in their own ECs.

Access Condition 3:

[USER] θ EC.leader.proxy AND
[PERMISSION] θ {read}

For the sake of completeness, the semantic interpretation of this access condition is: **Give read access to all users to all resources, which are at least leader (or proxy of a leader) in one EC.**

"EC" without "[]" hereby denotes any element of the set "EC". Please do not mix this up with "[EC]", which denotes the current allocation of the set "EC".

7.2.5. CDL Expression

To create these structures on a system using the Community Design Language, several elements, sets and relations must be created to allow access conditions and tests to be executed. To create these elements, the following CDL expressions can be used. For a complete syntax explanation of the CDL syntax refer to section 9.1.

- CREATE SETS

  User: {M1, M2, M3, M4, M5},

  EC: {EC1, EC2, EC3, EC4, EC5, EC6},

  Permission: {read, write};

Create three independent sets "User", "EC" and "Permission". Create and assign the denoted elements to the sets, e.g. for the set "Permission" the elements "read" and "write".

- CREATE RELATIONS

  perm_super (Permission, Permission) REFLEXIVE:
  {(write,read)},

  leader (EC, User):
  {(EC1,M1), (EC2,M1), (EC3,M2), (EC4,M3), (EC5,M1),
  (EC6,M3)},

  member (EC, User):
  {(EC1,M1), (EC1,M2),
  (EC2,M1), (EC2,M2), (EC2,M4),
  (EC3,M2), (EC3,M3), (EC3,M4), (EC3,M5),
  (EC4,M1), (EC4,M2), (EC4,M3), (EC4,M4), (EC4,M5),
  (EC5,M1), (EC5,M4), (EC5,M5),
  (EC6,M3), (EC6,M5)},

proxy (User, User) REFLEXIVE:
{(M1,M3), (M2,M3), (M2, M4), (M3,M1)};

Create four different, independent relations, “perm_super”, “leader”, “member” and “proxy”. The relation “perm_super” is defined on the source set “Permission” and the target set “Permission”. The relation is reflexive. After creating the relations, link the denoted elements by the relation. E.g. link the element “write” to “read” by the relation “perm_super”.

- CREATE TESTS
  test_isleader: ([User], [EC].leader.proxy),
  test_ismember: ([User], [EC].member);
- CREATE TESTS
  test_perm_read: ([Permission], {read}),
  test_perm_write: ([Permission], {write}.perm_super);

Create four (two and two) tests. Each test consists of a test set and a comparative set. We will explain the test “test_isleader” indetail: The expression “[USER]” is the test set, the comparative set in the expression “[EC].leader.proxy”. Both result sets are tested to have a non-empty intersection.

- CREATE ACCESSCONDITIONS
  ac1: (test_isleader, test_perm_write),
  ac2: (test_isleader, test_perm_read),
  ac3: (test_ismember, test_perm_read);

Create three access conditions, each as list of tests. Access Condition “ac1” consists of the tests “test_isleader” and “test_perm_write”.

Please note, that the prototype only supports the boolean operator “non-empty intersection”, which is not denoted.

7.2.6. SPARQL Query

To query the CAP in order to retrieve information about access grants for specific access allocations, the RDF-layer of the CAP can be used. To

check e.g., whether M4 can read on EC3, the corresponding SPARQL query would look as follows:

```
select ?grant where
{?situation rdf:type wki:AccessSituation.
?situation wki:Permission wki:read.
?situation wki:EC wki:EC3.
?situation wki:User wki:M4.
?situation wki:isGranted ?grant.}
```

8. Conclusion

The Community Administration Platform provides extensions to existing state-of-the-art access control methodologies like RBAC or ACLs. By introducing n-tuples instead of the traditional 3-tuple (User, Permission, Object) the CAP enables more and flexible conditions to formulate access rights. By the introduction of tests on sets as boolean functions we reach a higher expression power than in existing approaches. The Community Design Language as a structured definition language enables a systematic, provable and clearly defined way, to formulate policies in the CAP. Policies and the access decisions derived from these policies are made available via Web Service and as RDF data via a SPARQL interface.

The CAP is technically organized in seven layers. The persistence storage is a relational database system. It is accessed by the Persistence Layer managing the database connections and tables. Select, update and delete access are made transparent by this layer. Above, the Cache Layer supports a fast, scalable and efficient methodology to store and access CAP objects. The Core Layer implements the CAP logic, creates and deletes objects, ensures consistency and does the reasoning. The Parser Layer is interpreting CDL statements for the Core Layer. Textual input and output are processed by the Parser Layer. A direct access through the parser's API is possible for other work packages and services. The last two layers provide a web service for the communication with the Parser Layer, a textual user interface for remote CDL execution and a SPARQL interface for retrieving access decisions as RDF data. By this organization we hope to support a flexible, well-structured implementation of the CAP.

We introduced an example based on the Emergency Case Scenario to demonstrate the feasibility of the CDL. Also we have shown, that besides its high expressiveness, the CDL/CAP offers a simple way to realize policies in the CAP system.

The Community Membership Life Cycle Model is an approach to systematically describe potential roles in virtual online communities. We hope to support the interests and needs of such role inhabitants by an explicit description of these roles. Through linking social roles to authorative roles we hope to enable automatic authorative role assignment by next version CAP.

In future work we plan to extend the functionality of the CAP by enabling automatic role assignment. We plan to realize a compound service Task T4.1 “Cross-usage of intelligence” by combining the functionality of User Profiles (WP1), the Community Analysis Tool (T4.1) and the CAP. Furthermore it is planned to develop an interface between the CAP and Distributed Groups (WP5) and use the CAP in joint services of other work packages.

9. Appendix

9.1. Syntax of the Community Design Language

9.1.1. General Operations

<CDL Expression> → [<Action>;]*

<Action> → CREATE <Creation> [“,” <Creation>]*
| DELETE <Deletion> [“,” <Deletion>]*
| CHECK <Access Test> [“,” <Access Test>]*
| LIST <Listing> [“,” <Listing>]*
| VERSION

9.1.2. Action Statements

<Creation> → ELEMENTS <EleDef> [“,” <EleDef>]*
| SETS <BasicSetDef> [“,” <BasicSetDef>]*
| SETASSIGNMENT <BasicSetAssignment>
[“,” <BasicSetAssignment>]*
| RELATIONS <RelDef> [“,” <RelDef>]*
| LINKS <LinkDef> [“,” <LinkDef>]*
| TESTS <TestDef> [“,” <TestDef>]*
| ACCESSCONDITIONS <ACDef> [“,” <ACDef>]*

<Deletion> → ELEMENTS (<element> [,])+
| SETS (<setname> [,])+
| SETASSIGNMENTS <BasicSetDef>
| RELATIONS (<relname> [,])+
| LINKS <LinkDef>
| TESTS (<testname> [,])+
| ACCESSCONDITIONS (<acname> [,])+

<Access Test> → CHECK TEST <testname>: <AccessAllocation>
| CHECK ACCESSCONDITION <acname>:
<AccessAllocation>
| CHECK ACCESS: <AccessAllocation>

<Listing> → ELEMENTS <ListELements>
| SETS [ANONYMOUS]
| RELATIONS [ANONYMOUS]
| TESTS [ANONYMOUS]
| ACCESSCONDITIONS [ANONYMOUS]

9.1.3. General Data

<EleDef> → ([<setname> ":"] <ElementList> [","])+

<BasicSetDef> → (<setname> [":" <ElementList>] [","])+

<RelDef> → (<relname> "(" <setname>, <setname> ")"
["REFLEXIVE"] ["SYMMETRIC"] ["TRANSITIVE"]
[":" <ElementTuples>]
[","])+

<TestDef> → <testname>: "(" <set>, <set> ")"

<ACDef> → <acname>: "(" <testname> [, testname]* ")"

9.1.4. Sets

<set> → <basic_set> | <meta_set>

<basic_set> → <named_set> | <anonym_set> | <allocated_set>

<meta_set> → <basic_set>(.relation)+

<named_set> → <setname>

<anonym_set> → <ElementList>

<allocated_set> → "[" <setname> "]"

9.1.5. Set Assignments / Links

<BasicSetAssignment> → (<setname> ":" <ElementList> [","])+

<LinkDef> → (<relname> ":" <ElementTuples> [","])+

9.1.6. Elements

<ElementList> → "{" (<element> [,])+ "}"

<ElementTuples> → "{" (<ElementTuple> [,])+ "}"

<ElementTuple> → "(" <element>, <element> ")"

9.1.7. Allocations

<AccessAllocation> → "(" (SetAllocation [,])+ ")"

<SetAllocation> → <setname> "=" <element>

9.1.8. Listing

<ListElements> → [ANONYMOUS] [WITH SETASSIGNMENTS] [WITH TYPE] [IN SETS <setname>]

9.1.9. Atomic Elements

<element> → <identifier>

<setname> → <identifier>

<relname> → <identifier>

<testname> → <identifier>

<acname> → <identifier>

<identifier> → (A-Za-z)[A-Za-z0-9]*

9.2. Usage of the RDF-layer (CAP2SPARQL)

How work the SPARQL2CAP Translator, using the Sesame Framekwork? This is a short overview about the most important steps:

1. Create a new memory store

Code:

```
MemoryStore memory = new MemoryStore();
```

2. Create a new CAPSail and set the base Sail that this Sail will work on top of. This method will be called before the initialize() method is called. Sail is an interface for an RDF Storage. It can store RDF statements and evaluate queries over them. Statements can be stored in named contexts or in the null context.

Code:

```
CAPSail sail = new CAPSail();
sail.setBaseSail(memory);
```

3. Create a new SailRepository, initialize it and get the RepositoryConnection. SailRepository is an implementation of the Repository interface that operates on a (stack of) Sail object(s). The

behaviour of the repository is determined by the Sail stack that it operates on.

Code:

```
Repository rep = new SailRepository(sail);
rep.initialize();
RepositoryConnection con = rep.getConnection();
```

4. Now create a SPARQL-query and evaluate it

Code:

```
String query = "select ?result where { ?a rdf:type ?result. };
TupleQuery tupleQuery =
con.prepareTupleQuery(QueryLanguage.SPARQL, tupleQuery);
TupleQueryResult result = tupleQuery.evaluate();
```

The method prepareTupleQuery parses depending of the QueryLanguage the given TupleQuery and checks the syntax. Evaluate() compute the result of the Query. Every node of it will be visited and the specific CAPQuery-Informations are collected and build together a specific CAPQuery Object, which extends the AbstracCAPQuery class. The class AbstracCAPQuery implements the interface CloseableIteration.

5. Print result with Iteration

Code:

```
while (result.hasNext()) {
        BindingSet bs = result.next();
        System.out.println(bs);
}
```

This CloseableIteration is linked with the Iteration Implementation of the AbstractCAPQuery. The first call of next() instantiates a CAP API-Call an stores results in an ArrayList and the Iteration just return the entries of it.

[1]Different Authors. WeKnowIt Project. Description of Work, 2007.

[2] D7.1 Consumer and Emergency Response Use Case Initial Requirements, WeKnowIt Project, 30.9.2008

[3]Howard Rheingold. The Virtual Community: Homesteading on the Electronic Frontier, revised edition. The MIT Press, November 2000. ISBN 0262681218.

[4]Sandhu, R.S. AND Samarati, P. Access Control. Communications Magazine, IEEE, Volume 32, Number 9, p. 40 – 48. 1994.

[5]Joshi, James B D and Bertino, Elisa and Ghafoor, Arif. Temporal Hierarchies and Inheritance Semantics for GTRBAC". In Proceedings of the 7th ACM Symposium on Access Control Models And Technologies, p. 74-83. Published by ACM in 2002.

[6]Bruce W. Tuckman. Developmental Sequence in Small Groups. Psychological Bulletin, Volume 63, Number 6, p. 384-399. 1964.

[7]Bruce W. Tuckman and M. A. Jensen. Stages of Small-Group Development Revisted. Group Org. Studies, Volume 2, p. 419-427. 1977.

[8]Caroline Haythornthwaite and Michelle M. Kazmer and Jennifer Robins and Susan Shoemaker. Community Development Among Distance Learners: Temporal and Technological Dimensions. Journal of Computer-Mediated Communication, Volume 6. June 2000.

[9]Christopher M. Johnson. A survey of current research on online communities of practice. The Internet and Higher Education, Volume 6, p. 45-60. 2001.

[10]Jeremiah K. Owyang. Online Community Best Practices. Forrester Research. February 2008.

[11]Amy Jo Kim. Community Building on the Web. Peachpit Press. 2000. ISBN 0-201-87484-9.

[12]Rena M. Palloff and Keith Pratt. Building learning communities in cyberspace. Jossey-Bass, San Francisco, Calif. 1999. ISBN 0-7879-4460-2.

[13]Wasserman, S., & Faust, K. (1994). *Social network analysis: Methods and applications*. Structural analysis in the social sciences, 8. Cambridge: Cambridge University Press.

[14]Coase, R. H. The Nature of the Firm. Economica, vol. 4, number 16, p. 386-405, Nov. 1937

[15]The Hackers Dictionary. Weblink. http://www.catb.org/~esr/jargon/html/T/troll.html

[16]R.S. Sandhu und P. Samarati. Access control: principle and practice. Communications Magazine, IEEE 32, no. 9 (1994): 40-48.

[17] Klaas Sikkel, A Group-based Authorization Model for Cooperative Systems. In Proceedings of the Fifth European Conference on Computer-Supported Cooperative Work (Springer, 1997).

[18] Andreas Geyer-Schulz and Anke Thede. Implementation of hierarchical authorization for a web based digital library. In 3RD INT CONF ON CYBERNETICS AND INFORMATION TECHNOLOGIES, SYSTEMS, AND APPLICATIONS, p. 139-144.

[19] David F. Ferraiolo und D. Richard Kuhn. Role-Based Access Controls. In Proceedings of the 15th National Computer Security Conference, Baltimore MD, 1992, p. 554 – 563.

[20] "Oracle rechnet mit Erlaubnis für Sun-Kauf". Financil Times Deutschland. Monday, 29 June 2009. Page 7.