Chapter 7

Implementation

In this chapter the main issues encountered during the implementation will be discussed. Technology selection and overview are in Appendix A. It worthwhile to mention here that as long as this project implements a prototype which is part of a major system and focused on the reasoning approach, is not supposed to implement a GUI.

7.1 System packages

There are four main packages in the system:

![Diagram of System Packages]

- **Residence**: All classes related to the residence management.
- **Analyzers**: All classes related to analyze sensorial data.
- **Reasoning**: All classes needed to analyze the reasoning approach.
- **Utils**: Various useful classes.

7.2 Ontology

As the ontology is going to be used by a Reasoner which has only as purpose to classify scenario individuals, the ontology needs to be consistent and satisfactible. It is also required the ontology to be based on OWL-DL logic.

Except unsafe scenarios, all the rest of classes defined in chapter 5, are *primitive classes*, which means they only need to have specified properties as necessary conditions.
Chapter 7

Implementation

In this chapter the main issues encountered during the implementation will be discussed. Technology selection and overview are in Appendix A. It worthwhile to mention here that as long as this project implements a prototype which is part of a major system and focused on the reasoning approach, is not supposed to implement a GUI.

7.1 System packages

There are four main packages in the system:

Figure: System Packages

Residence: All classes related to the residence management.
Analizers: All classes related to analyze sensorial data.
Reasoning: All classes needed to achieve the reasoning approach.
Utils: Various useful classes.

7.2 Ontology

As the ontology is going to be used by a Reasoner which has only as purpose to classify scenario individuals, the ontology needs to be consistent and satisfactible. It is also required the ontology to be based on OWL-DL logic.

Except unsafe scenarios, all the rest of classes defined in chapter 5, are primitive classes, which means they only need to have specified properties as necessary conditions.
Unsafe scenarios are defined classes which means they need to have specified properties as necessary and unsafe conditions. Without this, the Reasoner could not decide whether an individual of scenario is also an individual of unsafe scenario.

Figure: Unsafe composite implementation

7.3 Rule set

The rule set has been implemented using Jena rules through an external file. Jena rule syntax is based on a low level triples and statements which depend directly on how the ontology is written. It follows this structure:

][name: (Subject Property Object ) -> (Subject Property Object )]

This syntax\(^1\) makes the whole rule set implementation complicate and also makes difficult to read and modify afterwards.

The following example shows how is written a rule to detect a temperature getting too cold in an ambient because of an open window.

```

-> makeActionQuestion( ?ambient, ?sensorialDevice, 'window'\^xsd:string, 'Temperature is getting too cold, close the window please.'\^xsd:string )
```

Jena framework offers some basic built-ins\(^2\) to be able to apply some algorithms to rules (for example, to know the minimum between two values) and also lets developers to implement they own built-ins. This is the only way to access to java classes from Jena rules.

As rule set is supposed to generate a response to some specific situations, some built-ins were developed in order to create the appropriate responses and send them to ResponseMngr. (see chapter 3). This was a design change of the conceptual model which had to be done at implementation time. There is one built-in for each type of response. Rules use this built-ins depending on the kind of response the want to give.

\(^1\) Rule Jena Syntax: http://jena.sourceforge.net/inference/#RULEsyntax/
\(^2\) Jena builtin primitives: http://jena.sourceforge.net/inference/#RULEbuiltins/
To use own built-ins, first have to be registered to Jena Built-in Registry, a work currently doing SystemMngr when setting up the system:

```java
BuiltInRegistry.theRegistry.register(new MakeActionQuestion());
```

### 7.4 Ontology querying: ambient information retrieval

When the system is loaded one of the first things it does is obtain all the relevant information from the ontology this includes: ambients, physical elements and sensorial devices.

Jena Framework offers ARQ\(^1\) as an implementation of SPARQL query language offering as features: complex queries, filters and use of regular expressions, query optimizer, remote queries and other facilities. At this point it was necessary to evaluate the necessity of using ARQ instead of Jena API to query an ontology. In the end, ARQ was not decided to use because:

- Queries done for this property are not so complex to require the facilities given by SPARQL.
- Jena Models used to load ontologies seems to offer some kind of indexation for properties, classes and individuals which made them quite fast to be queried.
- ARQ API\(^2\) is not yet stable.
- Although it is not confirmed, it seems the query optimization offered by ARQ is done every time a query is done.

### 7.5 Sensorial data analysis

An AAL system will not have only one sensorial data analyzer, but a set of them. There should be one analyzer for each sensorial device type as the incoming data gives very different information from one device to the other. Similarly, the information inferred from this data depends on each device and where they are attached to.

This project is supposed to use the results of those analyzers but not to create them. For this reason the sensorial data analyze is simulated by obtaining the results from an external file. This facilitates the process of modifying the results given for all the sensorial devices without having to rewrite/compile the application again. By modifying this analysis results at any time, different status for each ambient of the residence can be simulated and see how the system react to them.

---


The external file has a determined structure which has to be followed when adding new sensorial devices to the system. It is composed by blocks for each sensorial device type and for each sensorial device individual. Figure illustrates the structure of the file.

Sensorial devices of the same time are put together in a block type. This block starts with the symbol @ followed by the name of the type. The type block ends with the symbol * . Inside type block, there are several sensorial devices blocks which describe the sensorial devices in the system and its status. Every device separated from the others by a discontinuous line and for each of them, properties related to their status are defined. These properties can be written in any order.

This is an example of how the file should be written for two sensorial devices of type ContactSD and properties name, contact, changeStatus and duration trend:

```plaintext
@ContactSD
  name=contactSD1
  @ambient=bedroom1 (window1)
  contact=true
  changeStatus=false
  durationTrend=continuous

  name=contactSD2
  @ambient=mainEntrance1 (door1)
  contact=true
  changeStatus=false
  durationTrend=continuous

*End ContactSD
```

### 7.6 Reasoning and Pellet Reasoner retraction

To create scenarios from sensorial status, ScenariosProducer adds to a Jena Model the corresponding individuals of simple and composite scenarios for each of the status found in an ambient.

This model is given to SafetyClassifier which use Pellet Reasoner through Jena API to classify those individuals which are unsafe scenarios as well. This takes some time, as Reasoner prepares the model and does all the reasoning process. It was thought to use Event Model Mechanism¹ offered by Jena to detect every time a scenario was classified as unsafe.

The main problems which appeared during the implementation were:

1. The way to use an external Reasoner with Jena API is by using a DIG Interface. However this could not be done with Pellet (as Pellet developers warned) because there are some times problems with restrictions and datatype

¹ Model Event Mechanism: http://jena.sourceforge.net/how-to/event-handler.html/
properties from ontologies. Due to this, a direct connection to the reasoner had be done, removing the possibility to use some Jena Model features with it as it was initially planned.

2. Using Pellet Model to apply rules over its results and, layering in this way Pellet Reasoner with Jena Rule Engine, appeared to be really slow. This was explained because every time the rule engine infers a new statement in the model, the Reasoner does all the classification process again.

3. Next test was obtaining the model from Pellet Reasoner and giving it to Jena Rule Engine, without layering both engines. However the model still points to the Reasoner, so again the classification process is repeated

4. It was decided then to obtain manually the scenarios classified as unsafe and update with them the base model which is used by the rule engine. Every time a query was done to Pellet Model (e.g. search the class "UnsafeScn", obtain its individuals, etc.) the classification process was done again.

5. Finally, it was decided to obtain a complete new model from Pellet Model after the classification is done. This takes also time but not so much as doing the classification over and over again. This in a large ontology could end in completely unacceptable times for an AAL System.

7.7 Response management

Due to time reasons, response management has been kept too simple. It receives all the responses generated by SafetyChecker, gives them a treatment depending on the type they are and prints them. In chapter 9, the main features it should have the response manager are discussed.
Chapter 7

Implementation

In this chapter the main issues encountered during the implementation will be discussed. Technology selection and overview are in Appendix A. It worthwhile to mention here that as long as this project implements a prototype which is part of a major system and focused on the reasoning approach, is not supposed to implement a GUI.

7.1 System packages

There are four main packages in the system:

![System Packages Diagram]

- **Residence**: All classes related to the residence management.
- **Analizers**: All classes related to analyze sensorial data.
- **Reasoning**: All classes needed to achieve the reasoning approach.
- **Utils**: Various useful classes.

7.2 Ontology

As the ontology is going to be used by a Reasoner which has only as purpose to classify scenario individuals, the ontology needs to be consistent and satisfactible. It is also required the ontology to be based on OWL-DL logic.

Except unsafe scenarios, all the rest of classes defined in chapter 5, are primitive classes, which means they only need to have specified properties as necessary conditions.
Chapter 7

Implementation

In this chapter the main issues encountered during the implementation will be discussed. Technology selection and overview are in Appendix A. It worthwhile to mention here that as long as this project implements a prototype which is part of a major system and focused on the reasoning approach, is not supposed to implement a GUI.

7.1 System packages

There are four main packages in the system:

- **residence**
- **reasoning**
- **analyzers**
- **utils**

Figure: System Packages

**Residence:** All classes related to the residence management.

**Analyzers:** All classes related to analyze sensorial data.

**Reasoning:** All classes needed to achieve the reasoning approach.

**Utils:** Various useful classes.

7.2 Ontology

As the ontology is going to be used by a Reasoner which has only as purpose to classify scenario individuals, the ontology needs to be consistent and satisfactible. It is also required the ontology to be based on OWL-DL logic.

Except unsafe scenarios, all the rest of classes defined in chapter 5, are primitive classes, which means they only need to have specified properties as necessary conditions.
Unsafe scenarios are defined classes which means they need to have specified properties as necessary and unsafe conditions. Without this, the Reasoner could not decide whether an individual of scenario is also an individual of unsafe scenario.

Figure: Unsafe composite implementation

7.3 Rule set

The rule set has been implemented using Jena rules through an external file. Jena rule syntax is based on a low level triples and statements which depend directly on how the ontology is written. It follows this structure:

```
[name: (Subject Property Object ) ] -> (Subject Property Object )
```

This syntax\(^1\) makes the whole rule set implementation complicate and also makes difficult to read and modify afterwards.

The following example shows how is written a rule to detect a temperature getting too cold in an ambient because of an open window.

```
[tempTooCold: (?SCN rdf:type ont:CompositeSCN) (?SCN
ont:isScenarioOf ?ambient) (?ambient ont:hasAmbientName
?ambientName)(?SCN rdf:type ont:UnsafeSCN) (?SCN
ont:hasSimpleSCN ?temp) (?temp rdf:type
ont:TempGettingTooCold) (?temp ont:isSimpleSCNOf
?sensorialDevice)(?SCN ont:hasSimpleSCN ?window) (?window
rdf:type ont:WindowSCN) (?window rdf:type ont:OpenStatus

-> makeActionQuestion( ?ambient, ?sensorialDevice,
'window'"^xsd:string, 'Temperature is getting too cold, close
the window please.'"^xsd:string )
```

Jena framework offers some basic built-ins\(^2\) to be able to apply some algorithms to rules (for example, to know the minimum between two values) and also lets developers to implement they own built-ins. This is the only way to access to java classes from Jena rules.

As rule set is supposed to generate a response to some specific situations, some built-ins were developed in order to create the appropriate responses and send them to ResponseMngr. (see chapter 3). This was a design change of the conceptual model which had to be done at implementation time. There is one built-in for each type of response. Rules use this built-ins depending on the kind of response the want to give.

---

\(^1\) Rule Jena Syntax: http://jena.sourceforge.net/inference/#RULEsyntax/

\(^2\) Jena built-in primitives: http://jena.sourceforge.net/inference/#RULEbuiltins/
To use own built-ins, first have to be registered to Jena Buit-in Registry, a work currently doing SystemMngr when setting up the system:

```
BuiltinRegistry.theRegistry.register(new MakeActionQuestion());
```

### 7.4 Ontology querying: ambient information retrieval

When the system is loaded one of the first things it does is obtain all the relevant information from the ontology this includes: ambients, physical elements and sensorial devices.

Jena Framework offers ARQ\(^1\) as an implementation of SPARQL query language offering as features: complex queries, filters and use of regular expressions, query optimizer, remote queries and other facilities. At this point it was necessary to evaluate the necessity of using ARQ instead of Jena API to query an ontology. In the end, ARQ was not decided to use because:

- Queries done for this property are not so complex to require the facilities given by SPARQL.
- Jena Models used to load ontologies seems to offer some kind of indexation for properties, classes and individuals which made them quite fast to be queried.
- ARQ API\(^2\) is not yet stable.
- Although it is not confirmed, it seems the query optimization offered by ARQ is done every time a query is done.

### 7.5 Sensorial data analysis

An AAL system will not have only one sensorial data analyzer, but a set of them. There should be one analyzer for each sensorial device type as the incoming data gives very different information from one device to the other. Similarly, the information inferred from this data depends on each device and where they are attached to.

This project is supposed to use the results of those analyzers but not to create them. For this reason the sensorial data analyze is simulated by obtaining the results from an external file. This facilitates the process of modifying the results given for all the sensorial devices without having to rewrite/compile the application again. By modifying this analysis results at any time, different status for each ambient of the residence can be simulated and see how the system react to them.

---

\(^1\) Introduction to SPARQL + Jena: http://www.ibm.com/developerworks/xml/library/j-sparql/

\(^2\) ARQ API: http://jena.sourceforge.net/ARQ/javadoc/index.html/
The external file has a determined structure which has to be followed when adding new sensorial devices to the system. It is composed by blocks for each sensorial device type and for each sensorial device individual. Figure illustrates the structure of the file.

Sensorial devices of the same time are put together in a block type. This block starts with the symbol @ followed by the name of the type. The type block ends with the symbol *. Inside type block, there are several sensorial devices blocks which describe the sensorial devices in the system and its status. Every device separated from the others by a discontinuous line and for each of them, properties related to their status are defined. These properties can be written in any order.

This is an example of how the file should be written for two sensorial devices of type ContactSD and properties name, contact, changeStatus and duration trend:

```
@ContactSD
name=contactSD1
#ambient=bedroom1 (window1)
contact=true
changeStatus=false
durationTrend=continuous

name=contactSD2
#ambient=mainEntrance1 (door1)
contact=true
changeStatus=false
durationTrend=continuous

*End ContactSD
```

### 7.6 Reasoning and Pellet Reasoner retraction

To create scenarios from sensorial status, ScenariosProducer adds to a Jena Model the corresponding individuals of simple and composite scenarios for each of the status found in an ambient.

This model is given to SafetyClassifier which use Pellet Reasoner through Jena API to classify those individuals which are unsafe scenarios as well. This takes some time, as Reasoner prepares the model and does all the reasoning process. It was thought to use Event Model Mechanism\(^1\) offered by Jena to detect every time a scenario was classified as unsafe.

The main problems which appeared during the implementation were:

1. The way to use an external Reasoner with Jena API is by using a DIG interface. However this could not be done with Pellet (as Pellet developers warned) because there are some times problems with restrictions and datatype

\(^1\) Model Event Mechanism: http://jena.sourceforge.net/how-to/event-handler.html/
properties from ontologies. Due to this, a direct connection to the reasoner had
be done, removing the possibility to use some Jena Model features with it as it
was initially planned.

2. Using Pellet Model to apply rules over its results and, layering in this way Pellet
Reasoner with Jena Rule Engine, appeared to be really slow. This was
explained because every time the rule engine infers a new statement in the
model, the Reasoner does all the classification process again.

3. Next test was obtaining the model from Pellet Reasoner and giving it to Jena
Rule Engine, without layering both engines. However the model still points to
the Reasoner, so again the classification process is repeated

4. It was decided then to obtain manually the scenarios classified as unsafe and
update with them the base model which is used by the rule engine. Every time a
query was done to Pellet Model (e.g. search the class “UnsafeScn”, obtain its
individuals, etc.) the classification process was done again.

5. Finally, it was decided to obtain a complete new model from Pellet Model after
the classification is done. This takes also time but not so much as doing the
classification over and over again. This in a large ontology could end in
completely unacceptable times for an AAL System.

7.7 Response management

Due to time reasons, response management has been kept too simple. It receives all
the responses generated by SafetyChecker, gives them a treatment depending on
the type they are and prints them. In chapter 9, the main features it should have the
response manager are discussed.
Chapter 8

Testing

In order to test the system, equivalence partitioning black box tests have been used based in four main contexts:

1. **All correct values.** All sensorial devices give safe readings.

2. **One unsafe value.** One sensorial device gives unsafe readings while the others give safe readings.

3. **Combination values.** Some sensorial devices give safe readings while others give unsafe readings.

4. **All incorrect values.** All sensorial devices give unsafe readings.

8.1 Methodology

As is not possible to write all the possible tests for all the scenarios implemented in the system, the tests are going to be explained in base of two of them: TemperatureSCN and WindowSCN. The rest follows a similar structure.

On one hand, TemperatureSCN is a temperature scenario with information based on temperature level and environment trend. Each possible values are described in Table 8.1 and Table 8.2.

<table>
<thead>
<tr>
<th>Temperature level</th>
<th>Thermometer value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cold</td>
<td>20&lt;value&lt;18</td>
</tr>
<tr>
<td>Normal</td>
<td>20&lt;=value&lt;25</td>
</tr>
<tr>
<td>Warm</td>
<td>25&lt;=value&lt;28</td>
</tr>
<tr>
<td>Unacceptable</td>
<td>28&lt;=value and value&lt;=18</td>
</tr>
</tbody>
</table>

*Table 8.1 Temperature level values*

<table>
<thead>
<tr>
<th>Environment trend</th>
<th>Thermometer value history</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decrease</td>
<td>Thermometer values have been decreasing for the last 5 reads</td>
</tr>
<tr>
<td>Increase</td>
<td>Thermometer values have been increasing for the last 5 reads</td>
</tr>
</tbody>
</table>

60
Table 8.2 Environment trend values

| Stable          | Thermometer values have been stable for the last 5 reads |

On the other hand, WindowSCN is a window scenario which basically says whether the status of the window it represents. Table 8.3 each of its possible values.

<table>
<thead>
<tr>
<th>Window status</th>
<th>Contact sensor value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open</td>
<td>false</td>
</tr>
<tr>
<td>Closed</td>
<td>true</td>
</tr>
</tbody>
</table>

Table 8.3 Window status values

For TemperatureSCN these are the expected values.

<table>
<thead>
<tr>
<th>ID</th>
<th>Input</th>
<th>Expected result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Temperature level: cold</td>
<td>Alarm1: Temperature getting too cold.</td>
</tr>
<tr>
<td></td>
<td>Environment trend: decrease</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Temperature level: cold</td>
<td>accepted</td>
</tr>
<tr>
<td></td>
<td>Environment trend: stable</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Temperature level: cold</td>
<td>accepted</td>
</tr>
<tr>
<td></td>
<td>Environment trend: increase</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Temperature level: normal</td>
<td>accepted</td>
</tr>
<tr>
<td></td>
<td>Environment trend: decrease</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Temperature level: normal</td>
<td>accepted</td>
</tr>
<tr>
<td></td>
<td>Environment trend: stable</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Temperature level: normal</td>
<td>accepted</td>
</tr>
<tr>
<td></td>
<td>Environment trend: increase</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Temperature level: warm</td>
<td>accepted</td>
</tr>
<tr>
<td></td>
<td>Environment trend: decrease</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Temperature level: normal</td>
<td>accepted</td>
</tr>
<tr>
<td></td>
<td>Environment trend: stable</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Temperature level: normal</td>
<td>Alarm2: Temperature getting too warm.</td>
</tr>
<tr>
<td></td>
<td>Environment trend: increase</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Environment unacceptable</td>
<td>Emergency: Temperature unacceptable</td>
</tr>
</tbody>
</table>

Table 8.3 TemperatureSCN expected values

For WindowSCN these are the expected values.

<table>
<thead>
<tr>
<th>ID</th>
<th>Input</th>
<th>Expected result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Window status: closed</td>
<td>accepted</td>
</tr>
<tr>
<td>2</td>
<td>Window status: open</td>
<td>Detected unsafe but no alarm.</td>
</tr>
</tbody>
</table>

Table 8.3 WindowSCN expected values
<table>
<thead>
<tr>
<th>Environment trend: stable</th>
<th>Window status: closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>4  Temperature level: cold</td>
<td>accepted</td>
</tr>
<tr>
<td>Environment trend: stable</td>
<td></td>
</tr>
<tr>
<td>Window status: open</td>
<td></td>
</tr>
<tr>
<td>5  ...</td>
<td>accepted</td>
</tr>
<tr>
<td>6  Temperature level: warm</td>
<td>accepted</td>
</tr>
<tr>
<td>Environment trend: stable</td>
<td></td>
</tr>
<tr>
<td>Window status: closed</td>
<td></td>
</tr>
<tr>
<td>7  Temperature level: warm</td>
<td>Alarm2: Temperature getting too warm.</td>
</tr>
<tr>
<td>Environment trend: increase</td>
<td></td>
</tr>
<tr>
<td>Window status: open</td>
<td></td>
</tr>
<tr>
<td>8  Temperature level: warm</td>
<td>Alarm2: Temperature getting too warm.</td>
</tr>
<tr>
<td>Environment trend: increase</td>
<td></td>
</tr>
<tr>
<td>Window status: closed</td>
<td>Ask action: open window</td>
</tr>
<tr>
<td>9  Environment unacceptable</td>
<td>Emergency: Temperature unacceptable</td>
</tr>
</tbody>
</table>

*Table 8.4 TemperatureSCN and WindowSCN expected values*

### 8.2 Results

The rules were fired as expected in context 1 and 2. However, some errors were derived from context 3 and four. As there is no priority in showing rules, may occur that critical ones are shown at the end. May occur to ask resident to do an action to ambients where there has been an emergency (e.g. ask to close window in an ambient where it has been detected a fire). It may also occur to have an ambient with two contradictory scenarios (one safe and the other unsafe).
Chapter 9

Evaluation and future work

This project has meant a challenge in terms of working with several technologies and concepts which had never seen before. At the beginning point it was necessary to learn about AAL Systems, ontologies, rule engines and reasoner. There was also need to rapidly get enough knowledge about a completely new API and framework as Jena is and complement it with an other new engine, Pellet Reasoner.

Problems have been encountered during the realization of the project, but in the end of it it can be said although the prototype implemented in the project can be polished and improved, it achieves its main objectives:

- Reads sensorial data and creates envisage situations (scenarios) of all the ambients of the residence.
- Classifies the unsafe situations (unsafe scenarios) from what is defined (in the ontology).
- Applies rules to check the safety of the situations and generate responses to them when is needed.

Related to the project main objectives:

- It gives an overview to AAL systems and proposes some basic features and requirements.
- It proposes the use of an ontology to save the residence information and to define the possible envisage situations.
- It proposes different options to define the envisage situation by means of scenarios hierarchies in the ontology.
- It proposes the use of a set of technologies to achieve its purposes and shows the result of applying them.
- It presents a prototype of a reasoning approach about AAL systems which achieves its main objectives.
9.1 Problems encountered

Main problems have been listed through the report, especially in chapter 8.

**Jena Framework**: Jena Framework does not have any kind of GUI or IDE to facilitate the develop task. Even the definition of an ontology requires a significant amount of time. There is also a complete lack of tutorials, not all the API is documented and the given examples are quite simple. Some things can be done in too many different ways (due to previous versions) but they are not usually explained together. The way of learning how to use Jena is by means of Jena developers list, asking or searching similar questions over and over again.

**Jena rules**: Jena rules syntax is very basic and does not give too many facilities. It is based on statements and becomes difficult to read, modify or checking for errors after having some rules. There are some basic built-ins to be used by rules and there is the possibility of creating own built-ins which, again, is not too much documented. By the moment, there is no other way to access to Java code from rules, not even global variables. Jess, by its side, seemed to be prepared to offer this kind of necessities to developers as it lets create Java objects, call Java methods and implement Java interfaces without compiling code. It is also a limitation to only be able to use rules with data based on RDF.

**Jena integration with Pellet**: Jena and Pellet which presume to be the best couple in terms of API and Reasoner have also problems. The DIG Interface offered by Jena to let developers use Jena features through other reasoners, does not work with Pellet when datatype properties and some kind of restrictions are used.

**Pellet retraction**: this is the worst issue found during the realization of the project. The idea to use a Reasoner was to obtain unsafe scenarios faster than doing it all by only rules. Having to obtain and create the data model one time and other makes the system go slow. In AAL System Timeless is one of the non functional requirements as it is stated in chapter 2. It is also the core and almost most important of them.

**Planification**: There was probably too much time used in researching about AAL systems and existent projects. The design of the ontology it also needed a lot more time that the one it was expected and in the end, this have mean less time for implementing and avoiding features initially planned to have. With more time, the scenarios, the unsafe scenarios and the rules would have been more polished. More over, the response Manager would have act more similarly as what it is expected by an AAL System, giving some time to resident to do actions, establishing priorities in responses, detecting repeated responses on letting to fusion the similar ones, etc.

9.2 Future work

Chapter 3 stated some requirements an AAL system should have. Being aware of these ones it would be interesting to:

- System customization from resident profile.
- Resident profile based as well in medical conditions.
- Behavior learning from behavioral patterns.
- Maintain history of what is happening in the residence.
- Use fuzzy logic to obtain scenario status.
- Save ontology in database instead of a file.
- Use different levels of safety (not only unsafe, but also concerning, etc.)
Bibliography


Glossary

**AAL169**: Article 169 from the European AAL programme that will provide the necessary legal and organisational framework as well as research activities around the Ambient Assisted Living systems.

**A-box**: In description logic terminology, contains the assertions about the individuals in the domain.

**Activities of Daily Living (ADLs)**: Activities focused on assessing ability to perform basic self-care activities such as eating, dressing, bathing, toileting, transferring in and out of bed/chair and walking.

**Ambient**: Any room or limited space in a residence where the resident may be and where the well-being and safety has to be measured.

**Ambient intelligence (AmI)**: Enabling technology for a new generation of systems, which provide their services in a flexible, transparent, and anticipative manner requiring minimal skills for human-computer interaction.

**ARQ**: Implementation of SPARQL by Jena Web Semantic Framework team.

**API**: Application Programming Interface.

**Concept Satisfiability**: Given an ontology \( O \) and a class \( A \), verify whether there is a model of \( O \) in which the interpretation of \( A \) is a non-empty set.

**Concept Subsumption**: Given an ontology \( O \) and two classes \( A, B \), verify whether the interpretation of \( A \) is a subset of the interpretation of \( B \) in every model of \( O \).

**Close world assumption (CWA)**: assumption which says that a statement in a knowledge representation that cannot be inferred to be true, is false.

**Description Logic (DL)**: Family of knowledge languages which can be used to represent the terminological knowledge of an application domain in a structured and formally well-understood way.

**Java Agent DEvelopment Framework (JADE)**: Framework based in FIPA standard to create applications through an agent based architecture. Advanced features such as: distributed security, fault tolerance, support for replicated agents and services, persistence.

**Interface for Description Logic Reasoners DIG**: Standardised XML interface developed by the DL Implementation Group. Provides uniform access to Description Logic Reasoners via HTTP protocol and a XML Schema that describes the concept language and the accompanying operations.
Information Society Technologies Advisory Group (ISTAG): Advisory body to the European Commission in the field of Information and Communication Technology (ICT or IST). It reflects and advises on the definition and implementation of a coherent policy for research in and on ICT in Europe.

Instance Checking: Given an ontology, an individual a and a class A, verify whether a is an instance of A in every model of the ontology.

Instrumental activities of Daily Living (IADLs): Instrumental activities such as mobility, medication management, and grocery shopping.

Scenario: Description of an ambient situation which might become unsafe for the resident.

Ontology Consistency: Check whether a given ontology has at least one model.

Open world assumption (OWA): assumption which says that a statement in a knowledge representation that cannot be inferred to be true, still cannot be inferred to be false.

Context-awareness: the computer is aware of its environment and acts accordingly, e.g. time, temperature, location, user interests, etc.

ICT: information and communication technology

Java Rule Engine API (JSR 94) program, defines a Java runtime API for rule engines by providing a simple API to access a rule engine from a Java Platform, Standard Edition (Java SE, formerly known as J2SE) or a Java Platform, Enterprise Edition (Java EE, formerly known as J2EE) Java technology

JENA is a Java framework for building Semantic Web applications. It provides a programmatic environment for RDF, RDFS and OWL, SPARQL and includes a rule-based inference engine. Jena is open source and grown out of work with the HP Labs Semantic Web Programme.

JfuzzyLogic API enables us to analyse the multiple data coming from the sensors in such a way that makes possible a better classification of the different scenarios with respect of the comfort and well-being of the resident

N 3 (Notation 3) is a language which is a compact and readable alternative to RDF's XML syntax, but also is extended to allow greater expressiveness. It has subsets, one of which is RDF 1.0 equivalent, and one of which is RDF plus a form of RDF rules.

N-triples is a line-based, plain text format for representing the correct answers for parsing RDF/XML test cases as part of the RDF Core working group. Was designed to be a fixed subset of N3 and hence N3 tools such as cwm, n-triples2kif.pl and Euler can be used to read and process it. cwm can output this format when invoked as "cwm -triples".

OWL Web Ontology Language constitutes the formal W3C38 Recommendation for an ontology representation language, builds on RDF and RDF Schema and adds more vocabulary for describing properties and classes. Is designed for use by applications that need to process the content of information instead of just presenting
information to humans. Has three increasingly-expressive sublanguages: OWL Lite, OWL DL, and OWL Full.

**Sensor**: A device that responds to a physical stimulus, such as thermal energy, electromagnetic energy, acoustic energy, pressure, magnetism, or motion, by producing a signal, usually electrical.

**SWRL**: Rule Language based on a combination of the OWL DL and OWL Lite with the Unary/Binary Datalog RuleML sublanguages of the Rule Markup Language. SWRL includes a high-level abstract syntax for Horn-like rules in both the OWL DL and OWL Lite sublanguages of OWL. A modeltheoretic semantics is given to provide the formal meaning for OWL ontologies including rules written in this abstract syntax. An XML syntax based on RuleML and the OWL XML Presentation
Appendix A

Technology review

A.1 Technology selection

3.2.1 Software

- Netbeans 5.5.1 with UML Plug-in for coding and UML designs.
- Java(TM) SE Development Kit 6 Update 2 for implementation.
- Jena 2.5.4 Semantic Web Framework for ontology management through Java.
- Pellet Reasoner 1.5.0 for scenarios classification.
- Protégé 3.3.4 for ontology creation.
- Protégé plug-ins OntoViz and OWLViz plug-in for ontology design images.
- Graph Visualization Software 2.14.1 required for Protégé plug-ins.
- Freemind 0.8 for drafts and summaries.
- Subversion server with Tortoise 1.4.5 client for version control.
- Windows Office (Microsoft word, Microsoft PowerPoint and Microsoft Visio) for report and presentation.
- Freehand and Photoshop for image editing.

3.2.2 Hardware

The machine used for the development of this system was a Toshiba Satellite m40 with 1GB RAM and 1,86 GHz

A.2 Rule engines

AAL systems need to infer knowledge from what it is known from a residence. A way to do it is using rules to detect strange situations and decide how to act consequently. Chapter 6 gives a more proximate idea of what a rule is.

Rule Engines help developers to manage rules and integrate them with the existent technologies and their own applications. Nowadays, there are a lot of possibilities in the market so we can choose the one that fulfill best our purposes specifying some decision points.

Decision points

To choose a rule engine we specified these decisions points:

- Open source solution.
- Preferably use of standards.
- Easy integration with own application. (most of them use their own API which is not always easy to integrate).
- Easy integration with Java language.
- Facilities to change rules without needing to rewrite applications.
- Easy language used to describe rules.

**Existent options**

From the amount of rule engines, these ones were considered (although not all of them fulfill the decision points mentioned above):

<table>
<thead>
<tr>
<th><strong>Jess</strong>&lt;sup&gt;1&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Free use for academic use. Commercial use requires a license.</td>
</tr>
<tr>
<td>- Standards support: Java Rule Engine API (JSR-94)</td>
</tr>
<tr>
<td>- Integration with Java language application.</td>
</tr>
<tr>
<td>- Specific rule language based on XML: JessML. This language is easy to transform to other XML rule languages.</td>
</tr>
<tr>
<td>- Easy to manage rules as Jess offers its own tools for reading, writing and transforming.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Jena 2 Framework</strong>&lt;sup&gt;2&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Open source solution.</td>
</tr>
<tr>
<td>- Jena 2 is itself a Java framework which includes a generic rule based inference engine and the possibility to integrate other reasoners (such as Pellet to offer more functionalities). Therefore, it offers complete integration with Java applications.</td>
</tr>
<tr>
<td>- It additionally offers an extensive API to work with ontologies.</td>
</tr>
<tr>
<td>- Uses a specific rule language parsed afterwards to Java Rules,</td>
</tr>
<tr>
<td>- Rules can be loaded from external files. Thus, modification of these rules may not affect the application.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>JBoss Rules</strong>&lt;sup&gt;3&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Open source solution.</td>
</tr>
<tr>
<td>- Standards support: Java Rule Engine API (JSR-94) and J2EE 1.4 compliance</td>
</tr>
<tr>
<td>- Integration with Java language application.</td>
</tr>
</tbody>
</table>

---

<sup>1</sup> Jess: http://herzberg.ca.sandia.gov/jess/

<sup>2</sup> Jena 2: http://www.hpl.hp.com/semweb/jena2.htm

<sup>3</sup> JBoss Rules: http://labs.jboss.com/drools/
- Rules more focused to business logics
- Rule language easy to learn as it is thought to be used for business people.
- Easy to modify rules that may change complete behavior of a system.

**SweetRules (Semantic WEb Enabling Technology)**\(^1\)

- Open source solution.
- Is not itself an Rule based engine but offers lots of tools to work with some of them such as Jess and Jena.
- Use of standards for rules such as RuleML and SWRL:
- Tools for writing rules in different languages and transform them.

**Selected option**

The decided option was Jena Framework and Pellet reasoner for being both open sourced and named by several people as a good option.

JBoss Rules seems to be centered too much in Business and does not seems to offer ontology management. Jess is really interesting but as it is not exactly open source has been dismissed.

---

\(^1\) SweetRules: [http://sweetrules.projects.scmwebcentral.org/](http://sweetrules.projects.scmwebcentral.org/)
Appendix B

User manual

B.1 How to use the system

From a console, go through AALReasoning directory and type:

```
java -jar "AALReasoning.jar"
```

Be sure the directories files and lib are in the same directory as the jar and that both of them contain the ontology, the rule set and the file with the status of the sensorial devices contained in the ontology.

The ontology specified in the right directory is loaded, the sensorial devices are upload with the status of the sensorial device status file while some printings appear in the console.

Immediately afterwards for each sensorial device and the physical elements attached to them, the corresponding simple and composite scenarios are created, classified and the rules fired. Each of the rules fired prints its response.

Just as an example, initial status detect a temperature getting too cold in ambient bedroom1 and asks for close the open window.

Has to be mention that Pellet Reasoner prints also a considerable quantity of verbose.

This are the individuals in the ontology:
B.1 How to change sensorial data status

Get the file SensorialDevices.status in directory /files/ which follows the structure explained in chapter 7 section 7.5. There has to be one SDType bloc for each of the sensorial devices in the ontology. The possible values for each sensorial device are commented at the end of the file:

```plaintext
#------------------------
#type=ContactSD
#name=
#ambient=
#contact={true,false}
#changeStatus={true,false}
#durationTrend={continuous,frequent,brief,tooLong}
#------------------------
#type=CurrentSD
#name=
#ambient=
```

75
#applianceStatus={on, off}
#durationTrend={continuous, frequent, brief, tooLong}
#type=FireSD
#name=
#ambient=
#isDetected={true, false}

#type=FloodSD
#name=
#ambient=
#changeStatus={true, false}
#durationTrend={continuous, frequent, brief, tooLong}
#hasOverflowed={true, false}
#switchStatus={open, closed}

#type=HumiditySD
#name=
#ambient=
#humidityLevel={moist, dry, normalHum, unacceptableHum}
#environmentTrend={stable, increase, decrease}
#maxTimeUnsafe=[-1, ...]

#type=LightSD
#name=
#ambient=
#lightLevel={bright, poor, dark, normalLight}
#environmentTrend={stable, increase, decrease}

#type=MovementSD
#name=
#ambient=
#movementLevel={excited, normalMovement, quiet}
#durationTrend={continuous, frequent, brief, tooLong}
#movementType={longRange, shortRange}

#type=NoiseSD
#name=
#ambient=
#noiseLevel={normalNoise, calm, loud}
#environmentTrend={stable, increase, decrease}
#durationTrend={continuous, frequent, brief, tooLong}

#type=ProximitySD
#name=
#ambient=
#isResident={true, false}
#durationTrend={continuous, frequent, brief, tooLong}

#type=SmokeSD
#name=
#ambient=
#isDetected={true, false}

#type=ThermometerSD
#name=
B.1 How to modify a residence

Every modification in the residence has to be done through the ontology. Protégé 3.3.4 can be used to facilitate this process.
Appendix C

Maintenance manual

C.1 Installation instructions

These instructions are supposed to be done under Windows operative system it is not guarantied to work under Linux operative system.

1. Install Java SE Development kit 6 Update 2:
   - Follow the installation instructions and obtain the installation file from: http://java.sun.com/javase/downloads/index.jsp/

2. Install Jena 2.5.4 Semantic Web Framework:
   - Obtain the Jena version file from: http://sourceforge.net/project/downloading.php?groupname=jena&filename=Jena-2.5.4.zip&use_mirror=dfn/
   - Extract contents of Jena File in a directory of your choice.
   - Modify CLASSPATH class with the path of all Jena jars. Example: <JENA_ROOT> /lib/stax-api-1.0.jar;

3. Install Pellet Reasoner 1.5.0:
   - Obtain Pellet version file from: http://pellet.owldl.com/
   - Extract contents of Pellet file to a directory of your choice.

4. Select a directory and extract the contents of the AALReasoning code archive.

C.2 Running AAL Reasoning

Option 1: From Netbeans 5.5.1 (or superior)
Create a new project with Netbeans IDE and add to it the contents of AALReasoning. Modify project properties in order to use the required jars found in /lib/jena directory and /lib/pellet directory.

Option 2: From batch file.
Execute the AALreasoning batch file.

C.3 Dependencies

The system depends on all libraries found in /aalreasoning/lib.

C.3.1 Jena Libraries
Some of this libraries are not currently used but offers Jena functionalities that could be interesting in future.

antlr-2.7.5.jar
arq.jar
arq-extra.jar
commons-logging-1.1.jar
concurrent.jar
icu4j_3_4.jar
iri.jar
jena.jar
jenatest.jar
json.jar
junit.jar
log4j-1.2.12.jar
lucene-core-2.0.0.jar
stax-api-1.0.jar
wstx-asl-3.0.0.jar
xercesImpl.jar
xml-apis.jar

C. 3.1 Pellet Libraries

Owlapi libraries offered by Pellet have not been included as the system is using Jena libraries.

aterm-java-1.6.jar
pellet.jar
servlet.jar
jetty.jar
junit.jar
relaxngDatatype.jar
xsdlib.jar
C. 4 Source files

/aalreasoning/
- SystemMngr: Main class which loads the program and prepares the rest of components.
- SystemConfig: Contains global variables such as paths to localize the ontology and the rule files.
- ResidentProfile: These are global variables which depend on the resident profile. Although current system version is not using them because there was no time to include as a system feature the customization of it for each kind of users, it is the way to go for future work.

/aalreasoning/analyzers
- SensorialDataAnalyzer: This class corresponds to an external system. It should analyze all the sensorial devices of the system and provide them an status knowing their currently value and a history of values. To simulate this, the different values for each existent sensorial device are read from an external file.

/aalreasoning/reasoning
- ActionQuestionResponse: Represents a response from the system to the residence environment reasoning. The system ask the resident to do an action and waits for some maximum time to have the action done.
- AlarmResponse: Represents a response of type alarm from the system to the residence environment reasoning. An alarm means there is a strange behaviour in the residence.
- EmergencyResponse: Represents a response of type EmergencyResponse from the system to the residence environment reasoning. An emergency means the resident is in a potentially dangerous situation and external help is required.
- HistoryResponse: Represents a history response from the system to the residence environment reasoning.
- InformResponse: Represents a response of type inform from the system to the residence environment reasoning. Used for information or advice purposes.
- MakeActionQuestion: This class is a builtin created to use along with Jena Rules Syntax. Creates a response of type ActionQuestionResponse with the relevant information about the residence.
- MakeAlarm: This class is a builtin created to use along with Jena Rules Syntax. Creates a response of type AlarmResponse with the relevant information about the residence.
- MakeEmergency: This class is a builtin created to use along with Jena Rules Syntax. Creates a response of type EmergencyResponse with the relevant information about the residence.
- MakeHistory: This class is a builtin created to use along with Jena Rules Syntax. Creates a response of type HistoryResponse with the relevant information about the residence.
- MakeInform: This class is a builtin created to use along with Jena Rules Syntax. Creates a response of type ActionQuestion with the relevant information about the residence.
- Response: Represents a response from the system to the residence environment reasoning. A response usually has occurred in an ambient and has a message. Other subclasses of response may have other relevant information.
- ResponseMngr: This class manages the responses given to the system environment situation. There are different types of responses that are treated in a different way. In a future work, this class should be a thread and be always listening for new responses. It also should be more complete, using priorities in the messages and joining messages for a same ambient, avoiding messages for a dangerous ambient, etc.
- SafetyAnnalyst: Checks the safety of the scenarios classified as unsafe and if necessary generates a response to them.
- SafetyClassifier: Does some reasoning to classify the unsafe scenarios from a model.
- ScenariosProducer: This class creates Scenario individuals for each of the sensorial devices status.

/aa1reasoning/residence
- ResidenceMngr: Manages all the operations with a residence.

/aa1reasoning/residence/elements
- Ambient: This class represents an ambient in the residence.
- PhysicalElement: This class represents a Physical Element and the sensorial devices attached to it.

/aa1reasoning/residence/sensorialDevices
- SDType: Encapsulate the different types of sensorial devices in the system. Every type corresponds to a name in the ontology.
- SDValue: Common class for all sensorial devices.
- SensorialDevice: Represents a Sensorial Device with its complete name, type, current value and current status.

/aa1reasoning/residence/sensorialDevices/factory
- ContactSDFactory: Creates the sensorial device ContactSD and associates it with the correct classes for its value and status. As it is created before doing any analyze, the status is a safe one.
- CurrentSDFactory: Creates the sensorial device CurrentSD and associates it with the correct classes for its value and status. As it is created before doing any analyze, the status is a safe one.
- FireSDFactory: Creates the sensorial device FireSD and associates it with the correct classes for its value and status. As it is created before doing any analyze, the status is a safe one.
- FloodSDFactory: Creates the sensorial device FloodSD and associates it with the correct classes for its value and status. As it is created before doing any analyze, the status is a safe one.
- HumiditySDFactory: Creates the sensorial device HumiditySD and associates it with the correct classes for its value and status. As it is created before doing any analyze, the status is a safe one.
- LightSDFactory: Creates the sensorial device LightSD and associates it with the correct classes for its value and status. As it is created before doing any analyze, the status is a safe one.
- MovementSDFactory: Creates the sensorial device MovementSD and associates it with the correct classes for its value and status. As it is created before doing any analyze, the status is a safe one.
- NoiseSDFactory: Creates the sensorial device NoiseSD and associates it with the correct classes for its value and status. As it is created before doing any analyze, the status is a safe one.
- ProximitySDFactory: Creates a sensorial device ProximitySD and associates it with the correct classes for its value and status. As it is created before doing any analyze, the status is a safe one.
- SDFactory: Abstract factory to create SensorialDevices and its related structures.
- SmokeSDFactory: Creates the sensorial device SmokeSD and associates it with the correct classes for its value and status. As it is created before doing any analyze, the status is a safe one.
- ThermometerSDFactory: Creates the sensorial device ThermometerSD and associates it with the correct classes for its value and status. As it is created before doing any analyze, the status is a safe one.
- WeightSDFactory: Creates the sensorial device WeightSD and associates it with the correct classes for its value and status. As it is created before doing any analyze, the status is a safe one.

/aalreasoning/reasidence/sensorialDevices/status
- ApplianceStatus: Encapsulate the different status that can have an appliance.
- CurrentSDStatus: This class represents the status for a sensorial device of type: CurrentSD.
- DurationTrend: Encapsulate the different duration trends analized by the SensorialDeviceAnalyzer.
- EnvironmentTrend: Encapsulate the different environment trends analized by the SensorialDeviceAnalyzer.
- FireSmokeSDStatus: This class represents the status for a sensorial device of type: FireSD or SmokeSD.
- FloodSDStatus: This class represents the status for a sensorial device of type:FloodSD.
- HumidityLevel: Encapsulate the different humidity levels recognized by the Sensorial Device Analyzer.
- HumiditySDStatus: This class represents the status for a sensorial device of type: HumiditySD.getMaxTimeUnsafe = -1 indicates is a safe humidity.
- LightLevel: Encapsulate the different light levels recognized by the Sensorial Device Analyzer.
- LightSDStatus: This class represents the status for a sensorial device of type: Light.
- MovementLevel: Encapsulate the different movement levels analized by the SensorialDeviceAnalyzer.
- MovementSDStatus: This class represents the status for a sensorial device of type: MovementSD.
- MovementType: Encapsulate the different movement types analized by the SensorialDeviceAnalyzer.
- NoiseLevel: Encapsulate the different noise levels recognized by the Sensorial Device Analyzer.
- NoiseSDStatus: This class represents the status for a sensorial device of type: NoiseSD.
- PresenceStatus: This class represents the status for a sensorial device of type: ProximitySD or WeightSD.
- SDStatus: abstract class for generic use with sensorial devices.
- SwitchSDStatus: This class represents the status for a sensorial device of type:
  * ContactSD or ProximitySD. getChangeStatus = true indicates that this switch needs to change its status due to safety reasons.
- SwitchStatus: Encapsulate the Switch status recognized by the Sensorial Device Analyzer.
- TemperatureLevel: Encapsulate the different temperature levels recognized by the Sensorial Device Analyzer.
- The temperatureSDStatus: This class represents the status for a sensorial device of type: ThermometerSD.

/aalreasoning/utils

- DataFileMngr: Loads and saves data from Files.
- DataMG: This interface have some basic operations to load and save data from the Storage Layer.
- OntoUtilities: Simplifies the common operations with an ontology model.

C. 5 Crucial name constants

All of them can be found in SystemConfig.java.

    strOntPath: Ontology path
    strOntFile: Ontology file
    strNameSpace: Ontology namespace
    strOntResultFile = Ontology file to save results
    strRulesPath: Path for rules = "files/rules";
    rulesCheckSafetyFile: file for rules.
    strSDStatusPath: Sensorial devices status path
    strSDStatusFile: Sensorial devices status file

C. 6 Variable name conventions

Every variable name starts with some letters which represents its type. This is an example:

<table>
<thead>
<tr>
<th>Type</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>lst</td>
</tr>
<tr>
<td>String</td>
<td>str</td>
</tr>
<tr>
<td>OntModel</td>
<td>ont</td>
</tr>
<tr>
<td>InfModel</td>
<td>inf</td>
</tr>
<tr>
<td>Reasoner</td>
<td>re</td>
</tr>
<tr>
<td>OntClass</td>
<td>oc</td>
</tr>
<tr>
<td>Individual</td>
<td>in</td>
</tr>
</tbody>
</table>