D3.2
Electronic institutions for community building (v2)

<table>
<thead>
<tr>
<th>Core authors</th>
<th>Dave de Jonge, Nardine Osman, Bruno Rosell i Gui, Carles Sierra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maintainers/Editors</td>
<td>Dave de Jonge</td>
</tr>
<tr>
<td>PRAISE research group leaders</td>
<td>Mark d’Inverno, François Pachet, Carles Sierra, Luc Steels</td>
</tr>
<tr>
<td>PRAISE project leader</td>
<td>Carles Sierra</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Grant agreement no.</th>
<th>318770</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project acronym</td>
<td>PRAISE - Practice and peRformance Analysis Inspiring Social Educa- tion</td>
</tr>
<tr>
<td>Date</td>
<td>July 19, 2014</td>
</tr>
<tr>
<td>Distribution</td>
<td>Public</td>
</tr>
</tbody>
</table>
Disclaimer

The information in this document is subject to change without notice. Company or product names mentioned in this document may be trademarks or registered trademarks of their respective companies.

This document is part of a research project funded by the European Community as project number 318770 acronym PRAISE under THEME 3: FP7-ICT-2011-8 Collaborative project.

Abstract

Keyword list: Electronic institutions, community building, human interface, web services, community rules
## Changes

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Author</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>30.07.13</td>
<td>Bruno Rosell i Gui</td>
<td>Creation</td>
</tr>
<tr>
<td>0.2</td>
<td>10.07.14</td>
<td>Dave de Jonge</td>
<td>Initial version</td>
</tr>
<tr>
<td>0.3</td>
<td>14.07.14</td>
<td>Nardine Osman</td>
<td>Add part for self-evolving communities</td>
</tr>
<tr>
<td>0.4</td>
<td>19.07.14</td>
<td>Dave de Jonge, Nardine Osman, Bruno Rosell i Gui</td>
<td>Improve document</td>
</tr>
<tr>
<td>0.5</td>
<td>25.07.14</td>
<td>Dave de Jonge</td>
<td>Included Ismel’s adaptations to the Simplified Institution Editor section, and expanded the Executive Summary</td>
</tr>
<tr>
<td>0.6</td>
<td>25.09.14</td>
<td>Dave de Jonge</td>
<td>Increased size of figure 2 and added an example to the first paragraph of Section 1.1.2 (to take comments from Advisory Board into account).</td>
</tr>
</tbody>
</table>
Executive Summary

The first part of this deliverable describes the work realised during the second year of the PRAISE project regarding the development of Electronic Institutions as a basic framework for building social e-learning communities. We explain how the new tools created in the first year have been further developed in order to make them suitable for the development of online learning communities.

The goal of this work package is to implement the infrastructure that facilitates the creation and management of communities. The backbone of this part of this infrastructure is formed by the so-called Electronic Institution technology. Electronic Institutions provide a computational analogue of human organisations in which intelligent agents play different organisational roles and interact to accomplish individual and organisational goals. Hence they enable users to implement and enforce the norms of a social community.

However, in order to make this existing technology useful in the context of PRAISE, we have had to extend it with a number of new features, as explained in the previous deliverable. That is, we have had to implement a user interface that allows humans to participate in an Electronic Institution, whereas the existing software only allowed software agents to participate. Furthermore, we implemented a P2P version of the infrastructure to improve scalability, privacy, and maintainability.

In part I of this new deliverable we discuss how we have improved and extended these features. Most importantly, we have implemented a database management system that enables users to share files over the P2P network and adapt them, while keeping the database updated. Also, we have introduced the possibility for the users to share Resources such as files. In order to allow the designer of an institution to control which users have access to such resources and when, we have implemented a generic agent, called the ResourceAgent. It is programmed in such a way that it can, with some limitations, participate in any Electronic Institution, so one does not need to program a new resource agent for every new institution specification. Also, we describe several improvements that we have made to the automatically generated user interface presented in last year’s deliverable. It now gives more control to the user, as some actions available to agents were not yet available to human users.

Another essential feature that was missing from the existing infrastructure was a user-friendly editor that allows users to design new specifications of Electronic Institutions. The existing tool for the design of specifications, called Islander, was designed to be used by computer scientists, and is therefore not suitable for average users. In PRAISE however, we want teachers without better-than-average computer skills to be able to design lesson plans, and we want the students to be able to build communities with customised norms. For this reason, we have implemented a new EI-specification editor, which we introduce in this deliverable. It is in fact a highly simplified version of the Islander tool. This has the advantage that it can be used by any person with average computer skills, which makes it useful for, for example, music teachers that want to create a lesson plan for their students.
Finally, in order to present these tools in a coherent way to the final user we have created a platform, called PeerFlow that integrates all of the aforementioned technologies into one single framework. It allows users to download the software necessary to connect to the P2P network, and having started this software, allows them to access the new editor to design an EI-specification, and publish it on the P2P-network such that other users connected to the network can find it and launch a new instance of an Electronic Institution based on that specification. It allows users to search for such specifications by providing key words or by filtering the published specification based on other properties, such as its name or the name of its designer. Once a new institution is launched, users can enter it through the same PeerFlow software and participate in it.

Unlike the first part of this deliverable, which presents our implementation details for the community tools, the second part presents the results of more theoretical research that studies self-evolving communities. We argue that just like human communities, e-communities (defined by their software) also need to self-evolve. Instead of creating numerous rigid systems, what we should aim at instead is providing tools for creating self-evolving systems that adapt to the community’s needs. We believe different communities should be governed by different rules. These rules should be an ever evolving set resulting from the aspirations of its members. Furthermore, for the community’s rules to be effective, they need to be tailored to the specific character traits of the community members as well as considering some other external influences. We note that in this second part of the deliverable, we talk about self-evolution, as opposed to evolution, since we are interested evolution that is designed and directed by the community itself. We propose a roadmap for self-evolving communities that is based on a set of building blocks needed for self-evolution, and we refer to these building blocks as the community charter. We also presents an approach for each of these blocks, helping us build the first prototype for self-evolving communities.
Contents

1 Community Tools for Social Learning Communities

1 Introduction

1.1 Electronic Institutions Overview

1.1.1 Main Concepts

1.1.2 Electronic Institution Development Environment

1.1.3 AMELI

1.2 Adaptations to the EI Infrastructure for PRAISE

2 PeerFlow

2.1 Searching and Launching an Institution

2.2 Searching and Joining an Institution

2.3 Publishing new Institution Specifications

2.4 Future Work

3 Peer to Peer Electronic Institutions

3.1 Concepts

3.2 Resource Repository

3.3 Searching for an Electronic Institution Specification

3.4 Searching for running Electronic Institutions

3.5 Resource Agent

3.5.1 Behaviour in a Performative Structure

3.5.2 Behaviour in a Scene

3.5.3 Starting and configuring

3.6 Future work

4 REST Governor

4.1 Concepts

4.2 Rest Actions

4.2.1 Create resource

4.2.2 Get resource properties

4.2.3 Set resource properties

4.2.4 Get file resource property

4.2.5 Set resource property

4.2.6 Get resource property values

4.2.7 Set resource property values

4.2.8 Search for resources

4.3 Future work

5 Simplified Institution Editor

5.1 Simplified Institutions

5.2 The Simplified Editor

5.3 Resources

5.4 An Example
11 Diagram of peer to peer Electronic Institutions components ........................................ 14
12 Diagram of the interaction of the REST Governor with the peer to peer Electronic
   Institutions components ......................................................................................... 21
13 A screenshot of the simplified institution specification editor ........................................ 26
14 Elements involved in the design of a simplified institution: Scene, Role, Resource
   and File .................................................................................................................. 26
15 Screenshot of the properties for scene ‘Examination’ and roles ‘Teacher’ and ‘Stu-
   dent’ ....................................................................................................................... 27
16 Electronic Institution obtained from the simplified Institution shown in Figure 13 ............. 28
17 Pseudocode for norm enforcement ........................................................................... 37
18 Interaction protocol example ..................................................................................... 38

Listings
1 Example for configuring the resource agent with name dbAgent and a polling time
   of 15 seconds .......................................................................................................... 19
Part I

Community Tools for Social Learning Communities

1 Introduction

In this first part of the deliverable we explain the work realised during the second year of the PRAISE project regarding the creation of tools to support social learning communities. This work is focused on satisfying the requirements described in deliverable D1.4 [27], which are mainly the following two: to adapt the current Electronic Institution software to support learning communities, and to facilitate the specification of community norms using Electronic Institutions (EI).

Before we explain more about the realised work we will, in the next section, introduce the concept of electronic institutions and the related software that has already been developed previously. Unfortunately we do not have the space here to go into a detailed description of electronic institutions, so we will only give a short overview and introduce the main concept that are necessary to understand for the rest of this deliverable. For a more in-depth discussion of electronic institutions we refer to, for example, [3, 6, 21, 29].

1.1 Electronic Institutions Overview

As the complexity of real-world applications increases, particularly with the advent of the Internet, there is a need to incorporate organisational abstractions into computing systems that ease their design, development, and maintenance. Electronic Institutions are at the heart of this approach [10]. Electronic Institutions provide a computational analogue of human organisations in which intelligent agents play different organisational roles and interact to accomplish individual and organisational goals. In this scenario, agent technology helps enterprises reduce their operational costs and speed-up time to market by helping distributed business parties, represented by agents, run smoother and in a better coordinated fashion. Electronic Institutions appear as the glue that puts together self-interested business parties, coordinating, regulating, and auditing their collaborations.

Just like any human institution, an Electronic Institution is a place where participants come together and interact according to some pre-defined protocol. It makes sure that the norms of the institution are enforced upon its participants and thus prevents them from misbehaving. An Electronic Institution therefore provides the infrastructure in which agents can interact in an autonomous way within the norms of the institution.

A commonly cited example is that of a fish market, with buyers and sellers engaging in interactions aimed at buying and selling fish. They have strict conventions by which fish is traded under strict negotiation protocols. More specifically, the fish market is an auction house that enforces certain conditions on the eligibility of traders, the availability and delivery of goods and the behaviour of participants. While the actual trading makes up the critical part of the fish market, there are other interactions that are also governed by rules. For example, before any trading can
be undertaken, sellers must deliver fish to the market, and buyers must register. Furthermore, once a deal has been agreed, the sellers must pay for and collect the fish, and the buyers must collect payment. Beyond this example, many other institutions have similar sets of distinct activities that can be identified, like hotels and universities.

Electronic Institutions have been under development for more than 15 years [3, 6, 21, 29] which has resulted in a large framework consisting of tools for implementing, testing, running and visualizing them. Therefore, before we continue, we need to introduce the basic concepts behind the Electronic Institutions framework.

1.1.1 Main Concepts

Scenes Just as there are meetings in human institutions in which different people interact, Electronic Institutions have similar structures, known as scenes, to facilitate interactions between agents. Scenes are essentially group meetings, with well-defined communication protocols (see Figure 1) that specify the possible dialogues that agents are allowed to have within these scenes. For example, an electronic fish market may include an auction scene in which buyers compete to purchase fish, with a protocol that involves making bids. There may be many simultaneous instances of such auctions within a fish market, each referred to as a Scene instance.

Scenes within an institution are connected in a network that determines how agents can legally move from one scene to another. This network is called the Performative Structure. In the fish market example a buyer can only enter the auction scene after passing the registration scene.

Actions Activities are dialogical as they are achieved via agent interactions composed of non-divisible utterances, that occur at discrete instants of time. These utterances can be modelled as messages that conform to a certain pattern, and physical actions are represented by appropriate messages of this form.

Figure 1: Login scene example

1. (request (?x guest) (?y staff) login(?user ?email))
2. (inform (ly staff) (!x guest) accept())
3. (failure (ly staff) (!x guest) deny(?code))
4. (request (?x guest) (ly staff) login(?user ?email))
5. (inform (!y staff) (all guest) close())
6. (inform (?y staff) (all guest) close())
In a fish auction, for example, a buyer commits to buy a box of fish at a certain price by making a bid, while the actual physical action of transferring money from the buyer to the auction house is triggered when the auctioneer declares that the box is sold. In the rest of the document we will therefore use the words ‘action’ and ‘message’ interchangeably.

For each message that can be sent, a number of parameters may be specified by the protocol. When making a bid in an auction for example, the maker of the bid should include the amount of money he bids in the message. The Electronic Institutions framework supports several basic parameter types, such as ‘Integer’, ‘String’ and ‘Boolean’. Apart from these basic types the designer of an institution can define custom types, which are composed of one or more parameters of a basic type.

**Scene Protocols** The interactions between the agents in a scene in an Electronic Institution have to follow a certain protocol. The protocol defines which agent can say what and when within the scene. At each moment during the execution of a protocol, the protocol is in a certain *state*, depending on the messages that have been said so far. The current state of the protocol determines what kinds of messages each agent can send.

In an auction for example, the protocol may start in a state in which the auctioneer introduces the next item under auction. Participants are not allowed to make any bid yet in this state. Once the auctioneer announces the start of the auction, the state changes to a bidding state, in which the participants are allowed to make their bids.

A protocol is therefore represented as a directed graph in which the nodes are the states of the protocol. Each edge of the graph is labelled with one or more message patterns (see for example Figure 1). A message can only be sent if it satisfies one of the patterns labelling one of the outgoing arcs from the current state.

**Roles** Scene protocols are not specified in terms of agents, but rather in terms of *roles*. Every agent plays a specific role that determines which actions it can take at which moment. Roles can be understood as standardized patterns of behaviour that agents, when instantiating a role, must respect. The identification and regulation of roles is considered as part of the formalisation process of any organisation [23]. Any agent within an Electronic Institution is required to adopt some role(s). A major advantage of using roles is that they can be updated without having to update the actions for every agent on an individual basis. Recently, the concept of role is becoming increasingly studied by software engineering researchers and even more recently by researchers in the agents community.

**Constraints** As explained above, the state of the protocol restricts the set of possible actions that can be taken by the agents. Moreover, this set of possible actions can be restricted even further by including constraints in the protocol. Constraints are given as sentences in a first-order logic attached to a message pattern. A message can only be sent if its corresponding constraints are satisfied.

**Governors** Each agent participating in an Electronic Institution has a special agent assigned to it, called its Governor. The Governor of an agent $\alpha$ has control over each message that is being sent by $\alpha$. Whenever $\alpha$ tries to send a message, this message first passes $\alpha$’s Governor, which checks whether the protocol is in the correct state and whether the corresponding constraints are satisfied. If so, the Governor forwards the message to its recipient. If not (for
example, because the agent made a bid that is higher than what he can afford), the Governor blocks the message.

**Ontology** As explained above messages can have parameters, which can be of a basic type or of a user-defined type. Each Electronic Institution has an Ontology associated to it that stores the definitions of these user-defined types. Also it stores for each message how many parameters it has and which types those parameters have.

**Dialogical framework** Some aspects of an institution such as the objects of the world and the language employed for communicating are fixed, constituting the context or framework of interaction amongst agents. In a dialogic institution, agents interact through speech acts. Institutions establish the acceptable speech acts by defining the ontology (vocabulary) — the common language to represent the "world" — and the common language for communication and knowledge representation which are bundled in what we call dialogic framework. By sharing a dialogic framework, we enable heterogeneous agents to exchange knowledge with other agents.

**Transition** When moving from one scene to another an agent always passes a so-called transition. When the agent is at a transition it can choose which role it will play in the scene it is going to, and it can wait for other agents so that they can move to the next scene simultaneously.

**Performative structure** Scenes and transitions are connected, composing a network of scenes which is called the performative structure, and which captures the existing relationships among scenes (see figure 2). The specification of a performative structure contains a description of how the different roles can legally move from scene to scene. Agents within a performative structure may be participating in different scenes, with different roles, at the same time.

**Normative rules** Agent actions in the context of an institution may have consequences that either limit or enlarge its subsequent acting possibilities. Such consequences will impose obligations to the agents and affect their possible paths within the performative structure.

### 1.1.2 Electronic Institution Development Environment

The design and execution of an Electronic Institution is realised through a framework called The Electronic Institutions Development Environment (EIDE), which is implemented in Java. It consists of a set of tools aimed at supporting the engineering of intelligent distributed applications as Electronic Institutions [9]. EIDE allows for engineering both Electronic Institutions and their participating software agents. Notably, EIDE moves away from machine-oriented views of programming toward organizational-inspired concepts that more closely reflect the way in which we may understand distributed applications. It supports a top-down engineering approach: firstly the organization, secondly the individuals.

In the context of PRAISE we may for example imagine a teacher designing a lesson plan for its students. This teacher would then use EIDE to implement this lesson plan as an Electronic Institution. Such an institution may for example forbid students to enter the second level course, if the student hasn’t yet passed the first level course. Once the teacher has fully specified the
The two main components of EIDE are:

**ISLANDER** A graphical tool that supports the specification of the rules and protocols in an electronic institution [8] (see Figure 3). It allows you to visually define the scenes, roles, protocols, message patterns, constraints, ontology and other components of the institution. It then converts the visual representation into xml format (the EI-specification) that can be read by AMELI.

**AMELI** Software platform to run electronic institutions [11]. Once an electronic institution is specified with ISLANDER it can be executed by AMELI. AMELI comprises of a set of agents that control the execution of scene instances, and is responsible for assigning a Governor to each participating agent. An Electronic Institution can be executed by starting AMELI with an EI-specification. When it is running, agents can join it by requesting entrance to the institution, and, once entered, they can start communicating according to the protocols of the institution.

### 1.1.3 AMELI

The software that provides the execution of electronic institutions, is called AMELI[11]. It enables agents to act in an electronic institution and controls their behaviour. The main features of AMELI are:
To provide a way for different agents with different architectures to communicate with one another without any assumption about their respective internal architectures.

To enforce a protocol of behaviour as specified in an institution specification upon the agents. This means that AMELI makes sure that the agents can only do those actions that the protocol allows them to do.

Figure 4 describes the different parts of AMELI. It consists of three layers: a communication layer, which enables the agents to exchange messages, a layer consisting of the agents that are acting in the institution, and in between a social layer, that controls the behaviour of the participating agents. The social layer consists of the following agents:

**Governor** For each participating agent in an institution there is one Governor. This Governor forms the link between a participating agent and the Electronic Institution. Every time the participating agent wants to make an action in the institution, this is communicated to the governor in the form of a message. The governor determines whether the participating agent is indeed allowed to do that action at that moment and, if so, forwards the message to other agents in the social layer in order for the action to have effect. If the participating however is not allowed to do that action, the governor simply blocks the message and hence prevents the action from being executed. Furthermore, the Governor keeps the status of the agent in the Electronic Institution and its properties.

**SceneManagers** There is a scene manager for each instance of a scene protocol that is active in
the electronic institution. This agent controls and keeps the status of a scene protocol of an Electronic Institution. So, it controls when an agent can enter or exit the scene protocol, the scene state and of when an agent can say a message to another.

**TransitionManagers** There is one transition manager for each transition in each active performative structure. This agent controls and synchronizes the movements of the agents between the nodes in a performative structure.

**EInstitutionManager** This agent is unique to each running instance of an Electronic Institution and it keeps the status of the Electronic Institution. It also allows or denies an agent to enter into the Electronic Institution.

The communication between the agents of the social layer is taken care of by a generic communication layer that can be implemented using various technologies. This allows us to distribute AMELI over multiple machines. The communication between the agent and its Governor on the other hand does not take place in this communication layer. The communication channel between them is created when the agent tries to enter into the electronic institution. They could communicate through a socket connection or directly at the code-level if the agent is created as a service in the Electronic Institution.

### 1.2 Adaptations to the EI Infrastructure for PRAISE

AMELI and Islander have been under development for more than 15 years. However, we considered that some essential elements were missing in order to apply this technology to PRAISE. These elements are:
• A user interface that allows humans to participate in Electronic Institutions.

• A peer-to-peer infrastructure.

• A simplified version of Islander, that allows music teachers with average computer skills to define institutions.

This work package focuses on the implementation of these three components. In the previous deliverable we have already described the first two of these, so we will only explain the improvements we have made on these components during the past year. The simplified institution editor is a new tool that we have started working on in the past year, so we will give a complete introduction on it.

The rest of part I is organised as follows: in Section 2 we will introduce a new framework that integrates the three elements mentioned above. In Section 3 we will explain the improvements made to the P2P infrastructure. Specifically the possibility to create resources and share files. In Section 4 we will describe advances on the RestGovernor introduced in last year’s deliverable. Then, in Section 5 we will introduce the new EI-specification editor. In Section 6 we explain the new features added to the user interface, and finally in Section 7 we will conclude and describe what work is still to be done.

2 PeerFlow

As explained above, the work to be done for this work package roughly consists of three parts:

• The implementation of a P2P runtime for Electronic Institutions.

• The implementation of an automatically generated user interface for Electronic Institutions.

• The implementation of a new tool for the design of simplified institution specifications.

In last year’s deliverable (D3.1) these components were described as three separate, independent, components. In the past year however we have managed to integrate these three components into a single framework that encapsulates them all. In this section we present this new framework, which we call PeerFlow. More details about the specific components are given in the following sections.

The idea behind PeerFlow is that anyone can create his or her own online institution where users can come together and interact according to predefined protocols. Specifically, within PRAISE, the designer of an institution could be a teacher with the wish to create a lesson plan for his students. Using PeerFlow he would be able to design a lesson plan, publish it, and invite his students to participate in the class. Moreover, PeerFlow would automatically generate the required infrastructure so the teacher does not need to program anything. Everything can be done through a user-friendly interface and no knowledge of programming languages is required.

Such a lesson plan could for example consist of a number of tasks that every student needs to finish. The Electronic Institution software behind PeerFlow would make sure that no student can advance to the next stage of the program before having finished all required previous tasks. The fact that PeerFlow runs over a P2P network means that the teacher does not need to have
access to any server. He can simply launch the institution from his own desktop computer and his students, as long as they have downloaded the PeerFlow software, will be able to find his class and participate in it.

    Working with PeerFlow goes as follows:

    1. A user downloads the PeerFlow software from the PeerFlow-website and installs it on his computer.
    2. When starting the software a browser window will open, where the user will be able to create an account.
    3. From this browser window the user can open the ‘institution editor’ where he can design an institution through a user-friendly interface.
    4. When finished designing the institution, he can publish the specification, simply by clicking a ‘publish’ button.
    5. When other users that have PeerFlow installed click on the ‘search’ button, they should see the name of this institution appear in a list of found institutions.
    6. Any user can now launch this institution and optionally join the institution (probably the designer of the institution will often also be the one who launches it, but this is not necessary).
    7. Once an instance of the institution has started other users can find the instance when clicking on ‘search running’, after which they can join the institution.
    8. Inside the institution, the user interface allows the users to send messages to each other, but restricts those messages in such a way that they satisfy the protocol, as defined by the designer of the institution.

Next, we will go into detail about these steps.

    All of the PeerFlow framework is built upon the API described in last year’s deliverable D3.1. The search screens offer a number of buttons that can be clicked by the user and that trigger the several actions from the API.

2.1 Searching and Launching an Institution

The first screen the user will see when he is logged in is the ‘Search’ screen. He can request an exhaustive list of all institution specifications that are published in the P2P network by clicking on ‘Search’ directly. However, you may also first enter a number of search criteria. You can search for example for all institutions that where you can enter with a specific role. For example, you may want to see all institutions where you can enter as ‘student’.

    Also, you may search for an institution with a specific name, or for institutions that have specific keywords associated with them. For example you could search for all institutions with the keywords ‘music’ and ‘education’.

    After clicking the Search button in the Search screen, the user is taken to a screen with the search results. This window will display a list of published institution specifications. For each
such specification it shows the name and the possible roles you can adopt in that institution. When selecting a specification from the list, more details appear such as its description and its key words. In order to launch a new instance of an institution the user should select a specification and click ‘Launch’ or ‘Launch and Join’ in case he himself also wants to participate in it.

2.2 Searching and Joining an Institution

If you do not want to launch a new institution, but instead want to participate in one that is already running, you can click on the ‘Search Running’ button in the menu. Searching for running institutions works in exactly the same way as searching for institution specification and you can use the same kinds of search criteria. Once you have selected the institution you want to join, you click the ‘Join’ button. You are then asked to choose a role to adopt when entering in the institution. Next, after choosing a role, a new window opens and that displays the interface to participate in the institution. This interface is generated with the automated GUI generator explained in the previous deliverable.
Figure 7: Choose between ‘Launch’ or ‘Launch and Join’.

Figure 8: Choose a role before entering the institution.

Figure 9: Once you have entered an institution you can interact through this screen, generated by GENUINE.
2.3 Publishing new Institution Specifications

Designing a new institution specification can be done in two ways: either by using the old Islander tool, which is a standalone application outside of PeerFlow, or using the simplified editor which is completely integrated with PeerFlow.

In the first case one would have to download Islander separately from PeerFlow. When the specification is finished, it is stored as an xml file. Next, you would have to start PeerFlow, and click on the ‘Publish’ button in the menu. This will open a window that allows you to select the xml-file from your disk and publish it. The publication mechanism as described in D3.1 will then make sure it is placed in your local PeerFlow-repository so that it can be found by other users on the P2P network. In the second case, when the specification is finished you can simply click ‘Publish’ directly inside the editor, and it will directly be published.

2.4 Future Work

We are going to add a visualiser that displays a dynamic overview of the interactions that have taken place between the participant in an institution. It will consist of graph in which every node represents a participant, and its color will represent the role played by that participant. Links between the nodes will represent the past interactions between the participants. These links will get thicker as participants exchange messages, hence clearly showing the strength of relationships between the participants. Moreover, the nodes of the graph will get bigger as participants are more active. This visualiser will increase the social awareness of the participants and will generate a true sense of community amongst the community members.

Furthermore, we will add a chat functionality that will enable users to communicate outside the institution. Any interactions through this chat function will not have any institutional meaning, but does allow the participants to communicate and hence coordinate their actions within the institution.

We will add profile pages that allow users to create a personalised profile with profile picture and other personal information, such as relationships with other users. When participating in an institution one will be able to see the profile information of the other users present in the current scene instance.
Finally, we will also add the option for users to remove previously published specifications.

3 Peer to Peer Electronic Institutions

In this section we describe the new features and improvements that we have made with respect to the Peer-to-Peer Electronic Institution infrastructure in the past year. We here repeat only the most important details of last year’s P2P implementation. For more details we refer to last year’s deliverable D3.1 [28], section 2 “Peer to Peer Electronic Institutions”.

3.1 Concepts

As explained in previous year’s deliverable, in the first year of the project we have started to develop a new version of AMELI that can run over a peer-to-peer (P2P) network, rather than on a centralised server.

A peer-to-peer network provides a number of advantages over a centralised approach, such as:

- improved scalability
- more easily maintainable
- improved privacy

A P2P system provides better scalability, because with each new user that connects to the network, also a new machine comes available to the network, whereas the computing power of a server always remains constant. This means that in a P2P system the total supply of computing power increases exactly as fast as the demand for it.

Moreover, since the network runs entirely on the machines provided by its users, no external server is necessary, and hence there is no need for anyone maintaining a server. Once a community of users is established, the network will maintain itself.

Finally, user information and resources such as files are stored locally in a P2P system, so the individual users stay in control over these resources, while a centralised approach would require the users to upload all their resources to an externally controlled server.

A premature P2P version of AMELI already existed before PRAISE, but this only allowed the creation of a P2P EI based on a configuration file that stored the information about the machines in a network. This file needed to be created before the instantiation of the institution and hence it was impossible to add new nodes to the network at runtime. Since in PRAISE we want to be able to have dynamic user communities this was clearly not sufficient for our needs. The new P2P implementation of AMELI, created for PRAISE, does allow dynamic networks.

Figure [11] displays a schematic overview of the P2P EI infrastructure. It is distributed over several hosts, which are connected by a peer-to-peer network. Each host with a P2P connection is called a node of the network, and is managed by an agent called the DeviceManager. The main tasks of this agents are:

- To provide and manage a repository of Electronic Institution specifications.
To maintain the peer-to-peer network.

To execute Electronic Institutions.

Whenever a user wants to publish, search or delete a specification this request is handled by the DeviceManager of user’s node. The DeviceManager manipulates the local file repository to execute the request.

In the deliverable of the previous year we explained that EI-specifications were stored in a directory in the file system, and the information of each specifications was indexed using the Apache Lucene\(^1\) library. Whenever a user published a new specification this meant that the DeviceManager would store the file in this directory and would extract the name, description, keywords and roles from the file and store this information in a data base using Apache Lucene. This could then be used by other users to find the specification using the search tool. This allowed user to make highly descriptive queries containing wild cards ( *, ?), or make fuzzy queries and range queries. Unfortunately, every time that a specification was added or removed it was necessary to re-index all the specifications.

In the past year however, we have migrated the system to another content management system called Apache Jackrabbit\(^2\) that allows us to store other types of content as well, rather than only files. It provides the same functionality previously described but for any kind of content (see Section 3.2).

\(^1\)http://lucene.apache.org/
\(^2\)http://jackrabbit.apache.org/
The `DeviceManager` agent uses the `Freepastry` library for creating the peer-to-peer network. This library provides mechanisms to define the network and to send and receive messages between the nodes of the network. Unfortunately, it does not have any discovery mechanism, so in order to find the other nodes in the network we have implemented a simple broadcast mechanism. More specifically, every node listens for `UDP` broadcasts, and periodically sends `UDP` broadcasts. So, in this way we can find any new node that is started in the local network and add to the peer-to-peer network.

Whenever a user requests to start a new Electronic Institution its `DeviceManager` receives the request, with an identifier that identifies the EI-specification that the user wants to instantiate. The `DeviceManager` will then contact the other `DeviceManagers` in the P2P network in order to make sure the necessary agents are started, in a distributed fashion (see Figure 11).

Thus, the `SceneManager`s, `TransitionManager`s, the `EIManager` and the `Governor`s are started which are necessary for any fully running Electronic Institution. Since they are not running on one single machine, but distributed over the machines in the P2P network, the workload is distributed.

On the other hand the new `DeviceManager` can also launch internal agents once the infrastructure has begun (see Section 3.5). This kind of agent is equivalent to the staff of an institution.

In the previous year users could only search for running Electronic Institutions if they provided the identifier of the corresponding EI-specification, but this year we have added the possibility to search using the same queries used to search for specifications (see Section 3.4).

### 3.2 Resource Repository

A new feature that we have added to the P2P framework with respect to last year, is the ability for users to share files. Previously, it was possible for users to publish institution specifications and share them, but we had no mechanism for users to share other types of files, and adapt them.

Above, we have explained that we have migrated our database management system from `Apache Lucene` to `Apache Jackrabbit`, in order to allow the users to share files amongst themselves and dynamically change them.

`Apache Jackrabbit` allows us to store, access and manage content. Also it provides us with services for access control, versioning and search. Any entity stored in this database is called a `Resource`, and is stored in a hierarchical tree structure. A resource consists of a number of properties, which in turn consist of a property-name and a property-value. The property-name is a string, while the property-value can be either a string, a file or a string array. The possible actions one can perform on the database are currently restricted to: creating resources, creating properties, setting property values, getting property values, and searching for resources with certain properties.

The creation of a resource generates a new node in the tree structure of the database, with a universal unique identifier (UUID). This UUID is necessary in order to obtain or modify the properties of the resource. One can search for resources with specific string-valued properties by sending a query to the database. Each required property is specified in the query as a string with the name and required value of the property separated by a colon. The several properties are connected by boolean operators `AND` or `OR`. Moreover, one can use `wildcards` in the property name and/or value. Specifically, the symbol ‘?‘ matches any single character, and the symbol ‘*‘ matches any single character, and the symbol ‘*‘ matches any single character, and the symbol ‘*‘ matches any single character, and the symbol ‘*‘

3[^1]

[^1]: [http://www.freepastry.org/FreePastry/](http://www.freepastry.org/FreePastry/)
matches any sequence of characters.

### 3.3 Searching for an Electronic Institution Specification

*Apache Jackrabbit* provides a mechanism for searching files, similar to *Apache Lucene*, but is less expressive. Nevertheless, it is still powerful enough to satisfy our requirements.

When a user publishes an Electronic Institution specification, its *Device Manager* creates a new resource, stores the specification as a file-property of the resource and adds the following string-properties to the resource:

- **name** The name of the Electronic Institution, given by the user.
- **description** A description of the Electronic Institution in natural language, given by the user.
- **keywords** A list of keywords defined for the Electronic Institution, given by the user (the keywords are given as a comma-separated list of words, starting with the text `{@keywords` and ending with `}`. For example: `{@keywords auction,market,electricity}`).
- **input_roles** A list of roles that a user can adopt when entering the institution.

When a user searches for Electronic Institution specifications within PeerFlow, the search query is handled by its local *Device Manager* which will search through the local repository for any resources that match the query and have an institution specification associated to them. Furthermore, it asks the other *Device Managers* in the peer-to-peer network to search in their internal repositories for specifications that satisfy the query. They reply with the names and other information of the specifications they find and these are then displayed in the GUI of the user.

For example, to search for Electronic Institutions that have a name that starts with 'Auction' and that contain the role 'student', the query is `name:Auction* AND initial_role:guest`.

If we compare this with the search mechanism described in the deliverable D3.1[28], we see that we can not use fuzzy queries or range searches. However, the advantage is that with the new system we can define attributes dynamically and re-indexing is automatically done.

### 3.4 Searching for running Electronic Institutions

In the version of the P2P EI presented last year the users could only search for running institutions by providing the identifier of the corresponding specification in the search query.

To make things easier for the user we have now added a new functionality that enables the user to search for running institutions using the same query structure as used when searching for institution specifications (see Section 3.3). This means you can search for running institutions for which the corresponding specification has a specific name, description, keywords and/or input roles.

As in the previous case the *Device Manager* lists all running Electronic Institutions with certain properties. The *Device Manager* searches in its local repository for specifications that match the query and then asks any of the *EIManagers* that are running on the local device if they are running...
any of the specifications that match. If any manager is indeed running such an institution, it will reply, and the Device Manager will then make sure these institutions are listed in the browser window of the user. And finally, it ask to the other Device Managers in the peer-to-peer network if there is any other Electronic Institution running that matches the query.

Running Electronic Institutions that match the search query but that do not allow any more agents to enter are however filtered from the search results.

3.5 Resource Agent

The formal definition of an Electronic Institution distinguishes between two kinds of roles: internal roles and external roles. The internal roles are those that should be adopted by trusted agents. That is: agents that have been programmed specifically to make the institution work correctly. External roles on the other hand can be adopted by any agent that comes from an unknown source. Also, generally speaking one can say that internal roles are often adopted by software agents, while external roles are adopted by humans, although this does not always have to be the case.

This means that whenever a new instance of an Electronic Institution is started one must start the necessary internal agents (agents that adopt internal roles), in order to make the institution work correctly. For this reason we have modified the Device Manager to be able to start those internal agents.

Furthermore, we have introduced the possibility for the users to share Resources (see Section 3.2) (currently only string values or files). The designer of an Electronic Institution specification, may want to determine how the users can share some specific resources, which would often mean he would have to implement an internal agent to manage this. In order to simplify this, we have created the ResourceAgent.

This agent is programmed in such a way that it can, with some limitations, participate in any Electronic Institution, so one does not need to program a new resource agent for every new institution specification. It is programmed to interact with the resource repository. Because the messages that agents in the institution can not contain files the ResourceAgent is currently only capable of managing string resources. In the next sections we describe the behaviour of this agent depending on its state in the running Electronic Institution instance.

3.5.1 Behaviour in a Performative Structure

The ResourceAgent navigates through the performative structure according to the following rules:

- When a ResourceAgent has entered in the electronic institution, it has entered in sub performative structure or it has exited from a scene, it tries to move to the first accessible transition.

- When a ResourceAgent has entered in a transition, it tries to move to all the accessible scenes, if that transition is of the type AND. If one of the accessible scenes is of type ONE or SOME, it waits until at least one scene instance exists and then enters.

- If it can not move a warning message is shown in the logging system.
3.5.2 Behaviour in a Scene

When a ResourceAgent is inside a Scene, it acts according to the following rules:

- When a ResourceAgent has entered a scene, it directly asks to exit. This means that as soon as the protocol reaches a state where the agent is allowed to leave, it will leave the scene and move to another scene (see Section 3.5.1).

- If a ResourceAgent receives a message of which the content is a function without any arguments and it can reply to the sender with a function with an argument that is of the type string, then the ResourceAgent tries to create a new resource in the repository. If it succeeds it replies with the Universal Unique Identifier of the created resource as the argument of the content function of the reply. Otherwise, if it cannot create the resource, it replies with the same function but with an empty string.

- If a ResourceAgent receives a message of which the content is a function with a string and a data type as arguments, the string is a valid Universal Unique Identifier and all the fields of the data type are Strings or list of Strings, then it updates the resource identified with the first arguments, with the properties defined by the second one. Thus the data type elements are considered as the properties of the resource.

- If it receives a message of which the content is a function with one argument of the type string, depending on the string value, it will take one of the following actions:
  - If the string is a valid Universal Unique Identifier and it can reply to the sender with a function with a data type where all the fields are Strings or list of String, then it gets the current property values of the resource defined by the identifier and replies with the property values serialized into the data type.
  - If the string is not a valid resource identifier and it can reply to the sender with a function with a list of data type where all the fields are strings or list of String, then it searches for all the resources that satisfy the first string. So the first argument is considered as a search query. Remember this kind of query is a list of key:value pairs separated by spaces where the key has to be the property name and the value can be a pattern that use * and ?. When it has obtained all the resources that satisfy the query, for each one it asks about their properties and serializes them on the possible data type and store in a list, that finally is sent to the sender of the first message.

- If it cannot reply with any message, it only prints a log message.

3.5.3 Starting and configuring

The designer of an EI-specification can make sure a ResourceAgent is started when necessary, by adding certain XML tags to the specification. Specifically, the designer needs to add them in the description of the internal role that he wants to be adopted by the ResourceAgent.

The tags that need to be added are displayed in Code Listing 1. When the DeviceManager finds these tags it creates a new ResourceAgent, configured using the tag parameters, and launches the agent to play in the started Electronic Institution with the role that was tagged.
Listing 1: Example for configuring the resource agent with name \textit{dbAgent} and a polling time of 15 seconds

\begin{verbatim}
<ResourceAgent>
  <Name>dbAgent</Name>
  <PollingTime>15000</PollingTime>
</ResourceAgent>
\end{verbatim}

The configuration parameters for a \textit{ResourceAgent} are:

- \textit{Name}: the name that the \textit{ResourceAgent} will use when it participates in the running electronic institution. If it is not defined it uses \textit{resourceAgent} as its name.

- \textit{PollingTime}: time in milliseconds that the agent will wait between requesting the information it needs to participate in the Electronic Institution. For example, if the agent has to enter a scene, but it is not created yet, it waits the polling time before asking again if the required scene is created. If it is not defined it will wait 5 seconds.

\section*{3.6 Future work}

An important problem with peer-to-peer networks is that nodes may fail either because they unexpectedly stop working, or because they lose connection with the network. Since such a node may hold critical components of an electronic institution that the entire institution, which is spread out across the network, may fail to work correctly. It is therefore crucial that we have a mechanism that allows an institution to recover after such a node failure.

Currently, if the execution of an institution fails, the entire institution needs to start all over again from the beginning. There is no way to suspend its execution temporarily and continue running later. We will therefore focus on implementing the functionality of restarting an institution after failure.

The resource database already provides a starting point for the implementation of this functionality as it allows us to maintain the state of each node between executions of the peer-to-peer infrastructure. However we do not yet have any mechanism that allows to continue running an Electronic Institution from such a stored state. So, we will continue working on this matter.

We need a mechanism to discover P2P nodes spread around the Internet. We started using \textit{ApacheZookeeper}, but we do not have significant improvements yet, so will continue working on it.

Furthermore, \textit{Apache Jackrabbit} allows to define access control over the resources, so we will work on a simplified credentials system that allows users to define who can access, modify or remove resources.

Finally, in a similar way as we have created the \textit{ResourceAgent} to manage the resource repository, we will create a similar internal agent to manage generic web services. Alternatively, a more ambitious option could be to allow the user to define any Internal agent behaviour with a language...
similar to *If This Then That*.

## 4 REST Governor

In this section we first give a short summary of the work done previously regarding the Rest Governor, and then explain the new features and improvements that we have made in the past year. For more details on the implementation of the Rest Governor we refer to deliverable D3.1 [28], Section 3.

### 4.1 Concepts

In the first year of the project we have created a new piece of software called RESTGovernor that can manage interactions from humans in an institution using different devices as such as tablets, computers or mobile phones. The RESTGovernor is a combination of a Jetty web server and an agent that can participate in a Peer-To-Peer Electronic Institution.

As can be seen in Figure 12 whenever a user does some kind of action on his or her device (such as a tablet, phone or laptop) the device sends a HTTP request to the RESTGovernor which then sends the request to the proper agent in the institution. For example, if the user requests to search for a published specification, a HTTP request containing the search query is sent to the RESTGovernor. The RESTGovernor forwards this request to the DeviceManager (because the DeviceManager is responsible for managing the data base of published specifications), and when the RESTGovernor receive the response it sends the result back to the user in the form of a JSON object.

The actions that an user can do through the RESTGovernor are:

- **Registering a Governor** this is used to log in and create a RESTGovernor for a user.
- **Logging Out** finish the RESTGovernor of a user.
- **Searching for an Electronic Institution** with this action a user can obtain the published Electronic Institutions that satisfies a given search query.
- **Registering an Electronic Institution Specification** this is used by the user to publish an Electronic Institution specification so that it can be found by other users.
- **Getting Information of a Registered Electronic Institution Specification** a user can use this action to obtain the name, description, keywords and input roles of a published Electronic Institution.
- **Searching for all Running Electronic Institutions** with this action a user can obtain the identifiers of the running Electronic Institutions.
- **Starting a New Electronic Institution** a user can use this action to start a new Electronic institution that instantiates a given EI-specification.

---

[28]: https://ifttt.com/
[29]: http://www.eclipse.org/jetty/
Enter to Participate in an Electronic Institution with this action a user can join a running Electronic Institution and interact with other agents following the rules of a published EI-specification.

Obtaining Information about All Electronic Institutions the Agent is Participating in this action returns the identifiers of all the Electronic Institutions in which the user is participating.

Obtaining Information of an Electronic Institution Conversation this action can be used to obtain information of the state of a conversation in which the agent is participating.

Getting information about the Possible Movements with this action a user can obtain the possible ways it can move from one scene to another.

Moving to Another Conversation with this action an agent specifies to which scene instances it wants to move.

Getting Information about the Possible Messages to Say this action returns the patterns of the messages that the user can send in the current state of a scene.

Saying a Message in a Scene is used by an agent to send a message inside a scene.

Exiting from a Scene is used to request to exit form a scene in order to move to another scene.

Do a ‘Stay And Go’ in a Scene this is used by an agent to go to another scene, while staying in the current scene at the same time.
4.2 Rest Actions

In this section we describe the new requests that an user can do over a RESTGovernor. This new requests allow to the users to interact with the new resource repository that we have been described on Section 3.2. An agent in the P2P network can manipulate these resources by sending a request to its governor. These actions are controlled by web services following a REST methodology. For this reason the description of each available action will start with a box with the HTTP request method (GET, POST, ...), followed by the base URI that identifies the action. After that follows a short description of the action and the required or optional arguments, and finally the expected JSON object returned as a result of the action.

4.2.1 Create resource

POST /resource

This web service is used to create a new resource in the database. The new resource is empty and is identified by a universally unique identifier (UUID) that is returned as plain text. The agent needs this identifier so that it can later perform actions on the created resource.

4.2.2 Get resource properties

GET /resource/{id}/properties

This web service is used to obtain the string properties associated to a resource in the database. The \{id\} tag in this request must be replaced with the identifier obtained when the resource was created. It will return a JSON object where the fields are the property names and theirs values the associated string properties.

4.2.3 Set resource properties

PUT /resource/{id}/properties

This web service is used to change the string properties associated to a resource in the data base. The \{id\} tag in this request must be replaced with the identifier obtained when the resource was created. A JSON object should be included in the content of this request, in which each key-value pair represents a property name and its new value.

4.2.4 Get file resource property

GET /resource/{id}/file/{name}

This web service is used to obtain a file associated with a given resource.
The \{id\} tag in this request must be replaced with the identifier obtained when the resource was created, and the \{name\} tag must be replaced with the property name with which the file is associated. It will return the file associated with the resource, as a set of bytes.

### 4.2.5 Set resource property

\[\text{PUT} \ /\text{resource}/\{id\}/\text{file}/\{name\}\]

This web service is used to associate a file to a resource property. The \{id\} tag in this request must be replaced with the identifier obtained when the resource was created, and the \{name\} tag must be replaced with the property name that you want to set for the file. The content of the message should contain the file as a set of bytes.

### 4.2.6 Get resource property values

\[\text{GET} \ /\text{resource}/\{id\}/\{name\}/\text{values}\]

This web service is used to obtain all the values associated to a resource property. The \{id\} tag in this request must be replaced with the identifier obtained when the resource was created, and the \{name\} tag must be replaced with the property name with which the values are associated. It will return a JSON string array with all the values associated with the resource property.

### 4.2.7 Set resource property values

\[\text{PUT} \ /\text{resource}/\{id\}/\{name\}/\text{values}\]

This web service is used to associate to a resource property a set of values. The \{id\} tag in this request must be replaced with the identifier obtained when the resource was created, and the \{name\} tag must be replaced with the property name that you want to set the values. The content of the message should contain a JSON string array with the values to associate to the property.

### 4.2.8 Search for resources

\[\text{POST} \ /\text{search}\]

This web service is used to search the database for resources that have certain string properties. The query is send as a form parameter, and it represents a set of name-value pairs representing the name and value of a required property. The property name and value are separated by : and in each value you can use the wildcards ‘?’ , to match any single character, or ‘*’, to match multiple characters. Also one can use the boolean operators \text{AND} or \text{OR} between each pair.

For example, in order to obtain all the resources for which the property ‘author’ has a value containing the string ‘alice’ or for which the property ‘author’ ends with ‘bob’, and for which the
property ‘user’ starts with the string ‘user’, followed by exactly one more character, the query would be:

\[
\text{author:”*alice*” OR author:”*bob” AND user:”user?”}.
\]

The returned result will be a JSON array of strings representing the identifiers of the resources that match the query.

### 4.3 Future work

Future work for this tool will be to focus on providing access using other web technologies as websockets or Server-sent Events, to reduce the necessity of using polling technologies and reduce the workload of the server. Furthermore, we may implement the possibility of using other languages for the exchange of information, such as XML or Prolog.

### 5 Simplified Institution Editor

Until now institution specifications had to be created using the Islander tool. The problem with this is that it can be very difficult to use this tool as the designer of the institution needs to think about many technical details such as scene transitions, scene protocols and constraints. This is because Islander was mainly designed to be used by computer scientists and not by average users. The Islander tool is therefore not very useful in the context of the PRAISE project, since we want music teachers with average computer skills to be able to design lesson plans.

We have therefore started developing a new tool with a simplified user interface that allows average people to design institution specifications. A consequence of the simplification is however that it will be less flexible than Islander. The set of institution specifications that can be created with the simplified editor is only a subset of the full set of valid institution specifications. Nevertheless, we think that the simplified version of Islander we propose is a first step in motivating average users to specify interaction models as Electronic Institutions.

#### 5.1 Simplified Institutions

We will now define the concept of a *Simplified Institution*. A simplified institution is a specific kind of Electronic Institution that satisfies certain properties that make it more simplistic. It is important to note that each ‘simplified institution’ is still a valid Electronic Institution according to the full definition of an Electronic Institution. For details about the terminology used we refer to last year’s deliverable D3.1.

A simplified institution is an Electronic Institution with the following characteristics:

- Users cannot change roles.
- Each scene has only one instance.
- Messages are always sent to all participants in the same scene instance (so private messages are not allowed).
• Users cannot start new scene instances. For every scene its only instance is created automatically when the institution is launched.

• Users can only be in one scene at a time.

• It is not possible for two or more users to enter a scene simultaneously: users enter one by one.

• There are no constraints that restrict the possible actions of the user, other than those defined by the basic finite state machine of the protocol.

An important consequence of these simplifications is that the concepts of Initial scenes, Final scenes and Transitions can be completely hidden to the user, as well as to the designer of the institution. Although any participant still passes a transition as he or she moves from one scene to another, there are no decisions to make at the transition, so the user does not need to notice it.

5.2 The Simplified Editor

The simplified editor does essentially the same as Islander, but it has a number of important differences:

• The simplified editor only allows to specify simplified institutions.

• The simplified editor is used in the browser, rather than as a standalone application.

• It is more user friendly: more focus on drag-and-drop rather than writing equations.

The fact that it allows only simplified institutions also makes it possible to simplify the user interface, since there are a lot less decisions to be made by the designer. In the backend side of the simplified Editor, each simplified institution is automatically transformed into an Electronic Institution. To do that, we make the following decisions for the designer:

• For each scene in the simplified Institution there is a scene with the same name and description in the automatically generated Electronic Institution.

• For each role in the simplified Institution there is a role with the same name in the dialogical framework of the automatically generated Electronic Institution.

• Each role within a scene in the simplified Institution can enter, play and exit the corresponding scene in the automatically generated Electronic Institution.

• For each role in a scene that does not come from another scene in the simplified Institution, we add the Arcs and Transition to allow the role to move from the Initial Scene to that scene in the Electronic Institution.

• For each role in a scene that does not go to another scene in the simplified Institution, we add the Arcs and Transition to allow the role to move to from that scene to the End Scene in the Electronic Institution.
We add a Transition and its corresponding Arcs in the Electronic Institution between scenes in which a role moves from one to another in the simplified Institution.

Figure 13 displays a screenshot of the simplified editor. The editor window is split in five areas: the menu bar, the toolbars that appears at the top and left side of the window, the main screen or edition area and the property panel. The menu bar contains a set of actions for creating, saving, importing and publishing simplified institutions, whereas the toolbar at the top of the main screen includes the shortcuts to some of the actions appearing in the menu bar. The four necessary elements for creating a simplified institution are shown in the left-side toolbar. That is, Scene, Role, Resource and File as shown in Figure 14. The main screen is the area where designer drops and links the elements in order to create a simplified institution. On the other hand, the right-side panel shows the properties of the element that gets the focus.

A scene can be created by dragging and dropping an icon representing a Scene (first icon, from left to right, in Figure 14) from the toolbar to the central screen. Roles (second icon, from left to right, in Figure 14) can be added to the scenes by dragging and dropping a role-icon to the scene. The designer draws arrows between the roles in different scene to determine how different roles can move from one scene to another. Resource and File are the remaining icons that appear in the left-side toolbar and that can take place in building simplified Institutions. In fact, Resource and File are not present in the current version of Islander. File element is a special kind of Resource. In the next section we introduce Resource as the general concept and explain how it can be modelled with the tools provided by Islander and Electronic Institutions.
5.3 Resources

A new feature that we have added to the simplified editor, which is not present in Islander, is the ability to define ‘Resources’. A resource can represent a file, a web page, or a web service that can be opened by the user in a separate screen. In the context of education this could for example represent an online dictionary, or a pdf-file with an assignment for the students.

One can drag-and-drop a resource icon into a scene, in order to give the users in that scene access to the resource, or one can drag-and-drop it into the central area outside of any scene, in which case it is accessible from any other scene. Furthermore, the designer needs to draw arrows from one or more role-icons to the resource in order to give those roles access to that resource. Finally, the designer needs to specify a URL to the resource. In the GUI the resource will appear as a link in the menu, so that the user can click it to open the resource in separate screen.

File is a special kind of Resource. However, we decided to split these concepts into two icons in the simplified Editor because they offer different types of functionalities for the designer. Unlike Resource, which can only be consulted, a File is a Resource that can be generated and consumed by users during the execution of the institution. As with Resources, one can drag-and-drog a file icon into or outside scenes. The former makes possible to share files in the scope of a scene, whereas the latter allows to share files created in different scenes. The designer also needs to draw the arrows between roles and files. In such way, the designer grants permission to a role for consuming or generating files.

The question that arises here is, of course, how to deal with these new concepts that are not present in Electronic Institution. Our approach is essentially based on the usage of internal resource agents within the Electronic Institution. Those non-human agents are in charge of providing services to facilitate the access and creation of resources to the users of the Electronic Institution. The proposed approach is completely hidden for designers and users of the Electronic Institution so they do not need to explicitly interact with those internal agents.

5.4 An Example

The example shown in Figure 13 shapes the examination procedure carried out by a teacher and a set of students. The scene ‘Examination’ is intended to measure students’ knowledge in a concrete topic. In the course of the exam, the students are permitted to use an online English dictionary (i.e. http://dictionary.reference.com/). The teacher then assesses students’ exams in the scene ‘Evaluation’ in which students are also notified about their marks.

Figure 15 displays the properties of scene ‘Examination’ and roles ‘Teacher’ and ‘Student’. The scene includes the name and description of the scene as well as some graphical properties.
such as the fill color of the scene square. The properties of the a role basically set the number of agents that can play that role simultaneously in a given a scene. The specification described in the aforementioned figure states that scene ‘Examination’ requires exactly one teacher and at least one student.

Once the designer has finished his specification in the simplified Editor it can be saved as an xml file, in exactly the same format as a file created with Islander. This file contains the Electronic Institution specification the simplified editor has automatically generated from the simplified Institution built by the designer. Therefore, any institution specification created in the simplified editor can also be opened with Islander.

As an example, Figure 16 illustrates the Electronic Institution that results from transforming the simplified institution shown in Figure 13. As can be noticed, both include scenes ‘Examination’ and ‘Evaluation’, and roles ‘Teacher’ and ‘Student’. In the Electronic Institution the scene ‘begin’ and ‘end’ correspond to the initial and end scene, respectively. They were automatically generated as well the agent ‘File’, which is the internal agent in charge of providing services to facilitate exam exchange between the teacher and students. As ‘Dictionary’ is an intra scene resource, we have accordingly enriched the description of the protocol of scene ‘Examination’ in the Electronic Institution specification by adding an extra annotation. That information only contains the URL of the resource we want GUI lets users to view during the exam.

5.5 Future Work

The simplified editor currently does not allow protocols to be defined for the scenes. Instead, when the institution specification is generated, every scene currently gets a pre-defined default protocol assigned to it, which basically allows every user to do anything. The ability to design custom protocols is still to be added to the editor. We will allow the institution-designer to define
a protocol by stating a set of rules in the form of “if user x does this, then user y can do that”.

6 User Interface

In last year’s deliverable we have introduced the user interface for electronic institutions, named GENUINE (GENerated User INterface for Electronic institutions), although we did not have a name for it back then. In the past year we have made several improvements to this interface. We will here simply give a list of the main improvements. Apart from this, much time has been spent on solving small bugs and improving the efficiency of the framework.

Let us first restate the basic principles behind this user interface. AMELI only allows software agents to take part in Electronic Institutions, we have implemented a dummy agent that acts as a layer between a human user and the institution. It establishes a connection with the browser of the user, and translates any user action in the browser into an action in the Electronic Institution. Likewise, every message received by this agent is sent to the user’s browser and displayed in the user interface. This agent is called the GUIAgent.

6.1 New Features

The main new features that we have added to GENUINE in the past year are:

- When the user is at a transition between two scenes, he can now choose which scene instance to enter the new scene.
- When the user is at a transition between two scenes, he can now choose with which role to enter the new scene.
- When a user starts a new scene instance he can choose which protocol he wants to apply that new scene instance.
- When a user starts a new scene instance he now also has the possibility of setting values for the parameters of that new scene instance.
- It is now possible to send messages with list-variables in the content.

Since all these options listed here were not possible in the previous version of GENUINE, it would not work correctly on institutions where those options were relevant.

We should note however, that most of these options are not necessary in the simplified institutions that are created with the simplified editor described in Section 5. Therefore, these changes are not directly relevant for PeerFlow, but rather for more general applications based on Electronic Institutions.

Another concept that we have added is the notion of a resource (see Section 5.3). A resource corresponds to a file or web service that can be accessed by the user. The designer of an institution can add resources to the institution using the simplified editor. This resource will then appear as a link in a ‘resources’ menu of the user interface.
6.2 Future Work

There are still a number of properties of Electronic Institutions that currently cannot be represented by the GUI.

- If a user is in more than one scene at the same time, this cannot be represented in the GUI.
- The GUI does not allow a user to set the properties of the protocol when creating a new scene instance.
- The GUI does not work correctly if the institution contains more than one ontology.
- The GUI does not work correctly if the institution contains more than one Dialogical Framework.

7 Conclusions and Future Work

In this part of the deliverable we have explained the work done during the second year of the PRAISE project for the tool created to build social learning communities. The main achievements of the past year consist of the creation of a new, simplified editor to create institution specifications, and the integration of this editor, the p2p architecture and the automatically generated GUI, into one platform called PeerFlow.

Furthermore we have implemented a database management system that enables users to share files over the P2P-network and we have made several improvements to the automatically generated user interface.

For the future, we need to add a mechanism that ensures the institution keeps on running if or more nodes in the peer to peer network fail, the capability of storing information between executions of an electronic institution. Also we need to improve the user interfaces that allow users to search for electronic institutions and to participate in electronic institutions, because at this point they are not as user friendly as they could be. Furthermore, the user interface for EI participation should allow users to participate in more than one scene at the same time.

An important thing to do next is to integrate our system with Music Circle. Ideally a teacher would be able to do all the things he can do in PeerFlow, inside Music Circle. That is: create a lesson plan, publish it on Music Circle, and invite his students in Music Circle to participate. The students would then join the class by logging in to Music Circle and clicking a ‘join class’ button. At this point however, Music Circle and PeerFlow are still two entirely separate platforms.
Part II

Community Tools for Self-Evolving Communities

Self-organisation and self-evolution is evident in physics, chemistry, biology, and human societies. Despite the existing literature on the topic, we believe self-organisation and self-evolution is still missing in the IT tools we are building and using. Instead of creating numerous rigid systems, we should aim at providing tools for creating self-evolving systems that adapt to the ever evolving community’s needs. This part of the deliverable proposes a roadmap for self-evolution by presenting a set of building blocks, which we refer to as community charters. This part of the deliverable also presents an approach for each of these blocks, helping build the first prototype for self-evolving communities. Unlike the previous part of this deliverable, which presented our implementation details for the community tools, this part presents the results of more theoretical research that studies self-evolving communities.

8 Introduction

Whilst many approaches have been proposed for defining and implementing communities (such as using the notion of organisations [14] or institutions [7]), we believe the current literature lacks the in-depth study of computational accounts of communities’ evolution. The very foundation of autonomous agent research is based on the idea that agents evolve, for instance by assuming their beliefs or goals change over time. But how communities as a whole evolve is an overlooked question that this deliverable aims at raising and addressing.

We argue that just like human communities, e-communities (defined by their software) also need to self-evolve. Instead of creating numerous rigid systems, what we should aim at instead is providing tools for creating self-evolving systems that adapt to the community’s needs. We believe different communities should be governed by different rules. These rules should be an ever evolving set resulting from the aspirations of its members. Furthermore, for the community’s rules to be effective, they need to be tailored to the specific character traits of the community members as well as considering some other external influences. We note that in this part of the deliverable we talk about self-evolution, as opposed to evolution, since we are interested evolution that is designed and directed by the community itself.

This part of the deliverable proposes a roadmap for self-evolving communities. It proposes a set of building blocks needed for self-evolution, and we refer to these building blocks as the community charter. It also presents an approach for each of these blocks, helping us build the first prototype for self-evolving communities.

We say communities are formed around certain goals that community members are interested in fulfilling. Their evolution is driven by the fulfilment (or unfulfilment) and the evolution of these goals. Their fall is usually triggered either by the fulfilment or the abandoning of such goals by community members.

In addition to goals, we say interactions are another fundamental constituent of communi-
ties. A community, by definition, is a group of interacting peers (where peers may be a combination of humans, agents, and services). Interactions are the backbone that glues a community together. Without interactions, a community is simply a number of individuals. However, we say a community’s interactions should aim at fulfilling its goals. Otherwise, either its interactions are ineffective, or its goals have not been properly thought-out.

Between goals and interaction protocols lie the community’s norms. Norms may be thought of as describing the declarative rules, i.e. statements that describe what a rule is without going into the details of how to implement it. Norms provide generic guidelines that help fulfil the community’s goals, and the interaction protocols (or the procedural rules) should be designed to abide by these norms.

As such, we propose to define a community by its charter, which we say is composed of the community’s goals, norms, and interaction protocols. The concept of such a charter maps with the notion of traditional human communities, which are usually defined by their mission statement (the goals), their bylaws (the norms), and their standard operating procedures (the interaction protocols) [13].

Finally, we say unfulfilled goals can be one automated way for triggering evolution, suggesting a flaw with the charter’s components. Different elements of the charter may be modified during the evolution stage in the hope that a more coherent charter is achieved.

The remainder of this part of the deliverable elaborates further on each of the charter’s three components in Sections 9, 10, and 11, respectively. Section 12 then provides a brief insight on how the proposed model helps promote self-evolution, before concluding with Section 13.

9 Goals

A community usually has a general goal which may then be divided into more concrete sub-goals. In human communities, these are defined by the community’s mission statement.

For instance, consider an online social learning music community composed of a tutor and a number of students. The community’s mission statement may be defined by the following goal:

\[ G1. \] To improve the students’ performances in playing the piano.

The goal is defined by the community members that establish this community. Although goals may evolve over time like any other charter component, their evolution is usually much less frequent in comparison with the evolution of norms, and interaction protocols. This is because communities are essentially viewed as defined by their goals.

9.1 Goal Specification

In multiagent system, goals have been studied extensively as being part of an agent’s belief-desire-intention (BDI) model [19]. In such models, goals are viewed as the desires that the agent adopts for active pursuit. When an agent commits to a specific plan for achieving a given goal, then the goal becomes an intention.
In this part of the deliverable, we do not discuss agent goals, but community goals. Different approaches and/or languages may be adopted for the specification of these goals. We say one way to specify a goal is as a tuple:

\[ (G_{id}, G_{specification}, G_{description}) \]

where \( G_{id} \) is the goal’s unique identifier, \( G_{specification} \) is the goal’s specification in first-order logic, and \( G_{description} \) is the description of the goal in text, which may be used to aid human users understand which goals have been fulfilled or not during the evolution stage of a community (we assume community members may be a mix between agents and human users).

9.1.0.1 Goal Specification Example. The online music community’s goal \( G \) may be specified as follows:

\[ (G, \exists S' \subseteq S \cdot (\forall s \in S' \cdot \text{final grade}(s) > \text{Passing Grade}) \land \text{majority}(S', S), \]

"Students finish the course with an acceptable level of playing the piano."

where, \( S \) is the set of all students, \( \text{majority}(S', S) \) represents that the elements of set \( S' \) constitute a majority with respect to the elements of the set \( S \) (and the exact definition of the function \( \text{majority} \) will need to be defined; for example, one can have \( \text{majority}(S', S) = \{ \text{true, if } \frac{|S'|}{|S|} > 0.5 \}, \text{false, otherwise} \)). \( \text{final grade}(s) \) specifies the final grade that the student \( s \) obtains by the end of the course, and \( \text{Passing Grade} \) represents the course’s passing grade. Goal \( G \) states that the majority of students pass the course.

9.2 Checking Goal Satisfaction

We say the unfulfillment of community’s goals is one of the triggers for self-evolution. When such a situation arises, community members should be alerted by the system, which would then suggest that the charter may need to be revised as it is failing to fulfil its goals. Of course, there may be other triggers specified by the norms and interaction protocol, which we do not discuss here, such as stating which members are allowed to initiate evolution.

When to check for the satisfaction of community goals is also something that needs to be specified by the interaction protocol. For instance, should this check happen on a daily basis? Should it happen every time a new set of 100 requests is issued? This is an issue to be decided by the community itself (or those members who are given the right to do so) and specified accordingly by the interaction protocol.

Nevertheless, we say the minimum requirement for implementation is for interaction protocols to be capable of calling the goal satisfaction checker. Section 12 elaborates further on this.

10 Norms

Norms describe the rights and duties of community members. We say there are two different types of norms, those that are regimented by the interaction protocol, and those that are enforced by other means (such as punishments and rewards) as they cannot be regimented by the interaction protocol. An example of the former is the norm that states that a student cannot submit its assignment after
the deadline, and the system prevents the student from doing so. An example of the latter is the norm that states that a student is obliged to submit its assignment, although the system cannot force such a thing.

10.1 Regimented Norms

10.1.1 Regimented Norms Specification

Although numerous logics have been proposed in the literature [12], mostly using some kind of modal logic, the most common approach for specifying norms is through deontic logic, the logic of duties that deals with concepts like permissions and prohibitions. We believe the literature is rich enough with logics for one to choose from.

In this part of the deliverable, we propose to specify regimented norms in a simplified deontic-based approach:

\[ \langle \text{NormId}, \text{NormType}, \text{Agents}, \text{Action}, \text{Condition} \rangle \]

where NormId is the norm’s unique identifier, NormType = \{permissible, omissible, obligatory, impermissible, optional\} specifies the type of the norm (we define the main five deontic operators of the Traditional Scheme [13] that describe what is: permissible, omissible, obligatory, impermissible, and optional), Agents describes the set of agent roles that this norm applies to, Action specifies the action the norm addresses, and Condition specifies the circumstances under which the norm holds. Of course, as with goals, one may add a descriptive field to norms to aid humans in comprehending the norms they are discussing or voting on.

10.1.1.1 Regimented Norms Example. As an example, we specify a couple of regimented norms for the online social music learning community.

1. It is obligatory that each student’s assignment is assessed, either manually by a community member, or automatically by the system.

\[ \langle 1, \text{obligatory}, \text{system}, \text{auto\_assess(Assignment)}, \neg \text{manual\_assessment(Assignment)} \rangle \]

2. Students are prohibited from submitting an assignment after the deadline has passed.

\[ \langle 2, \text{prohibited}, \text{student}, \text{submit\_assignment(X)}, \text{deadline\_passed(X)} \rangle \]

Regimented norm (1.) states that if an assignment has not been assessed by a community member (\(\neg \text{manual\_assessment(Assignment)}\)), then the system is obliged to provide an automated assessment for that assignment (\text{auto\_assess(Assignment)}). Regimented norm (2.) states that a student is prohibited to submit an assignment (\text{submit\_assignment(X)}) if the assignment’s deadline has passed (\text{deadline\_passed(X)}).

10.1.2 Verifying Norm Regimentation

Naturally, when norms need to be regimented by the interaction protocol, there should be means for automatically verifying this regimentation each time the norms or the interaction protocols
change. As such, we say there is a need for automatic verification, which should happen once at the formation of the community and then again after each evolution.

Automated theorem proving or model checking are popular approaches with rich existing literature [25]. We propose to use the model checker of [22] that can help verify on the fly whether the norms specified in the proposed syntax above are satisfied in an interaction model specified in LCC [24], which is a lightweight process calculus for specifying multiagent interactions. Naturally, other languages may be used for specifying both the norms and the interaction model and other model checkers may be used for verification. However, in such cases, an appropriate translator is needed to translate the chosen language into the input language of the chosen model checker.

10.2 Enforced Norms

10.2.1 Enforced Norms Specification

Enforced norms are norms that cannot be regimented by the system. They are then enforced by alternative methods, such as applying sanctions (punishments and rewards). Sanctions usually apply to prohibitions and obligations. As such, while other modal logics may be used for specifying regimented norms, enforced norms usually rely on deontic logic [1], as it is the logic of prohibitions and obligations.

Although any of the existing logics may be chosen, in this part of the deliverable, we propose to specify enforced norms by extending the specification of regimented norms with sanctioning information. The proposed approach follows the same style as regimented norms and is coherent with many existing approaches [17, 2].

\[
\langle \text{NormId}, \text{NormType}, \text{Agents}, \text{Action}, \text{Condition}, \text{Reward}, \text{Punishment}, \text{Deadline} \rangle
\]

As in the case of regimented norms: NormId is the norm’s unique identifier, Agents describes the set of agent roles that this norm applies to, Action specifies the action the norm addresses, and Condition specifies the circumstances under which the norm holds and it is specified in first order logic. However, NormType $\in \{\text{impermissible, obligatory}\}$ is now restricted to obligations and forbiddances only, as we shortly explain. Reward and Punishment specify the rewards and punishments that an agent receives if they abide to or break the norm, respectively. Last, Deadline specifies the deadline for an action to be prohibited or obligatory. The use of deadlines is clarified further in Section 10.2.2.

Concerning the restriction of the norm type to obligations and prohibitions, we say enforced norms rely on the concept of sanctions, and sanctions are usually assigned not to permissible actions that one is free to perform or not, but to negative permissions that describe what one is not permitted to perform. The concept of punishment and reward only makes sense when addressing negative permissions. We note that when defining deontic operators, one can pick any of the operators to be the basic operator, and the remaining operators may be defined in terms of the chosen basic operator. To illustrate our argument, we choose the permissible operator to be the basic operator, as such:
permissible $\phi$

omissible $\phi = \text{permissible } \neg \phi$

impermissible $\phi = \neg \text{permissible } \phi$

obligatory $\phi = \neg \text{permissible } \neg \phi$

optional $\phi = \text{permissible } \phi \lor \text{permissible } \neg \phi$

As enforced norms focus on negative permissions only, then the two operators that describe negative permissions are prohibitions (describing impermissible actions) and obligations.

10.2.1.1 A Note on Sanctions. It may be argued that punishment and reward is not always the right approach for motivating the abidance to norms. Furthermore, what may be considered a punishment for one may be viewed as a reward for another. In this part of the deliverable, we label post-conditions as rewards or punishments, although they may simply be interpreted as the post-conditions of abiding with or breaking the norm.

10.2.1.2 Enforced Norms Example. As an example, we specify the following two norms:

1. Students are obliged to submit assignments.

$\langle 1, \text{obligatory}, \text{student}, \text{submit}\_\text{assignment}(X), \text{requested}\_\text{assignment}(X), \text{nil, fail}\_\text{course, assignment}\_\text{deadline}(X) \rangle$

2. Students are forbidden to make comments that contain spam.

$\langle 2, \text{impermissible}, \text{student}, \text{make}\_\text{comment}(X), \text{spam}(X), \text{nil, prohibited}\_\text{to}\_\text{comment}(\text{Next}_{10}\_\text{days}), \text{nil} \rangle$

The first enforced norm states that a student is obliged to submit an assignment ($\text{submit}\_\text{assignment}(X)$) before the assignment’s deadline ($\text{assignment}\_\text{deadline}(X)$) if they have been requested to do so ($\text{requested}\_\text{assignment}(X)$). If they fail to do so, then they will fail the course ($\text{fail}\_\text{course}$). Note that students are not rewarded for submitting their assignments (reward is $\text{nil}$).

The second enforced norm states that a student is forbidden to post comments that contain spam ($\text{make}\_\text{comment}(X)$ and $\text{spam}(X)$), and if they do so, then they will be prohibited to post comments over the next 10 days ($\text{prohibited}\_\text{to}\_\text{comment}(\text{Next}_{10}\_\text{days})$). Note that this rule has no deadline (deadline is $\text{nil}$); i.e. it holds forever. Also, students are not rewarded for not posting spam (reward is $\text{nil}$).

10.2.2 Ensuring Norm Enforcement

Several approaches exist that propose norm enforcement mechanisms [20, 4]. In this part of the deliverable, we propose a basic norm enforcement algorithm, whose pseudocode is presented by Figure 17. The algorithm essentially states that norms become active (or are instantiated) for a given community member if the condition of the norm holds for that given community member. Community members are then either rewarded or punished either when they perform the action
in question or when the deadline passes and they have not yet performed the action in question, depending on the type of the norm. Note that the algorithm is event triggered: The event of having a norm’s condition satisfied triggers the activation (and instantiation) of that norm, and the events of performing an action or having a norm’s deadline pass trigger sanctions and the removal of the instantiated norm.

OnEvent: Constraint $C$ of norm $n$ is satisfied for $\alpha$
Do: Instantiate norm $n$ for agent $\alpha$

OnEvent: $\alpha$ performs action $A$ and there exists an instantiated norm $n$ on $\alpha$’s action $A$
Do: If norm $n$ is of type impermissible Then
Agent $\alpha$ is punished and
the instantiated norm is deleted
Else
Agent $\alpha$ is rewarded and
the instantiated norm is deleted

OnEvent: Deadline of instantiated norm $n$ passes
Do: If norm $n$ is of type impermissible Then
The corresponding agent is rewarded and
the instantiated norm is deleted
Else
The corresponding agent is punished and
the instantiated norm is deleted

Figure 17: Pseudocode for norm enforcement

We note that the system responsible for the execution of interactions should also be responsible for the norm enforcement algorithm, as it needs to keep track of which conditions are being satisfied, which actions are performed, and which deadlines have passed.

11 Interaction Protocols

Interaction protocols describe the operational rules governing the interaction between community members. In abstract terms, they may be specified via labelled transition systems or finite state machines. In multiagent systems, several approaches have been used that one is free to choose from, such as using process calculi [24], electronic institutions [7], or contracts and commitments [5]. We say any approach may be adopted, as long as a model checker can verify the specified interaction protocols against regimented norms. For example, given an electronic institution, the institution may be translated in an automated manner [15] into the lightweight coordination calculus (LCC) [24], which is the language of [22]’s model checker.
11.0.2.1 Interaction Protocol Example. Figure [18] provides an example of an interaction protocol, where the nodes present the state of a process (or agent). The protocol states that there can be a tutor and a number of students interacting, and a number of voters discussing the evolution of the community (one agent may play more than one instance for more than one role at the same time; that is, a student can play the role of the student and the voter at the same time). Note that || describes the parallel operator in process calculus. As an example, we explain the voter’s role, and we leave it to the reader to interpret the rest of the specification, as we believe the names of the agent roles and actions are self-expressive.

![Interaction protocol example](image)

The voter’s protocol states that when an agent plays the role of a voter, first, it will receive a message stating the unsatisfied goal $g$ that has initiated the evolution stage (get_unsatisfied_goal($g$)). Then, either the voter suggests a change $x$ in the charter (suggest_change($x$)), or it receives a suggested change $x$ by some other voter (get_suggested_change($x$)). In the first case, it will wait for others to vote on its suggested change, before receiving the final result $r$ of the vote (get_result($x, r$)). In the latter case, it can either vote $v$ (vote($x, v$)), or it can abstain from voting (abstain($x$)). In both cases it will then be informed of the final voting result $r$ (get_result($x, r$)). As the arrows illustrate, the protocol may loop several times with different voters suggesting new changes and voting on the suggestions, before the final charter $ch$ is agreed upon and the voter is informed (get_final_charter($ch$)).

12 Self-Evolution

The interaction protocols are expected to specify the details of evolution. They should specify when does evolution take place and how does the system trigger evolution (such as when goals are not satisfied), which community members are allowed to suggest evolution (such as permitting the community’s president to trigger evolution whenever he sees fit), the minimum number of people required to discuss evolution (such as stating that at least 30% of the community should be present...
for discussing evolution), or who can suggest changes and how evolution is discussed and agreed upon (such as following some predefined voting mechanism).

Ideally, there would be specific processes dedicated for evolution. That is if we are thinking of interaction protocols specified in a process calculus. For instance, if one uses electronic institutions to model interaction protocols, then one would think of a dedicated evolution scene. In the example presented by Figure 18, the dotted processes, states, and actions are those related to evolution. The processes defining the voters’ roles illustrate how voters may vote on recommended changes. And it is the goal_check(g) action in the tutor’s role that is responsible for initiating evolution. Although, we note that for simplification, the conditions that control the flow are omitted.

Naturally, there is the crucial issue of defining how the interaction needs to be paused at a safe state from which it can be resumed after evolution is completed. We leave this open issue for future research. However, we say after evolution takes place and a new charter is agreed upon, all community members need to be informed of the new charter.

13 Conclusion

This part of the deliverable argues the need for self-evolving communities. We believe self-evolution is still missing in the IT tools we are building and using. And we say that instead of creating numerous rigid systems, we should aim at providing tools for creating self-evolving systems that adapt to the ever evolving community members’ needs.

The deliverable proposes a roadmap for self-evolving communities. It proposes a set of building blocks, which we refer to as the community charter, and presents a concrete approach for each of the proposed building blocks, which helps build the initial prototype for self-evolving communities. We say a community charter should define: (1) the community’s goals, (2) the community’s norms, and (3) the interaction protocols, which include the evolution protocols.

One proposed approach for triggering evolution is for the system to signal unfulfilled goals, suggesting the modification of the charter’s various components in a way that helps address these unfulfilled goals.

Our plans for the final year of the project is to study the notion of community tools for self-evolving communities in further detail, and investigate the possibility of integrating this work with the PRAISE platform.

References


REFERENCES


