Hogeschool West-Vlaanderen

Master Industriële Wetenschappen: Electronica-ICT

Voorzitter: prof. ir. F. De Pauw

Service georiënteerde architectuur voor multi-sensor surveillance in smart homes

door

Pieter-Jan Huyghe

Promotoren: dr. ir. S. Van Hoecke, dr. ir. D. Vandeursen

Scriptie ingediend tot het behalen van de academische graad van

Master in de Industriële Wetenschappen: elektronica-ICT

optie ICT

Academiejaar 2011–2012

Acknowledgments

First of all I would like to thank Henry Houdmont and Vincent Haerinck, for providing good companionship during research, aiding in the development of the total application and helping where they could.

I would also like to thank Sofie Van Hoecke and Davy Van Deursen for following up on this project, reading this document, sharing their knowledge about computer science and adjusting the project where needed. They have been there when they needed to be and that made a big difference in the end.

Also a thank you towards HOWEST, for funding my research, providing knowledge and giving me a place to work.

Pieter-Jan Huyghe, juni 2012

Toelating tot bruikleen en publicatie op DSpace

Pieter-Jan Huyghe, juni 2012

Service georiënteerde architectuur voor multi-sensor surveillance in smart homes

door

Pieter-Jan Huyghe

Scriptie ingediend tot het behalen van de academische graad van
Master in de Industriële Wetenschappen elektronica-ICT:
optie ICT

Academiejaar 2011–2012

Promotoren: dr. ir. S. Van Hoecke, dr. ir. D. Vandeursen
Hogeschool West-Vlaanderen
Associatie Universiteit Gent

Master Industriële Wetenschappen: Electronica-ICT
Voorzitter: prof. ir. F. De Pauw

Samenvatting

Korte, Nederlandstalige samenvatting van deze scriptie

Trefwoorden

Reasoning, Semantic web, EYE, Euler, EulerSharp, Jena, Machine Learning, Logic, Healthcare.

Inhoudsopgave

Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
CARSA	Context-Aware Reasoning-based Service Agent
DDL	Dynamic Description Logic
EYE	Euler YAP Engine
HTML	HyperText Markup Language
N3	Notation 3
N3-Triple	A structure containing a subject, predicate and an object. Written in N3.
OWL	Web Ontology Language
OWL-S	OWL-Semantic
RDF	Resource Description Framework
REST	Representational State Transfer
REST-desc	Representational State Transfer Description
SOA	Service Oriented Architecture
SVM	Support Vector Machine
SWRL	Semantic Web Rule Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WSDL	Web Service Description Language
XML	Extensible Markup Language

Hoofdstuk 1

Introduction

Enhanced living has been available to the public for over a decade now, usually in the form of domotics applications designed to make living easier and more comfortable. In these applications sensors and remote human input have been used to control or alter the behavior of certain devices in and around the house. In a domotics application a sensor can be used to detect if the plants have enough water, while another sensor monitors the temperature in the living room and controls the heating accordingly. The possibilities are endless, but usually consist of a range of independent, very simple systems designed to each complete only one specific task, therefore none of them apply to the true meaning of a smart home.

The main difference between a domotics application and a smart home is the fact that a smart housing project is not limited to enhancing the comfort of the individual living in that house. A smart home will not use a single sensor to monitor the plants, but will combine the power of multiple sensors to monitor the individual itself. To most people, this would seem like a tremendous waste of money, but for the weak in our society, elderly or sick people, it is a very welcome technology aiding in their welfare and allowing them to live longer independently. As the healthcare industry is a resource-consuming machine that runs on time, knowledge and manpower. These resources all have one thing in common, they are very expensive. It is therefore wise to only use them when necessary and with utmost care. What if it was possible to cut back on personnel costs, use some source of knowledge to use manpower in a better and more efficient way, make a good evaluation of the situation at hand without consuming too much time and do all of that without being needlessly intrusive in a person's private environment? This is where the smart home comes into play. A great role model for the smart home on Belgian soil has always been the “Living Tomorrow” project situated in Vilvoorde, Brussels [1]. Since 1995 F. Beli n and P. Bongers have created multiple ”Houses of the future“ with multiple partners based on the slogan: ”How will we be living in the future“. Every five years a new house is built as the technology evolves too quickly to adapt the house. Since the first version, the project has expanded to contain a shop of the future, a senior flat and so much more.

The basic smart home consists of two primary features. It uses sensors like any other domotics system and it implements logic to make decisions based on the data collected from these sensors. The sensors in a smart home can measure temperature, light, sound level and intensity, detect motion, pressure and many more aspects of the variable inputs that a person encounters in his everyday life. Combining these sensors can make life easier, but also safer. The safety aspect is key in the healthcare industry since a safer environment costs less money on the long run.

Safety is very hard to comprehend because many different aspects of life have an influence on a person’s safety, thus making it hard to detect when a person is safe or not. There are some obvious reasons why a person could be in danger. A fire for instance can be life threatening and can also be detected by a machine in an easy way. The real challenge in providing a safe environment is to not only detect the obvious, but to concentrate on a person’s behavior or more importantly, his lack of normal behavior. Using multiple sensors can be of great value in situations where behavioral detection is key to knowing if a person is safe or not. A machine can compare the values of multiple sensors and form a conclusion based on what it knows about these variables, greatly increasing the certainty when detecting the severity of a given situation. That machine can then notify the correct person to check on the patient’s wellbeing, making it easier to spread the manpower over multiple smart homes thus saving time and money.

The lack of normal behavior could also be signaled to the individual directly, aiding him with some of his day-to-day tasks. For instance brushing teeth consists of taking a toothbrush, using toothpaste, brushing and so on. The system can follow the elder in those tasks and remind them if they miss a step or do something in the wrong order. A good example of a system capable of doing such a thing is the system researched by Jaeyong Sung et al [2]. For a great overview of other examples and reviews of multiple technologies used in smart homes, the paper written by Marie Chan [3] is the one to read. It features many technologies, explains the state-of-the-art of them and also features a list of relevant papers about these subjects.

The basic smart home still has many issues, indicating that a basic package that works out of the box is a myth and that almost every smart home has to be tailor made. The first problem is the fact that no two houses are ever the same, thus making it hard to have a basic package of sensors and services to implement. Programming everything is time-consuming and makes a smart home very expensive. One of the biggest hassles is the adding of sensors to the network and allowing them to communicate with all the others. Therefore, this thesis will address this issue, making sure that a failsafe mechanism is implemented in case communication would temporarily be impaired. Both classical sensors, such as pressure, light, temperature, etc, will be supported, as well as sensors capable of sensing multiple variables: cameras. These are the most advanced sensors known to date, since they can sense motion, light intensity, colors and many more. When collecting data from all these sensors, the next step is the reasoning on this data, automating this as much as possible with a high degree of certainty in as many situations as possible. In order to do so, this subject is divided in three pillars studied by different researchers.

- Sensor Communication

  First of all Haerinck [4] focusses his research on the service oriented architecture. His research includes the State of the Art of the different protocols for web service communication (e.g. REST, REST-desc and SOAP) and the hardware needed to set up a system like this. He configures the base of the architecture for the dynamic discovery of new sensors in the house and the communication with the gateway.

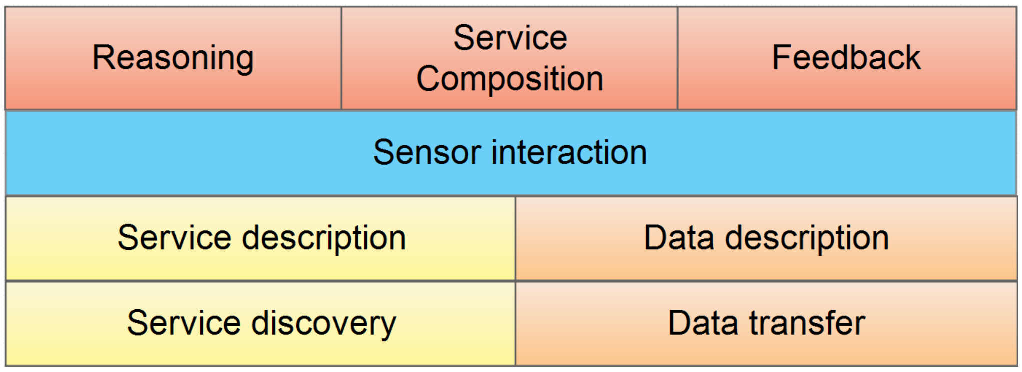
- Reasoning

  Reasoning is the second pillar and can be found in this paper. This study explores the possibilities of a reasoning module which is the artificial brain of the system. It also focuses on the best ways to analyze data delivered by the different sensors in the house. Based on the analysis, feedback is generated to tell the sensors if the data was correct. This feedback can be used to reconfigure the sensors in case of failures or corrupt data.

- Vision-based surveillance

  This is the last pillar of the research, which can be found in the paper by Houdmont [5]. It studies the possibilities of vision based surveillance in smart homes. The module keeps an eye on the general movements of a person in the room and analyzes specific situations. The analysis of images and thus surveillance as well in favorable as in bad lighting conditions is important as the lighting conditions in a house aren't always as good. Person tracking gives the reasoning module information about the position of the person in the room, their pose (e.g. standing, sitting, lying down) and the hotspots where the person often stops. Face and eye detection helps for detecting the state of the person sitting in the couch.

Figure 1.1 provides an overview of the high level sensor architecture. The "Sensor interaction" module is the gateway through which all communication between the different modules is processed. The "Reasoning" module gets its sensor data for analysis through this module and returns information through this module to the feedback block. The feedback serves the sensors with data about the correctness of the information delivered to the reasoning module. This can be used to reconfigure the sensors if necessary. The service composition serves as an intelligent module combining different services with the same semantic values to a bigger, more intelligent service. When a sensor transfers data as described by the service and data description, The service composition will guide it through the consecutive services for optimizing the usefulness of the data.



Figuur 1.1: High level architecture

Hoofdstuk 2

Requirements

The basic smart home consists of many sensors, the challenge is to hide these from the patient so that the system is not intrusive in his private environment. Due to the fact that sensors need to be hidden, not all sensors can be used to their full potential or placed where you would want to place them. The system also needs to be cost-efficient, so the sensors should not be expensive at all. The downside to this is that sometimes a shortage in data will occur. One of the challenges will be to build a reasoning module that can cope with the lack of data, by filling in the gaps combining multiple sensors at once to still get the result needed. The type of reasoning used must not be prone to errors, due to the environment where the system will be used. Errors could have very big consequences for the patient and he is always the number one concern.

Combining multiple values is an intensive task and requires rules and a reasoner to implement this and get a fairly decent result. Due to the nature of the task, speed will be one of the issues. The more time the system consumes in generating a conclusion, the less responsive the system will be and that is never what you want. Another downside to a longer execution time is the fact that it will consume more power, produce more heat and be tougher on the batteries when power from the grid is not possible. So the system does need to be fast but lightweight to run.

The system will need to be as flexible as possible, new sensors need to be added with ease and rules need to be easy to write and maintain by someone without programming knowledge.

Hoofdstuk 3

State of the art and related work

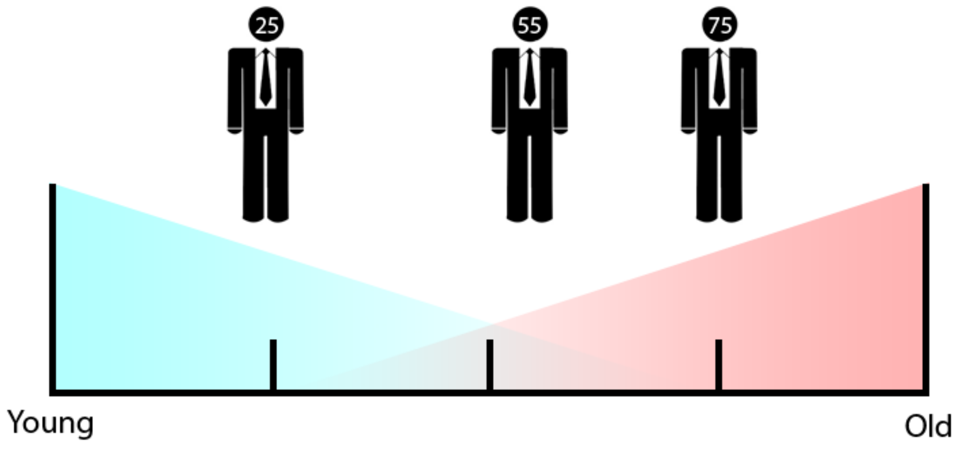
The biggest part of this thesis will be the implementation of a reasoning engine. The first question is, what type to use. In the world of reasoning many types of techniques exist who each have their advantages and in the end do roughly the same thing, but work in a completely different way. This study will investigate the biggest differences, sum up the advantages and disadvantages and form a conclusion based on the needs of the application. One of the big differences that will be examined is the difference between predefined logic and machine learning based reaoners. These are the two big classifications for reasoners.

3.1 Predefined Logic

Predefined logic is one classification of reasoners. A reasoner is called predefined if it follows a set of hardcoded rules or a certain pattern to come to a conclusion about data. These systems in their most basic form do not learn from the past.

3.1.1 Fuzzy logic

Making a decision, one of the easiest things in the world for any human. Humans do it every day, mostly without even noticing it, humans have a sense for what is the right decision without needing to think about most of them. This type of behavior does not translate to machines very easily. When a human makes a decision, he evaluates the situation as good as he possibly can by giving priorities to certain variables and mixing those with common logic. Machines can't do this very well, they do not intuitively give a weight to a variable, they simply compare it to a certain value they already know and take according action at a certain outcome of the comparison. As Zadeh [9] [10] states, the variables used in a machine can have the same meaning, but can be different in value. This means that a certain label is placed over a collection of variables because they are all similar in type and value, linking them to each other. This helps a machine tolerate imperfections in data, making it possible to still use the data, even though it is not exactly what the system is looking for. Dealing with imperfections is the first step to making a system think like a human, because it can relate data to facts without setting a hard boundary, this improves flexibility and robustness.



Figuur 3.1: Age represented by fuzzy logic.

Figure 3.1 illustrates this by describing a person's age. A person can be young or old, the words young and old give meaning to a variable, but the words themselves have no hard boundaries. Humans interpret the words young and old and relate them to an age, which is a number and therefore comparable to a different number. Nobody will deny the fact that a twenty-five-year-old is still very young and that a seventy-five-year-old is getting old, but what about a person is his mid-fifties? Some will find that this person is getting older, while others will state that he still has many days ahead of him. The fact that it is possible to find someone a little old but still very young is a problem for a machine. There seems to be a cross-over zone, as shown in Figure 3.1. This way of thinking isn't binary, fuzzy logic can offer a solution to this problem.

Fuzzy logic is designed to utilize the space between 0 and 1, introducing a degree of truth that allows a system to make a decision based on a variable amount of certainty. A person can now be 0.4 young and 0.6 old, making him relatively older than he is younger. That allows the system to take according action, without being completely sure of the situation at hand. In some cases this can be a big advantage since the system can now react in situations where the system is not completely certain.

In the healthcare industry, using fuzzy logic is not common practice since the matter of life or death is not something to judge based on an educated guess. Fuzzy logic is a great technique allowing quick conclusions in a complicated environment, yet does not apply to a real healthcare situation due to this downside. As a result, the technology is therefore not applicable on its own to this project. It is however possible to build a reasoner that is assisted by fuzzy logic as a check on the data it produces, giving a higher hit ratio in correct solutions.

3.1.2 Forward chaining

The forward chaining technique is one of the most commonly used techniques in small Artificial Intelligence (AI) projects. It combines logical rules programmed by a human or extracted from a knowledge network [18] and a set of data, generated by one or multiple machines. The forward chaining algorithm is a very powerful tool when used in smaller subsets of related data. It allows the user to funnel its data through a system to get to, in a best-case scenario, one solution with a high level of certainty. If the rules are chosen wisely, it is possible to get the desired degree of certainty in a fast and cost effective way, thus effecting battery life in a positive manner in mobile applications which is important for this project. Forward chaining looks at the facts, to form a conclusion. This conclusion can be considered as a fact as soon as the degree of certainty is high enough, making it possible to generate new facts and make the general situation easier to comprehend by providing a more complete answer.

To be able to use the chaining technique, data must have context to fill in the rules. This seems like a huge overhead of network traffic, but in this day and age with high bandwidth networks at an acceptable cost, this is not an issue anymore. The reasoner used in the CARSA project [6] uses a form of chaining where it first filters out the unnecessary data through reasoning, enabling the system to perform faster and generate more accurate conclusions on the long run.

As Yung-Chien Sun and O. Grant clark [7] stated in their research, a rule based engine is a time consuming approach to reasoning. The design of a better system using super rules was introduced. The system looked for frequently used rules, giving them a higher weight than others, thus cutting back on processing time by predicting what rule would be correct in case certain pieces of data where available. The combination of iterating through super rules and base rules proved to be a big gain in performance and also aided in getting good results from the reasoner.

A prime example of the forward chaining technique combined with OWL 2 data binding is the DLEJena reasoner [8]. This technique runs on the Jena Framework, making it robust and easy to setup. The DLEJena reasoner uses OWL 2 to its advantage by binding variables to the rules it knows. It does this by matching data and rules based on previous usage. After a few iterations the system will learn that certain variables are never used in certain rules, thus ignoring them later on, cutting back on processing time and memory usage. Jena uses RuleML as a syntax to interpret rules. RuleML has been around for a while and is limited in use due to some design flaws. The developers however have been working to fix these issues and all is looking good for the future. A W3C member submission called SWRL syntax [12] partially fixes all the shortcomings of the original RuleML but is not really usable yet, it does however look very promising.

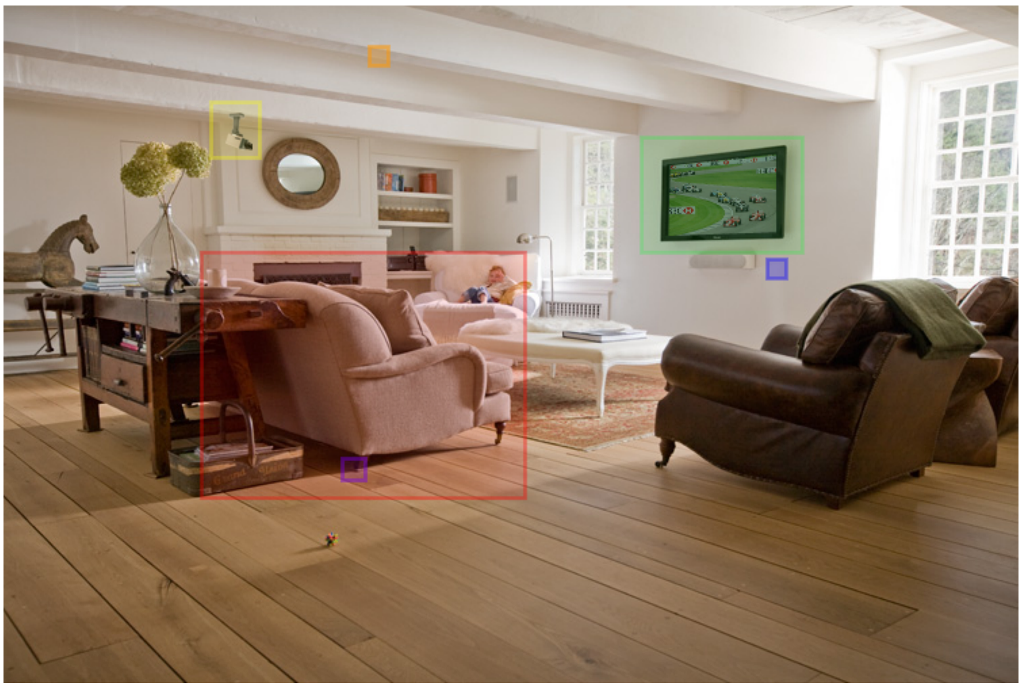
3.1.3 Backward chaining

Backward chaining is similar to the forward chaining method as it is able to use the same set of rules and data to come to a conclusion. However the conclusion using the backward chaining algorithm is not one single answer to your question, the answer you get from the machine is an entire subset of data extracted from the rules. In the backward chaining algorithm you start from an activity, not the dataset itself. The activity is then related to certain rules that prescribe that activity, making it possible to extract data about the activity. For example when a person is watching the television from his sofa, the rules state that the light sensor near the television should be fluctuating and the person should be sitting down, thus activating the pressure sensor in his seat. If the system is certain the person is watching the television but the light sensor is not fluctuating, this could mean that the sensor is malfunctioning meaning that error detection has now become a lot easier. When a sensor is told it should behave in a certain way, it knows that its values are off compared to what the sensor is registering, triggering a diagnostics report on that sensor and possibly notifying a technician to have a look at the problem. A system is also able to learn, when it detects that a certain sensor has been outputting the same data over and over, but the rules state that the data is wrong, the system can learn from that data and conclude that the rule could be faulty.

Besides using backward chaining for error detection, another way to use it is to check if all the data derived from the given situation is compliant with the data from the sensors. If the data is compliant, the degree of certainty rises once again to a level where it is safe to say that the detection is working as it should and the individual is performing the earlier detected activity. No contradictions in the data can also be a significant sign that the rules are well thought out and working as they should.

3.1.4 Chaining in a real world example

In the example of Figure 3.2 the forward and backward chaining technique is used to determine what the person is doing with an acceptable degree of certainty.



Figuur 3.2: Situation overview: Red - Hotspot A, Green - Hotspot C, Yellow - Camera, Orange - Ambient light sensor, Blue - TV light sensor, Purple - Pressure sensor

Figure 3.2 is an example of a classic living room filled with a television, a sofa and some windows. The behaviour of the person in this room can be tracked, as wel as the status of some of the appliances, due to the use of multiple simple sensors. In this particular room, four sensors monitor each and every move. Assume that the system has been turned on and that data collection is working as it should. To determine what a person is doing at any given time, the system must first read the sensor data and then use that data to come to a conclusion.

Sensor data

The sensor data is one of the most important parts of the project, gathering it without a big delay and making sure it is as accurate as possible is key. In this situation, the following data is collected:

- Ambient light sensor: high value, indicating that the room is well-lit
- Television light sensor: fluctuating value
- Seat pressure sensor: high value, indicating that an object is in the sofa
- Camera: X/Y position, in this case near hotspot A as last known position

Rules

The beating heart of the system is the rule engine. The rule engine itself is fed with the sensor data, that data is funneled through the rules until a conclusion is formed. The rules integrated for this example are:

1. Rules describing sensor values:
   (a) The sofa is situated in hotspot A, set of X/Y coordinates
   (b) The bed is situated in hotspot B, set of X/Y coordinates
   (c) The television is situated in hotspot C, set of X/Y coordinates
   (d) The camera's motion detection is reliable when there is enough ambient light
   (e) The camera can detect fluctuations in hotspot C when the ambient light is below a certain level
2. Rules describing behavior of the individual or an appliance:
   (a) A person is watching the television when he is seated and the television is on.

(b) A person is sleeping when the lights in the room are off and he has not been moving for a longer period of time near hotspot B

(c) A person is in need when he has been laying down in a non-logical place.

(d) The television is turned on when the televisions lightsensor is fluctuating or when the camera detects fluctuations in light intensity in hotspot C.

(e) The camera can detect fluctuations in hotspot C when the ambient light is below a certain level.

(f) A person is sitting down when the seat is detecting a lot of pressure or if the camera reliably detects that the person has stopped moving in hotspot A

Analysis

For the initial analysis, forward chaining will be used. Starting with the first sensor, the ambient light sensor has been transmitting a high value, the first set of rules describes what this sensor data means.

The system now knows that:

- The camera's motion detection is reliable, as stated by rule 1d
- The camera cannot detect if there are fluctuations in hotspot C, as stated by rule 1e

The system now knows more about the capabilities of the camera. The camera has detected the last known position of the individual near hotspot A, the system can now give a high weight to this value because it knows the camera is reliable.

Given that the system now has information about the person's last known position, it can search in the behavior rules if it finds that position in a rule, indicating that it is relevant to the information it has. Rule 2f states that the person is sitting down when he last moved in hotspot A. The rule also states that the person is sitting down when the seat's pressure sensor detects a high load. The next logical step is to verify if it is, giving a higher degree of certainty in case both conditions are met. The pressure sensor in this case is indicating a high load, making it very likely that the person is in fact sitting down.

The system now knows that a person is sitting down with a high degree of certainty, searching the rules for this behavior leads it to rule 2a and 2f. Since the systems knows it got this information from rule 2f, this rule is discarded from the list of possibilities. Rule 2a is now the one to investigate. Knowing that half of the rule is already filled in correctly, the system will try to use other rules and data from the sensors to conclude if rule 2f can be interpreted as true. If it is considered as true, the system knows that the individual is watching the television, which in this case is the correct outcome of the data.

The system now knows that the individual is watching the television.

The fact that a rule has an 'or' condition in it allows the system to detect an action with a higher degree of certainty if both conditions are met, it can also correct information it may or may not have received from a sensor, thus allowing the system to self-learn in some cases. Self-learning can be achieved by relaying data back to a sensor and allowing it to set the threshold in a different way or increasing sensitivity after detecting multiple „faults" in the data. Rules with an „and" statement are very dangerous as they affect the reliability in a bad way because contradictions in one rule can occur in case of sensor failure, making the system unable to conclude anything. They are however very good to be able to achieve a very high certainty, correct false information and avoid wrong conclusions. Rules with an „and" condition should not be avoided, but used with utmost care and intelligence.

Fault correction using backward chaining

After the determination of the person's activity, it is possible to check for faults in the data and possibly correct them. A fault is not always a malfunctioning sensor, it could for instance just be a miscalibration, a bad rule or a misjudged threshold. A contradiction in a rule cannot be corrected through code, the rule in question can be avoided if detected it is faulty, but generating a new one is a tricky and dangerous task, best left to the operator of the system.

Starting from a given activity, such as watching television, certain conditions need to be fulfilled.

The method to accomplish this is the exact opposite of the way the system detected the activity. There are multiple ways to describe an action, therefore there are multiple ways a system can come to the same conclusion. This gives the system the opportunity to examine an action based on the rules it has. The aim of doing this is to take a different route to the same solution in the reverse order, making it possible to get estimated values from the rules and to compare those to the actual values from the sensors, allowing it to self-learn and evaluate its solutions and conclusions.

In this case, the person is watching the television. The system will search for the keyword television and find all the rules that have a connection to it. Rules 1c, 2a and 2d apply:

- 1c: The television is located in hotspot C
- 2a: A person is watching the television when he is seated and the television is on
- 2d: The television is turned on when the televisions light sensor is fluctuating or when the camera detects fluctuations in light intensity in hotspot C

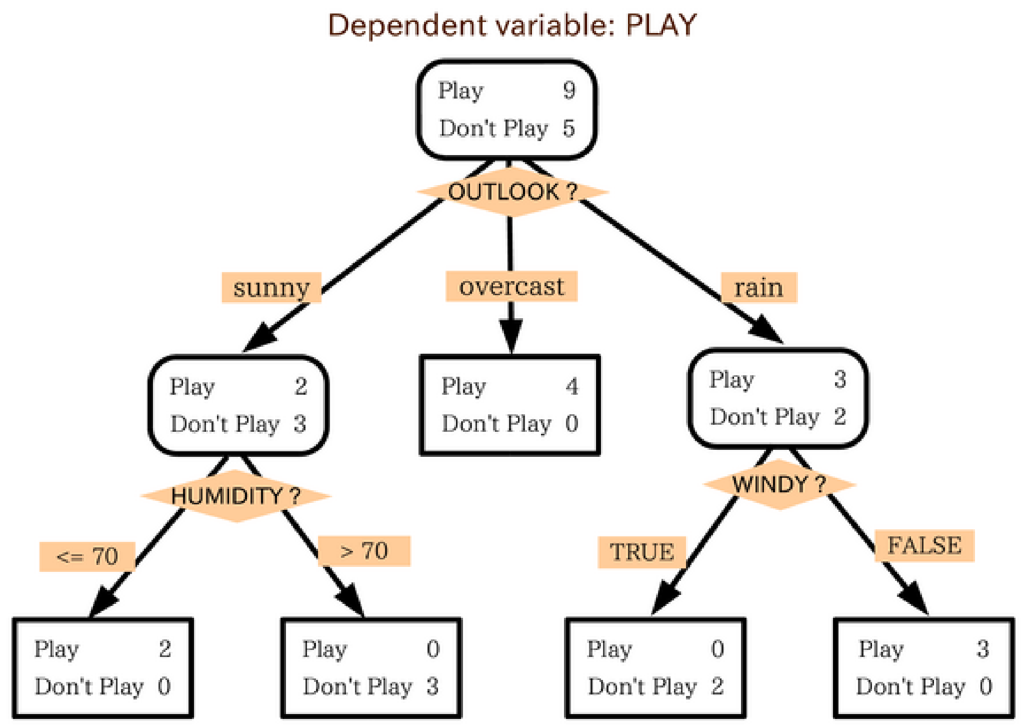
From these rules it is possible to extract conditions that can be verified with the available sensor data, if you consider that the facts are correct about the television being watched (and is therefore turned on). These conditions are:

- Hotspot C has a fluctuating light intensity
- Television light sensor is fluctuating
- The person is seated

This extra knowledge can then be tested. The camera can try to detect fluctuating light in hotspot C, if there is, this can be a sign that the television is indeed turned on, a first acknowledgement of the fact that the person is watching TV. If there is no fluctuating light, this can be a sign that the camera cannot detect the fluctuations. The system now knows that the camera could be malfunctioning if it does not find anything in the rules about detecting light in hotspot C. In this example the system will find the rule that states that the camera cannot detect light in hotspot C if there is too much ambient light, thus ignoring the warning about the malfunction. If all the extracted conditions are tested and all of them pass, the level of certainty is very high, making the system robust and accurate in most situations. Again, the main conclusion to remember is the fact that the system is only as good as the rules it is given to work with. A chaining system's setup is initially very complex due to the rules, but once working properly it can become a very powerful and reliable decision making tool.

3.1.5 Decision tree

When visualized in a graphical way, the decision tree [11] and chaining algorithms resemble each other in multiple aspects. They both use rules to some extent, work their way through the data in a sequential way and correct branching is key for both of the systems. The biggest difference between the two is that chaining uses hard facts as a rule, where the decision tree is using data that is not always necessarily 100% correct.



Figuur 3.3: Simple example of a decision tree for evaluating if a game should be played or not

The decision tree tries to give a weight to the outcome of a branch, making it possible to factor in some sort of tolerance in the data. This tolerance is good in smaller applications, but in more complex situations it will make the system unreliable, unpredictable or even indecisive. The biggest issue is that multiple answers can be seen as true, due to the tolerance that accumulates in every branch. This makes decision trees very valuable if choices have to be made between options that can all be the right one, not in situations where sorting between right and wrong is the question at hand.

In order to be able to determine an action, the decision tree will be very large and complex, especially in a smart home application, due to the already complex nature of human behavior and the amount of actions any person can take part in. One solution to this problem is the use of classification trees, thus sorting the data into classes by relevance to one certain subject. Effectively programming a system like this is very labor intensive and costs lots of system recourses at runtime. The biggest downside of this method is that the work is not yet done when the classification is complete. Once classifications are made, a regression tree is still needed to determine what action the subject is taking part in at that moment. The system needs to make multiple iterations on the data and it will be hard to be absolutely sure of the outcome. As a result, it is not a good method for smart home applications where a high density of data needs to be examined and errors could be very costly as described in the requirements.

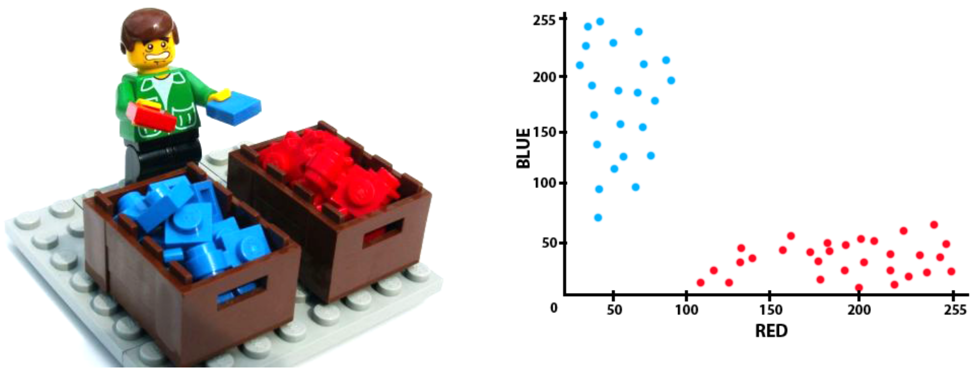
3.2 Machine Learning

Although predefined logic is a very powerful tool, it does have its shortcomings. The logic is predefined, this means that a system does not react very well to a new situation, one that it is not programmed to handle. Another downside is the labor intensity for the programmer because a person can do many things, so many things also need to be covered by the code. Designing such a program could be very expensive when the application is very complex. Machine learning is a partial solution to both these problems.

Humans tend to be influenced by previous events when making a decision. An accountant wearing a silk suit for instance will experience that leaning against a freshly painted wall is not the best idea he has ever had, because of the consequences. The accountant will realize that a suit and wet paint do not mix very well, avoiding the same mistake later on. This is called learning, something machines can also do by creating a profile for a certain event and coupling variables to it or describing an event in characteristics. This can be done in various ways, some more successful than others.

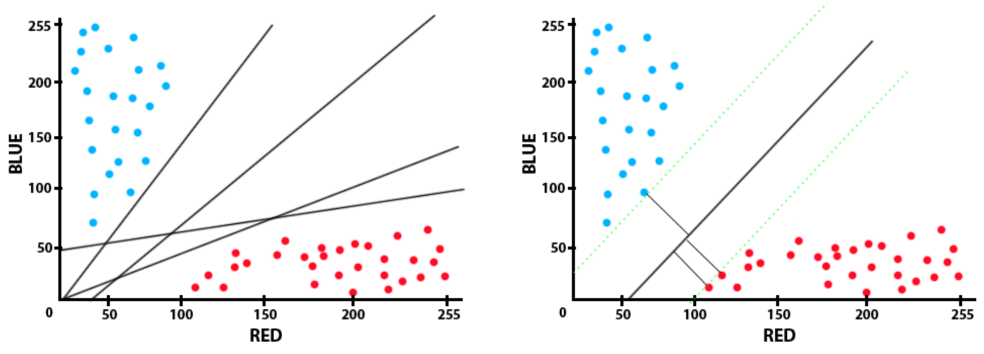
3.2.1 Support Vector Machine

The Support Vector Machine (SVM) is basically an algorithm that compares data to a benchmark, looks for similarities and places that data under a certain classifier or label. An SVM can be trained by sampling through sets of training data, this way forming a profile of what defines an object or action. Every object can be identified based on basic data about it. For instance when an SVM is trained to sort blue and red blocks of Lego (see figure 3.4), the training data could be a bunch of pictures of different blocks and their matching color. The SVM will detect that there are two big differences in color, followed by lots of different shades of each color due to lighting differences in each photograph. The different shades will be mapped out based on their values as seen in figure 3.4.



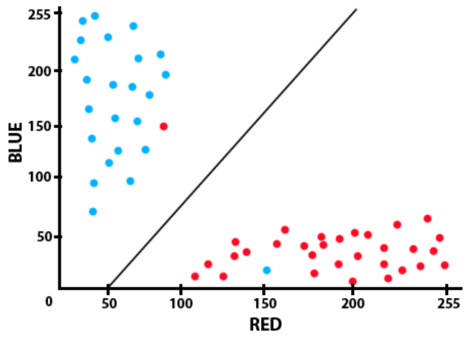
Figuur 3.4: Mapping out red and blue, based on shades

Humans are very good at pattern recognition, even in the shortest amount of time a human will recognize the fact that two clusters of data have been formed. These two clusters do not seem to overlap, making it possible to separate these clusters by a single line. When a new block needs to be color sorted, it can now be stated that if the color value falls above the line, the block is blue or if it falls below, the block should be red. This technique also works in a three dimensional environment. It is the first step in SVM reasoning and is called separation via a hyperplane. The biggest problem in all this is choosing the correct separation line, because multiple ones exist as shown by figure 3.5 (left). When a separation line is not chosen correctly, the SVM could be prone to errors under certain circumstances. A good solution to this problem is the maximum-margin hyperplane as explained by Muller et al. [15] and further explained by Byun et al. [16].



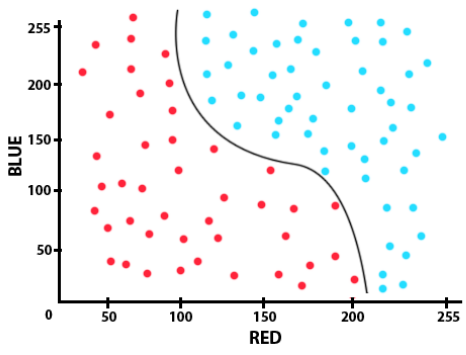
Figuur 3.5: Left: multiple hyperplanes can be found. Right: Maximum-margin hyperplane

The maximum-margin hyperplane is the hyperplane with the biggest possible distance between the points of data and the plane itself. The distance is measured perpendicularly from the data point or points closest to the possible hyperplane as seen in figure 3.5 (right), this for both data clusters. Creating the biggest possible margin ensures the best accuracy of the SVM when new data needs to be analyzed. The SVM is now ready to use in its most basic form, yet could use some more optimization as explained in the next paragraph.



Figuur 3.6: Allowing errors via the soft margin

Suggesting that data sets can always be separated by one single straight hyperplane is a bold claim and a limitation to the system. Most real world examples will feature datasets that cannot be separated so easily, due to data pollution, incompleteness or just simply very complex situations. If polluted data would be entered into the system it could cause the hyperplane to shift position, corrupting the SVM for further use. Allowing a small amount of „errors” prevents this from happening. The soft margin lets a few data points cross the hyperplane between the members of the opposite cluster as seen in figure 3.6. Allowing certain „errors” is dangerous and could corrupt the data set, highlighting the necessity of a control function that specifies how many points can get to the other side of the plane and by how far they are allowed to go.



Figuur 3.7: Complex separation using a kernel function.

The fourth and final step in the SVM building process is designing a kernel function for data that simply cannot be divided by a single line. A kernel function is a mathematical function that adds additional dimensions to a set of data. If the kernel function is functioning as it should, the data will be linearly separable in a higher dimension. Understanding kernel functions is a very complex matter since we cannot always visualize an n-dimensional space, but it is possible to project this line down to the original two-dimensional space. This projected separation line will be curved, allowing for separation in a more complex set of data as seen in figure 3.7.

The biggest downside to the basic SVM is its inability to cope with multiple classifiers at once. As stated by Hsu et al. [17], building a multiclass SVM is still an ongoing research topic. One could simply state that multiple SVMs can work together, which is true, yet inefficient. In smaller projects, a good approach is to build a one versus all system. For instance answering the question if a product is yellow, red or blue can be done by building multiple small SVMs that answer the questions: „Is the product yellow?”, „Is the product blue?” and „Is the product red?”. If trained correctly, one of the SVMs will be able to classify the product thus solving the question. Usually doubling the amount of SVMs tends to quadruple the amount of processing time needed due to the creation of overhead and extra data points needed to do the reasoning on. In smaller projects this is not an issue, but it could become the main performance limiting factor in the more complex situations.

3.2.2 Case-based reasoning

Humans adapt to situations hundreds of times a day. They assess situations by recognizing certain parameters that define a situation they have experienced before and then react to the situation at hand based on previous knowledge. A system can learn to do this as well, by breaking the situation into comprehendible and measurable parameters and thus creating a profile for a certain situation. But what if not all parameters are met?

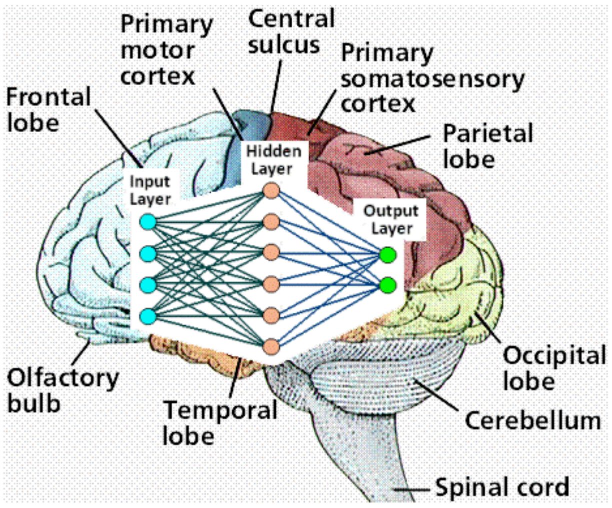
Case-based reasoning [13] is a variation on normal machine learning reasoning techniques. Case-based reasoning is based on what the system has learned from previous events, but now adapts to slight differences in situations where a non-flexible reasoner would just become indecisive. The ability to relate situations to each other and act accordingly depends on the experience the system has and is one of the biggest key success factors to making case-based reasoning work. Experience however is very relative, a machine can be very experienced when it comes to detecting if a television is turned on, but when it suddenly needs to detect if there is a fire in the room, all of its information becomes useless. If this type of reasoner is used for small projects it is one of the best [13], considering it will be able to recognize situations and adapt to them very quickly if the data density in the reasoners „brain" is sufficient for the problem at hand. The ability to separate actions from one another and evaluate the changes between these situations is also of utmost importance because the reasoner would otherwise spoil its data with false information. Getting the facts straight is one of the key challenges in building a case-based reasoner, just like in any other learning system.

The best way to explain case-based reasoning is through a real world example. If someone spills a glass of Coca-Cola on the carpet, cleaning it up and treating the stains is the action that needs to be performed. From previous events a person will know how to do this, but what happens if someone spills some Sprite? The case remains the same, there is some spillage on the carpet and it needs to be cleaned, this can be detected by a human or machine due to pattern recognition. The variable in this case, is the soda itself, but the action afterwards will remain the same. The person will know what to do in case of Coca-Cola spillage, Sprite will fit this case because of its characteristics. The person will then try to clean it in the same way, although the case is not 100% the same.

The biggest challenge to implement Case-based reasoning into a smart home will be the amount of situations a person can come across. The machine will need time to learn from situations that occur, this could take a very long time. Due to the complex nature of some tasks, the system will need various examples with the same characteristics before it will be able to make a generalization of a task. No human works in a sequential way, so many parameters will differ from time to time, making case-based reasoning hard to implement in a smart home environment.

3.2.3 Neural network

The Von Neumann-based machines have been dominating the computer industry for years and will continue to do so for many years to come. However they are not the most ideal platform to run software that is capable of thinking for itself. They feature fast arithmetic's, work sequentially and follow the path that the programmer has embedded into the system, but they are not good at adapting to situations, intelligent fault tolerance or parallelism on a large scale. This limits the potential of some machines and places a bottleneck in the innovation of computer systems as Hertz et al. [19] and Jain et al. [20] state. The best solution to these problems up till now is the concept of neural networks. Neural networks can help in situations where vast amounts of data need to be structured and processed, where an algorithmic solution is not easy or possible and where lots of solutions to a problem are possible. Neural networks basically consist of an input layer which contains input nodes, collecting the data that needs to be processed. The second step in the process is the hidden layer, a layer that is never accessed by the public and does all the processing. The hidden layer has many individual processing nodes, connected to the input and output layer nodes. The hidden layer's nodes work individually, making the system faster by creating massive parallelism. After the processing is done, the hidden layer outputs its solution to the according node on the output layer.



Figuur 3.8: Representation of a neural network and its nodes.

Neural networks are based on the parallel architectures of animal brains, they use simple processing elements with a high degree of interconnectivity between them. These elements can adapt their interaction with each other in order to work more efficiently in known situations, without creating high processing latencies for lesser known situations. This means that you have a system that can alter its behavior in a reasonable amount of time, without the trade-off of wasting processing power on the structural maintenance of the network itself. Again this is an indication of massive parallelism possibilities, like in an animal brain. One of the first examples of this architecture is the „Half a mouse brain" project [25], an ambitious project mimicking half of a mouse's brain which consisted of over 8000 neurons on IBM's BlueGene super computer. The project got a respectable amount of media attention and showed the world that sequential systems may be coming to an end.

The project started its life as just a proof of concept, but as interest grew, some actual tests where ran, some ending in success. The only real conclusion to this project was that the world was not ready yet, because of the vast amount of resources needed to run neural networks on such a large scale. A small scale project may seem very reasonable to run in this day and age, but can still require more resources than expected due to the fact that the network must be given the opportunity to learn in the beginning, adapt in real-time and form conclusions all on the same machine. The network itself will never stop learning and continues to evaluate and adapt at runtime, this creates overhead even for simple tasks where a normal sequential machine could be much faster.

Due to the complex nature of a neural network and the processing power that is needed in these complex environments, it would be very expensive to use this technology in a smart home. Every home would have to be equipped with a server room making this type of system very power consuming and intrusive to people who are not familiar with technology. The second issue with an application in a smart home is the amount of learning that is needed to be done before the system is able to function. No two people are the same, so their behavior will differ in most situations, making it hard to install a ready to use system. Due to the amount of different actions a person is able to do, the system would need to much time to learn. A solution to these problems could be to build a system that only does basic things, making it less needy of processing power and learning time, but then the advantages of a neural network would not be used at their full potential, making the installation needlessly complex for the tasks it would be performing. Probabilistic neural networks [14] can train easier and faster, but need more resources to make their generalizations, so they do not apply to the smart home due to infrastructural problems.

3.3 Conclusion

There is no reasoning method that will suit every situation, choosing the right one is a complex task that depends on many factors of the project at hand. Hybrid systems that combine multiple reasoning methods are a good compromise, but tend to use up a lot of resources and are often very complex to set up. Table **??** gives an overview of the strengths and weaknesses of the different techniques reviewed in this study. As stated in the requirements, a reliable system is one of the most important features a system should have. Therefore, a system must also be decisive at all times.

	Setup Complexity	Reliability	Resource usage	Decisive
Fuzzy logic	Moderate	Moderate	Moderate	Yes
Chaining	Low	High	Moderate	Yes
Decision tree	Low	Moderate	Low	No
SVM	Moderate	Moderate	High	Yes
Case-based	Moderate	Moderate	Moderate	Yes
Neural network	High	High	High	Yes

Tabel 3.1: Comparison between reasoning methods

Human behavior differs from person to person, making it hard to implement a pre-learnt system into a smart home. Training a machine like the case-based system or neural network, would take too long and could cause an inconvenience to the user. A pure machine learning based system is not ideal for this application, because the application itself is too complex for each of the systems to adapt effectively over a short period of time.

Another issue with smart homes is the fact that they are a medical application. In medical applications, mistakes can be very costly with the highest price being the life of a human itself. Making sure the reasoner makes the right conclusions as much as possible is the biggest hurdle in the development of a smart home, adding sensor or network failure to the equation makes it virtually impossible to build a system that is right all the time. This illustrates the importance of building a reasoner with built in mechanisms to make it fail-safe, even under the toughest of conditions. At the time of writing, the system with the highest degree of certainty is a rule engine that combines both chaining methods to come to a conclusion and then check that conclusion with values it has as a benchmark. This makes it possible to detect a failing sensor, thus aiding the user to get a better experience in the end. The chaining method could however benefit from a joint-venture with a fuzzy logic machine and a neural network to extract rules [18] from the data it has processed, but this would cause very high resource usage, without a big increase in reliability.

Hoofdstuk 4

Application design and further research

4.1 Types of reasoners

As noted in Chapter 3, choosing the correct type of reasoner is essential. Research has shown that the chaining-based reasoners suit the smart home application best for various reasons, yet not every reasoner of the chaining type is the same. During the first phase of research, two chaining-based reasoners did attract some attention to themselves, these will be looked at further.

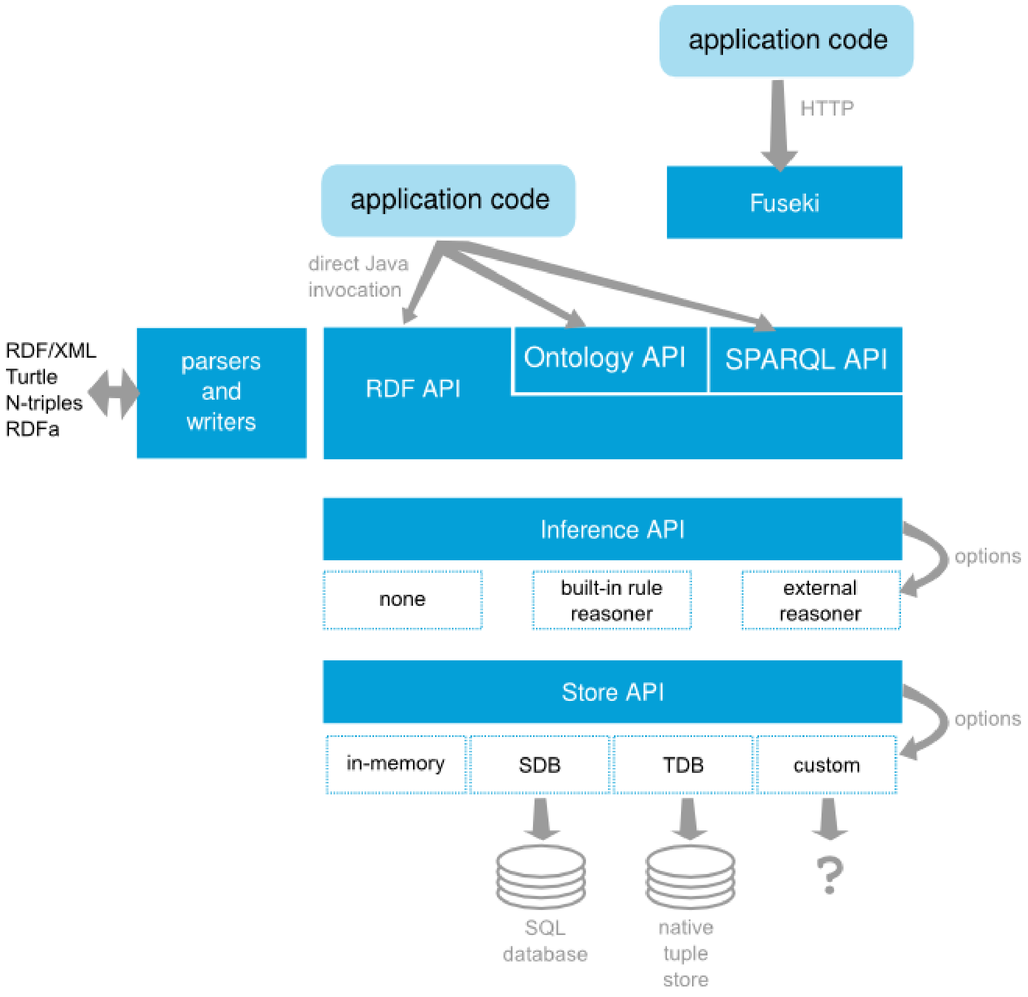
4.1.1 Jena

Jena [22] is, at the time of writing, almost twelve years old. Development began in the early part of the year 2000, under the wings of Hewlett-Packard's development lab. It soon gained a lot of interest among technical experts and leisure programmers. Due to Jena being an open source initiative, many developers were given an opportunity to experiment with semantic content and reasoning. This caused a boost in growth for the project as its user base became larger every day. In late 2010, HP's team decided to try and get Jena to be adopted by the Apache Software Foundation. Due to the fact that over the course of ten years Jena had become increasingly popular, they succeeded.

Jena has now grown to become a powerful Java framework for building semantic web applications. The Jena API supports:

- An API for reading, processing and writing RDF data in XML, N-triples and Turtle formats
- An ontology API for handling OWL and RDFS ontology's
- A rule-based inference engine for reasoning with RDF and OWL data sources
- Stores to allow large numbers of RDF triples to be efficiently stored on disk
- A query engine compliant with the latest SPARQL specification
- Servers to allow RDF data to be published to other applications using a variety of protocols, including SPARQL

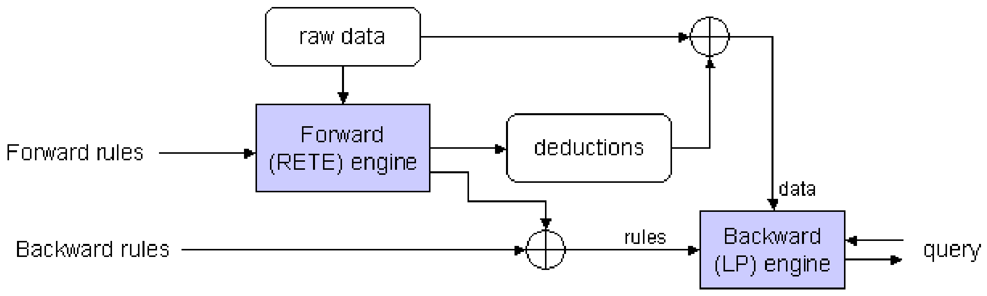
Jena has been under development for quite some time, and the result is an extensive API with a lot of built in functionality as listed above. The Jena framework consists of a number of smaller pieces that together form a large and usable tool. As shown in figure 4.1, Jena consists of three big sections: an RDF API, the inference API boasting the internal rule-based reasoner and the store API which makes it possible to store the outcome of the reasoner in a number of ways. In this project, the inference API will be the most important one.



Figuur 4.1: Graphical representation of the Jena framework.

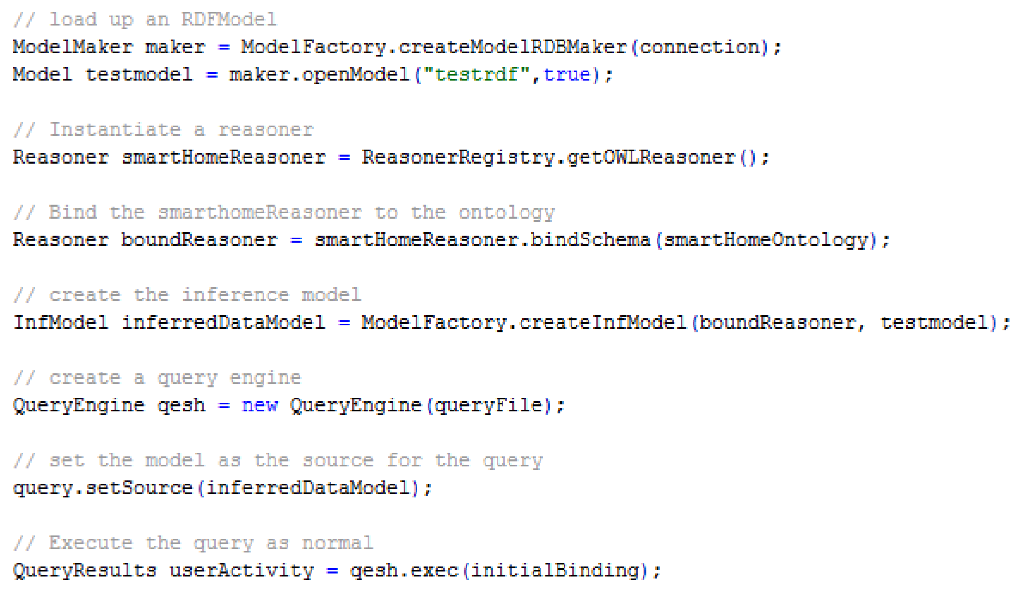
The RDF API is where it all begins. This piece of the code parses your data and rule files into a usable format. The RDF API allows you to add data in a variety of formats and supports the adding or removing of data in the collection. Data can be imported from files or URL's, thus creating a really usable machine that can easily gather data from across the globe if it has to. This makes Jena a flexible cloud application when it needs to be. Jena also supports RDFS and OWL, these are ontology languages that further implement data in a correct way and add support for linked-data applications, which are a necessity for every serious semantic web application.

Once the RDF API is done loading and parsing the available data, it passes the data on to the next step, the inference engine. The inference engine's goal is to deduce new data using the data from the previous step. It does this by applying rules to the initial data sets in order to form new conclusions that can then again be used as data. The machine outputs these fragments of new data into the old data as if it was always there, in the same format. Jena has two built-in reasoners, but does offer support for external options to reason in a more specific way if necessary. Jena supports forward, backward and hybrid rules to form conclusions based on data. If no parameters are given, the framework does its best to gather as much new data as possible by combining all available techniques and both reasoners as shown in figure 4.2. This is the most powerful mode and infers as much data as possible, yet is very intensive to run. To cut back on execution time, the framework can be configured to only use the forward or backward chaining reasoner.



Figuur 4.2: Hybrid mode using both reasoners.

Jena's built-in reasoners seem very complicated at first, but using them is not so hard. The hardest part of using Jena is knowing what every component is for and how to use them. The documentation on Jena's website is a good guideline featuring many tutorials and examples and can be a guideline to create an application in a short amount of time. Using the reasoner in its most basic form, without any configuration or parameters, is possible and not very hard as can be seen in figure 4.3. This example shows the code needed to execute a simple command, imported from a file on the hard drive.



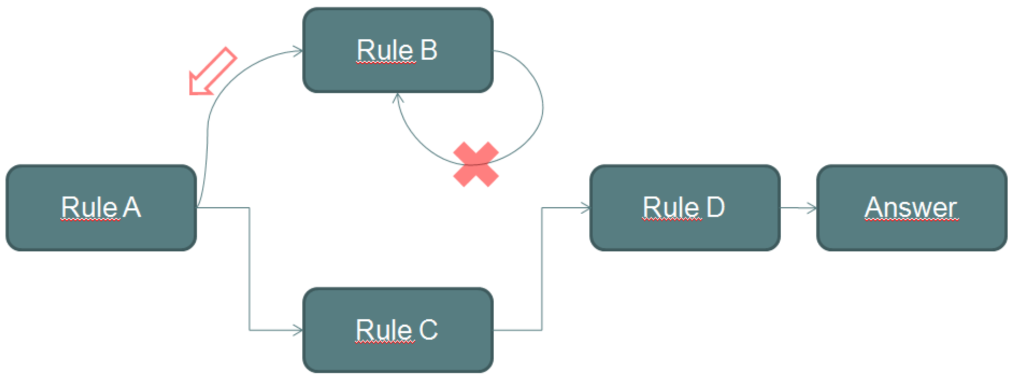
```
// load up an RDFModel
ModelMaker maker = ModelFactory.createModelRDBMaker(connection);
Model testmodel = maker.openModel("testrdf",true);

// Instantiate a reasoner
Reasoner smartHomeReasoner = ReasonerRegistry.getOWLReasoner();

// Bind the smarthomeReasoner to the ontology
Reasoner boundReasoner = smartHomeReasoner.bindSchema(smartHomeOntology);

// create the inference model
InfModel inferredDataModel = ModelFactory.createInfModel(boundReasoner, testmodel);

// create a query engine
QueryEngine qesh = new QueryEngine(queryFile);

// set the model as the source for the query
query.setSource(inferredDataModel);

// Execute the query as normal
QueryResults userActivity = qesh.exec(initialBinding);
```

Figuur 4.3: Executing a simple command using Jena.

The result of the inference engine is then passed on to the output module, or store API. The Store API supports the storing of data in SQL databases, the machine's internal memory and on the machine's discs by default but can be expanded to support custom storage solutions if one would need it to. Storing triples in an efficient way is a much overlooked fact, but handled by Jena very well.

4.1.2 Euler

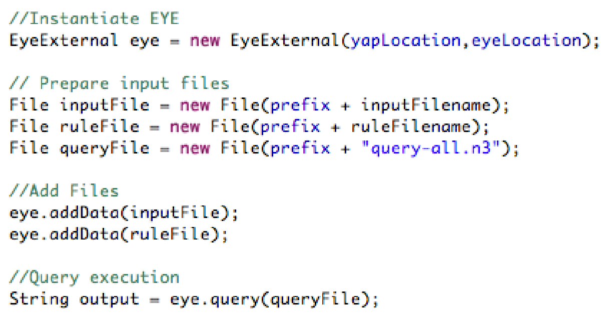
Euler is a backward-chaining reasoner with Euler path detection [24]. The project was started in 2002 and the first stable version of Euler saw the light of day in early 2004. Euler is still under heavy development and is updated on a monthly basis. Euler features Euler path detection, a feature not found in other reasoners. The Euler path detection makes sure that the reasoner can not get into a trivial loop, meaning that the reasoner does not have the issue of stepping in its own tracks. In execution time, this is a huge advantage when things go wrong. Backward chaining reasoner do tend to loop more often than forward chaining reasoners, so path detection was a welcome addition to the system. Euler achieves this by chaining through the data and saving a list of what rules it has gone through, if a rule is called upon twice in succession, the rule is ignored and the chain is backtracked to where it was before the looping occurred as shown in figure 4.4.



Figuur 4.4: Euler path detection at work

Euler is capable of generating proofs for any question you ask it, if supplied with the correct data and rules, another feature not often seen. Analyzing proof can aid in understanding data and the way it was formed. Proof generation shows the steps taken to come to a certain conclusion, sometimes it is more important to look at how a solution was found than the solution itself. Proper proof analysis can aid in writing better and faster rules, Euler can give you just the right output to do this. Euler offers implementations in C#, Java, Python and JavaScript. This makes the reasoner very versatile as it can even be deployed as a web application as seen in Ruben Verborgh's N3 tutorial [23].

EYE is a backward-forward-backward chaining reasoner and implements Euler fully. EYE is the reasoner that is used for this comparison, not Euler in its basic form. EYE is a fairly easy to use reasoner, with no configuration required. After downloading and correctly installing EYE, it can be run from the command line using basic commands. Already after doing this first test, the speed of the EYE reasoner will not go unnoticed. Implementing EYE into your own program will be just as easy, all you need to do is instantiate it, call it and show it where the rules are. Figure 4.5 shows how to execute a query using EYE in four simple steps.



```
//Instantiate EYE
EyeExternal eye = new EyeExternal(yapLocation,eyeLocation);

// Prepare input files
File inputFile = new File(prefix + inputFilename);
File ruleFile = new File(prefix + ruleFilename);
File queryFile = new File(prefix + "query-all.n3");

//Add Files
eye.addData(inputFile);
eye.addData(ruleFile);

//Query execution
String output = eye.query(queryFile);
```

Figuur 4.5: Using EYE in Java

After query execution is complete, EYE outputs a string containing old and new data in combination with the necessary identifiers. This output is ready to be saved or processed in any way you want.

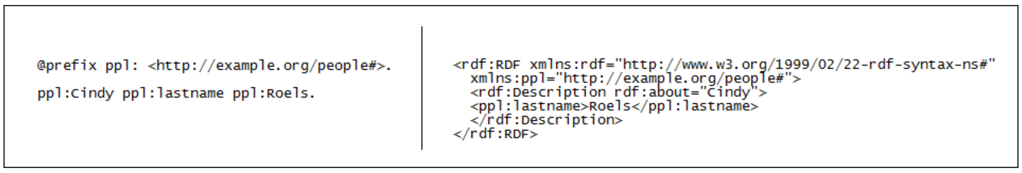
4.1.3 Testing

Choosing the correct reasoner for the job based on features alone is not accurate nor possible. To really get a good eye for what can or can not be done by both reasoners, objective testing is the only way to go.

In this use case, the application will be used to monitor humans. Humans are very slow compared to computers, they do not tend to change their status very quickly, as they perform tasks like watching TV which can go on for hours. This means that the reasoner does not have to be very fast to keep up with what the human in doing, but does this mean speed is not important?

In the world of computing, speed is everything. Raw speed might not really be necessary to reason with great accuracy, as explained before, but can still make a difference where one would normally not expect it to. Reasoning is a very intensive task when used to perform jobs that require a lot of inputs and have many possible solutions, thus can take up a lot of time and system resources. Each millisecond a system is under load, it consumes a big amount of electricity. This affects battery life in a negative way which can be an issue if power from the grid is not available. Another side effect of being under load is that each and every clock cycle, more heat is produced, heat that needs to be taken away by electric fans and thus again affects battery life in a negative way. So it is safe to say that speed is important, but just not for the obvious reason.

To perform the speed test, each rule engine is given the same rule set and performs an equal amount of iterations. The rule sets are designed to test certain features of the reasoner. Both rule sets are basically the same, yet have their syntax differences as Jena uses an RDF/XML syntax and EYE uses the much easier to read N3 format.

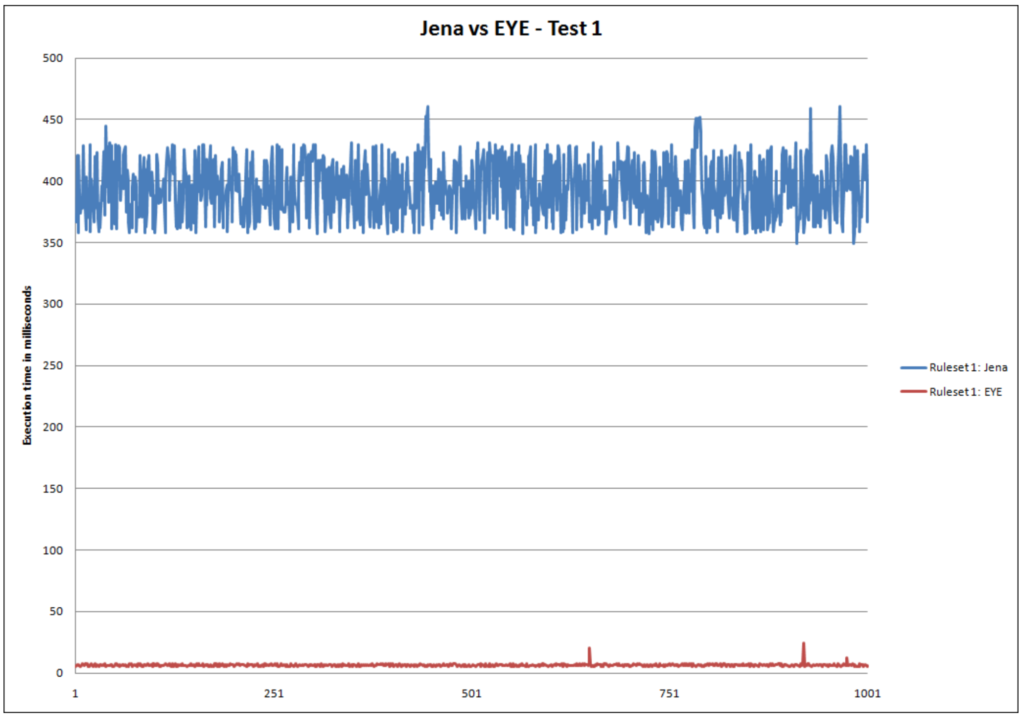


```
@prefix ppl: <http://example.org/people#>.

ppl:Cindy ppl:lastname ppl:Roels.
```

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:ppl="http://example.org/people#">
  <rdf:Description rdf:about="Cindy">
  <ppl:lastname>Roels</ppl:lastname>
  </rdf:Description>
</rdf:RDF>
```

Figuur 4.6: Difference in data formatting. Left: N3 data representation, Right: RDF/XML data representation

The first test is the easiest and most basic of all available operations, just a plain and simple match-and-output of data operation based on a simple rule without variables or inference. Image 4.6 represents the first set of data, also clearly showing the difference between RDF/XML and N3. The reasoner was then asked to get Cindy's last name. No variables, aliases or other operations where used to keep load to the bare minimum. As Figure 4.7 shows, the EYE reasoner was by far the fastest in this early stage of testing.

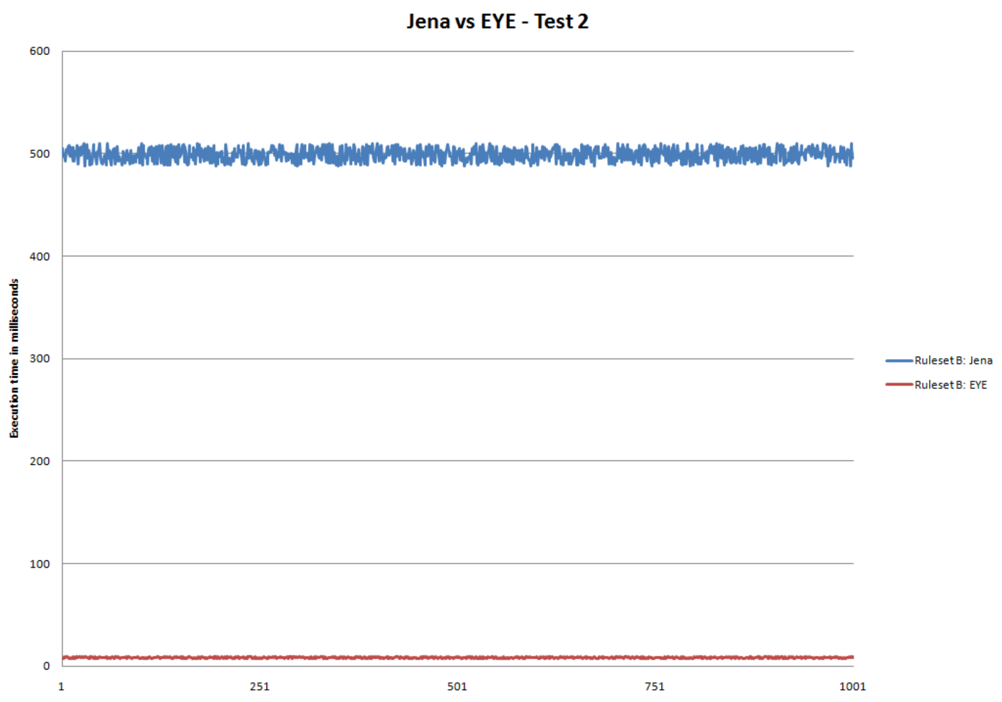


Figuur 4.7: Test 1: execution time in milliseconds, Blue: Jena, Red: EYE

In this first test, Jena clocks an average execution time of 394ms. Although this is fast enough to cover most of a human's moves, EYE certainly impresses with a very fast 6ms of reasoning time. If startup time of the respective reasoners is included in the test, EYE once again emerges victorious with an average startup time of 143ms against 221ms for Jena. The total average time of one complete reasoning session would then be 149ms for EYE and 615ms for Jena, thus meaning that EYE is roughly four times faster in a basic operation.

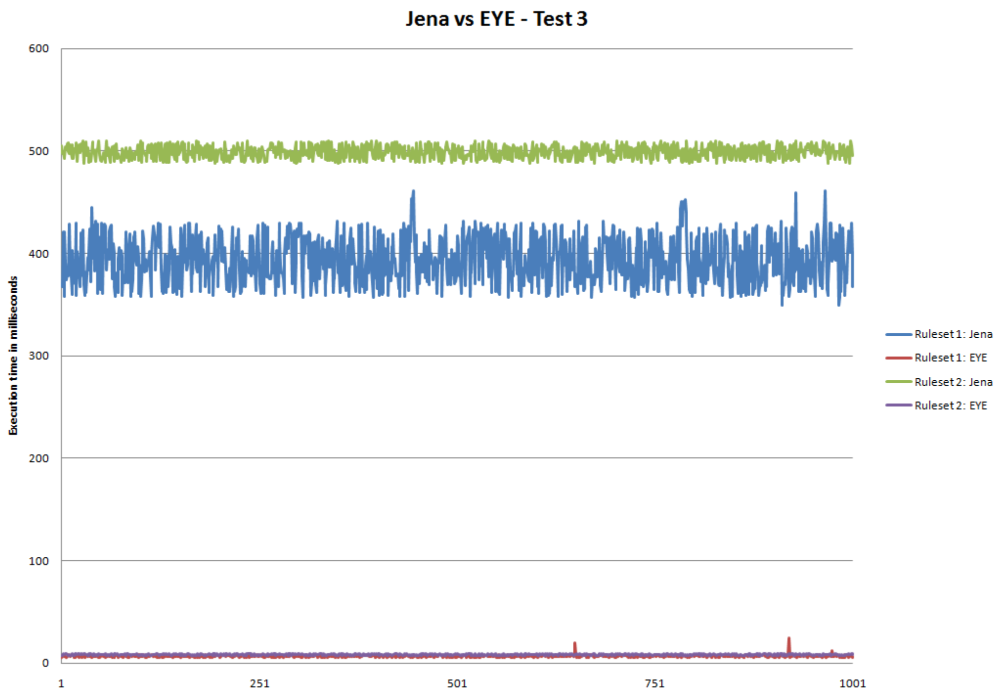
The basic operation is a good indication of startup and minor execution time, yet things get interesting when a more intensive task is demanded from the reasoner. Both reasoners have the ability to work with variables, link those to existing data and then inference on that data to form new fragments of data. The reasoner in this project will mainly be used to inference new data, based on what it knows about a situation so testing how both perform is a logical next step. In the next test, the reasoner is given a set of data with a matching set of rules based on the examples found at JenaRules [21]. These where modified to fit the EYE reasoner and ran 1000 times as a test on both systems.

As figure 4.8 shows, EYE emerges fastest once again. EYE seems fairly unaffected by the increased complication of rules and data and remains very stable as load increases. During testing no memory usage issues were recorded, both reasoners remained within acceptable limits. Jena did seem to increase memory usage slightly in the second test, while EYE remained roughly at the same level. Average processing time for Jena was 499ms, while EYE checked out at an impressive 8ms, or 63 times faster in pure reasoning if startup time of the reasoners is ignored. It is to be noted that both reasoners performed very well during both tests with no errors, crashes or incomplete inferences.



Figuur 4.8: Test 2: Execution time in milliseconds, Blue: Jena, Red: EYE

Although the result of both reasoners was predictable, the real difference was to be found when both data sets are compared to one another as seen on figure 4.9. Immediately noticeable is the fact that the increase in difficulty makes the processing time of the Jena reasoner increase significantly. On a single run, this does not form a problem, but when more rules are added and the application expands further than the reach of this use case, this could become a problem to guarantee a responsive system at all times.



Figuur 4.9: Test 1 and 2 compared: Blue: Jena 1, Red: EYE 1, Green: Jena 2, Purple: EYE 2

4.1.4 Conclusion

Two reasoners, both very similar, yet very different. Table 4.1 shows an overview of the most important features of both tested reasoning engines.

	Jena	EYE
User License	Open Source	Open Source
Version	2.6.4	2012-05
Last updated	2010-12	2012-05
Install Size	23.3 Mb	13.9 Mb
Type of reasoning	Forward, Backward or Hybrid	Backward-Forward-Backward
Ease of use	Complex	Easy
Execution Speed	Moderate	Fast
Accuracy	High	High

Tabel 4.1: Comparison between Jena and EYE

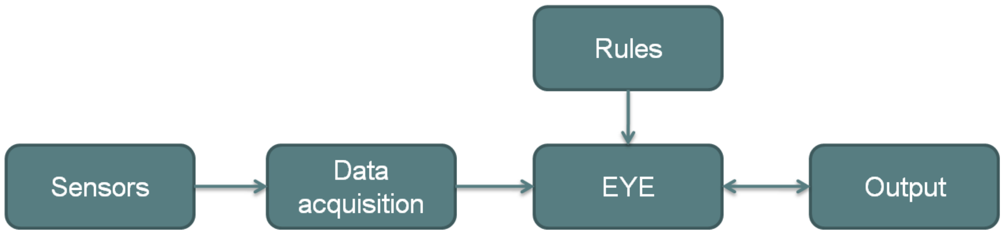
As stated in the requirements, a fast and reliable reasoner is needed to maintain a responsive system and keep energy costs to a minimum. EYE is by far the favorite based on execution time alone. Jena does have the advantage of offering a more complete API, but is harder to use and most of the features it offers will not be used anyway. The clear winner in this test is the EYE reasoner and will be implemented in this project.

4.2 Structure of the application

After all research is done, a good idea is formed of how the application should work and how it is to react to changing data. The main structure of the application will feature four main ingredients:

- A data acquisition module
- A set of rules
- EYE: reasoning engine
- An output module

These four blocks will be structured as seen in figure 4.10. They will interact with each other when needed, adding to the flexibility of the system itself.



Figuur 4.10: Structure of the smart home application

4.2.1 Data acquisition

The data acquisition module will be in charge of gathering the data from the sensors through the network that was designed by Vincent Hearinck [4]. Once the rough data has been gathered, it will be formatted into a usable N3 format and saved to the hard drive for further use.

4.2.2 Rules

The rules will determine how the application uses the sensor data and try to deduce new data from the already existing data. The rules will be loaded from a rule file from the hard drive of the machine, and then used by the reasoning module in combination with the data that was gathered earlier. The rules will specify what actions can be detected and what sensors to use to detect a certain activity. The rules will also be used to check the data provided by the sensor, with the intention of finding malicious data and thus malfunctioning sensors.

4.2.3 Reasoning

The reasoning module will be the heart of the application. It will combine the rules and the data while providing conclusions based on its inputs. EYE will be the reasoner of choice after testing. The reasoner will output the conclusions to the output module, but will also get input from the output module when it will be asked to verify sensor data.

4.2.4 Output

The smallest of the four main pieces of the application will be the output module. The output of the reasoner module, as discussed in the previous section, will be in the N3 format. This set of N3-triples needs to be examined for new data and processed into a clean set of variables, with the most important one being the activity of the person in the end. This activity will then be relayed back to the reasoner to check the values as a built-in fail-safe mechanism.

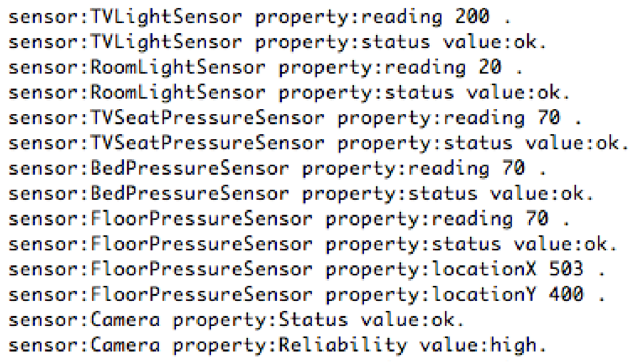
Hoofdstuk 5

Implementation

5.1 Data acquisition

The first step in reasoning, is gathering the data to reason on. If data is not delivered correctly, the system would be flawed and not work at all so this is the first important step in the whole process.

At initial startup, or after a certain period of time, all sensors are queried and deliver their data. This is the first stage in the data acquisition process.



```
sensor:TVLightSensor property:reading 200 .
sensor:TVLightSensor property:status value:ok.
sensor:RoomLightSensor property:reading 20 .
sensor:RoomLightSensor property:status value:ok.
sensor:TVSeatPressureSensor property:reading 70 .
sensor:TVSeatPressureSensor property:status value:ok.
sensor:BedPressureSensor property:reading 70 .
sensor:BedPressureSensor property:status value:ok.
sensor:FloorPressureSensor property:reading 70 .
sensor:FloorPressureSensor property:status value:ok.
sensor:FloorPressureSensor property:locationX 503 .
sensor:FloorPressureSensor property:locationY 400 .
sensor:Camera property:Status value:ok.
sensor:Camera property:Reliability value:high.
```

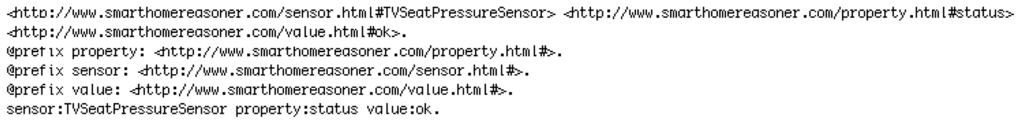
Figuur 5.1: Rough sensor data as received from the webservices

A typical example of received sensor data can be seen in figure 5.1. The collected sensor data is still in a rough format and is not yet ready for use, it still needs prefixes and URI's to work properly. The data that needs to be added to the received sensor data is read from a file on the hard drive and added to the sensor data. A correctly formatted data file consists of three parts:

- A set of URI's
- A set of prefixes
- The actual data in N3 format

A URI is the definition of the data that is being used, usually in the form of a URL. The goal of a URI is to define data in a semantic model, allowing the data to have a meaning on the web. Adding this property to data makes it linkable to a subject or usable in a rule. The advantage of doing this is the fact that data can be reused in other applications and universally identified on the web.

The prefixes are abbreviations of the URI used in the data model. These are used to make the data and rules easier to read and write. If prefixes would not be used, the data would look like the top two lines in figure 5.2. For each and every value, a semantic definition is needed. If no prefixes would be used, the value could still be described by using the URL every time. In a complex application, this would make it harder to work with the data manually and also increase the file size and thus adding load on the network that should not be there.



```
<http://www.smarthomereasoner.com/sensor.html#TVSeatPressureSensor> <http://www.smarthomereasoner.com/property.html#status>
<http://www.smarthomereasoner.com/value.html#ok>.
@prefix property: <http://www.smarthomereasoner.com/property.html#>.
@prefix sensor: <http://www.smarthomereasoner.com/sensor.html#>.
@prefix value: <http://www.smarthomereasoner.com/value.html#>.
sensor:TVSeatPressureSensor property:status value:ok.
```

Figuur 5.2: URI notation (top two lines) vs prefix representation of data (bottom three lines)

Prefixes are not a complicated thing, nor are they very large, so they should be able to be stored in the source code and ready for use. If these prefixes would be stored in code, two less file I/O operations would have to take place making the application just a bit faster. The reason why the prefixes are not stored in the source is because a technician would need the source code to be able to make an adjustment to the system. The system is designed to accept new sensors without any complex configuration, so it would be counter-productive to need a programmer every time a small adjustment or new sensor needs to be added. With the prefixes read from just a single text file, it should be easy to add or adjust parameters even if the person doing the adjustments does not have any programming skills.

The data itself consists of an N3-formatted string, containing a subject, predicate and an object. In this use case, the subject can only be a sensor, a person or a warning. The predicate states what will be said about the subject, this can be almost anything. A good example of a much used predicate is „position”, meaning that the value will be the position of the subject, in most cases the patient.

Once the data has been formatted and enriched with prefixes and URI's, it can be seen as a complete and ready to use collection of N3 Data. Before this collection is passed on to the reasoner, is it written away to a file for future use. These files can be used later on as a source of data for machine learning, or simply as a backup or log entry to map activities and warnings.

In a later stage of the reasoning process, it may occur that there is some doubt about some of the delivered values. It is of course possible to query specific sensors, although this will be used less frequently than a normal query to all sensors at once.

5.2 Rules and reasoning

In the smart home project, two types of rules are used. One set will focus on determining every aspect of the situation in the home, while the other will check if the deduced data is correct. Both of these actions require the usage of the reasoner.

5.2.1 Reasoning with EYE

Deducing new facts from already existing data is what a reasoner is designed for, it can have multiple ways of doing this. The EYE Reasoner has support for variables, implements semantic linking of data and some built-in easy to use functions. These features combined give the user the power to deduce facts with great precision in a fast and effective way. EYE does more than only reasoning, it can do basic functions such as adding and multiplying but also string or time based operations, making the reasoner a tool that also works with the data and not just processes it. This functionality comes in handy in certain aspects of the project, such as person detection based on coordinates provided by multiple sensors.

The built-in functions used in this project are primarily math related and used to compare values or perform basic mathematical operations with the data.

The reasoning itself happens in multiple stages. The first stage of the process is the importing of data that was saved on the hard drive earlier. File import and export is supported by the EYE Reasoner out of the box, getting the data into the reasoner is merely a matter of calling the correct methods and supplying the correct filename and location. The reasoner itself needs three things to work properly:

- A correctly formatted set of data
- A set of matching rules
- A goal to reach

Once the reasoner is asked to query the data and rules, it automatically fetches the data from the hard drive. This makes the reasoner easy to use and keeps the code easy to maintain. File I/O does cost some time, so the reasoner also supports direct import from data stored in memory, as long as the data is in the correct N3 format.

The rules combined with the sensor data define what the output of the reasoner will be. The rules use the data to deduce new facts in an easy to adjust and very flexible way. The rules are basically a definition of what type of data can be used to deduce new facts, they are filled in by the data and generate an output in the form of an activity or warning.

5.2.2 Forming a conclusion based on sensor data

The sensor data from the previous section, means nothing without a benchmark to compare them to. Sensor data needs to be combined and analyzed to be able to form a correct conclusion, the best way to do this, is by filling in a set of predefined rules. The rules specify where a sensor value is used, what it can be compared to and what the outcome of a combination of values could be.

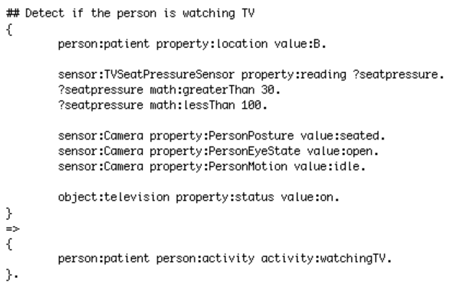
At first, working with rules may seem complex and not worth the effort. As soon as an application grows however, the true advantages of using rules soon come to the surface. Using rules is a great way to maintain different inputs in an orderly fashion, without the clutter of writing hundreds of different if/else constructions or switch/case code blocks. A good tutorial on writing your first rules using N3, can be found at http://n3.restdesc.org

[23], the website offers a clear look at what basic rules are and why one would want to implement them. Rules can also be tested in real time on that website which makes it a valuable resource to those making their first steps in rule-based reasoning.

There is no real golden approach to rule design, but a good way to start is by defining what your ultimate goal will be. The next step is to define what data is needed to reach that goal and to work with that as a starting point. Most of the time a combination of inputs will be enough to reach the end goal, but hard coding of every goal including its steps to reach it is not good design practice.

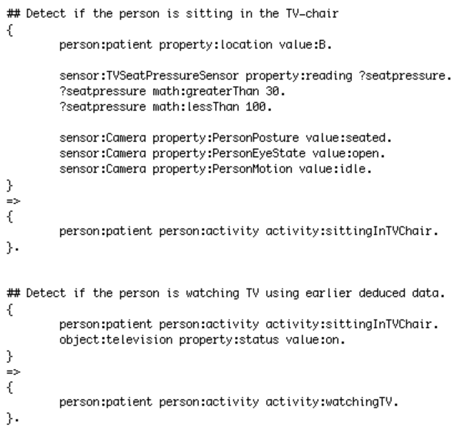
A better way to implement the data received from the sensors, would be to design the rules based on a cascading effect. The reuse of sensor data directly can cause a rule file to become hard to maintain and read, causing anomalies in the reasoning process later on. Good practice would be to have some steps in between start and ending of the reasoning. After every step, a new fact will be deduced if all is according to plan, this new fact can be used as an input in multiple other rules. This design will be a lot less prone to errors and will shorten the rule file itself.

To illustrate the usefulness of this design principle, a comparison between figures 5.3 and 5.4 is made.



```
## Detect if the person is watching TV
{
        person:patient property:location value:B.

        sensor:TVSeatPressureSensor property:reading ?seatpressure.
        ?seatpressure math:greaterThan 30.
        ?seatpressure math:lessThan 100.

        sensor:Camera property:PersonPosture value:seated.
        sensor:Camera property:PersonEyeState value:open.
        sensor:Camera property:PersonMotion value:idle.

        object:television property:status value:on.
}
=>
{
        person:patient person:activity activity:watchingTV.
}.
```

Figuur 5.3: Rule using all data at once



```
## Detect if the person is sitting in the TV-chair
{
        person:patient property:location value:B.

        sensor:TVSeatPressureSensor property:reading ?seatpressure.
        ?seatpressure math:greaterThan 30.
        ?seatpressure math:lessThan 100.

        sensor:Camera property:PersonPosture value:seated.
        sensor:Camera property:PersonEyeState value:open.
        sensor:Camera property:PersonMotion value:idle.
}
=>
{
        person:patient person:activity activity:sittingInTVChair.
}.


## Detect if the person is watching TV using earlier deduced data.
{
        person:patient person:activity activity:sittingInTVChair.
        object:television property:status value:on.
}
=>
{
        person:patient person:activity activity:watchingTV.
}.
```

Figuur 5.4: Rule using cascading

At first glance, figure 5.3 seems to be the best option as there is only one rule needed to determine the end result. The amount of code is also shorter and easier to read since no references are made to data from other rules. The downside to this approach, is that only one value will be deduced from a large amount of input variables. In this example, the outcome of the reasoning process is that the patient is watching the television. Although this is correct, if the input data is examined, it soon becomes clear that more than one fact can be deduced from that data set.

To be able to deduce more facts that the reasoner can use later on, the cascading style of rule building is a good option. The reasoning now happens in multiple steps on smaller fragments of input data. Each time a rule passes its preconditions, a new conclusion is formed. The more conclusions are formed, the more accurate the end result can be determined. This type of reasoning is illustrated in figure 5.4. The output of that combination of rules will still be that the patient is watching the television, but that conclusion will be accompanied by the new fact that the person is also sitting down. So more data has now been deduced, without adding new inputs.

This extra data can from now on be used in other rules, the main advantage of this technique is that sensor data only needs to be processed once and is from then on carried through the rules as a new fragment of data. Combining data into new shorter fragments makes it reusable in other rules, this way data does not have to be repeated in the same file keeping everything easy to adjust and maintain. For instance, if a system supports six different ways of determining where a patient is in the room, the system could combine all six ways into one fragment of new data. This new fragment can later be used if a position is required, instead of using all six once again in a different rule.

After all input data has been used and all possible connections have been made, the ultimate end result will be one activity and a set of possible warnings or extra data that has been generated. Once the main activity of the patient has been determined, the first round of reasoning is done.

5.2.3 Double-checking deduced data

Although the first round of reasoning is enough to get an accurate result in most cases, double checking when possible has never hurt anyone. The advantage of the second check is that some actions or parts of actions can be influenced by a single sensor and that one sensor can be faulty. Now that the activity of the patient has been detected, it is possible to work the other way around through the data. It is always true that one activity can be linked to multiple segments of data, so there has to be a value that is expected to match the activity at all times, else the activity can not exist. This theory can be proven by the fact that EYE is capable of generating proof of the conclusion it forms. It illustrates the path towards the solution, so going backwards should be possible at any given moment.

The idea behind all this, is that the real time sensor data can be checked against the data one would expect if a patient is doing a certain activity. Comparing these two sets of data can highlight misconfiguration issues or malfunctioning sensors. This form of error control is a nice extra that can be achieved without much effort since the data and reasoning engine is already there. ————————————————- verder afwerken!!!!!!!!!

5.2.4 Dealing with malicious data

To be continued ————————————————- verder afwerken!!!!!!!!!

5.3 Reporting

The reporting module is designed to catch the output of the reasoning module. It has the ability to output data to a GUI, save the deduced facts to the hard drive or save the data in the systems memory for fast reuse. Since the deduced facts contain information about a patient, it could be useful in the future to relay that information to a family member or a caretaker of choice in certain scenarios.

The EYE Reasoner outputs all the data it has gotten in the first place, with newly deduced data added to it. The output module has the ability to save the entire stream to a file, including prefixes and URI's for later use or separate the new data from the old and save only the new data.

Hoofdstuk 6

Results

The end result of this study is a multi sensor semantic reasoner designed to be implemented in a smart home. It uses rules to define what a person is doing or what types of actions are going on in the smart home. The reasoning module has proven to be fast and reliable, even if reasoning needs to be done with only a few input values. The use of EYE was clearly the correct choice as Jena would have been much slower, looking at testing, thus negatively affecting battery life and usability. The reasoner has the ability to warn if a sensor has been malfunctioning, adding to the reliability of the entire system.

The output module has been kept fairly simple, as really implementing a warning system is something specific for every application. There is however a list of warnings and actions ready for use in any way required.

This application is capable of running on its own. The only problem with the application as it is right now, is the fact that furniture is movable. This problem will be addressed in the future work section.

Hoofdstuk 7

Future work and conclusion

7.1 Future work

The next big step in designing the smart home, would be to make the system self learning. Based on the sensor data, it could try to keep track of the living patterns of a human, and try to detect if someone has deviated from his normal path in such a way, that it could be a health risk. For instance, when the patient goes to sleep between 10 and 11 pm for a long period of time, the machine could track this behavior using a support vector machine. The machine will learn that this is normal, when all of a sudden the patient isn't in bed by 2 am, the machine will pick this up and could signal this to one of the caretakers if deemed necessary.

This type of pattern recognition could also be used with people suffering from dementia, to protect them from making the wrong mistakes and letting the machine take action before the situation gets out of hand. Although the smart home was originally designed not to intervene in a patient's private atmosphere, it could one day save a person's life. This type of intervention will always be a touchy subject, because it could scare or confuse people, but in some cases it is definitely worth considering.

Anther useful type of machine learning that could be added to a smart home, is to register en log movements and adjust sensor data with it. As the reasoning module is set up right now, it looks at a patient as if it was a moveable object, the only one in the room. In real life, chairs and beds are moveable objects too, with their own position and status. If the sensors could track a person and learn that he has been sitting at a certain place, combining that with the seat's sensor registering that someone is seated, the machine could learn that the seat is indeed at that location. If the location was than to change due to someone moving the seat, the sensor data would not match and would generate contaminated data if no further action is taken. If the system would be capable to adjust the location of the seat, based on where a person has been sitting and triggering the seat's sensor, the system could learn that the seat has just moved instead of generating errors or cause illusive warnings. This would add to the user experience by causing fewer interventions from caretakers or the system itself due to the higher accuracy.

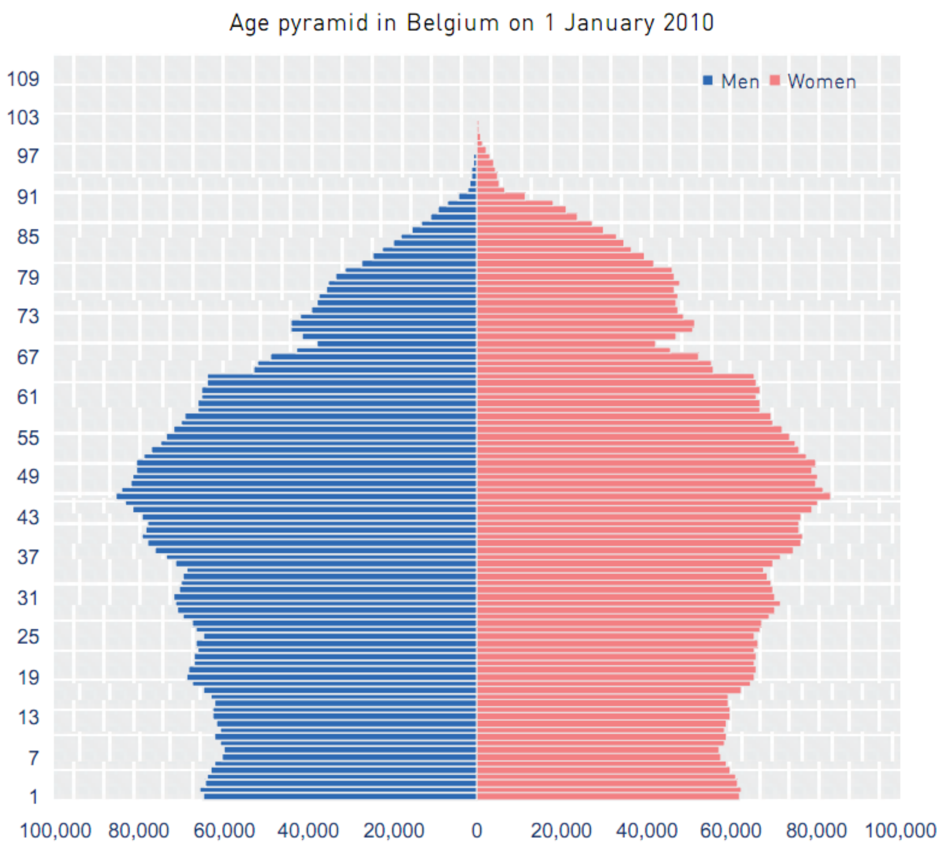
7.2 Final conclusion

The biggest issue in healthcare is the cost of maintaining a single patient. A single patient takes up expensive space in a retirement home or hospital, needs attention from caretakers and often does not feel like leaving his own trusted environment. Placing a person in a retirement home is a step that most will try to postpone as much as possible, even if this means that they are putting themselves in danger. Although not every patient can be placed in a smart home, this system does offer a start to a solution that one day could be implemented for certain types of patients. The smart home designed by Haerinck [4], Houdmont [5] and Huyghe provides a controlled environment that makes it possible for some patients to live without assistance from a caretaker without big concerns about health and safety. The main target group for this study will be the elderly who are still mobile to some extent, not the sick of severely injured.

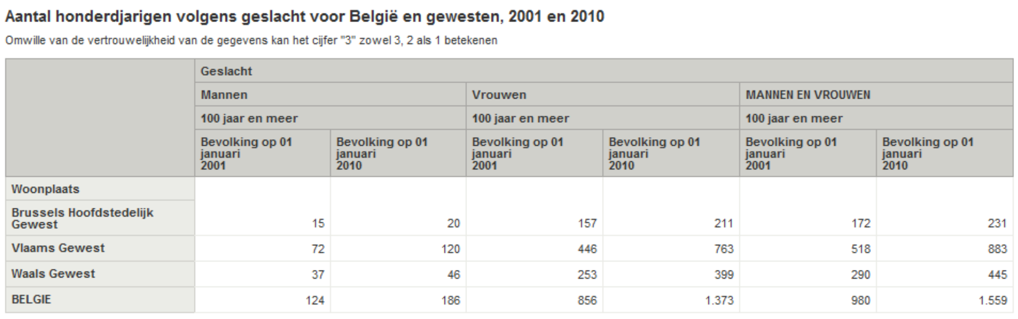
The smart home tackles the problems of everyday healthcare by keeping an eye on the patient in a non-intrusive way. The system is able to detect certain activities using sensors and a reasoner and report if needed to a caretaker for further assistance. The system is designed to relieve a caretaker from his tasks as a guardian by automatically detecting when a person is in distress, not to take over the actual healthcare. The system can be implemented in an existing home if a patient would chose to do so, thus cutting back on the costs of renting a room in a retirement home.

Bijlage A

Appendix A - Belstat statistics



Figuur A.1: Age pyramid for Belgium

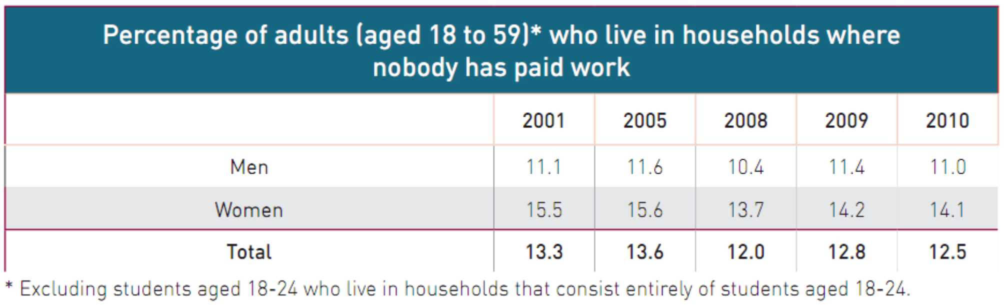


Aantal honderdjarigen volgens geslacht voor België en gewesten, 2001 en 2010

Omwille van de vertrouwelijkheid van de gegevens kan het cijfer "3" zowel 3, 2 als 1 betekenen

	Geslacht					
	Mannen		Vrouwen		MANNEN EN VROUWEN	
	100 jaar en meer		100 jaar en meer		100 jaar en meer	
	Bevolking op 01 januari 2001	Bevolking op 01 januari 2010	Bevolking op 01 januari 2001	Bevolking op 01 januari 2010	Bevolking op 01 januari 2001	Bevolking op 01 januari 2010
Woonplaats						
Brussels Hoofdstedelijk Gewest	15	20	157	211	172	231
Vlaams Gewest	72	120	446	763	518	883
Waals Gewest	37	46	253	399	290	445
BELGIE	124	186	856	1.373	980	1.559

Figuur A.2: Population aged over 100 years in Belgium

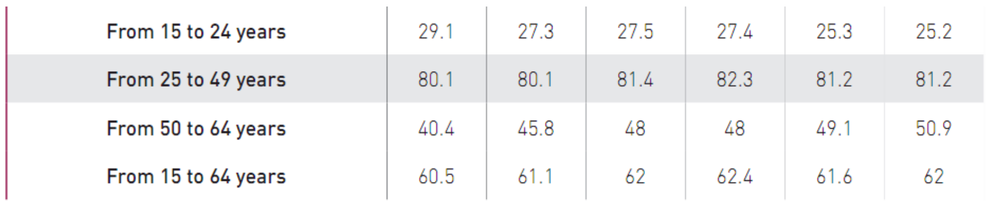


Percentage of adults (aged 18 to 59)* who live in households where nobody has paid work

	2001	2005	2008	2009	2010
Men	11.1	11.6	10.4	11.4	11.0
Women	15.5	15.6	13.7	14.2	14.1
Total	**13.3**	**13.6**	**12.0**	**12.8**	**12.5**

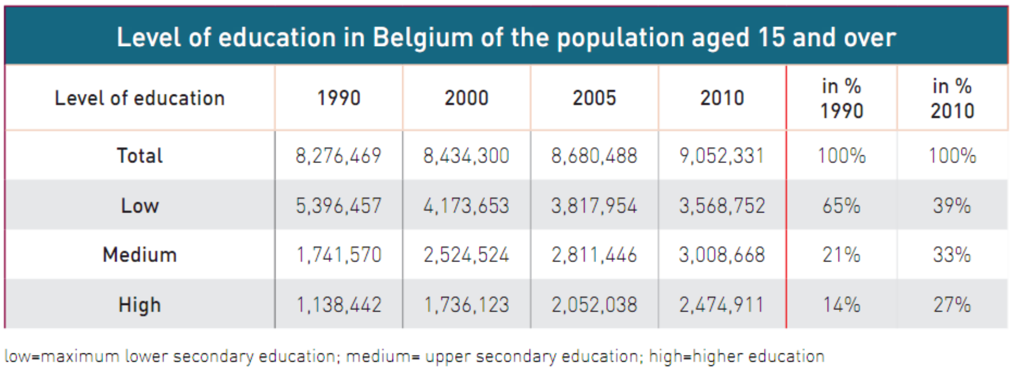
* Excluding students aged 18-24 who live in households that consist entirely of students aged 18-24.

Figuur A.3: Number of households without an income in Belgium



From 15 to 24 years	29.1	27.3	27.5	27.4	25.3	25.2
From 25 to 49 years	80.1	80.1	81.4	82.3	81.2	81.2
From 50 to 64 years	40.4	45.8	48	48	49.1	50.9
From 15 to 64 years	60.5	61.1	62	62.4	61.6	62

Figuur A.4: Population by employment rate in Belgium



Level of education in Belgium of the population aged 15 and over

Level of education	1990	2000	2005	2010	in % 1990	in % 2010
Total	8,276,469	8,434,300	8,680,488	9,052,331	100%	100%
Low	5,396,457	4,173,653	3,817,954	3,568,752	65%	39%
Medium	1,741,570	2,524,524	2,811,446	3,008,668	21%	33%
High	1,138,442	1,736,123	2,052,038	2,474,911	14%	27%

low=maximum lower secondary education; medium= upper secondary education; high=higher education

Figuur A.5: Population by education level in Belgium

Bibliografie

[1] Living tomorrow.

[2] Jaeyong Sung, Colin Ponce, Bart Selman, and Ashutosh Saxena. Human activity detection from rgbd images. *CoRR*, abs/1107.0169, 2011.

[3] Marie Chan, Daniel Estève, Christophe Escriba, and Eric Campo. A review of smart homes-present state and future challenges. *Comput. Methods Prog. Biomed.*, 91:55–81, July 2008.

[4] Vincent Haerinck. Semantic soa-based smart sensor networks in smart homes. 2012.

[5] Henry Houdmont. Vision-based assistance in soa-based smart sensor networks in smart homes. 2012.

[6] Wenjia Niu, Gang Li, Hui Tang, Xu Zhou, and Zhongzhi Shi. CARSA: A context-aware reasoning-based service agent model for AI planning of web service composition. *JOURNAL OF NETWORK AND COMPUTER APPLICATIONS*, 34(5):1757–1770, SEP 2011.

[7] Yung-Chien Sun and O. Grant Clark. Two learning approaches for a rule-based intuitive reasoner. *Expert Syst. Appl.*, 38:6469–6479, June 2011.

[8] Georgios Meditskos and Nick Bassiliades. DLEJena: A practical forward-chaining OWL 2 RL reasoner combining Jena and Pellet. *JOURNAL OF WEB SEMANTICS*, 8(1):89–94, MAR 2010.

[9] L.A. Zadeh. Fuzzy logic = computing with words. *Fuzzy Systems, IEEE Transactions on*, 4(2):103 –111, may 1996.

[10] Lotfi A. and Zadeh. Toward a theory of fuzzy information granulation and its centrality in human reasoning and fuzzy logic. *Fuzzy Sets and Systems*, 90(2):111 – 127, 1997. ¡ce:title¿Fuzzy Sets: Where Do We Stand? Where Do We Go?¡/ce:title¿.

[11] S. R. Safavian and D. Landgrebe. A survey of decision tree classifier methodology. *Systems, Man and Cybernetics, IEEE Transactions on*, 21(3):660–674, 1991.

[12] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. Technical report, World Wide Web Consortium, May 2004.

[13] Janet L. Kolodner. An introduction to case-based reasoning. *Artificial Intelligence Review*, 6:3–34, 1992. 10.1007/BF00155578.

[14] Donald F. and Specht. Probabilistic neural networks. *Neural Networks*, 3(1):109 – 118, 1990.

[15] KR Muller, S Mika, G Ratsch, K Tsuda, and B Scholkopf. An introduction to kernel-based learning algorithms. *IEEE TRANSACTIONS ON NEURAL NETWORKS*, 12(2):181–201, MAR 2001.

[16] H Byun and SW Lee. A survey on pattern recognition applications of support vector machines. *INTERNATIONAL JOURNAL OF PATTERN RECOGNITION AND ARTIFICIAL INTELLIGENCE*, 17(3):459–486, MAY 2003. 1st International Workshop on Pattern Recognition with Support Vector Machines, NIAGARA FALLS, CANADA, AUG 10, 2002.

[17] CW Hsu and CJ Lin. A comparison of methods for multiclass support vector machines. *IEEE TRANSACTIONS ON NEURAL NETWORKS*, 13(2):415–425, MAR 2002.

[18] R Andrews, J Diederich, and AB Tickle. Survey and critique of techniques for extracting rules from trained artificial neural networks. *KNOWLEDGE-BASED SYSTEMS*, 8(6):373–389, DEC 1995.

[19] John Hertz, Anders Krogh, and Richard G. Palmer. *Introduction to the theory of neural computation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1991.

[20] AK Jain, JC Mao, and KM Mohiuddin. Artificial neural networks: A tutorial. *COMPUTER*, 29(3):31+, MAR 1996.

[21] Brian McBride. Jena Rules Syntax. `http://hydrogen.informatik.tu-cottbus.de/wiki/index.php/JenaRules#The_Syntax_of_Jena_Rules`, 2011.

[22] Jena. Jena API. `http://incubator.apache.org/jena/index.html`, 2012.

[23] Ruben Verborgh. Semantic Web Reasoning With EYE. http://n3.restdesc.org, 2011.

[24] Jos De Roo. The home of EYE reasoner. http://eulersharp.sourceforge.net, 2006.

[25] BBC News. Mouse Brain Simulated on Computer. http://news.bbc.co.uk/2/hi/technology/6600965.stm, 27 April 2007.

Lijst van figuren

Lijst van tabellen