EsperSens - A Complex Event Processing System for Automated Home Care

William W.F.D. Almnes - wwalmnes@ifi.uio.no
Master's Thesis - August 1, 2014
Abstract

In automated home care systems (HCSs), sensor technology is employed to monitor patients, thereby aiming to reduce the workload of healthcare professionals and improving the patient’s quality of life. A complex event processing (CEP) system can be used to analyze sensor readings and identify interesting events.

CommonSens is a CEP system that is specifically designed for automated HCSs, which simplifies the work of the application programmer by adapting to different environment and sensor configuration. It provides fundamental data models for the environment, sensors and events, which enables the user to identify elements in the home environment and to reuse these elements. However, some required features are not fully implemented, such as support for processing multiple queries simultaneously and a solution for notifying the caregiver when an important event has occurred.

Esper is considered the leading open source CEP provider and can be applied to a variety of domains. As Esper is designed in a generalized way, it has to be extended in order for it to be usable as an underlying CEP system for an automated HCS.

In this thesis we investigate if Esper can be used as a CEP system to build an automated HCS by implementing the three fundamental data models of CommonSens with Esper. The resulting system is called EsperSens. To simplify the work for the application programmer, by using the query language of CommonSens, we create a translator that takes a CommonSens query and constructs an Esper query. We implement a web application to serve as a user interface for the system so that it can be accessed by multiple devices, e.g., a smartphone or a laptop.

To evaluate if EsperSens correctly detects events we performed simulations with synthetic workloads. We measure the time Esper needs to process events during these simulations and compare them with the results of running these simulations with CommonSens. Our results show that EsperSens correctly detect events, but runs on average slower than CommonSens. Compared to the time measured when delivering a notification over a network, the difference between CommonSens and EsperSens is insignificant. We conclude that EsperSens satisfies the most crucial requirement of automated HCS, which is to correctly detect events in near real-time and deliver notifications to a healthcare professional.
Acknowledgements

First of all, I would like to thank my supervisor, Professor Dr. Vera Goebel, for her guidance, support and incredible patience. Being able to have frequent meetings where we could discuss my thesis has really helped and I have learned a lot. Thank you Professor Dr. Thomas Plagemann, for randomly encountering me along the corridors and providing me with a boost to morale.

I would also like to extend my thanks to the following individuals: Thank you James Trotter and Ivar Tryti for proofreading my thesis. Thank you Khiem Xuan and Endri Hysenaj for spending time together at IFI and at the gym. Thank you Marcel Eggum and Stein Kristiansen for being able to discuss the various features of Esper with me. Thank you to the members of my martial art club for allowing me to vent out my frustrations (recommended to all programmers!). To family, friends and anyone who have listened to me babble on about my thesis, thank you!

Lastly, thank you my extraordinary Camilla. This thesis has been a long journey and during this period there have been many hardships. I am tremendously lucky for having you with me and supporting me every step of the way.
# Contents

Abbreviations 5

List of Figures 7

List of Tables 9

1 Introduction 11
   1.1 Background and Motivation .......................... 11
   1.2 Problem Statement .................................. 13
   1.3 Outline ............................................. 13

2 Automated Home Care Systems 15
   2.1 Traditional Home Care ............................... 15
   2.2 Automated Home Care ................................. 16
   2.3 Requirements Analysis ............................... 17
   2.4 Sensor Technology ................................ 20
   2.5 CommonSens for Automated HCS .................... 21

3 Complex Event Processing And Data Stream Management Systems 25
   3.1 Complex Event Processing ............................ 25
   3.2 Data Stream Management Systems .................... 30
   3.3 Using Data Stream Management Systems for Complex Event
        Processing ......................................... 32
   3.4 Esper for realising CommonSens .................... 32

4 CommonSens 35
   4.1 Event Model ......................................... 35
   4.2 Environment Model .................................. 36
   4.3 Sensor Model ....................................... 37
   4.4 Query Based Event Language ....................... 38
## Contents

5 Esper .......................... 39  
5.1 Event Representation ................. 39  
5.2 Event Stream Queries ............... 44  
  5.2.1 Windows ......................... 44  
  5.2.2 Streams ......................... 45  
  5.2.3 Context ......................... 45  
5.3 Event Patterns ..................... 47  
  5.3.1 Pattern Expression .............. 47  
  5.3.2 Pattern Operators .............. 48  
  5.3.3 Pattern Atoms ................. 51  
5.4 Receiving output data ............. 52  

6 Design .......................... 57  
6.1 Data Models ...................... 57  
6.2 Query Translation ............... 60  
6.3 User Interface ................. 62  

7 Implementation .................... 67  
7.1 Data Models ...................... 67  
  7.1.1 Event Model .................... 67  
  7.1.2 Sensor Model ................... 69  
  7.1.3 Environment Model .......... 72  
7.2 Queries and Query Translation .... 74  
  7.2.1 Query Deconstructor .......... 74  
  7.2.2 Query Constructor .......... 75  
  7.2.3 Queries ...................... 75  
7.3 User Interface ................... 77  
7.4 Workflow ....................... 80  

8 Evaluation ....................... 83  
8.1 Qualitative Tests ................. 83  
  8.1.1 Workload ...................... 85  
  8.1.2 Code ......................... 86  
8.2 Quantitative Tests ............... 88  
  8.2.1 Code ......................... 90  
  8.2.2 Workload ................... 93  
8.3 Results ......................... 94  
8.4 Discussion of Results .......... 95
## Conclusion

9.1 Summary and Contributions .............................................. 99
9.2 Critical Review ............................................................. 100
9.3 Future Work ................................................................. 101

### Bibliography

103

### Appendices

A Code .............................................................................. 109
B Workloads .................................................................. 111
C Results ........................................................................ 117
# Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAL</td>
<td>Ambient Assisted Living</td>
</tr>
<tr>
<td>ADL</td>
<td>Activities of Daily Living</td>
</tr>
<tr>
<td>BAP</td>
<td>Battery Assisted Passive</td>
</tr>
<tr>
<td>EPL</td>
<td>Event Processing Language</td>
</tr>
<tr>
<td>CEP</td>
<td>Complex Event Processing</td>
</tr>
<tr>
<td>CQ</td>
<td>Continuous Query</td>
</tr>
<tr>
<td>DBMS</td>
<td>Data Base Management System</td>
</tr>
<tr>
<td>DSMS</td>
<td>Data Stream Management System</td>
</tr>
<tr>
<td>HCS</td>
<td>Home Care System</td>
</tr>
<tr>
<td>MOM</td>
<td>Message Oriented Middleware</td>
</tr>
<tr>
<td>RFID</td>
<td>Radio-Frequency Identification</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Population pyramids ............................................. 16
3.1 Order of events .................................................. 27
3.2 Generic CEP system architecture .............................. 29
3.3 Generic DSMS architecture ..................................... 31
3.4 Generic CEP system and DSMS architecture .................. 33
5.1 POJO - event representation ..................................... 41
5.2 POJO - properties .................................................. 42
5.4 Map - nested properties ......................................... 42
5.3 Map - event representation ....................................... 43
5.5 Object[] - event representation ............................... 43
5.6 Object[] - nested properties ..................................... 43
5.7 Input stream and output stream .............................. 46
5.8 Example of an event listener in Esper ....................... 52
5.9 Example of a subscriber in Esper ............................ 52
6.1 Conceptual architectural overview of EsperSens ............ 64
7.1 Event hierarchy .................................................... 67
7.2 Generic get- and set method .................................... 68
7.3 UML diagram of the sensing package .......................... 69
7.4 Architectural overview of EsperSens .......................... 77
7.5 Web application - load phase ................................... 78
7.6 Web application - AMQP consumer ............................ 79
7.7 Mathematical and SVG coordinate space ...................... 80
8.1 Web application - environment ................................ 85
8.2 Quantitative Test 1 .............................................. 94
8.3 Quantitative Test 1 - Event processing distribution ....... 96
A.1 EsperSens - AMQP consumer ................................... 110
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Comparison between DBMS and DSMS</td>
<td>30</td>
</tr>
<tr>
<td>5.1</td>
<td>Pattern operators - part 1</td>
<td>54</td>
</tr>
<tr>
<td>5.2</td>
<td>Pattern operators - part 2</td>
<td>55</td>
</tr>
<tr>
<td>8.1</td>
<td>The possible values returned from the functionality tests and</td>
<td>84</td>
</tr>
<tr>
<td></td>
<td>their associated meaning.</td>
<td></td>
</tr>
<tr>
<td>8.2</td>
<td>Quantitative Test 2 - Workload</td>
<td>90</td>
</tr>
<tr>
<td>8.3</td>
<td>Quantitative tests 2</td>
<td>95</td>
</tr>
<tr>
<td>8.4</td>
<td>Quantitative tests 3</td>
<td>95</td>
</tr>
<tr>
<td>B.1</td>
<td>Movement patterns part1</td>
<td>111</td>
</tr>
<tr>
<td>B.2</td>
<td>Movement patterns - part2</td>
<td>112</td>
</tr>
<tr>
<td>B.3</td>
<td>Complex queries - part1</td>
<td>112</td>
</tr>
<tr>
<td>B.4</td>
<td>Complex queries - part2</td>
<td>113</td>
</tr>
<tr>
<td>B.5</td>
<td>Complex queries - part3</td>
<td>113</td>
</tr>
<tr>
<td>B.6</td>
<td>Functionality tests - part1</td>
<td>114</td>
</tr>
<tr>
<td>B.7</td>
<td>Functionality tests - part2</td>
<td>115</td>
</tr>
<tr>
<td>C.1</td>
<td>Functionality tests comparing CommonSens and Esper - part1</td>
<td>117</td>
</tr>
<tr>
<td>C.2</td>
<td>Functionality tests comparing CommonSens and Esper - part2</td>
<td>118</td>
</tr>
<tr>
<td>C.3</td>
<td>Functionality tests comparing CommonSens and Esper - part3</td>
<td>118</td>
</tr>
<tr>
<td>C.4</td>
<td>Functionality tests comparing CommonSens and Esper - part4</td>
<td>119</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Background and Motivation

There is an increase in the ratio of elderlies. Due to this trend, we will not have enough professional caregivers per elderly, which means traditional home care will not suffice in ensuring the elderlies well-being. Therefore alternatives, such as automated home care, must be considered.

Automated home care systems (HCSs), or ambient assisted living (AAL), use sensors that are placed at strategic places in the homes of the elderlies and a system which is able to process the information provided by these sensors. This allows the system to, e.g., notify the elderly to remember to take his or her medicine, or alert the professional caregiver that something potentially dangerous has occurred. The aim of automated home care is to extend the time elderly people can stay in their preferred environment by increasing their autonomy while maintaining their security and well-being.

Sensors by themselves do not explicitly give information to, e.g., the helping personnel, they provide low level data which has been created by converting analogue signals to digital values. Not all sensor readings are of interest, thus there is a need for a system to analyze these readings and filter out only the correct readings that are of interest. One possible approach is to use complex event processing (CEP).

CEP allows a user to associate low level data provided by sensors with events and to further analyze and aggregate these events. It uses a declarative language to understand what happens in the home, e.g., the elderly falling down or forgetting to take the (correct) medicine at the right time. This is done by searching through patterns of events, how they relate to each other and when they occur.

Although sensors can provide events for a CEP system to analyze, sensor
readings may give false information. They may report that motion was detected in the living room, but in reality there was no movement there. This can occur when the coverage area of, e.g., a camera intends to only cover the living room but covers parts of the kitchen too. Thus movement in the kitchen will be interpreted as movement in the living room. The type of signals that sensors use may also be affected by the environment and objects within the environment. Sensors relying on light will have their coverage area limited by non-transparent objects, while sensors relying on radio signals will only be limited by non-metallic objects. In automated home care, issues such as these must be considered.

CommonSens is a multimodal complex event processing system for automated home care that adapts to the home, i.e., the environment, sensors and persons. As an open system, it expects a heterogeneous set of sensors. This means that it needs a model that contains the important properties of the sensors, such as the signal type and the sampling rate. It is also possible to get the actual coverage area, i.e., the coverage area after calculating for obstructing objects, of a sensor given an environment. By combining information from sensors CommonSens is able to eliminate some of the false readings that might occur. It uses an environment model to keep track of locations where interesting events may occur. These key locations may be the same for several homes, although the position may differ. CommonSens has a declarative query language that enables the programmer to describe capabilities, i.e., what sensors are able to detect, rather than specific sensors. It allows the user to query for events without considering where every sensor is placed.

There are some features yet to be implemented in CommonSens for it to work well for automated home care. The first issue is that CommonSens currently supports only one complex query running at a time. This is limiting the automated HCS as there is usually the need to monitor multiple events. The second issue is that CommonSens only considers home environments with a single person in it. This is not always the case as there can be married couples living together, or a single person who has visitors. Another feature CommonSens lacks is a way to report results of queries to the user in a more meaningful and customizable way. In a CEP architecture, event observers provide information as an incoming data stream for the CEP engine and the CEP engine has event sinks to catch when certain events match the query registered. Having event sinks, or event listeners, gives the possibility of doing additional computation with the matching event. It can also give the programmer the ability to choose where to send the notification; if it should be delivered through message-oriented middleware, over the network or to a mobile phone application.
Esper is considered the leading open-source CEP provider. It is not created for any specific domain, but is more generalized so that it can be applied to a wide variety of domains, such as network analysis and assets management. It offers a rich declarative language and has high performance with regards to throughput, latency and accuracy. Good performance with regards to latency and accuracy is essential in automated home care, as the speed and the accuracy of which it detects events, i.e., if the system is able to detect what is actually happening, is important for the elderly’s well-being.

1.2 Problem Statement

In this thesis we will design a solution, EsperSens, which uses Esper as the CEP system to realize the concepts of CommonSens to create a better solution for automated HCS. In order to achieve this, we will first do an analysis of the requirements of automated home care. Based on these requirements we will redesign and reimplement the three fundamental data models of CommonSens; the environment model, the sensor model and the event model with Esper. We will also design a solution for delivering notifications to the helping personnel when something of interest occurs in the home. To determine whether EsperSens satisfies the requirements we will perform tests that measure the correctness and speed of the system.

1.3 Outline

In this master thesis, we start by describing the application domain in Chapter 2. We look at traditional home care and the change in demography to motivate the use of automated home care. Further in this chapter, we investigate the characteristics and requirements of home care and present an analysis of the requirements we will have for EsperSens. We examine the different types of sensor technology and how they work. We end the chapter by looking at how CommonSens analyze the domain and what requirements it has for automated HCS.

In Chapter 3 we describe how we can analyze the information provided by sensors. As this information is a continuous stream of data, rather than static data, both CEP systems and data stream management systems (DSMSs) have tools to work with this type of data. We investigate the capabilities of these systems and discuss how a combination, or a hybrid solution can be used for automated HCS. To conclude the chapter, we show how Esper can be used as the CEP system with the concepts and ideas of CommonSens.
Chapter 4 describes the three fundamental data models in CommonSens and the query based event language. In Chapter 5 we investigate the different features of Esper that can be used to implement the features in CommonSens (described in Chapter 4).

The design of EsperSens is presented in Chapter 6. We look at how we can implement the fundamental data models of CommonSens using the features of Esper and we investigate how we can translate queries so that users may write CommonSens queries, but still be understood by Esper. Different solutions for the user interface are discussed and we present how we intend to implement it. Chapter 7 presents the implementation of the topics discussed in Chapter 6.

The evaluation is presented in Chapter 8. We show how the tests are implemented, our results from the tests and what these results signify.

The last Chapter of this thesis (Chapter 9) is the conclusion. We provide a summary of what we have done and the contributions we have made. We present our critical review of this thesis. Finally, we discuss what we can do for future work.
Chapter 2

Automated Home Care Systems

2.1 Traditional Home Care

Traditional home care allows a person to stay in their own home, rather than living in an institution, such as a nursing home, when that would otherwise be necessary. The person receiving home care is typically an elderly person, although it may also be a young adult suffering from illness, mental or physical disability. Helping personnel or professional caregivers regularly visit the home of the individual being cared for to ensure that the activities of daily living (ADLs) may be carried out. ADLs may include simple errands such as getting dressed, taking the proper medication, or more complicated tasks such as preparing meals or managing money.

A person may require care for many different reasons. They may lack the physical strength to perform their ADLs, or they may suffer from dementia, so they cannot be fully self-reliant. The amount of aid that they require depends on their exact needs.

Unfortunately, the time and availability of professional carers is a limited resource. If a carer does not have enough time to give the proper assistance to an individual, it may limit the potential well-being of that person.

It is well known that the ratio of elderly people will greatly increase in the near future compared to other age groups. According to the United Nations [1], 21.1% of the world’s population will be 60 years or above in year 2050 compared to the 10% in the year 2000. This is illustrated by the population pyramids in Figure 2.1.

Consequently, the demand on professional home care will increase. Unless the availability of professional carers increases accordingly to handle this
change in demography, there is a risk that less time is spent caring for each individual. This in turn may compromise the quality of life of the world’s elderly population.

Therefore, alternatives to the traditional home care must be considered in order to sustain and improve the well-being of the elderly population. One solution is automated home care, also known as Ambient Assisted Living (AAL). The European AAL Joint Programme is a funding activity, supported by 22 countries, that aims to further the research and development of automated home care solutions [2].

Figure 2.1: Population pyramids[1].

2.2 Automated Home Care

Automated home care systems assist in and, in some cases, replace the need for tasks that are traditionally performed by home care professionals. The cared for persons, whether they are elderly or require aid due to a disability, are monitored by the system to observe their ADLs. This is made possible by placing various types of sensors in the home to detect specific events. For example, one may deploy sensors to detect whether or not the cared for persons have taken their medicine. If they have forgotten, the system may remind them, or, alternatively, notify the professional carers. It is also possible to use actuators to automatically perform certain actions, such as adjusting the room temperature or turning off the oven, if it was turned on and left unattended.

In essence, automated home care allows the helping personnel to monitor an individual without being in the home. This is opposed to traditional home care, where the helping personnel is in the home of the person requiring aid and devotes his or her full attention to this individual, which is called active monitoring. Although necessary for the traditional home care approach, active monitoring consumes a great deal of time from the monitored person.
When using automated home care the helping personnel will only be notified about something noteworthy happening, or about events that were expected, but did not take place. This means that automated home care gives the professional more time to care for more people.

The sensors alone are usually not enough to understand what happens in the home. Typically, the information provided by the sensors is consolidated and subjected to some form of analysis before meaningful interpretations can be made. In Chapter 3, we will discuss technologies that can be used to process and analyze the sensor information.

2.3 Requirements Analysis

To successfully deploy and run an automated HCS we must look at the requirements posed in this domain. The home care scenario is a complex domain, involving many stakeholders with many requirements. In order to define these requirements, we must first establish the goal. In essence, the purpose of automated home care is to allow the elderly or people suffering from various impairments or disabilities to live in their own homes while receiving care that ensures their well-being. There are many things to consider in order to achieve this goal. The following is a list of some key characteristics of HCSs [3]:

- **Multi-user:** Anyone involved in the care of the monitored person is a potential user of the HCS. This can be the helping personnel, which is not restricted to a single person, and it can be family or friends who help the monitored person and thus engages with the HCS.

- **Dynamic:** People have different needs depending on their disabilities or impairments. However, their needs are not necessarily constant. Needs and medical condition may change over time [4].

- **Shared interaction spaces:** The person that requires care may share the home with, e.g., a spouse or relatives. These people have their own perspectives, preferences and perhaps impairments.

- **Multimodal:** The use of different devices to provide input to the home care system allows the monitored person a more personalized interaction and more adaptability when installing it to the home.

- **Usability:** The monitored person may not want to put effort into providing the HCS with input. In some cases, the monitored person might
not even be capable of the cognitive effort to provide the HCS with the desired input [5].

There may be many stakeholders involved in home care. These stakeholders can be categorized into the following:

- **The cared**: The person with whom the home care is revolved around. It may be one or more reasons the person requires aid, but commonly it is because their age has affected their physical or mental abilities, or they are suffering from various disabilities.

- **The carer**: This may be a helping personnel who provides assistance to the the cared/monitored person. The responsibilities the helping personnel has for the monitored person is medical, but it may also serve a social purpose. However, there can be others who serves as a carer. The family can support the monitored person and, in many cases, the spouse is the primary carer. This is the person who is with the monitored person most of the day. Although the spouse may not necessarily be able to provide enough assistance and may even require some himself/herself.

- **Visitors**: This category includes social workers, paramedics, relatives, community nurses etc who visits the monitored person when possible or necessary. The distinction between this category and the carer category might not be clear-cut. For example, a non-residential carer could be in both categories. However, by differentiating between visits at regular intervals and sporadic visits, we could categorize them as a carer or visitor, respectively.

- **Remote users**: The HCS may provide some method of remote communication that allows people to connect to the cared person. Family members unable to frequently visit the cared person might use the HCS system to maintain awareness of the condition of the cared person. Helping personnel may also use remote communication to ensure the well-being of the cared person.

- **Institutional stakeholders**: People who are not direct users of the HCS may still have an interest and have the possibility to influence the context of the HCS. For instance, a landlord might place constraints on the technology used, especially if it influences the other apartments in the housing complex.
Based on the characteristics of home care and the stakeholders involved in home care, we present the following as requirements of automated home care:

- **Well-being:** The primary goal of an automated home care system is to allow the monitored person to stay in a familiar environment and ensuring their well-being, and possibly increasing their well-being compared to traditional home care. To ensure their well-being, the automated HCS must be able to detect events that are of interest and it must be able to notify the people responsible in near real-time.

- **Non-intrusive:** The deployment should not interfere with the daily life of the monitored person. However, the system should be able to deliver notifications to the monitored person when he or she has, e.g., forgotten something. It is important that it should not require too much effort to receive the message. For example, notifications could be sent to the mobile phone, but some individuals may suffer from cognitive disabilities, which means this type of interaction may prove to be too extraneous.

- **Adaptive:** A person’s condition is not necessarily permanent. The condition might get better or worse and the system should ideally adjust itself. However, this is not a trivial task. This requires the system to be able to detect changes in the condition of the monitored person and know what should be adjusted. Advanced pattern recognition and machine learning must be utilized in order to fully satisfy this requirement. As a temporary solution, the system should be able to take input from either the application programmer or the helping personnel in order to handle the change in condition of the monitored person.

- **Shared space:** The home should be able to distinguish between the person it is monitoring and other residents or visitors. This is important as to not trigger events for the monitored person when in reality it was actually the monitored person’s spouse.

- **Security:** It is important that the information provided by sensors and through analysis of sensor readings is not accessible by unauthorized people. If the necessary precautions are not met, ill-intended people may take advantage of the situation.

There are three basic consequences of not considering or properly identifying the requirements [3]. The first consequence, and the most important one, is that the system may fail in delivering the services it is intended for.
This means that it may be unable to properly detect what happens in the home, thus lowering the well being of the monitored person. The second consequence is poor usability and unintentional misuse. In order to provide a more natural user experience the automated HCS should be capable of providing implicit and multimodal means of interaction [6]. By integrating the automated HCS with standard devices in the home, the experience and existing knowledge of the monitored person can be used. The last basic consequence, which relates to the second consequence, is the reluctance to use the system. Poor usability and obtrusive integration may affect the attitudes of the monitored persons and reduce their inclination to use the system.

2.4 Sensor Technology

Advances in the field of sensor technology have given us smaller and more efficient sensors at affordable prices. Sensors can be found in many things such as cars, refrigerators, computers, mobile devices, tags/cards, doors, animals, security systems etc. There are many types of sensors such as motion sensors, temperature sensors, weight sensors, humidity sensors, radio-frequency identification (RFID) sensors, cameras etc. Some devices, such as smartphones, may have multiple sensors e.g.: GPS, accelerometer and gyroscope. These sensors can be divided into three categories [7]; RFID tags/readers, programmable sensors and sensors that require substantial processing to give meaningful results.

- **RFID**: RFID tags can be quite small and can be placed in many different items like cards and clothing. There are three types of tags: active, passive and battery assisted passive (BAP). Active tags have a battery attached and sends an ID signal at regular intervals. Passive tags have no battery, but uses energy from the radio energy transmitted by RFID readers instead. BAP tags have a small battery attached and are activated when in the presence of a RFID reader.

- **Programmable sensors**: Programmable sensors consists of a sensing unit, processing unit, transceiver unit and power units (although not necessary if connected to a power line). These sensors have a local processor which computes the information the sensor provides. The information is only single values such as temperature readings, on/off states for lights, open/close states for doors etc.

- **”Advanced” sensors**: Sensors that produce complex data types (arrays/data objects) require substantial processing before any meaningful can be interpreted from it. E.g. cameras have many pixels of data
which constitutes a frame and there are many frames per second. These must be analysed before being able to tell if, for example, motion has been detected.

Sensors can use batteries or be connected by wire. By using a battery the sensor can be mobile, but after a certain period of time the battery needs to be changed or recharged. How long a sensor using a battery can last depends on how often the sensor sends information to the system. It is possible to extend the time it is active by having the sensor sleep during inactive periods. If the sensor is connected by wire, it can continuously observe and provide readings. However, it is limited to a stationary location.

The way sensors communicate with the system can either be wired or wireless. The same limitation for sensors using wire for electricity applies here. Wired communication is more secure than wireless as information cannot be intercepted as easily, i.e., the wire must be "tapped" or the system hacked for information to fall in the wrong hands. Wireless communication allows the sensors to be mobile but network packages can be intercepted which means information about what happens in the home can be known to, e.g., a thief/robber.

Regardless of the type of sensors, they can all give inaccurate readings. That means false positives and false negatives may occur. When sensors report false positives they registered something that did not actually happen or a false state. Examples may be reporting motion in a room when there is no activity in the room or that the temperature in the oven has reached a critical high point while in reality it is not even turned on. False negatives is the opposite; when something actually does happen, but the sensor fails to register it. An example may be that the oven is on fire but the sensor registers it as a normal temperature. False readings are unwanted, where false negatives are the least desirable in the home care scenario. Given false negatives help may arrive too late, however, with false positives there is only the economic cost of dispatching help to the monitored person.

2.5 CommonSens for Automated HCS

CommonSens identifies three important roles in automated HCS [7].

- **Monitored person**: The person who has some disabilities or impairments and requires care. Sensors are used to observe the monitored person to ensure that everything is as expected, thus ensuring his/her well being.
2.5. CommonSens for Automated HCS

- **Helping personnel:** The person, or persons, who interacts and takes care of the monitored person. The helping personnel knows the needs and requirements of the monitored person, and conveys this knowledge to the application programmer.

- **Application programmer:** The person who implements the automated HCS. The application programmer is the one responsible for placing sensors in the home of the monitored person and ensures that they cover key locations. By instantiating the environment, the application programmer can make sure that the sensors are able to obtain the required information. He/she does not need to have extensive knowledge in low level programming of sensors to achieve this.

Based on current research and observations in the field of automated home care, CommonSens emphasises two user requirements and two system requirements.

1. **Safety:** One of the reasons of using automated home care is ensuring that nothing bad happens with the monitored person while the helping personnel is not present. The system must be able to detect possibly dangerous events, e.g., if the monitored person falls and is injured, and alert the helping personnel. Situations which are not as dire, can notify either the monitored person or the helping personnel, depending on the situation.

2. **Non-intrusive deployment:** The automated HCS should not interfere with the monitored person’s daily routine. This means that the automated HCS should not require direct input by the monitored person in order to provide the safety for the monitored person. The method of notifying the monitored person to, e.g., take medicine should be considered. One solution would be to have the system send a notification to the monitored person smartphone, however, the monitored person may not notice the notification which conflicts with the safety requirement.

The two system requirements are as follows:

1. **Personalisation:** The system should be suited to the needs and requirements of the monitored person. However, in order for the automated HCS to be suited for large scale deployment the system must also be easy to instantiate.

2. **Near real-time detection of dangerous and critical situations:** The system should immediately report to the appropriate personnel,
e.g., helping personnel or paramedics, when something dangerous or critical occurs.

One of the goals of CommonSens is to simplify the work of the application programmer. This is done by having a query language that provides the abstraction of capabilities and location of interests (LoIs) that facilitates the task of specifying that [8]. By having a repository of query templates, the application programmer can simply choose, after consulting with the helping personnel, which is suited for the monitored person. If there are no query templates suited for the monitored person, the application programmer can either adjust a query template or write a new one. Considering that homes mostly have the same type of rooms and can contain same type of objects, it is possible to reuse these from a repository too. The application programmer does not need an extensive knowledge of low level sensor programming. By using a simple interface, the system can display the coverage area of sensors and the possibility of false positives, which helps the application programmer placing the sensors correctly. All this speeds up the process of instantiating the automated HCS which coincides with the goal of making the automated HCS suited for large scale deployment.

CommonSens is an open system and as such, it expects a heterogeneous set of sensors. This means that it uses a model that contains the important properties of the sensors, e.g., signal type and sampling rate. The sensor model also contains the capability of the sensor, i.e., a description of what the sensor can observe. Sensors provide the system with information and CommonSens has divided the sensor model into three types based on how they provide information. The first sensor type is the physical sensor, which is the sensor placed in the home of the monitored person. It converts analogue signals into digital values and sends it to the system. The second sensor type is the logical sensor, which provides input to the system by aggregating information provided by other sensors. The last sensor type is the external source, which is static data.
2.5. CommonSens for Automated HCS
Chapter 3

Complex Event Processing And Data Stream Management Systems

CEP systems and DSMSs are similar because they both process data streams and produces results from them. However, DSMSs leave clients responsible of associating a semantics to the data being processed while CEP systems associate a precise semantics to the information being processed [9]. CEP systems are also responsible for analyzing data from the stream to give an understanding of what happens.

3.1 Complex Event Processing

A basic definition of an event is "a thing that happens or takes place, especially one of importance" (Oxford dictionary). The key word in this definition is importance - what may be regarded as important depends on the domain.

Carlson defines an event as a "particular type of action or change that is of interest to the system, either occurring internally within the system or externally in the environment with which the system interacts" [10]. This definition creates a scope for where events may occur. Events may occur either internally in a system or in the environment with which the system interacts. It also regards events as something of significance, not just anything that can happen.

The event definition that Luckham uses is that an event is "an object that is a record of an activity in a system" [11]. An activity is something that happens that an information source has made a record of. Luckham assigns three aspects to an event:
3.1. Complex Event Processing

- **Form** - The form of an event is an object, which contains one or more attributes or data components. Events may have both temporal and spatial properties, such that an event may contain an integer to represent the timestamp and a string to represent the location of the event.

- **Significance** - The significance of an event is the activity the event represents. The form of an event usually contains an attribute describing the activity the event represents.

- **Relativity** - Activities, or events, may be related to each other. They may be related to each other by time, causality and aggregation. Relativity is the relationship between an event and other events.

These events are created in the following steps[11]:

- **Observation step** - The system needs to be able to observe activities at any level of the hierarchical system.

- **Adaptation step** - What the system observes must be translated into events so that it can be processed by the CEP engine.

A CEP system may receive events from many different sources, e.g., it can receive events from message-oriented middleware (MOM), from a network or from sensors. Luckham assigns three principal sources of events:

- **IT layer** - The components in an IT layer, such as MOM and databases, may provide the CEP system with events.

- **Instrumentation** - Components of a system may generate events to signify activities. It can be an Operating System (OS) providing information such as disk activity, memory performance and CPU usage. Network-level instrumentation can signify node failures, traffic overloads, network packages from certain addresses. Sensors, e.g., motion sensors or temperature sensors, can provide a system with events from the real world.

- **CEP** - Events can be created by the CEP system itself. In some cases, if a certain sequence of specific events signify another event, the CEP system aggregates the set of lower-level events into a higher event.
There are three important types of relationships that can exist between events:

- **Time** - Events have timestamps which gives us an order when events occur, i.e., timestamps can tell us if an event A happened before an event B. Babcock et al. differentiate between two types of timestamps: implicit and explicit timestamps [12]. Implicit timestamps are when the system adds a data attribute to every incoming tuple, while when tuples already have a data attribute designated as a timestamp it is called explicit timestamps. The drawback with explicit timestamps is that tuples with later timestamps may arrive before tuples with earlier timestamps, as seen in Figure 3.1. However, it is possible to buffer events so that the CEP system can process the tuples in the correct order. When using implicit timestamps the CEP system will always process tuples in order of their timestamps, which is the order they are received by the CEP system. However, this may not be consistent with the causal order, i.e., the order of when the tuples were detected by information sources.

An event may have multiple timestamps, e.g., if an event happens in a time interval, it may contain a timestamp for when the event begins and a timestamp for when it ends [7, 11]. Events may be detected or registered by multiple observers, thus the need of timestamps representing each observer may be appropriate [11].

- **Cause** - Events that are related with each other through causality have a dependency between them. The causal relationship between events

Figure 3.1: Order of events.
is both asymmetric and transitive [11]. The relationship is asymmetric because if an event A caused an event B, event B cannot have caused event A. It is also transitive because if an event A caused an event B, and B caused an event C, then C was transitively caused by A.

- **Aggregation** - When an event A consists of a set of other events B, then A is an aggregation of all the events in B. Event A is a complex event where its members are the events that caused event A.

Events can happen consecutively or concurrently. In CommonSens, there are five classes of concurrency and one class for consecutiveness [7]. When two events are consecutive there are no overlapping periods of time, i.e. only when an event A ends can event B start.

\[ e_A t_e < e_B t_b \]  \hspace{1cm} (3.1)

\( t_b \) is the timestamp for when an event begins, \( t_e \) is the timestamp for when an event ends and \( e_X \) is an event X. The five classes of concurrency are:

- **Equals** - When two events have the same start time and end time, we say A equals B.

\[ (e_A t_b = e_B t_b) \land (e_A t_e = e_B t_e) \]  \hspace{1cm} (3.2)

- **Starts** - If two events have the same start time, but different end times.

\[ (e_A t_b = e_B t_b) \land (e_A t_e < e_B t_e) \]  \hspace{1cm} (3.3)

- **Finishes** - If two events have different start times, but the same end time.

\[ (e_A t_b > e_B t_b) \land (e_A t_e = e_B t_e) \]  \hspace{1cm} (3.4)

- **Overlaps** - When an event A has a start time between B’s start time and end time, but A’s end time is after B’s end time.

\[ e_B t_b < e_A t_b < e_B t_e < e_A t_e \]  \hspace{1cm} (3.5)

- **During** - When an event is contained within the duration of another event.

\[ e_B t_b < e_A t_b < e_A t_e < e_B t_e \]  \hspace{1cm} (3.6)
Events are commonly divided into two groups: atomic events and complex events, also known as compound or composite events [7, 11, 9]. Atrey et al. states that an atomic event is an object involved in exactly one activity over a period of time [13]. While a complex event is "[...] an event that could only happen if lots of other events happened" [11]. That is, a complex event is a set of (two or more) events. These events may either be atomic or complex, which introduces an event hierarchy. For example, if a complex event C consists of two other complex events, C would be considered a "higher level" complex event than the others.

To be able to detect complex events, we need to be able to describe the pattern of events that the complex event consists of. One approach is to use an event pattern language (EPL). It is similar to using, e.g., grep, which allows a user to search for string patterns in files. It is possible to specify which characters you want to appear in the string and it is also possible to specify variables. These variables can, e.g., match any character or any sequence of characters. When looking for event patterns, it is not enough to merely specify events, but also their causal dependencies, timing and data parameters. An example would be to search for the appropriate time to buy stocks, which may depend on when certain other stocks increases in value while some stocks plummets in value.

![Figure 3.2: Generic CEP system architecture.](image)

CEP systems have an interaction style known as publish-subscribe [9, 14]. It is a message-oriented interaction paradigm where those who send messages, called publishers, do not directly send the message to the receiver, called the subscriber. Instead, the subscriber express interest in certain classes of messages and the publisher sends messages categorized into certain classes. The messaging system will ensure that the subscribers will receive messages based on their interest.

In CEP systems, publishers can be seen as event observers and subscribers can be seen as event consumers, which is illustrated in Figure 3.2. Event observers detect what happens in the external world and sends notifications of events to the engine. Event consumers receive events that they have declared.
interest in. A key difference between CEP systems and a traditional publish-subscribe system is that, in most cases, the events sent by the event observers are not directly passed to the event consumers. Events are usually filtered and aggregated by the CEP engine to get a better understanding of what happens. Events that have been filtered and aggregated are called complex events. The event consumers “subscribe” to particular complex events. CEP systems can be seen as an extension to the traditional publish-subscribe paradigm, where subscribers express interest in the aggregated information provided by publishers.

3.2 Data Stream Management Systems

A DSMS is a type of system to manage large and continuous streams of data. It is similar to a DBMS, but has some key differences (see the summary in Table 3.1). For one, a DBMS works mostly with static data that is stored on disk while a DSMS works with a continuous stream of data where it temporarily stores parts of the data stream in main memory to process it and when it is done, it discards or archives the data. This is because in most use-cases of a DSMS, users have real-time requirements, such as financial tickers where they need to identify trends and discover correlations as fast as possible to know whether to sell or buy stocks.

<table>
<thead>
<tr>
<th>DBMS</th>
<th>DSMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static data</td>
<td>Continuous stream</td>
</tr>
<tr>
<td>One-time queries</td>
<td>Continuous queries (CQs)</td>
</tr>
<tr>
<td>Random access</td>
<td>Sequential access</td>
</tr>
<tr>
<td>&quot;Unbounded&quot; disk store</td>
<td>Bounded main memory</td>
</tr>
<tr>
<td>Current state of data</td>
<td>Historical data</td>
</tr>
<tr>
<td>No real-time requirements</td>
<td>Real-time requirements</td>
</tr>
<tr>
<td>Assume precise data</td>
<td>Approximation / Data imprecise</td>
</tr>
<tr>
<td>Relatively low update-rate</td>
<td>Possibly multi-GB arrival rate</td>
</tr>
</tbody>
</table>

Table 3.1: Comparison between DBMS and DSMS.

There is, in most cases, no correlation between when data was added to a DBMS and when it is accessed. In a DSMS, however, data is processed sequentially, i.e., when data arrives it is processed immediately and data arriving later will never be processed before earlier data.

In a DBMS, the data represents only the current state. However, how data has changed over time is of interest to a DSMS. This is also represented
in how the two systems queries for data. A DBMS uses one-time queries, i.e., queries that are evaluated once and returns an answer to the user. While a DSMS uses continuous queries (CQs) [12, 9, 15], also known as standing queries. CQs continuously produce an answer which reflects what the engine has processed thus far. This introduces a new type of interaction model where users do not explicitly ask for updated information, instead they register CQs with the system which will report when the sought after information occurs (see Figure 3.3). This form of interaction is commonly called database-active human-passive [16]. In traditional DBMSs one-time queries are used to fetch the desired information. This type of interaction is usually called human-active database-passive.

![Diagram](image)

Figure 3.3: Generic DSMS architecture.

Data being analysed stays in memory for only as long as it needs to be processed, but some aggregation operators, e.g., SUM, AVG, MIN and MAX, are blocking, i.e., processing will potentially go on forever as the data stream is potentially “infinite”. To be able to use aggregations a DSMS uses the concept of windows. Windows is a technique to divide the “infinitely” large stream into finite parts of data. There are different ways for windows to divide an infinite stream:

- **Windows based on time**: A window that lasts for a specified period of time. It gathers all data tuples that occur during this period of time and does the desired aggregation on this finite set of data tuples. Although this technique gives control over how long a window should last, it does not control how many data tuples will be received during this time. That means the aggregation could be done for a window with only one data tuple or a window with 1000 data tuples.

- **Windows with fixed data tuples size**: A window that lasts until the specified amount of data tuples have been received. This technique allows the user to specify a fixed size of data tuples per window, but
3.3 Using Data Stream Management Systems for Complex Event Processing

offers no control over the time used. That means a window with a limit of 100 data tuples could be filled in one second or it could take one hour.

- **Adaptive sized windows:** A window that has its window size (can be time based or tuples based) determined by the value from, e.g., another data stream [17].

3.3 Using Data Stream Management Systems for Complex Event Processing

A DSMS alone is not enough for automated home care systems. In home care, the order of events is important and if events occur concurrently. For example, if the oven has been turned on (without being turned off) followed by the monitored person laying down on the bed and not getting up within 30 seconds, we want the system to give a notification as this may indicate that the monitored person has forgotten about the oven.

A traditional DSMS lacks the ability to determine a concurrent or consecutive relation between events. However, there are some features that are of interest. A DSMS provides several types of aggregations, such as `COUNT` and `AVG`. These aggregations can be used to, e.g., analyze the trends of patterns of a monitored person. As specified in Chapter 2.3, the needs of the monitored person may change over time and having the system automatically detect this change can improve the well-being of the individual.

If we use functionality from both DSMS and CEP systems it is possible to do concurrent or consecutive relations with aggregations, such as `COUNT` and `AVG`. However, when doing aggregations it is also necessary to have some means of dividing the data stream, which a DSMS provides through windows.

By using a CEP system it is possible to associate events with the data produced from CQs. Thus, it is possible to continuously search for interesting events and attach event consumers to the CQs (see figure 3.4). Event consumers are responsible of notifying either the monitored person or the helping personnel when a certain event has been detected.

3.4 Esper for realising CommonSens

CommonSens queries for events by specifying the type of event (capability), location, time and a percentage that specifies how long of the time duration should satisfy the condition. The time specifies either a specific point in
Chapter 3. Complex Event Processing And Data Stream Management Systems

Figure 3.4: Generic CEP system and DSMS architecture.

time (with duration), or duration starting from when the first event occurs. These events can be related through logical operators, by concurrency and by consecutiveness.

Esper allows the user to combine pattern searches, as seen in CEP systems, and (continuous) queries, as seen in DSMSs. The queries and patterns can be registered with a listener, which serves as an event consumer. When the queries and patterns evaluate to true, they will alert the listener. If specified, the listener will receive relevant information of the events involved and can do further processing or notify the helping personnel or monitored person.

We choose to use Esper because it is the leading open source CEP system [9] and is easily customizable, e.g., events can be modelled in many different ways and there are different ways of providing notifications (see Chapter 5). It is designed in a generalized way, in order for it to be applicable to multiple domains, thereby reducing the overhead one might have from a system that is specifically designed for a domain other than home care. This also enables us to implement the concepts and models of CommonSens mostly unhindered as there are no opposing concepts, except for, perhaps, the event model. As a minor benefit, both CommonSens and Esper are written in Java, which might prove to be a more natural process for implementing the models of CommonSens with Esper.

CommonSens and Esper will be discussed further in the following chapters (Chapter 4 and Chapter 5).
3.4. Esper for realising CommonSens
Chapter 4
CommonSens

4.1 Event Model

CommonSens defines an event as "a state or state transition in which someone has declared interest". A state can, e.g., be the humidity or the temperature in a room. Where a state transition can be a light with state "on" that changes to the state "off". States or state transitions that are of interest are described by the application programmer using declarative queries.

An atomic event consists of four attributes and is defined as follows:

$$e_A = (e, loi, t_b, t_e),$$

(4.1)

where $e$ is the event, $loi$ is the location of interest (if specified), $t_b$ and $t_e$ are the timestamps for when the event begins and when the event ends, respectively. A location of interest (LoI) is a set of coordinates that describes the boundaries of a location in the home where events may occur. A timestamp is an element that specifies a specific point in time.

A complex event is defined as follows:

$$e_C = \{e_{A0}, ..., e_{AN-1} \},$$

(4.2)

where $e_C$ is the complex event, $e_{AN}$ are atomic events and $N$ specifies the number of atomic events. CommonSens defines five classes of concurrency and one class of consecutiveness of which atomic events can be related to each other. The five concurrency classes are: equals, starts, finishes, during and overlaps (described in Chapter 3.1). The consecutive class is called before and denotes when an event has occurred (and completed) another event occurs.

The application programmer may specify patterns of events (a complex event) to describe things that may go wrong or behavior the monitored person should avoid. However, there are many things that may go wrong and it is
implausible to specify every single pattern of "bad" behaviors. Therefore, in some cases, it is better to specify the deviation of a desired (good) behavior. CommonSens offers deviation detection, which is the process of identifying the absence of events [18]. For instance, the application programmer may want to know when the temperature in the living room is either too warm or too cold. This can easily be done by specifying the desired temperature of the room and ask for deviations from this state.

A deviation can be specified as follows:

\[ D = N / E, \] (4.3)

where \( D \) is a set of data tuple sequences (the deviation), \( N \) is a set that contains all possible sequences of state values that might be read by the sensors that instantiate a given complex query and \( E \) contains all sequences of data tuples that match the query.

### 4.2 Environment Model

The environment model is used to describe the home, or the physical space that the monitored person inhabits. This includes LoIs, people that live in the home and objects, such as walls and furnitures. This enables CommonSens to identify how the signals of sensors are affected so that it is possible to determine the coverage area of sensors.

An environment consists of many objects. Objects consists of two core properties: the shape and the permeability. The shape is a set of coordinates that describes the boundary of the shape. The shape is defined as follows:

\[ s = \{(x, y, z)_{0}, \ldots, (x, y, z)_{N-1}\}, \] (4.4)

where \( s \) is the set, \( (x, y, z)_{N} \) are the coordinates of the shape and \( N \) is the amount of coordinates. The permeability describes how permeable an object is to different signal types. For instance, a wall is not permeable to light, but is permeable to radio signals. The permeability is defined as follows:

\[ p = (val, \gamma), \] (4.5)

where \( p \) is a tuple, \( val \) specifies how permeable the object is to the signal type (\( val \in [-1, 1] \)) and \( \gamma \) is the signal type. If \( val \) is 0, the signal cannot pass the object, if \( val \) is positive the signal passes through the object and if \( val \) is negative the signal is reflected.

Objects may be reused in multiple environments. For instance, a wall of concrete in one environment will have the same permeability as a concrete
wall in another apartment; only their physical dimensions (their shape) differs. The same applies for LoIs. The same locations may be used multiple times, but their shape may differ to suit the layout of the home environment better.

### 4.3 Sensor Model

CommonSens uses the sensor model to achieve three objectives. The first objective is that CommonSens should be able to determine which sensors should be used for detecting specified events. The second, as events may happen over time, sensors should be able to utilize historical data in conjunction with current data tuples. The last objective is to be able to aggregate data tuples from multiple sources.

To achieve the first objective, every sensor in CommonSens provides a set of capabilities. A capability is a state variable a sensor can detect, e.g., motion or temperature. This enables the application programmer to only specify states that are of interest without regarding which sensors have the ability to detect these states. To meet the last two objectives CommonSens has defined three types of sensors: the physical sensor, the logical sensor and the external source.

The physical sensor is responsible for converting analogue signals into data tuples and is only able to detect single states in the home. It is defined as follows:

\[
\phi^P = (\text{cov}, \gamma, f, C),
\]

(4.6)

where \(\phi^P\) is the physical sensor, \(\text{cov}\) denotes the coverage area of the physical sensor, \(\gamma\) is the signal type it sends, \(f\) is the maximal frequency and \(C\) is the set of capabilities the sensor provides.

The external source only contains a single attribute and is defined as follows:

\[
\phi^E = (C),
\]

(4.7)

where \(\phi^E\) is the external source and \(C\) is the set of capabilities the external source provides. The external source does not obtain readings directly from the environment, but rather from historical and stored data, e.g., from a DBMS or files.

The logical sensor is responsible for aggregating information from multiple sources, e.g., historical data with recent data tuples. It is defined as follows:

\[
\phi^L = (C_d, \text{agg}^L, C_p, f),
\]

(4.8)

where \(\phi^L\) is the logical sensor, \(C_d\) is the set of all capabilities the sensor depends on, \(\text{agg}^L\) is the user defined function that aggregates the data tuples
from $C_d$. $C_p$ is the set of capabilities the logical sensor provides, i.e., the type of information it provides which is the result from $agg^l$. $f$ is defined as for the physical sensor, but it depends on $agg^l$ and $C_d$.

## 4.4 Query Based Event Language

The query language of CommonSens is designed with reuseability in mind. To describe an event the application programmer only needs to address the capability, LoI and temporal properties. This allows the query to be used in different environments as CommonSens does not specify where the information should come from. That means that although the physical space a given LoI occupies may vary for each home environment, the name/label of the LoI may still be used in a query.

A query that addresses a single atomic event is called an atomic query and is defined as follows:

$$q_A = (\text{cond}, \text{loi}, t_b, t_e, \text{preg}),$$

(4.9)

where $q_A$ is the atomic query. $\text{cond}$ is the condition that consists of three attributes: $(c, \text{op}, \text{val})$. $c$ is the capability, $\text{op}$ is the operator and $\text{val}$ is the expected value of the capability. $\text{loi}$ is the location of interest, $t_b$ and $t_e$ denotes when the event begins and when it ends, respectively, and $\text{preg}$ specifies the amount of data tuples should evaluate to true for the query to evaluate to true.

A complex query is a list of atomic queries and logical operators and is defined as follows:

$$q_C = (q_{A0} p_0, \ldots, p_{N-2} q_{AN-1}),$$

(4.10)

where $q_C$ is the complex query, $q_{AN}$ are the atomic queries $p_0$ are the logical operators and $N$ is the amount of atomic queries. The logical queries in the query language of CommonSens are $\land$, $\lor$ and $\neg$. An example of a complex query is that after the monitored person has used the toilet, the toilet should be flushed and the monitored person should wash his or her hands before leaving the bathroom. This complex query needs the followed by $(\rightarrow)$ query relation in order to specify that an event should happen only after another event.
Chapter 5

Esper

Esper is considered the leading open-source CEP provider [9]. It is designed by EsperTech and allows for processing event streams and doing event pattern recognition in Java and .NET (as NEsper).

The Esper engine works a bit like "a database turned upside-down" [19]. It stores queries and has data flowing through it rather than storing data and having queries run against the stored data (as in the case of a traditional database).

There are two methods of processing events in Esper: event patterns and event stream queries. The event pattern language is used to specify expression-based event pattern matching. The engine that implements this method is a state machine. The event stream queries provide windows, aggregation, joining and analysis functions, which resembles SQL in its syntax. It is possible to have a statement that uses both these methods, i.e., a statement that uses event patterns and event stream queries.

5.1 Event Representation

An event is an immutable record of something that has occurred (either an action or a state change). Events uses event properties to capture the state information of an event. There are four types of properties that are supported:

- **Simple**: A simple property is a primitive in the Java language, such as a long or a String. There is only a single value that can be retrieved.

- **Indexed**: An indexed property is a collection of objects, which must be of the same type that is accessed by a non-negative integer. The get-method could either return the whole collection in the form of an
array or an iterable, or it may return a single value but it requires an index as a parameter.

- **Mapped**: A mapped property is a collection of objects, which must be of the same type, and is accessed by a string (a key).

- **Nested**: A nested property is not directly accessible. It resides in another class and the class object itself is a property of an event.

There are six approaches on how to represent an event:

- **Plain Old Java Objects (POJOs)**: Any Java POJO, which provides get- and set-methods following JavaBean conventions.

- **Map**: Events using the `java.util.Map` interface where each map entry is a property value.

- **Object-array**: Two arrays (of type `Object[]`) representing property values and property names.

- **XML document object model (DOM)**: Events represented as `org.w3c.dom.Node` instances.

- **XML - Streaming API for XML (StAX)**: Apache Axiom provides access to XML event data on top of StAX.

- **Application classes**: Plug-in event representation via the extension API.

According to the documentation provided by Esper, XML event representations do not have comparable performance with POJO, Map and Object-array event representations. In the home care domain, it is crucial that the system processes events as fast as possible in order to detect when something dangerous occurs in real-time. For this reason, we chose to compare the performance of the POJO, Map and Object-array event representations and investigate which is most suited for the home care domain.

Plain old java objects (POJOs) are object instances that uses get- and set-methods to access and update, respectively, event properties. Events that share properties can use a superclass and inherit these properties, which means class hierarchy can be used to represent event hierarchy.

When accessing properties through an EPL statement the get-methods for those properties are called. The engine assumes that the property has a function with the property name appended to get, meaning a property named `temperature` would have a function called `getTemperature()`.
public class BaseEvent {
    private long timestamp;
    private String LoI;
    public BaseEvent(long timestamp, String LoI) {
        this.timestamp = timestamp;
        this.LoI = LoI;
    }
    public long getTimestamp() {
        return this.timestamp;
    }
    public String getLoI() {
        return this.LoI;
    }
    public void setTimestamp(long timestamp) {
        this.timestamp = timestamp;
    }
    public void setLoI(String LoI) {
        this.LoI = LoI;
    }
}

Figure 5.1: POJO - Base Event.

A set-method is needed if an EPL statement updates the properties of an event, however only indexed or mapped properties are possible. It has similar syntax to the get-method; setProperty\(\text{PropertyName}(index, \text{newValue})\). When updating the properties we need to specify where we can find the value we want to change, \(\text{index}\), and what the new value will be, \(\text{newValue}\). \(\text{index}\) is an \texttt{Integer} or a \texttt{String} if the property is indexed or mapped, respectively. The \texttt{newValue} must be of the same type as the previous value. An event with get- and set-methods is illustrated in Figure 5.1.

Events can be modelled as \texttt{Map} objects through the \texttt{java.util.Map} interface (see Figure 5.3). The event properties are the values in the map and are accessed through the get-method exposed by the interface. As with POJO events, Map events may have superclasses, or supertype Maps, which allows for event hierarchy.

Map events have four property types, similar to POJO events. The difference is that a map event may inherit properties from more than one (supertype) map event, which is illustrated in Figure 5.4.
public class Personnel {
    private long id;
    //more properties + get methods
}

public class MovementEvent extends BaseEvent {
    private boolean motion;
    private Personnel[] detectedPersonnel;
    public MovementEvent(long timestamp, String LoI, Personnel[] dp, boolean motion) {
        super(timestamp, LoI);
        detectedPersonnel = dp;
        this.motion = motion;
    }
    //An indexed and nested property.
    public Personnel getPersonnel(int index) {
        return detectedPersonnel[index];
    }
    //A mapped and nested property.
    public Personnel[] getAllPersonnel() {
        return detectedPersonnel;
    }
    //A simple property.
    public boolean getMotion() {
        return motion;
    }
}

Figure 5.2: An example of properties used in the POJO event representation.

//Define Map event type
Map<String, Object> movementEvent = new HashMap<String, Object>
movementEvent.put("motion", boolean.class);
//Personnel is a user defined class -> Figure 5.2
movementEvent.put("detectedPersonnel", Personnel[].class);
//Add the base event type to the movement,
//so all properties are inherited
movementEvent.put("baseEvent", baseEvent);
//May inherit from multiple map events.
movementEvent.put("someOtherEvent", someOtherEvent);

//Register the movement Map event type
epService.getEPAAdministrator().getConfiguration().addEventType("MovementEvent", movementEvent);

Figure 5.4: Map - nested properties.
Object-array events are represented using two arrays; (1) a string array which contains the property names and (2) an object array which contains the property types (see Figure 5.5). This event representation may only have a single supertype. The supertype is added in the same way as when adding properties (see Figure 5.6).

```java
// Define Map event type
Map<String, Object> baseEvent = new HashMap<String, Object>;
baseEvent.put("timestamp", long.class);
baseEvent.put("loi", String.class);

// Register the Map event type
epService.getEPAdministrator().getConfiguration().addEventType("BaseEvent", baseEvent);
```

**Figure 5.3:** Map - Define Map and register with configuration.

```java
// The property names
String[] propertyNames = {"timestamp", "loi");

// The property types (should match order of property names)
Object[] propertyTypes = {long.class, String.class};

// Register the Object-array event type
epService.getEPAdministrator().getConfiguration().addEventType("BaseEvent", propertyNames, propertyTypes);
```

**Figure 5.5:** Object[] - Define Object-array and register with configuration.

```java
// Showing simple, nested and index property types
String[] propertyNames_movement = {"base", "motion", "detectedPersonnel"};
Object[] propertyTypes_movement = {"BaseEvent", boolean.class,
Personnel[].class};

epService.getEPAdministrator().getConfiguration().addEventType("MovementEvent", propertyNames_movement,
propertyTypes_movement);
```

**Figure 5.6:** Object[] - nested properties.
5.2 Event Stream Queries

The event processing language is similar to SQL in its syntax, containing clauses such as `select`, `from`, `where` and `order by` and aggregations such as `sum`, `max` and `avg`. In a traditional DBMS a query would fetch information from one or more tables, but in Esper the query fetches information from one or more streams of events. A simple query can be seen below:

\[
select * \text{ from eventA}
\] (5.1)

This engine will invoke the listeners/subscribers every time an `eventA` occurs on the infinite stream. However, if the query would do an aggregation on the infinite stream the listeners/subscribers will never be invoked as there is no end to the data of which to do the aggregation on. To be able to do aggregations on infinite streams of data, Esper provides `windows`.

5.2.1 Windows

Windows splits the stream into a finite amount of data. There are many ways of splitting a stream that Esper has provided. The Esper engine can create windows based on how many events have arrived:

\[
select * \text{ from eventA}.\text{win:length(s)}
\] (5.2)

where `s` is the maximum number of events the window shall keep in memory. Windows may also be based on time:

\[
select * \text{ from eventA}.\text{win:time(t)}
\] (5.3)

where `t` is how long the window should last.

\[
\begin{align*}
\text{select temperature from WeatherEvent}\cdot &\text{win:length(5)} \\
\text{select temperature from WeatherEvent}\cdot &\text{win:time(10 seconds)}
\end{align*}
\] (5.4)

These two methods (Example 5.4) of creating a window will send an incoming event to a listener/subscriber the moment it arrives. However, it is possible to `batch` these events (Example 5.5) and send them to the listener when the window is "full" - either it has received the specified amount of events or the specified amount of time has passed.

\[
\begin{align*}
\text{select temperature from WeatherEvent}\cdot &\text{win:length\_batch(5)} \\
\text{select temperature from WeatherEvent}\cdot &\text{win:time\_batch(10 seconds)}
\end{align*}
\] (5.5)
When to use the different ways of splitting a window is based on what is known about the stream and what is the desired information. Batching based on number of events and batching based on time both may suffer from erratic behavior on the stream, as a window can take more time than expected before getting filled.

5.2.2 Streams

Esper queries fetches information from streams of data, but there are options as to which type of stream the query should use. A query may fetch information from the insert/input stream (default), the remove stream or both. Example 5.1 selects from the insert stream (as it is the default), and thus every eventA the engine processes will invoke the listeners/subscribers.

\[ \text{select rstream * from eventA.win:length(2)} \]  

(5.6)

The insert stream is when the tuples enter the window. If no window has been specified, all tuples received will be considered as part of the insert stream. The remove stream contains tuples The remove stream will receive an eventA when the window is full, receives a new event and has to push out an event to make room for the new event (see Figure 5.7). The removed event will enter the remove stream if the query has specified that query streams should be included.

5.2.3 Context

Oxford dictionary defines a context as “the circumstances that form the setting for an event [..]”. In event processing, context is “a declaration of dimension” [19]. This dimension can be of a temporal nature, e.g., the temperature in a room should be above a certain threshold during daytime. A context may also be segmentation-oriented, e.g., only network packets coming from outside the company’s IP range should be analyzed.

Esper’s EPL allows the user to declare contexts explicitly, which gives certain benefits [19]:

- A context can be applied to multiple statements, thus common shared context dimensions can be factored out in favor of an explicitly defined context.
- As common predicate expressions can be factored out into a context, EPL may become easier to read.
- Context partitions can be temporally overlapping.
5.2. Event Stream Queries

- Context partitions provide a fine-grained lifecycle that is independent of the lifecycle of statement lifecycle.

- Fine-grained lock granularity:
  - A user may specify nested contexts, which are composed of two or more contexts. These contexts may be temporal or segmentation-oriented.
  - Using contexts, an application can aggregate events over time periods (overlapping or non-overlapping) without retaining any events in memory.
  - Using contexts, an application can coordinate time boundaries for multiple statements.

To declare a context, Esper provides the create context syntax (see Example 5.7). When a context has been declared, one or more EPL statements

---

![Figure 5.7: Input Stream (New Events) and Output Stream (Old Events)](image)

- **Incoming Events**
  - *W₁(500)*
  - *W₂(100)*
  - *W₃(200)*
  - *W₄(50)*
  - *W₅(150)*
  - *W₆(300)*

- **Length Window – 5 Events**
  - *W₁*
  - *W₂* → *W₁*
  - *W₃* → *W₂* → *W₁*
  - *W₄* → *W₃* → *W₂* → *W₁*
  - *W₅* → *W₄* → *W₃* → *W₂* → *W₁*
  - *W₆* → *W₅* → *W₄* → *W₃* → *W₂* → *W₁*

- **New Events**
  - *W₁*
  - *W₂*
  - *W₃*
  - *W₄*
  - *W₅*
  - *W₆*

- **Old Events**
  - *W₁*
  - *W₂*
  - *W₃*
  - *W₄*
  - *W₅*
  - *W₁*
may refer to the context by specifying the context name. By default, all EPL
statements are considered using a single context that lives as long as the EPL
statement lasts, i.e., from when the EPL statement starts until it ends.

\[
create \ context \ ContextName \ partition \ [by] \ \EventProperty\ \[and\ \EventProperty\ \[and \ ..\]\] \ from \ StreamDef\\
[, \ EventProperty \ [..] \ from \ StreamDef]\\n[, ..]
\]

(5.7)

5.3 Event Patterns

Patterns are a set of events occurring in a distinct fashion. It consists of
pattern atoms and pattern operators. An atom can be an event, an observer
specifying time intervals or schedules; or a custom plug-in. These atoms are
connected by logical or temporal operators and the operators can also be
used to control sub-expression repetition and lifecycle. Operators supported
by Esper can be seen in Table 5.1.

5.3.1 Pattern Expression

A pattern expression may be registered in two ways. The first approach is
to use the \texttt{createEPL} method to create a statement that may specify one
or more pattern expressions. A pattern expression may appear in the \texttt{from}
clause of an EPL statement (see Example 5.8).

\[
\text{SELECT} \ * \ \text{FROM} \ \text{PATTERN} \\
[\text{EVERY} \ \text{TemperatureEvent}(\text{temperature}>30)]
\]

(5.8)

The second approach is to use the \texttt{createPattern} method. This approach
cannot use any of the EPL clauses, such as \texttt{select}, \texttt{where}, \texttt{group by} and \texttt{having}
clauses.

Events can be filtered by event properties, by specifying the event prop-
erty values within parenthesis after the event type name (see Example 5.8).
Multiple filters can be used by separating each filter with a single comma.
The comma represents a logical AND between expressions. Filters can refer
to the same type of event properties of other events, happening prior to the
current event (see Example 5.9).

\[
[\text{EVERY} \ t=\text{TemperatureEvent}(\text{temperature}>30) \ \Rightarrow \ \text{TemperatureEvent}(\text{temperature}<0, \ \text{location}=t.\text{location})]
\]

(5.9)

47
For every event that is processed by the engine, the event will apply to every filter expression for which the event matches. This means that an event $A$ will apply to every pattern expression referring to event $A$. A user may specify the @consume annotation as part of the expression to limit the event so that the event only applies to specific expressions.

\[
\begin{align*}
\text{[TemperatureEvent(temperature}>30)@consume} \\
\text{AND TemperatureEvent(location='Oslo')}
\end{align*}
\] (5.10)

In example 5.10, if @consume was not specified, we would have a match if a single TemperatureEvent arrives that has a temperature greater than 30 degrees and is located in Oslo. It will also match if we receive one TemperatureEvent with temperature greater than 30 degrees and another TemperatureEvent located in Oslo. However, with the @consume annotation, the expression would only match for the latter case.

### 5.3.2 Pattern Operators

There are four types of pattern operators:

- Operators that control pattern sub-expression repetition, such as *every*, *every-distinct* and *until*.
- Logical operators, such as *and*, *or* and *not*.
- Temporal operators, such as *->* (followed-by)
- Pattern guards that control the lifecycle of sub-expressions, such as *timer:within*, *timer:withinmax* and *while*.

The *every* operator restarts the sub-expression, i.e., the expression qualified by the *every* keyword, whenever the sub-expression evaluates to true or false. Without the *every* operator, the sub-expression ends when it evaluates to true or false. The engine will start a new sub-expression looking for more events or timing conditions when the *every* sub-expression evaluates to true. The *every* operator should be used with care as, depending on the data stream, this can possibly lead to many sub-expressions resulting in a high memory consumption. Using the *timer:within* operator and the *and not* operators, the user may end the sub-expression after a certain amount of time or when certain conditions occur.

The *every-distinct* operator is similar to the *every* operator, except that it eliminates duplicates with regards to the specified event property. It is also possible to specify multiple event properties and only eliminate duplicates
which have the combination of these event properties in common. If an expiration time period is given, a duplicate event may occur after the time period has expired. In Example 5.11, the pattern expression will report a temperature event for, e.g., Oslo once every hour.

\[ \text{every-distinct}(t.\text{location}, 1 \text{ hour}) t=\text{TemperatureEvent} \]  
(5.11)

The repeat operator is similar to the \textit{every} operator, except that it specifies the amount of times the sub-expression should occur before evaluating to true. In Example 5.12, the pattern evaluates to true when five events, in any combination, of temperature events and rain events.

\[ [5] (\text{TemperatureEvent OR RainEvent}) \]  
(5.12)

The repeat-until operator (using the \textit{until} keyword) provides an additional method of ending the sub-expression. The range for the repeat operator can be bounded or unbounded. If it is unbounded, the only way the sub-expression evaluates to true is if the expression in the \textit{until} operator is evaluated to true. In Example 5.13, the pattern will continuously look for temperature events until a rain event occurs.

\[ \text{TemperatureEvent until RainEvent} \]  
(5.13)

Bounded repeat can have an open ended range, high endpoint range or a bounded range. An expression using bounded repeat will end when either the high endpoint is reached or the until sub-expression evaluates to true. An open ended range specifies only a low endpoint and not a high endpoint. This means the sub-expression will only evaluate to true if the amount of events arrived is equal to or greater than the low endpoint, and it will only end when the until sub-expression has evaluated to true (see Example 5.14).

\[ [2:] \text{TemperatureEvent until RainEvent} \]  
(5.14)

For the high endpoint range, the sub-expression will evaluate to true if the amount of events arrived is at maximum the high endpoint (see Example 5.15). A bounded range specifies both a low endpoint and high endpoint. As long as the amount of events arrived is equal to or greater than the low endpoint, the pattern evaluates to true (see Example 5.16).

\[ [:2] \text{TemperatureEvent until RainEvent} \]  
(5.15)

\[ [2:5] \text{TemperatureEvent until RainEvent} \]  
(5.16)
The followed-by operator (\texttt{->}) specifies that the expression on the left hand side of the operator must evaluate to true before the right hand side can be evaluated. This means that something must occur before something else can occur, e.g., look for door.

\begin{equation}
every \text{StatusEvent}(status='\text{warning}') \rightarrow \text{StatusEvent}(status='\text{error}')
\end{equation}

(5.17)

In Example 5.17, a new sub-expression starts for every StatusEvent that occurs, which can potentially be more sub-expressions than desired. The user can limit the amount of sub-expressions as in Example 5.18. Now there can only be at maximum three active sub-expressions.

\begin{equation}
every \text{StatusEvent}(status='\text{warning}') -[3]\rightarrow \text{StatusEvent}(status='\text{error}')
\end{equation}

(5.18)

There are three pattern guards: \textit{timer:within}, \textit{timer:withinmax} and \textit{while}. The \textit{timer:within} pattern guard specifies a time limit for when the sub-expression can evaluate to true. If this time limit is passed, the sub-expression ends and will be false. It should be noted that the \textit{where} condition of the pattern guard has no relationship to the \textit{where} clause found in event stream queries. Example 5.19 searches for a warning status from the system followed by an error happening within five seconds of the warning status.

\begin{equation}
every \text{StatusEvent}(status='\text{warning}') \rightarrow \text{StatusEvent}(status='\text{error}')
\text{where}\ timer:within(5\ sec)
\end{equation}

(5.19)

The \textit{timer:withinmax} is a pattern guard similar to \textit{timer:within}, except that it has an additional method of ending the sub-expression. By specifying a max count the sub-expression ends when it has found an amount of matches equal to the max count.

The \textit{while} pattern guard specifies a condition that returns either true or false. Prior to the \textit{while} keyword is a sub-expression. When this sub-expression finds a match the pattern guard condition is evaluated and if true, the match passes and if false, the sub-expression ends. Example 5.20 shows us that the pattern matches while we receive warning status events (parenthesis around the every sub-expression is needed because of the precedence rules).

\begin{equation}
(every\ s=\text{StatusEvent})\ while\ (s.status='\text{warning}')
\end{equation}

(5.20)
5.3.3 Pattern Atoms

A pattern atom can be a filter expression, which specifies an event to look for. It can also be a time-based event observer, which specifies time intervals or time schedules. There are two time-based observer atoms: `timer:interval` and `timer:at`. These can be controlled by an engine timer or an external timer event.

The `timer:interval` observer waits for a specified amount of time before it evaluates to true. Example 5.21 searches for a status event and when it occurs, it will wait for five seconds before the pattern returns a match. This is similar to the `timer:within` pattern guard, except that the pattern guard may report a match before the specified time limit while the `timer:interval` will always wait the specified time amount.

\[
\text{every StatusEvent} \rightarrow \text{timer:interval}(5 \text{ sec}) \quad (5.21)
\]

The `timer:at` observer will evaluate to true at a specified time. It is similar to the Unix `crontab` function where you can schedule jobs to be done at specific times. The syntax for the `timer:at` observer is:

\[
\text{timer:at}(\text{minutes, hours,}
\text{ days of month, days of week [, seconds [, time zone]]}) \quad (5.22)
\]

The fields `seconds` and `time zone` are the only fields not mandatory. It is possible to use wildcard (*) values, which will evaluate the field to true for any value. Ranges can be used by specifying a lower bounds and a higher bounds (with a colon between them). The `timer:at` observer in Example 5.23 will be true between 08:00 AM and 16:59 PM every weekday (1 = Monday).

\[
\text{every timer:at}(*, 8:16, *, *, 1:5) \quad (5.23)
\]

Keywords can be used instead of specifying with integers, e.g., Example 5.23 can use the `weekday` keyword in the `days of month` field instead of specifying the range in the `days of week` field. The only restriction to the parameters is that it is not possible to define both the `days of month` field and the `days of week` field.

Instead of using static values such as, keywords or simple integer values, in the fields it is possible to invoke user-defined functions (see Example 5.24). These functions must return a value that is accepted as a parameter.

\[
\text{every timer:at}(*, \text{calculateHour}(), *, *, *) \quad (5.24)
\]
5.4 Receiving output data

Esper provides three alternatives for receiving output data, or results, from queries; two of which are push-based and one that is pull-based.

```java
public class BaseEventListener implements UpdateListener {
    public void update(EventBean[] newEvents, EventBean[] oldEvents) {
        // Output the timestamp of the event
        System.out.println("Timestamp: ", newEvents[0].get("timestamp"));
        // Do whatever is necessary, e.g., calculations or
        // outputting more data.
    }
}
```

Figure 5.8: A listener.

- **Push-based mode**: When the engine discovers that a query pattern occurs it reports to either a listener (Figure 5.8) or a subscriber (Figure 5.9), depending on which one was registered to the query. A listener is able to handle multiple queries, meaning that multiple queries can be registered to a single listener. A subscriber is a direct binding of query results to a Java object, meaning there may be at most only one query per subscriber. This gives some performance benefits as the query results does not need any intermediate representation (EventBean) and the subscriber receives strongly-typed parameters which tends to make the code more simple [19].

- **Pull-based mode**: If not all updates are interesting iterators are used to perform frequent or infrequent polls to fetch the latest data.

```java
public class BaseEventSubscriber {
    // Matches the EPL statement
    public void update(long timestamp, String loi) {
        // Output the timestamp of the event
        System.out.println("Timestamp: ", timestamp);
    }
}
```

Figure 5.9: A subscriber.
Listeners and subscribers are both POJOs, however, the listener is required to implement either the \textit{UpdateListener} or the \textit{StatementAwareUpdateListener} interface.

It is possible to get some performance benefits by using the \texttt{EventSender} interface. It "can reduce the overhead of event object reflection and type lookup as an event sender is always associated to a single concrete event type" [19].
### Table 5.1: Pattern operators - part 1.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>every</td>
<td>every X</td>
<td>continuously look for events of type X</td>
</tr>
<tr>
<td>ever-distinct(value)</td>
<td>every-distinct(x.value)</td>
<td>continuously look for events of type X, drop duplicates of specified value</td>
</tr>
<tr>
<td></td>
<td>x=X</td>
<td></td>
</tr>
<tr>
<td>repeat</td>
<td>[5] X</td>
<td>similar to every, but specify how many events should occur before evaluating to true</td>
</tr>
<tr>
<td>repeat-until</td>
<td>[5] X until Y</td>
<td>Matches X a number of times equal to the range specified and if Y arrives before or after this range, the expression evaluates to false</td>
</tr>
<tr>
<td>unbound-repeat</td>
<td>X until Y</td>
<td>The expression keeps looking for X events until an Y event occurs</td>
</tr>
<tr>
<td>bounded-repeat: open-ended range</td>
<td>[5:] X until Y</td>
<td>similar to repeat, X can be repeated for the specified range or more and if Y occurs after 5 or more events, expression will be true</td>
</tr>
<tr>
<td>bounded-repeat high-endpoint range</td>
<td>[5:] X until Y</td>
<td>similar to repeat, if Y occurs before 5 X events, the expression will evaluate to true</td>
</tr>
</tbody>
</table>

---
<table>
<thead>
<tr>
<th>and</th>
<th>X and Y</th>
<th>logical AND operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>or</td>
<td>X or Y</td>
<td>logical OR operator</td>
</tr>
<tr>
<td>not</td>
<td>not X</td>
<td>logical NOT operator</td>
</tr>
<tr>
<td>followed-by</td>
<td>X → Y</td>
<td>if X event occurs, look for a Y event</td>
</tr>
<tr>
<td>timer:within</td>
<td>X where timer:within(1 hour)</td>
<td>expression will look for X with 1 hour limit</td>
</tr>
<tr>
<td>timer:withinmax</td>
<td>X where timer:withinmax(1 hour, 5)</td>
<td>same as timer:within except expression may end before 1 hour has passed if 5 events of X has occured</td>
</tr>
</tbody>
</table>

Table 5.2: Pattern operators - part 2.
5.4. Receiving output data
Chapter 6

Design

In Chapter 2.3 we presented our requirements analysis of automated home care. To consider all of these requirements encompasses more than we are able to do in this master thesis. Our primary aim is to achieve the first requirement: well-being. Well-being is an ambiguous term with no single metric that is easily measured, but it revolves around the automated HCS’s ability to be able to detect events that are of interest and notify the people responsible in near real-time. Another important requirement is that the HCS should be easy to deploy, as this supports large scale deployment. Although we create models that enables us to reuse configurations, and the architecture of our system enables us to easily deliver notifications, we have not done any tests that measure the deployment speed. Our design, implementation and evaluation is mainly focused on the first requirement.

6.1 Data Models

There are three models from CommonSens which we will implement; the event model, the sensor model and the environment model. In CommonSens an event is a state or state transition in which someone has declared interest. The event model consists of four attributes; $e$, $loi$, $tb$, $te$. $e$ is the event, $loi$ is the location of interest, $tb$ and $te$ are the timestamps of when the event begins and when the event ends, respectively.

Esper provides different alternatives of modeling events. Each event must be registered with the engine for Esper to be able to process them. There are two design possibilities on how to model the events.

- **Single type**: Create a class which will represent all events.
- **Multiple type**: Create one type of event to represent each individual
Design alternative one is similar to CommonSens' implementation. The event model will contain the name/type of the event, the location of which it occurs and the timestamps. Using this alternative provides a clean solution and makes it easy for the application programmer to register events as there is no need to do any configuration with the event model when setting it up in the environment. However, this does not give a clear overview of an event hierarchy nor does it support for any additional information or functionality some events may have need for.

Design alternative two uses different event types to represent each individual event, i.e., every event that is of interest is mapped to an event type. There are common properties that every event shares that are placed into an abstract class, called `BaseEvent`, which all the events extend. Events may also extend other events, thus creating a hierarchy of events. The drawback with this solution is managing all the events. Although having a repository of commonly used events, such as motion and temperature events, may lessen the burden of the application programmer, some events may still need to be manually programmed. However, being able to have events formed in a hierarchy gives a great overview over all events and how they relate to each other. We may also extend a specific event with more variables or functionality, depending on the need. In spite of the drawback, the benefits this design alternative provides made us choose this solution.

`BaseEvent` contains the common properties of all events. This means that it contains `loi`, `tb` and `te`. `e` is not a property as the event classes themselves are the representation of the event. It might seem unnecessary to have separate events when `BaseEvent` is similar to CommonSens' event model, with the exception of `e`. However, having separate event models enables us to further extend the functionality of a given event, if it is needed.

The events that extend `BaseEvent` contains at least one property/attribute:

- **value**: This is the value associated with the event, if any. For example, `TemperatureEvent` specifies the temperature in the value attribute. Although, we might consider using a more appropriate name in place of `value`, such as `temperature`. However, using different names for the value might be confusing for the application programmer whom is the one writing the queries. Using a single name might be more beneficial.

Another issue of heterogeneity is what type value should be. It is possible that each event may use a different type depending on events (`Long`, `String`, `Boolean`, etc). However, application programmers may
have different interpretations of which type a value property for an event should hold. Using the TemperatureEvent as an example, the value property may be interpreted by some as an Integer while others may interpret it as a Double.

CommonSens describes a Sensor model that should achieve three objectives:

1. Determine which type of sensors to use when describing events.
2. Able to utilize historical and stored data together with recent data tuples.
3. Able to aggregate information from several sources of informations.

In order to achieve the first objective, CommonSens sensors implement a set of capabilities. A capability describes the type of state variables a sensor can detect. The Sensor model contains a list of capabilities the sensor provides. The application programmer specifies capabilities and the system alleviates this individual from having to specify every sensor.

CommonSens uses three types of sensors to achieve the objectives stated above; the physical sensor, the logical sensor and the external source. The physical sensor is responsible (and limited to) detecting single states in the home.

Our PhysicalSensor is conceptually the same as in CommonSens, except that we need the sensor to send the readings into the Esper engine. It contains the following variables:

- **signalType**: The type of signal, e.g., light or radio
- **coverageArea**: The shape that describes where the sensor is able to do detection.
- **objectCovered**: The object that has the physical sensor, e.g., a person carrying a device (such as a RFID active tag).
- **samplingFrequency**: How often the sensor outputs data tuples.
- **esperEngine**: The Esper engine the sensor uses to send events with.

When a physical sensor detects a state or state transition, it should send this information to the Esper engine. We will also need a variable, or more, that allows us to connect to a sensor, but this depends on how the sensor is connected to the system. It could either be directly connected to the
same machine EsperSens is running on, or it could transmit readings over
the network. If it is the former we need to connect to the `SerialPort` and
if it is the latter we can connect to an IP address.

The `ExternalSource` contains:

- **type**: What type of information the external source is.
- **value**: The value of the information.

The logical sensor is responsible of aggregating data tuples from several
sources of information. In EsperSens, however, this responsibility can be
assigned to the Esper engine. This means that to aggregate data tuples
we may simply create a query that uses information provided by a physical
sensor and an external source and pass it to the engine.

The Environment model in CommonSens is used to identify how the
sensor signals are affected when the sensors are placed in the home. The
environment is described by a configuration of objects. In EsperSens we
want to extend the responsibility of the Environment. The Environment
should describe everything related to the home, which includes:

- **Objects**: An arraylist of objects that exists in the home / environment.
- **LoIs**: An arraylist of LoIs that specifies key points of interest in the
  home.
- **Sensors**: An arraylist of sensors that are placed in the home.
- **Persons**: An arraylist of persons that resides in the home.
- **Queries**: An arraylist of queries that run in the home.

This provides a clear overview of what is related to a given home when
multiple home environments are under observation. If information of the
environment needs to be passed to other parts of the code, the environment
model knows of everything and can pack a message containing the relevant
information and send it to, e.g., the web application.

### 6.2 Query Translation

When specifying an atomic query in CommonSens, it is possible to specify
up to five attributes; `cond`, `loi`, `tb`, `te` and `preg`. The condition (`cond`) consists
of a capability, an operator and a value. Complex queries is a set of atomic
queries that are related to each other through time or aggregation.
Chapter 6. Design

A feature of CommonSens’ queries is that it does not specify which sensors to use. A capability is specified and the system finds the sensors that are able to detect the capability in that location. This feature gives the application programmer “more ease of use”. To achieve the same result for Esper, which needs to specify the sensors, we need to make the system find which sensors can be used for the specified query. This means that when we translate a CommonSens query into an Esper query, we first need to deconstruct the CommonSens query into the attributes that comprise the query. We need to analyze the attributes and also see which sensors has the capability referred to in the CommonSens query. When we have analyzed the CommonSens query, we can construct an Esper query. This process is done in two phases called the deconstruction phase and the construction phase.

Deconstruction phase:
The deconstruction phase uses a class, QueryDeconstructor, to parse a CommonSens query. When the QueryDeconstructor receives a query, it starts by separating a potential complex query into a set of atomic queries. We know that each atomic query is encased in parenthesis and between each of the atomic queries there is an operator. A complex query may look like: \[ (aq1) \text{ op1 } (aq2) \text{ op2 } (aq3) \], where \( aq \) is an atomic query and \( op \) is an operator. The QueryDeconstructor takes each of these atomic queries and parses them separately. When deconstructing an atomic query we look for all attributes it may contain: \( \text{cond}, \text{ loi}, \text{ tb}, \text{ te} \) and \( \text{preg} \). To register the attributes we use an AtomicQuery object to store these values. Each atomic query that is deconstructed is registered in their own AtomicQuery object. All the AtomicQuery objects are added to an ArrayList that resides in the Query object. The operators that associate the atomic queries are also added to an ArrayList in the Query object. The Query object contains the full CommonSens queries and its deconstructed parts, and will contain the translated query after the Construction phase has completed (see below).

The alternative would be to “force” the application programmer to write queries in Esper’s query language. This solution would be against our goal of providing the application programmer an intuitive and easy way to describe an event as Esper’s query language is not specifically designed for automated home care.

Construction phase:
The construction phase uses a class, QueryConstructor, to interpret the parts in the Query object and create an Esper query based on the parts. QueryConstructor starts by looking at each atomic query and determining which combination of attributes the atomic query uses. The five possible combinations are:
Based on these combinations the **QueryConstructor** can determine which tools of the Esper query language it should use. For instance, for combination 3 we need to consider how we can specify an event with duration equal to $tb$. For combination 4 we need to specify a duration at a specific point in time. One solution could be the use of *contexts*. As described in Chapter 5.2.3, contexts enables to define, among other things, a time period. This can be useful for the combination 4, as the context can define the time period denoted by $tb$ and $te$.

Another option is to use the **timer:at()** operator with the operators **until** **timer:interval()**. A similar approach can be done for combination 3, where the **timer:at()** function is ignored because the time duration is specified after a given event has been detected. We chose these solutions as it provided consistency across the queries, but we see that contexts is certainly a viable solution.

For very common queries, it is possible to write highly optimized queries that do not have to go through the deconstruction and construction phase. This can significantly lower the event processing time, but for that we need to know more about the input stream. It is only possible to deliberately write optimized queries if we know about the input stream and thereby know how the queries will process them.

### 6.3 User Interface

The application programmer should be provided with an intuitive interface as to make the work of deploying an automated home care system relatively easy. A graphical user interface (GUI) gives the application programmer an immediate and visual response that is easier to familiarize oneself with than a terminal or command line. The use of icons and color palettes are tools that can be used to improve the interaction. There are different ways of implementing a GUI:
Chapter 6. Design

1. Desktop application: One solution would be to create a desktop application with Java through the frameworks/toolkits AWT, Swing or JavaFX.

2. Web application: Another solution is to create a web application that provides the interface through a browser.

Solution 1 requires the device to be able to run Java. Granted, many devices are able to run Java, but not all. Using a Java GUI requires that the application is deployed to various devices, which can be a complex task, prone to errors and increases deployment time. Further, updates to the application will also have to be done to every device, which is also complex and prone to errors. This solution is also directly paired with the automated home care system, i.e., the application running the user interface is the same application that is processing events (etc). CommonSens uses this solution.

In solution 2, any device that has access to a browser will be able to run the GUI. This means that there are no requirements to install the application onto different devices. If there is an update to the web application, it is only necessary to refresh/restart the browser and the update is done. This solution is not directly paired with the automated HCS. It communicates with the automated HCS via the network. This gives the benefit that the user interface does not need to be on a powerful device, nor does the interface have any bearings on the actual performance of the automated HCS. However, one drawback is that it is susceptible to errors inherent in network communication, such as packet losses or latency issues.

There are different options of implementing solution 2:

1. Integrate a Java web server application with the EsperSens implementation.

2. Use sockets to transfer information to a separate server.

3. Use message oriented middleware (MOM) to transfer information to a separate server.

Esper provides a method of sending and receiving information with sockets (through the EsperIO library), which can be used in solution 2. This means that we can send warnings or notifications to the user interface with EsperSens. To connect to the web application server (which provides the user interface), Esper needs to know the exact address to the server. This means that the address needs to be hardcoded into the system. It also means that if the address changes, this change needs to be accounted for in the Esper code.
Option 3 allows EsperSens to send messages to the server, but with a key difference. It does not need to consider various operating systems or network interfaces. EsperSens publishes messages and the MOM ensures that the messages arrive to the correct recipient. The servers can be programmed in any language as long as it can communicate with the MOM. Figure 7.4 displays a conceptual architectural overview of how deployment can be done. Multiple EsperSens instances, each connected to a home environment, publish and subscribe to messages from the MOM. Web applications, each possibly programmed in a different language, serve as the interface for the users. The web applications also publish and subscribe messages to the MOM and the MOM ensures that every message is sent to the correct recipient(s).

![Conceptual Architectural Overview](image)

Figure 6.1: A conceptual architectural overview of the whole system, which is EsperSens, a MOM and the web application.

The users of the web application (that provides the GUI) may be the application programmer, the helping personnel or even the monitored person. For each of these persons, the web application must be designed to handle their circumstances (knowledge).

The web application for the application programmer should assist in the matter of deploying the HCS. This includes:

- Creating the environment.
- Placing sensors and determining their coverage area.
- Run simulations based on environment configurations, movement patterns and queries.

The web application for the helping personnel is to give assistance in monitoring the people the helping personnel is responsible for. This includes:

- Giving notifications when something noteworthy, but not crucial has happened.
Chapter 6. Design

- Giving alerts when something crucial has happened.
- Allowing communication between the monitored person and the helping personnel.

The monitored person may have his/her cognitive abilities reduced, which is why he/she has need of the HCS. However, due to this circumstance the user interface must be carefully designed. The user interface must be understandable for the monitored person. This requires more analysis and research that is beyond the scope of this thesis as mentioned in Chapter 9.2.
6.3. User Interface
Chapter 7

Implementation

7.1 Data Models

7.1.1 Event Model

We use an abstract class called \texttt{BaseEvent} that contains the common properties of all events. All other events extend \texttt{BaseEvent} either directly, i.e., an event that extends \texttt{BaseEvent}, or indirectly, i.e., an event that extends a different event that extends \texttt{BaseEvent}. \texttt{BaseEvent} by itself is never used.
as an event, which means that sensor readings are never translated into 
*BaseEvent*. The *BaseEvent* class contains three attributes:

- String *sensor*
- String *loi*
- long *timestamp*

Although we presented in our design that the *BaseEvent* should contain two
variables which is a timestamp of when the event begins and a timestamp
of when an event ends, this is not practical to implement. In Esper an
event is only "an immutable record of a past occurrence of an action or state
change", meaning an event represents a state only at a given point in time..
This conflicts with our idea of an event. For instance, we consider that act of
taking a shower to occur over time, and not only in a single point in time. To
represent the duration of an event in Esper, we need to get multiple readings
of an event that states, e.g., the shower is on. We may thereby invoke a
listener stating that a person has showered for a time period because we
received many events over time stating the shower was on.

As shown in Figure 7.1, the events that extend *BaseEvent* may contain
extra variables to provide further information with what they can detect.

For each of the variables in a class we added a set- and get method as
this is required by Esper.

```java
// generic get-method:
// public <variableType> get<variableName>() {...}

public String getLoi() {...}

// generic set-method:
// public void set<variableName>(<variableType> <variableName>)
// {...}

public void setLoi(String loi) {...}
```

Figure 7.2: Generic get- and set method.

*DetectPersonEvent* is one event that extends *BaseEvent* (it is the most
frequent used event in the Qualitative Tests, see Chapter 8.1). *DetectPerson
Event* contains these variables:

- **String person**: The person which the sensor detects moving. This
  variable is the one described as **<value>** in the design.

- **int epoch**: Not needed for the real application, but is used when doing
  the Qualitative Tests. It notes at which epoch the event happened. We
could have used either \texttt{tb} or \texttt{te} but using a separate variable for this logic makes the code more easy to read.

\texttt{DetectPersonEvent} contains set- and get-methods for the variables mentioned above. We have overridden the \texttt{toString()} method to properly provide a \texttt{String} that shows the event. All events are placed in the package \texttt{espersens.event}. However, when building repositories of events, the number of events might be a large amount which makes it harder to maintain and keep track of. We could have added new packages to the package \texttt{espersens.event} which are sorted by category. For instance, we could create a sub-package, \texttt{espersens.event.kitchen} for all events regarding the kitchen. As we currently only have five events in the package, we did not deem it necessary to create subpackages.

### 7.1.2 Sensor Model

![UML diagram of the \texttt{espersens.sensing} package.]

We implemented an abstract class called \texttt{Sensor} which holds the common properties of all the types of sensors. \texttt{PhysicalSensor} and \texttt{ExternalSource} extend this abstract class. \texttt{Sensor} has the following variables:
7.1. Data Models

- **Arraylist<Capability> providedCapabilities**: All the capabilities that the sensor provides.

- **String name**: The name of the sensor. This is useful when viewing the environment and seeing which sensors are which and see if this correlates with queries/events.

The `Sensor` class provides set- and get-methods for these variables.

A physical sensor, which is the type of sensor that is placed in the home environment, is implemented by the `PhysicalSensor` class. It has the following variables:

- **SignalType signalType**: The type of signal, e.g. light or radio.

- **Shape coverageArea**: The coverage area of the sensor, i.e. the area where it can do detection.

- **CommonSensObject objectCovered**: If the physical sensor is on an object or person.

- **double samplingFrequency**: How often the sensor provides data tuples.

- **EPServiceProvider esperEngine**: The Esper engine the sensor sends event into.

As shown in Figure 7.3, every `Sensor` provides at least one capability. Capabilities are represented through the `Capability` class. It contains two variables: `name` and `description`. The `name` variable is the name of the capability, e.g., DetectPerson or Temperature, and the description is of the `CapabilityDescription` type, which contains a `String` describing the capability.

There are four noteworthy functions in `PhysicalSensor`.

- **run()**: A function required by the `Runnable` interface. When executing a `PhysicalSensor` in a new thread this function is called. It polls the sensor for readings at a rate specified by the `samplingFrequency`.

- **fetchTuples()**: This function is periodically called by `run()`. It polls the sensor to do a reading and sends the result into the Esper engine.

- **getPolyReduced(Environment env, SignalType signalType)**: Calculates the real coverage area of sensors (considering how the signal is affected by various objects). Creates a new `Poly` that will have
Chapter 7. Implementation

the shape specified by `getShape().getPolyReduced()` which calls `Ray.reduceRay()`. `reduceRay()` uses the same solution as specified in the appendix in (A.2) [7].

- **toJSON():** Formats the coverage area of the sensor into the JSON format. Checks to see the signal type of the `PhysicalSensor`. Depending on the type it calls different `toJSON()` methods for those `Shape` objects to get the proper formatting.

`SignalType` is merely a class that holds a `String` describing the signal type. `Shape` is used to describe a physical dimension. In the case of `PhysicalSensor` it describes the coverage area of the sensor, but it can also describe an object in the environment. `Shape` consists of four variables:

- **Arraylist<Triple>** `boundary:` Boundary is the physical dimension the shape covers.

- **Arraylist<Triple>** `boundaryReduced:` This is only used by the shape for the coverage area of physical sensors. It describes the physical dimension the shape covers after considering how a signal is affected by objects / the environment.

- **Triple** `startPoint:` The middle point of the shape.

- **Polygon** `realCoveragePolygon:` The shape described, using a `Polygon`.

To calculate the `boundaryReduced` variable, `Shape` provides the `getPolygonReduced()` function. The function loops through the size of the boundary. It creates a new `Ray` object with parameters describing where the ray should start and where it should end. It loops through each `Triple`, which is a `Triple` with position somewhere in a line between start and end, in the `Ray`. By calling `reduceRay` it reduces the ray according to the objects it meets. The returned result from `reduceRay` is the new modified `Triple`. The `Triples` returned are added to the `boundaryReduced ArrayList`. Finally a polygon is created based on the `Triple` coordinates in `boundaryReduced` and this polygon is returned.

An external source, which is the type of sensor that does not provide data tuples through sensor readings, but rather through historical or static data, is implemented by the `ExternalSource` class. It contains two variables:

- **String** `type:` The type of information the external source holds.

- **String** `value:` The value of the information.
Using a `String` may be limited. Alternatively we could have a separate class for these variables which allows for different types of variables. Currently `ExternalSource` is not used, nor needed, in any of the tests. `Sensor`, `PhysicalSensor`, `ExternalSource`, `SignalType`, `Capability` and `CapabilityDescription` all exist in the package `espersens.sensing`. Although `PhysicalSensor` is heavily dependent on `Shape`, `Shape` is placed in the `espersens.environment` package as it is mostly used to describe things that exist in the environment.

### 7.1.3 Environment Model

To describe the environment we use the `Environment` class. The environment contains an `ArrayList` of objects, sensors, LoIs, persons and queries. There are three noteworthy functions in `Environment`:

- `calculateError(AtomicQuery aq)`: Calculates the chance that the query, with the current placement of sensors/lois, will provide a false positive. As a side effect: sets the `isec` and `noIsec` arraylist for `aq`.

- `findLoIsWithSensorInIsec(String sname)`: is a function that finds all LoIs that intersect with the sensor `sName`, where `sName` is the name of the sensor.

- `toJSON()`: Formats everything in the environment that has a shape to JSON (sensors, lois, persons, CommonSens objects). Used to send to WebEsper.

`calculateError()`

`calculateError()` fetches the `LocationOfInterest` from the parameter, `aq`, and assigns it to the variable `loi`. Secondly, it fetches the capability referenced in the query and assigns it to `cap`. It defines a `Poly` named `intersection` that will contain the intersection of where the coverage area of the sensors intersect with the LoI. `ArrayList`s are defined to hold the sensors that are calculated to go into the `isec` and `noIsec` lists.

When the basic assignments are done, the function defines and populates the `sensors ArrayList` with all tuple sources that provide the capability, `cap`. An `if` statement is used to see if the LoI has been specified in the query to determine whether there is a point in doing the calculation, i.e., if there is no LoI defined there will be no sensors in the `isec` and `noIsec ArrayList`s. Note that this `if` statement could have been placed before many of the variable instantiations, but to make the code more readable and maintainable we chose to do it after all the basic assignments were done. If
the LoI is not defined in the query we provide empty ArrayLists for isec and noIsec.

In the case where LoI is defined, we continue on by fetching the shape of the LoI. We loop through sensors to find all the physical sensors, as these are the only type of sensors that have a coverage area. For each physical sensor we find the intersection of its coverage area and the shape of the LoI and assign it to intersection. If the intersection covers the shape of the LoI, we may add it to the isec ArrayList.

After all the sensors with the specified capability have been checked, we do an if statement to see if isec was populated with any sensors. If isec is empty, there is no point in calculating noIsec and we return empty ArrayLists for isec and noIsec. If isec is not empty, we create the intersection of the coverage area of all sensors in isec. We then loop through all the sensors with a check for if the sensor is a physical sensor and has the specified capability. If both of these are true, we calculate if this sensor intersects with intersection and if it intersects with the shape of the LoI. When the case is that a sensor intersects with the intersection, but not with the shape of the LoI we may add it to the noIsec ArrayList.

findLoIsWithSensorsInIsec()
findLoIsWithSensorsInIsec loops through all the LoIs in the environment and checks to see if the sensor specified by sName intersects with the shape of any of these LoIs. This is done by calling the function isSensorInIsec of the LocationOfInterest object. The LocationOfInterests that have the sensor in their isec arraylist are added to an ArrayList, which will be returned when the loop is done.

toJSON()
This function loops through everything in the environment that has a shape. That means objects, sensors (where only the physical sensors are considered), LoIs and persons are looped through and a call upon their respective toJSON function is done. The result from these toJSON functions are added to the resulting string which is returned at the end of this function.

The Environment class contains set- and get methods for the variables it contains.

To create an Environment, we use the EnvironmentCreator class. It contains two variables: an Environment object, env, and an integer, arrPtr. When the EnvironmentCreator has created the Environment it is placed in env and returned to the caller. arrPtr is the index pointer to the variable fileArr (see below). By calling registerEnvironmentFromFile(String file) we start the process of parsing the file, which is sent as a parameter. This functions starts by splitting the file, with space as the delimiter, into an array of Strings called fileArr. We read the number of objects that is
in the environment and create a loop that calls on \texttt{readObjectFromFile()} equal to the number of objects. The object returned from the function is added to the environment instance \texttt{env}. The same process is applied for sensors, LoIs and persons.

### 7.2 Queries and Query Translation

Translating queries from the CommonSens query language into the query language of Esper is done in two phases: the deconstruction phase and the construction phase.

#### 7.2.1 Query Deconstructor

The class responsible for deconstructing CommonSens queries is \texttt{QueryDeco}\texttt{nstructor}. It contains two variables: a \texttt{String} called \texttt{query} and an \texttt{Environment} object, \texttt{env}. The CommonSens query is stored in \texttt{query} and the \texttt{Environment} object that will contain the \texttt{Query} object of the deconstructed parts is \texttt{env}. These variables are set when the constructor is called.

To deconstruct a CommonSens query \texttt{deconstructQuery()} must be called. This function creates an \texttt{ArrayList} of \texttt{AtomicQuery}s, \texttt{aqs}, as the CommonSens query consists of at least one atomic query. A \texttt{Query} object, \texttt{q}, is defined to store the deconstructed parts of the CommonSens query. We have defined two variables, \texttt{aq\_start} and \texttt{aq\_end}, to specify at which character in \texttt{query} an atomic query starts and where it ends, respectively. To parse the query we use a \texttt{for} loop to read a character at a time in \texttt{query}. If the character is a starting parenthesis, we know this is the start of an atomic query and thus note the index of this character in \texttt{aq\_start}. When a closing parenthesis is encountered we have an atomic query. By using \texttt{substring} we can get the atomic query from the \texttt{query} string and deconstruct it by calling \texttt{deconstructAtomicQuery()}. The result of \texttt{deconstructAtomicQuery()} is added to \texttt{q}. The loop also checks for operators between atomic queries and if this is the case, the operator is added by calling \texttt{q}'s function \texttt{addOperator}. When a closing bracket is encountered it means the CommonSens query has been deconstructed. The function both adds the query object to the environment specified and returns the object to the caller.

To deconstruct an atomic CommonSens query \texttt{deconstructAtomicQuery()} is called. However, it is a private function and is only called by \texttt{deconstruct Query()}. 

74
7.2.2 Query Constructor

QueryConstructor is the class responsible for analyzing the deconstructed parts of a CommonSens query and constructing an Esper query. QueryConstructor contains two variables: a Query object called query and an integer named qCounter. When the constructor is called it sets query to the Query object we want to use and it sets qCounter to zero. qCounter is used to differentiate between events of similar types in the query.

The main function of QueryConstructor is constructQuery(), which constructs the Esper query. It starts by fetching all the atomic queries and all the operators that relates each atomic query with each other. We do a for loop and call constructAtomicQuery(), which analyzes each of the atomic queries individually and constructs an Esper query. The result is appended to the String complex_query. If any operators exist, they are appended directly after an atomic query has been constructed. When this function has fully translated the CommonSens query, it adds the Esper query to the query object.

The function constructAtomicQuery() starts by fetching all the constituents a CommonSens query: condition, loi, tb and te. The condition is further divided into a capability, operator and value. We define a String called tmpAQ, which is where the constructed parts will be added to. The function first checks to see if both tb and te are set and if they are, we call translateTb() and adds the result to complex_query. Regardless if tb and te were set, we can now translate the condition by calling translateCondition() and adding the results to complex_query. After translating condition we recheck to see if te is set. If it is, we can call translateTe() and add it to the result. If it is not set, we call translateOnlyTb() because only the start time has been set. The resulting atomic query is then returned.

The reason why we call different functions depending on if tb and te is set or if only tb is set, is because they are two completely different timing functions (as described in Chapter 6.2). If only tb is set, we announce that the query should be delta timed. Meaning that when the desired event occurs, we should see if it lasts for a time period equal to tb. While if both tb and te are specified, we announce that the event should happen in that time period, i.e., between tb and te.

7.2.3 Queries

We use the class AtomicQuery to represent a CommonSens atomic query, i.e., a query that represents a single event. It does not contain an Esper atomic query as Esper does not have a clear distinction between atomic queries and
complex queries. When we construct a query with QueryConstructor we do not add the translated CommonSens atomic queries into Esper atomic queries as these translated parts would not necessarily be able to be processed by the Esper engine by itself. While if we divide a complex CommonSens query into the atomic queries that it is made of, each of these atomic queries is possible to be processed by the CommonSens engine.

AtomicQuery consists of the following variables:

- **String commonSensQuery**: The atomic query in CommonSens’ query format.
- **String translatedQuery**: The CommonSens query which has been deconstructed and reconstructed into an Esper query.
- **LocationOfInterest loi**: The location of interest specified in the atomic query.
- **Capability cap**: The capability the atomic query specifies.
- **String operator**: The operator which relates the capability and the value.
- **String condValue**: The value the atomic query expects the capability to have.
- **String tb**: The time when the event begins.
- **String te**: The time when the event ends.
- **HashMap<String, Sensor> isec**: The set of sensors which are in the isec with regards to loi. The key in the HashMap is the name of the sensor.
- **HashMap<String, Sensor> noIsec**: The set of sensors which are in the noIsec with regards to loi. The key in the HashMap is the name of the sensor.

AtomicQuery does not contain any noteworthy functions, it merely contains various set- and get-methods. Query is the class which represents a complex query. It consists of one or more AtomicQuery objects. Query has the following variables:

- **String commonSensQuery**: The full complex query in CommonSens’ query format.
• String translatedQuery: The CommonSens query which has had each of its AtomicQuery objects deconstructed and translated into this complex query (Esper format).

• ArrayList<AtomicQuery> subQueries: The atomic queries which constitutes the complex query.

• ArrayList<String> operators: The operators which relates each of the atomic queries with each other.

• Result result: The result of the query (used in the functionality tests).

Query does not contain any noteworthy functions, it merely contains various set- and get-methods. QueryDeconstructor, QueryConstructor, Query and AtomicQuery all exist in the esper.sens.query package.

7.3 User Interface

Figure 7.4: Our implementation has EsperSens running on the same server (server ”A”) as the MOM. The MOM we use is RabbitMQ, an AMQP service. EsperSens communicates with the web application, on server ”B”, via RabbitMQ. Users, which can include helping personnel and monitored persons, may interact with the web application through different devices such as smartphones, tablets or desktop computers. MOM; message oriented middleware, AMQP; advanced message queuing protocol.

We have created a web application to provide notifications to the helping personnel when something noteworthy has occurred. This is done by having EsperSens sending notifications to the web application server through RabbitMQ, an advanced message queuing protocol (AMQP) service. We can
run simulations on the web application and monitor the movement of the monitored person and see which sensors provide the readings. The web application will also report when a query has been matched. The simulation consists of two phases: the load phase and the simulation phase.

The load phase consists of three widgets:

- Load environment file
- Load movement file
- Load queries file

Each of these widgets contain an Add button, a Delete button and a Load button. The Add button uploads the file to the web server, but does not send it to EsperSens. The Delete button removes the file from the server and the Load button issues the server to send the file to EsperSens (via RabbitMQ). The order of which the files should be loaded is (1) the environment file, (2) the movement pattern file and (3) the queries file.

![Figure 7.5: Three widgets that are used during the load phase of the simulation in the web application. By clicking on the Add button, the user can upload the environment file, the movement pattern file or the queries file to the server. The Delete button allows the user to remove these files and the Load button issues the web application the order of sending these files to EsperSens via RabbitMQ.](image)

When the environment is loaded, EsperSens will process the environment and send the result to the web application. The web application listens for messages from EsperSens and when the environment message has been received, it will display the environment in the Environment overview widget. The different types of messages the server may receive are:

- **environment**: The environment, including all objects, sensors and persons within it.
- **movement**: The coordinates, for each epoch, of the movement of the monitored person.
Chapter 7. Implementation

- **danger**: Notifications about events that need to be taken care of immediately.

- **notification**: Notifications about events that the helping personnel may be interested to know about.

- **timestamp**: This is used to measure the time it takes for a message to be sent from EsperSens until it is displayed to the user.

**esper.js** has listeners that listen for these events. We have our AMQP consumer fetch incoming messages and parse the content type. Based on the content type we emit the desired message and message type to the user.

```javascript
var open = amqp.connect('amqp://<host>');</open.then(function(conn) {
    var channel = conn.createChannel();
    
    Consumer
    channel = channel.then(function(consumer) {
        consumer.assertQueue(<queue>);
        consumer.consume(<queue>, function(msg) {
            var type = msg.properties.contentType.toString();
            if (type === 'environment') {
                socket.emit('environment', { msg: msg.content.toString() });
            }
            else if (type === 'notification') {
                socket.emit('notification', { msg: msg.content.toString() });
            }
        });
    });
});

Figure 7.6: NodeJS code snippet from simulation.js. It shows how the web application server connects to the RabbitMQ server and listens for messages sent by it. The content type of these messages are parsed and the appropriate message type is sent to the user.

The simulation phase consists of three or four widgets.

- **Debug panel (optional)**: Alerts the user whenever a sensor detects something. This is optional and primarily used only for debugging.
7.4 Workflow

The lifecycle of EsperSens consists of three phases, similarly to CommonSens [20].

1. *A priori*: The environment is modelled. Objects, sensors, LoIs, person(s) are created from file. Queries are parsed and translated into
Chapter 7. Implementation

EPL so that it can be registered with the engine. Isecs and noIsecs for LoIs specified in the queries are found.

2. **Event processing**: When running the functionality tests the movement simulator sends events to the esper engine. The esper engine does the event processing and returns the result to a listener.

3. **System shutdown**: The last phase consists of doing adjustments. This may include rewriting queries or replacing sensors.

**A priori**

When we are running the tests described in Chapter 8.1 and in Chapter 8.2 we start the a priori phase in Tester. Tester is explained more into detail in Chapter 8.1. To run the system normally, the a priori phase starts in either WebApp or Terminal, depending on if you want to communicate with the web application or do testing in the terminal, respectively. We will describe WebApp as this is the primary class when running EsperSens.

**Event processing**

Simulator is the class responsible for starting the simulation. The constructor of Simulator sets up the communication with the MOM and assigns the environment it should work with. To start the simulator, run() is called. This function adds the various event types the simulation will need, it starts the Esper engine and it registers the queries sent from the web application. For the queries, it registers an event listener. Currently we have only tested DetectPerson events and therefore only register assign the DetectPersonListener. To handle different event types we could maybe have the user include a list of listeners EsperSens should use for specific queries. The last thing run does is calling runMovementPattern(), which runs the movement pattern file.

The function runMovementPattern() start off by fetching the person object. If this object is null it means that the person has not been registered with the environment and there is no point in running the movement pattern so the system will exit. The movement pattern, movementFile, is parsed in readMovementFile() and we receive an ArrayList of Triple objects (movCoords). We fetch all the sensors that has the capability DetectPerson, which we put in an ArrayList called sensors, as this is the only capability used when detecting movement. We loop through every Triple object in movCoords and call on we set the new position of the monitored person according to the Triple object. The new position is sent to the web application so that it may show the movement to the helping personnel. The function continues by looping through every sensor in sensors. We call on
generateTuples() for each of the sensors. This function checks if the coverage area of the sensor intersects with the monitored person. If so, we send an event to the Esper engine and we send a debug message to the web application describing which sensor detected movement.

**System shutdown**
Results.java provides feedback on how the different functionality tests performs. This can be used to determine if queries should be rewritten. Results.java should be extended in future work to provide more useful information.
Chapter 8

Evaluation

In this thesis we want to investigate if we can use Esper as a CEP system to build an HCS by implementing the three fundamental data models of CommonSens on Esper. The most crucial requirement of an automated HCS is that the system is able to correctly detect events in near real-time. We perform qualitative tests to determine if EsperSens is able to correctly detect events and we perform quantitative tests to determine if EsperSens is able to detect events in near real-time.

8.1 Qualitative Tests

In order to test if EsperSens is able to correctly detect events, we constructed Qualitative tests that are the same as the functionality tests specified in CommonSens. The language constructs we evaluated are language construct I, III and IV [7]:

- **Language construct I**: Timed and delta-timed atomic and complex queries.
- **Language construct III**: The logical operators $\wedge$, $\lor$ and $\neg$.
- **Language construct IV**: The consecutiveness relation $\rightarrow$.

Language constructs related to preg were not implemented. The reason is that the preg functionality proved difficult to implement with Esper.

When a test has completed it returns an integer based on how the test went. Table 8.1 displays the meaning behind the different values (it is the same as in CommonSens, except where otherwise noted).

In CommonSens the meaning behind the value 6 is specified as “the evaluation of the complex query has not started” [7]. We provide an alternative
8.1. Qualitative Tests

Table 8.1: The possible values returned from the functionality tests and their associated meaning.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The evaluation of the complex query has finished correctly.</td>
</tr>
<tr>
<td>3</td>
<td>A deviation is caused by temporal mismatch in one atomic query / A deviation is caused by a spatial mismatch in one atomic query.</td>
</tr>
<tr>
<td>6</td>
<td>The evaluation of the complex query has not started / has not yet completed.</td>
</tr>
<tr>
<td>7</td>
<td>A deviation is caused by mismatch in the concurrency.</td>
</tr>
<tr>
<td>8</td>
<td>A deviation is caused by temporal mismatch in a list of atomic queries.</td>
</tr>
</tbody>
</table>

meaning that is "the evaluation of the complex query has not yet completed". This is because a query may have matched some events in its subqueries, or atomic queries, but have yet to evaluate the whole query as the query is, e.g., waiting for an additional event (specified with the followed-by relation).

Each test consists of an environment, a movement pattern and a query. We use three different environment instances for the functionality tests. These environment instances do not represent a realistic environment. They are synthetically designed to evaluate the language constructs without having the added complexity of a typical home environment. Every environment has a different setup of sensors, but the same configuration of LoIs. Every environment has only a single person moving about.

Each LoI is represented by the color green with a black border (see Figure 8.1). In our tests, all LoIs are shaped as a square, but that is not necessarily the case for other environments. Each sensor has their coverage area represented by the color blue with a black border. As with LoIs, the shape of the coverage area varies from sensor to sensor, but in our tests all sensors have a circular coverage area. The sensors we use in our tests simulate RFID readers, which provide the DetectPerson capability. The monitored person is represented by the color orange with a black border in a square shape. The person, in these tests, carries a RFID tag that reacts to the RFID readers.

A movement pattern describes which coordinate the person (described in the environment configuration) should occupy at a given epoch. Each
coordinate is a set of Triples. The epoch is not specified in the movement pattern, but for each Triple of coordinates we increment the epoch with one. In our tests we assign the epoch to be equal to seconds, but that can be changed depending on what is desired in the test. The movement patterns described in CommonSens differ slightly from the files. We chose to use the movement patterns provided in the RunTime folder of the CommonSens project on Google [21], as these were used when we ran the functionality tests in CommonSens, which gave the expected results.

We use the same complex queries used in CommonSens for the functionality tests. Although some of the queries may not be considered “complex”, e.g., cq1.qry, we do not create the distinction between atomic queries and complex queries as to keep the presentation of the test clear. The first few queries (cq1.qry – cq10.qry) only test a single language construct. This is to first investigate if EsperSens properly evaluates each individual language construct correctly. We proceed with testing multiple language constructs as many real-life complex queries will combine these language constructs.

8.1.1 Workload

We do different combinations of the environment instances, movement patterns and complex queries to test if EsperSens is able to correctly detect events (see Appendix C.1 for the workload we used for the Qualitative Tests). The workload describes an environment in which a person moves around in a

Figure 8.1: The web application displaying the environment configuration of e1.env.
specific pattern that may trigger some sensors to give a reading. The sensors we use provide only the `DetectPerson` capability, which is the capability that allows us to detect the movement of the monitored person.

8.1.2 Code

We have implemented a class, `Tester`, which is responsible for running the Qualitative Tests (and Quantitative Test 1, see 8.2.1). The `Tester` class is called by EsperSens when running the application in test mode (see Chapter 7.4).

The constructor of `Tester` calls on functions that register environment, movement pattern and queries with the system. The constructor also sets `current_expected` to the value 6, as the complex query has not yet started. The constructor takes the following parameters:

- `String testnr`: The test number according to the functionality tests in CommonSens.
- `String env`: The name of the environment file to be used in the test.
- `String mov`: The name of the movement pattern file to be used in the test.
- `String qry`: The name of the query file to be used in the test.
- `int expected`: The expected result from the test.

The function `start()`, which is called by EsperSens, outputs the environment file, movement file, query file and the expected result to terminal. It then calls on `runTest()` that starts the test. The other notable functions are: `registerEnvironmentFromFile()`, `registerMovementFromFile()` and `registerQueryFromFile()`.

The function `registerEnvironmentFromFile()` takes a parameter that is the filename, including the relative path, of the environment configuration. We use `Scanner` to read the file from disk into a `String` called `content`. To parse the environment configuration, we create an instance of `EnvironmentCreator` and call `registerEnvironmentFromFile()` with `content` as a parameter. The function returns an `Environment` object that is set to the `Tester` variable `env`.

We use `registerMovementFromFile()` to register the movement pattern with the environment. As with `registerEnvironmentFromFile()`, this function uses `Scanner` to read the file from disk which is then set to the `String content`. The movement pattern does not need to be parsed,
thus we immediately add it to the `Environment` object `env` through the `setMovementPattern()` function.

To register queries we use `registerQueryFromFile()`, which also takes the same type of parameter as the previous two functions. This function uses `Scanner` to read the queries file from disk into the `String content`. The queries in the file are in the CommonSens query language and therefore needs to be translated into the query language of Esper. We create an instance of the `QueryDeconstructor`, then loop through each query in the queries file and let the `QueryDeconstructor` deconstruct the query. Each query will be added into a `Query` object and added to the `ArrayList` of queries in `env`. After all the queries have been deconstructed, we call the `calculateErrorFromAtomicQueries()` function of the `Environment`, which calculates the `isecs` and `noIsecs` of the queries. Lastly, we loop through the queries and construct Esper queries. It should be noted that, although we support multiple queries, the functionality tests only run a single query file at a time.

The class responsible for initializing the test, making the `MovementSimulator` run the movement pattern and output the results to a file is `runTest()`. This function starts by configuring which events are known by the engine. We add `BaseEvent`, `NoEvent` and `DetectPersonEvent` to the engine. This is done through `Configuration.addEventType("<EventName>", <Event-Class>).` To ensure that we have full control over the flow of time in the functionality tests, we disable the internal timer. After we have done the configuration, we can get the Esper engine instance and initialize the Esper runtime.

We register the translated queries with the Esper engine and we also add a listener. After the queries have been registered, we call `runMovementPattern()` to loop through the movement pattern and move the monitored person. The `runMovementPattern()` is called five times as to get the average event processing time. After this is done, we write the our results to file by calling `writeToTestFile()`.

`MovementSimulator` is responsible for running the movement pattern file and calling on sensors to see if they detect anything within their coverage area. The sensors sends events to the engine when the monitored person moves within their coverage area. When the `MovementSimulator` constructor is called, it assigns variables to store the environment configuration the test will run on, the movement pattern to be used, the epoch duration (which in our case is set to 1000 milliseconds), the epoch counter and the event sender. The event sender can ”reduce the overhead of event object reflection and type lookup as it is associated with a single concrete event type” [19], which in our case is `DetectPersonEvent`. 

87
The most significant function in MovementSimulator is `runMovementPattern()`, which is the function that actually moves the monitored person in the environment and commands the sensors with the DetectPerson capability to look for movement. The function starts by fetching the person object. It simply fetches the first person in the `ArrayList` of persons in the environment and if this is null, the function will exit as there is no person to move. To support multiple persons the function should be passed the name of the person as to move the correct person.

If the person exists, the movement pattern file is read and parsed in `readMovementFile()`, and the set of `Triples` is returned and stored in the `ArrayList movCoords`. The sensors with the capability DetectPerson are fetched, as this is the only capability used when detecting movement, and placed in `sensors`. We loop through every `Triple` in `movCoords`. For every `Triple` we loop through we set the new position of the monitored person to that `Triple`. We loop through `sensors` and check to see if a given sensor is a RFID reader, since the person is carrying a RFID active tag. If it is not a RFID reader, the sensor is ignored. If the sensor is a RFID reader, we call on the function `generateTuples()` in the `PhysicalSensor` model.

In `generateTuples()` we check to see if the person is contained within the reduced shape of the coverage area of the sensor. If that is the case, the sensor has detected motion. This means that we can call Tester and set the current result to 3, because based on the knowledge thus far, we may have either a temporal or spatial mismatch. This means that the query has encountered a DetectPersonEvent outside of the time period specified or that the query has encountered a DetectPersonEvent in a different LoI.

By calling `findLoIsWithSensorInIsec()` we can find all the LoIs (if any) that the sensor covers. The function then either sends a DetectPersonEvent for every LoI the sensor covers, or it sends a DetectPersonEvent with no LoI. In the case where the query listener is called, we can set the current result to 0, meaning that the query matched.

### 8.2 Quantitative Tests

We separated the quantitative tests into three parts:

1. Event processing – Using queries translated by QueryDeconstructor and QueryConstructor.

2. Event processing – Using specifically written queries for certain situations.
3. Notification delivery – The time used to pass a notification from EsperSens to the web application.

For Quantitative Test 1 we follow the suggestion posted by Thomas Bernhardt, the CTO and founder of EsperTech, by taking the timestamp before and after calling the `sendEvent(...)` function, which allows us to measure the event processing time [22]. However, it is not a fair comparison with the functionality tests performed in CommonSens as this includes the time Esper takes to package an event. We cannot simply ignore using a listener to avoid the time consumption due to packaging. If we omit the use of a listener, Esper will not execute the `SELECT` clause in the query thereby practically rendering the query invalid.

The complex queries used in the tests vary in complexity. Some may not be considered “complex” at all, but for simplicity each query in this set will be called complex. The first few complex queries (cq1.qry - cq10.qry) only tests a single language construct. The other queries test multiple language constructs. This is to investigate if EsperSens properly evaluates each individual language construct. If this works, we want to see if it correctly evaluates the different language constructs together. This is important as many real-life complex queries will combine these language constructs.

Quantitative Test 2 measures how Esper is affected when we have an increased number of events arriving at the same time. We investigate how long it takes for an event of interest, i.e., en event matching a query, arriving at a given second to be analyzed by Esper and sent to the listener, when multiple events arrive at the same time as the event of interest. The number of events we send per second is more than what is expected for a regular home environment, which in most cases do not register more than five events per second [23, 24]. However, the test is to see how the performance scales with Esper and how one could use better specified queries, as opposed to the queries generated dynamically by `QueryConstructor`.

A test consists of a complex query and a trace file. We ran three complex queries that used different trace files to trigger the queries. The workload can be seen in Table 8.2. The queries use features needed for a home care scenario, such as timing functions and the followed-by operator. For instance, q2 simulates a simplified hygiene test for when the monitored person has used the toilet. The workloads have been written with specific events to trigger the queries. Each of these events have `NoEvents` prepended and appended to the specific events. The load describes the number of `NoEvents` added, i.e., a load of 1000 means that for each specific event there are 1000 `NoEvents` added before, and 1000 events added after.
Table 8.2: Quantitative tests 2 - Workload.

To measure the time it takes to deliver a notification there are some things we need to consider. We cannot simply take a timestamp on one machine and compare it with a timestamp taken on a different machine. There is no guarantee that these machines are synchronized. Instead, when EsperSens delivers the message we take the timestamp and wait for the web application to respond back to EsperSens that it has displayed the notification to the user. This time difference is divided by two, as to not include the time used by responding back to EsperSens. There are, however, some issues with this method, see Chapter 8.4.

8.2.1 Code

Quantitative Test 1
Quantitative Test 1 is coupled with the Qualitative Tests and the code therefore exists in the EsperSens package using the same classes as mentioned in Chapter 8.1.2. Tester is the class responsible for running the Qualitative Tests and Quantitative Test 1. When the runMovementPattern() of MovementSimulator is called and the sensors detect the monitored person, a DetectPersonEvent is sent to the Esper engine. To measure the event processing time we take a timestamp, using System.nanoTime(), before calling sendEvent() and another timestamp directly after. The differ-
ence is then calculated and registered by calling `setTestDelay()` located in `Tester`. `setTestDelay()` checks the time difference to see if this is either the largest or smallest difference currently registered and will add it to `maxTime` or `minTime`, respectively. It will also add the time difference to the `ArrayList avgTime`, so that we can later calculate the average of all the time differences registered.

After a test is done `writeToTestFile()` is called, which writes the result to file. The average, minimum and maximum processing time are changed to milliseconds and is rounded to two decimals.

**Quantitative Test 2**
Quantitative Test 2 is in the `QuantitativeTests` application. We chose to have a separate application for this test to provide a cleaner solution. This test may be further extended in the future and it is beneficial to not have it clutter the main application - EsperSens.

The class responsible for running the test is called `QuantitativeTester`. The constructor of this class sets the test number and tracefile to be used. To run the test `runTest()` is called. It starts by adding all the different event types to be used in the tests to the Esper engine. The Esper engine instance is fetched so that the queries may be registered. Depending on the test number, we add the according query and listener. To run the tracefile we call `runEventFile()`.

The function `runEventFile()` is responsible for parsing the tracefile and sending events into the Esper engine. It starts by reading the contents of the tracefile. Each line in the tracefile corresponds to a sensor reading. The sensor reading is added to an array so that we may access it faster and not do the reading from file while we send events into the engine. When the tracefile has been read, we take the timestamp and add it to the local variable `eventDelay` before we loop through the tuple array. If the tuple we read is a specially constructed `pause` tuple, we pause the thread to simulate a pause. After the pause is over we do a new timestamp for `eventDelay`. We check what type of information the sensor provides and construct an event with the properties specified in the tuple before sending the event into the Esper engine. When all the tuples have been looped through we call `writeToTestFile()` to write the average, minimum and maximum event processing time to file.

**Quantitative Test 3**
`WebApp` is the class that is used to do the Quantitative Test 3. The constructor of `WebApp` initializes the RabbitMQ factory, channel and queue, which allows for communication with the MOM. When `run()` is called we send a message to the web application stating that EsperSens is ready for input.
This should be considered removed as neither part, i.e., the web application and EsperSens, need to know when they start and should be independent. However, they need some communication and this has been left in for convenience and debugging.

We have created a function called `awaitingAction()` (see Figure A.1) that listens for messages sent by the web application. We have surrounded the content of the function in a `try/catch` block, printing out errors if something happens with the communication or when handling the messages. We call on `basicConsume()` with a `DefaultConsumer` which will listen for messages sent by RabbitMQ to the defined `QUEUE`. The `DefaultConsumer` uses the function `handleDelivery()` to receive the messages and inspect the content type to determine what type of message it is and what we should do with the content of the message. For instance, environment files are sent to `createEnvironmentFromFile()` and movement pattern files are sent to `createMovementPatternFromFile()` (see Figure A.1).

The `startSimulation` type indicates that the user has indicated through the web application that he or she wants to start the simulation. The `startSimulation()` function is called which starts an instance of the `Simulator` class and initiates the simulation, assuming that the environment file, movement pattern and queries file have been uploaded. The simulation is started on a separate thread so that we may continue listening for messages from RabbitMQ with `handleDelivery()` while the simulation is running.

After calling the `basicConsume()` function we set `awaitingAction()` in a `while`-loop. We need to this so we can continue the system, otherwise the system will exit. Currently the condition of the `while`-loop is set to `true`, meaning it will run "forever", i.e., until EsperSens is forcefully shut down.

On the web application, the function `start()` in `server.js` is responsible for transmitting messages sent from EsperSens (via RabbitMQ) to the web browser of the user. It is also responsible for taking commands from the user and delivering these orders to EsperSens. By using a consumer (see Figure 7.6) we fetch messages from RabbitMQ, inspect the content type and call on the appropriate functions, similar to `handleDelivery()`.

When a listener is triggered because of a matching query, the listener will call `setEventSetTime()` of the instance of the `WebApp` class we passed to `Simulator`. This function will register the current time using `System.nanoTime()`. The listener will then send a notification to the web application (via RabbitMQ). The notification is received in the consumer part of `start()` in `server.js`, which will send the notification to the web browser of the user. The client-side JavaScript code will then display the notification and send a message back to the server to confirm that the notification has been displayed. The server will send this confirmation back to EsperSens (via RabbitMQ).
and it will be received in `handleDelivery()`. The function takes the current time and uses the previously set time to calculate the difference. This event delay is divided by two and displayed to the terminal.

### 8.2.2 Workload

For the Quantitative Test 1 we use the same workload as the Qualitative tests, as described in Chapter 8.

We use `TupleGenerator` to create the data tuples for Quantitative Test 2. The generator creates a specific pattern of sensor readings, but allows for input of `NoEvents`. `NoEvents` are events that are of no interest to any of the queries in the test. They are merely used to simulate a higher load. For example if we have a query looking for Temperature events we know that `NoEvents` will never trigger that query. We could have had the same result with this query using `MotionEvent` instead, but to avoid confusion with all the events we chose to use a separate event type for events that are of no interest. `TupleGenerator` takes a single integer as a parameter. `nrOfNoEvents` is the number of `NoEvents` before and after every other event of significance for the query. It has several public functions, which are `generateTuplesForQ<queryNumber>()`. For instance, we have `generateTuplesForQ1()` to generate data tuples for the query `q1`. These functions creates a list of Strings. Each string is one line in the tuplefile - this means it is either an event or a pause. We add specific events to trigger the queries and before and after each of these specific events we add a number of `NoEvents` equal to `nrOfNoEvents`. When the workload has been created, it is written to file with name `<query>_<nrOfNoEvents>-test`.

For the Quantitative Test 3 we use test number 4 from the Qualitative Tests. We chose this test as it is a simple test which triggers the listener only once. We chose a test that only triggers the listener once so that our timestamp is not overwritten by a second trigger of the listener. If the listener was triggered twice and the second trigger occurs before we receive a confirmation from the first notification, we would have gotten wrong results.

This test is not fully automatized as it requires the user to input the environment configuration, movement pattern and the queries into the web browser. We did this five times and calculated the average of the results.
8.3 Results

We performed the functionality tests for EsperSens and we can see (in Appendix C) that the language constructs I, III and IV have been evaluated correctly.

For the Quantitative Test 1, we measured clear differences on many of the tests between CommonSens and EsperSens, as seen in Figure 8.2. Although the many peaks on the graph for EsperSens, there are sets of tests where we can see that it has a faster event processing time than CommonSens, e.g., tests 68-76 and tests 103-111.

![Figure 8.2: Quantitative Test 1 - comparison between EsperSens and CommonSens.](image)

Our results from Quantitative Test 2 are shown in Table 8.3. The workload is the number of events occurring per second. The amount is divided into two and one part is placed before and the other part after a significant event that may trigger the listener.

Our results in Quantitative Test 3 show that we have on average 30 ms latency from when an event is detected by EsperSens until it is delivered to the browser of the user. This delay is mostly dependent on the quality of the network and the distance the network packets need to travel. The results presented in Table 8.4 sends a message from the server running EsperSens to RabbitMQ (on the same server), to the web application, which is on another server; and to a client accessing the browser from a computer in Oslo. The servers are located in London.
### Table 8.3: Quantitative tests 2.

<table>
<thead>
<tr>
<th>Test number</th>
<th>Workload</th>
<th>Average event processing time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2 events</td>
<td>4.37 ms</td>
</tr>
<tr>
<td>1</td>
<td>2000 events</td>
<td>50.94 ms</td>
</tr>
<tr>
<td>2</td>
<td>2 events</td>
<td>2.9 ms</td>
</tr>
<tr>
<td>2</td>
<td>2000 events</td>
<td>40.07 ms</td>
</tr>
<tr>
<td>3</td>
<td>2 events</td>
<td>2.84 ms</td>
</tr>
<tr>
<td>3</td>
<td>2000 events</td>
<td>47.35 ms</td>
</tr>
</tbody>
</table>

### Table 8.4: Quantitative tests 3.

<table>
<thead>
<tr>
<th>Test</th>
<th>Average latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>EsperSens to Web application</td>
<td>14ms</td>
</tr>
<tr>
<td>Web application to EsperSens</td>
<td>26ms</td>
</tr>
</tbody>
</table>

### 8.4 Discussion of Results

Our main concern in EsperSens is to satisfy the first requirement in Chapter 2.3: the well-being of the monitored person. As mentioned in Chapter 6, this revolves around the HCS’s ability to correctly detect events and deliver notifications of these events to the person responsible in near real-time. We see from our results of the Qualitative Tests (see Appendix C) that EsperSens is able to correctly detect events.

The Quantitative Test 1 measures how fast CommonSens and EsperSens process events. CommonSens measures the processing time as "the average processing time in milliseconds for each of the atomic queries" [7]. In Esper there is no explicit distinction between atomic queries and complex queries, therefore we measure the average processing time for the query as a whole. We see from Figure 8.2 that, on average, EsperSens has a higher processing time of events due to the many peaks in the graph.

The peaks in the graph are caused by two things. One is because of the time the Esper engine uses to construct a message for the listener, which is something CommonSens does not do. However, this only applies to the tests where the listener is called, i.e., the tests with the result 0 in the Quantitative Tests. The second is because of how Java optimizes code during runtime when the first event is sent to the engine. Java uses a just-in-time compiler that continuously analyses a program and searches for ”hot spots”, i.e., code
that is frequently run. These "hot spots" are turned into highly optimized machine code which increases the performance of the program, but has an initial cost, which is why the first event type generally uses more time.

This would make us assume that every test, especially the tests where the event listener was triggered (i.e., test result = 0, in Quantitative Tests), would show EsperSens having a much higher processing time. This is not true, as can be seen in, e.g., tests 67-76 and 103-111 (Figure 8.2). These events use the environment configuration specified in e3.env. The event configuration has multiple sensors covering multiple LoIs, which will cause more sensors to detect the movement of the monitored person. This in turn means that more DetectPersonEvents will be sent to the engine and the average processing time will be pushed to a lower amount (since only the first event, and events that trigger listeners cause the increase in event processing time).

Figure 8.3 shows the event processing time for the first event, the average processing time, the maximum processing time and the processing time when a listener has been called. This illustrates that the peaks in EsperSens’s performance (maximum processing time) is either related to the listener or the first event. We measured that some tests, e.g., 71-75 and 103-110, had higher maximum processing time than either the first event or the listener. This is because of the every operator, which causes the query to use more memory for each event sent into the engine (that matches the event type).

This is confirmed by the Esper documentation: “For time-based data windows, one needs to be aware that the memory consumed depends on the actual event stream input throughput. Event pattern instances also consume memory, especially when using the "every" keyword in patterns to repeat pattern
sub-expressions - which again will depend on the actual event stream input throughput" [19].

The results we got from Quantitative Test 2 show that in test number 1 the event processing time increases by about 11 times, for test number 2 it increases by about 14 times and for test number 3 it increases about 17 times. Even with this increase in processing time, it may still be considered in near real-time. However, having 2000 events per second is currently not a realistic scenario, as shown in datasets provided by Washington State University [23, 24]. This test merely shows that even with the unlikely amount of events per second, we can provide, with proper written queries, a HCS that can detect events in near real-time.

The final test we did was Quantitative Test 3. By adding the event processing time we received from Quantitative Test 1 with Quantitative Test 3, we can see the time it takes for EsperSens to process an event followed by delivering a notification to a user (assuming a query matched and the listener was triggered). If we add the maximum event processing time from Quantitative Test 1 (test number 98) with the average latency from Quantitative Test 3 we get 26 ms. This is over eight times as fast the average human reaction.

The Quantitative Test 3 is a simplified test to illustrate the time EsperSens uses to deliver a notification to the browser of the user. As Quantitative Test 1 and 3 measure the performance in different parts of the system, we are provided with insight into which part of the process consumes the most time and thereby we know which part can be improved. However, the Quantitative Test 3 has some issues. For instance, as explained in the introduction of Chapter 8.2, we have to take a timestamp in EsperSens when the listener has matched a query, and wait for the web application to display the notification to the user and send a message back to EsperSens that this notification has been displayed. We divide this time by two and present the time. However, this assumes that it uses the same amount of time both ways, which is not necessarily true.

Another issue is that our servers are located in London and the user who accessed the browser during the tests is located in Oslo. This increases the time used compared to having the servers and client running within the same city, or a similar geographical area confinement. However, the medium of which these network packets travel may be significantly better than what may be available in different locations in the city.

Security is an issue we have not considered in this thesis as this is beyond the scope of our thesis. Therefore we have not considered any security measures for our servers, such as a firewall. Adding the necessary security for communication between the servers for EsperSens, RabbitMQ and the web
application may greatly affect the performance.

With the Qualitative Tests we proved that EsperSens is able to correctly detect events and with Quantitative Test 1 and 3 we proved that it is able to detect events and deliver notifications in near real-time. The results of Quantitative Test 2 show that even with a much higher amount of events per second than what is probable for automated HCSs, we can still achieve the near real-time requirement.
Chapter 9

Conclusion

9.1 Summary and Contributions

In this thesis we have investigated how we can use a CEP system to realize the concepts of CommonSens and improve its use as an automated home care system. Our main contributions to this goal are:

1. We analyzed the requirements of automated home care. The analysis consists of looking at how traditional home care works, how automated home care can contribute, what sensor technology offers.

2. We investigated the different features of DSMS and CEP systems. We concluded that for automated HCS, Esper is the most viable solution for an underlying CEP system.

3. Based on our requirements analysis, we created EsperSens. EsperSens implements the three fundamental data models of CommonSens; the environment model, the sensor model and the event model. It uses the event processing engine of Esper to process events.

4. We designed and created a query translator that accepts CommonSens queries and translates it into Esper’s query language so that it may be processed with the Esper event processing engine.

5. It is important that the helping personnel are notified when something noteworthy happens in the home environment. Therefore, we created a web application that receives notifications from EsperSens (via RabbitMQ, a MOM) and displays the notifications to the helping personnel. The web application is designed to work for both desktop computers, tablets and smartphones, so that the helping personnel may access the web application at any time and any place.
A key factor for the safety of the monitored person is that the system is able to detect events correctly and in near real-time. In order to evaluate if EsperSens detect events correctly, we performed functionality tests that tested the language constructs. To investigate if EsperSens is able to detect events in near real-time we performed three quantitative tests that monitor the event processing time. The first one is the performance of the event processing during the functionality tests. The second one is the performance when running optimized Esper queries. Lastly, we measured the time it takes from when EsperSens detects something noteworthy until the web application is able to display it to the user.

We proved with the functionality tests that EsperSens is able to correctly detect events. Our results also show that EsperSens, in general, performs slower than CommonSens. EsperSens has peaks in performances due to the listener or the optimization when the first event is processed, where the biggest difference we measured was 6.4ms. We measured that it takes on average 26ms to send a notification from EsperSens until the web application is able to display it. This is over eight times as fast as the average human reaction [25]. We conclude that EsperSens is able to correctly detect events and in near real-time.

9.2 Critical Review

Our web application is intended to work on most devices and browsers. However, we have only tested on a few browsers and devices. Older versions of browsers or older operating systems on devices may not support all the features we have used and thus work incorrectly. It is crucial to map which browsers and devices the web application will work on as to not provide a solution that may contain false positives or false negatives.

We have attempted to design the web application to be as easy and intuitive as possible, but we have not considered any user interaction patterns. To ensure that the web application is easy to use, more research should be invested in user interaction. Currently, our web application only provides the helping personnel with information, but our goal is to also provide an interface for the monitored person. It is therefore of special importance to provide a good user interaction considering the monitored person may have reduced cognitive abilities.

The qualitative tests used workloads assuming only one person in the home environment, which is not a realistic scenario. There may be more than one person in a home that may need of being monitored. Also, a monitored person may live with a spouse or other family members. The
Chapter 9. Conclusion

home environment is not necessarily subjected to only permanent residents, but the home can have visitors, such as family, friends or helping personnel. The complexity of analysis greatly increases when considering multiple inhabitants and visitors. Both the qualitative and quantitative tests would be interesting to run with workloads simulating multiple people in the home environment.

During our work on this thesis, Esper has been further developed and seen many updates. We have tried to keep EsperSens up to date with the best practices that give the best performance, but it is not necessarily something we have achieved. If we had more time we would have thoroughly reviewed one major release of Esper and worked with that version. Perhaps we could have used other features that may have improved the performance more.

9.3 Future Work

For all of our tests we have only used synthetic workloads. It is necessary to test EsperSens in a real-life scenario as there may occur circumstances we have not considered. A real-life scenario may involve multiple inhabitants and multiple visitors. This means that we need to create a solution that allows us to distinguish between the people in the home environment. Some possible early solutions could be:

- Use RFID tags for both residents and visitors. Residents will always have a RFID tag on them while visitors get a “visitor-RFID” tag. This requires that each sensor reading should be appended with an RFID tag, or that each query should be aggregated with RFID tag readings.

- Considering that mobile devices are so prevalent and it is getting more common that everybody carries a smartphone when going outside (of their own home), we could possibly use an application that communicates with the smart home via Bluetooth.

We discussed a solution where multiple homes could each connect to a centralized server that would display the information it receives to the web application. It would be interesting to see how many homes could be connected before we would experience significant delay in delivering the notification from one of the homes.

Mobile devices are getting more prevalent in our daily lives. While our web application provides an interface for mobile devices, we have yet to test how the mobile network (3g/4g) will affect the delay on event notifications. We can start by simulating these networks and later extend with real-life
testing by, e.g., traveling around the city while we send notifications to the web application that displays the message to the user’s mobile device.

A major concern in automated HCS is security. When delivering information across the network we need to look into encryption methods. However, these methods should not be so complex that they affect the delay, but we need something that ensures the privacy of the monitored person. Also, we need to make sure how we can set up firewalls etc, without affecting the performance of the system.
Bibliography


Appendices
Appendix A

Code

The implementation of EsperSens can be found at this link:
https://github.com/william-almnes/EsperSens

The implementation provided by the link above was run on Ubuntu 14.04 (64-bit OS) using Oracle’s Java (jdk 1.7.0_45), with 15.6 GiB RAM and an AMD FX-8350 eight-core processor. I used Apache Ant (v1.7.1) to compile and run my code. The Qualitative Tests and the Quantitative Test 1 can be run using the shell script runTests.sh (the script must be given executable permission), but it requires Ant. To run a specific test: ant run -Darg0=test -Darg1=Tests.txt -Darg2=<testnumber>

To run EsperSens in web mode, RabbitMQ needs to be setup to enable communication and NodeJS must be installed to run the web application. This can be done on a single machine (localhost), but this will drastically differ from our results in Qualitative Test 3. The command to run EsperSens in web mode: ant run -Darg0=web

The web application, which communicates with EsperSens through RabbitMQ, can be found at this link:

https://github.com/william-almnes/WebSens
public void awaitingAction() {
    // ...
    channel.basicConsume(Queue, false, new DefaultConsumer(channel)
    {
        @Override
        public void handleDelivery(String consumerTag,
        Envelope envelope,
        AMQP.BasicProperties
        properties,
        byte[] body)
        throws IOException
        {
            String type = properties.getContentType();
            String delivery = new String(body);

            try {
                if (type.equals("envFile")) {
                    createEnvironmentFromFile(delivery);
                } else if (type.equals("movFile")) {
                    createMovementPatternFromFile(delivery);
                } else if (type.equals("CSquery")) {
                    createQuery(new String(delivery));
                } else if (type.equals("startApplication")) {
                    startSimulation();
                } else if (type.equals("eventDelay")) {
                    long now = System.nanoTime();
                    long diff = now − eventSentTime;
                    long diff2 = diff/2;
                    String timeUsed = new DecimalFormat("##.##").format((
                        double) diff2/1000000);
                    System.out.println("Time used: " + timeUsed);
                } else if (type.equals("timestampTest")) {
                    properties.setContentType("timestamp");
                    channel.basicPublish("", QUEUE, properties, timestamp.
                    getBytes());
                }
            }
            catch (Exception e) {
                e.printStackTrace();
            }
            channel.basicAck(deliveryTag, false);
        }
    // ...
}
Appendix B

Workloads

Table B.1: Movement patterns, part1

<table>
<thead>
<tr>
<th>File name</th>
<th>Movement pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1.mov</td>
<td>2 center, 2 bottom left, 2 center</td>
</tr>
<tr>
<td>m2.mov</td>
<td>2 center, 2 left, 2 center</td>
</tr>
<tr>
<td>m3.mov</td>
<td>1 center, 1 top center, 10 center</td>
</tr>
<tr>
<td>m4.mov</td>
<td>1 center, 1 top left, 1 center</td>
</tr>
<tr>
<td>m5.mov</td>
<td>1 center, 6 bottom left, 1 center</td>
</tr>
<tr>
<td>m6.mov</td>
<td>1 center, 5 bottom left, 2 center</td>
</tr>
<tr>
<td>m7.mov</td>
<td>1 center, 1 bottom left, 6 center</td>
</tr>
<tr>
<td>m8.mov</td>
<td>1 center, 3 bottom left, 4 center</td>
</tr>
<tr>
<td>m9.mov</td>
<td>1 center, 1 top right, 10 center</td>
</tr>
<tr>
<td>m10.mov</td>
<td>1 center, 1 bottom right, 10 center</td>
</tr>
</tbody>
</table>
Table B.2: Movement patterns, part 2

<table>
<thead>
<tr>
<th>File name</th>
<th>Movement pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>m11.mov</td>
<td>1 center, 1 bottom left, 10 center</td>
</tr>
<tr>
<td>m12.mov</td>
<td>1 center, 1 top left, 10 center</td>
</tr>
<tr>
<td>m13.mov</td>
<td>1 center, 6 top right, 1 center</td>
</tr>
<tr>
<td>m14.mov</td>
<td>1 center, 5 top right, 2 center</td>
</tr>
<tr>
<td>m15.mov</td>
<td>1 center, 3 top right, 4 center</td>
</tr>
<tr>
<td>m16.mov</td>
<td>1 center, 1 top right, 6 center</td>
</tr>
<tr>
<td>m19.mov</td>
<td>1 center, 1 bottom left, 1 top left, 1 top right, 1 bottom right, 7 center</td>
</tr>
<tr>
<td>m20.mov</td>
<td>1 center, 3 bottom left, 3 top left, 3 top right, 3 bottom right, 1 center</td>
</tr>
<tr>
<td>m21.mov</td>
<td>1 center, 2 bottom left, 3 top left, 3 top right, 3 bottom right, 1 center</td>
</tr>
</tbody>
</table>

Table B.3: Complex queries, part 1

<table>
<thead>
<tr>
<th>File name</th>
<th>Query description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cq1.qry</td>
<td>A complex query containing a single atomic query with no LoI or temporal properties.</td>
</tr>
<tr>
<td>cq2.qry</td>
<td>A complex query containing a single atomic query with no temporal properties, but specifying a LoI (LoI3).</td>
</tr>
<tr>
<td>cq3.qry</td>
<td>A complex query containing a single atomic query that specifies (tb = 5) epochs and (LoI = loi3).</td>
</tr>
<tr>
<td>cq4.qry</td>
<td>A complex query containing a single atomic query that specifies a time period from epoch 1 (tb) to epoch 6 (te) and (LoI = loi3).</td>
</tr>
<tr>
<td>cq9.qry</td>
<td>A complex query which contains two queries that are related to each other with a (\land) operator. The queries is the same as the query specified in cq1.qry.</td>
</tr>
<tr>
<td>cq10.qry</td>
<td>A complex query which contains two queries that are related to each other with a (\land) operator. The queries is the same as the query specified in cq2.qry.</td>
</tr>
<tr>
<td>cq11.qry</td>
<td>A complex query which contains two queries that are related to each other with a (\land) operator. The queries is the same as the query specified in cq3.qry.</td>
</tr>
<tr>
<td>cq12.qry</td>
<td>A complex query which contains two queries that are related to each other with a (\land) operator. The queries is the same as the query specified in cq4.qry.</td>
</tr>
</tbody>
</table>
### Table B.4: Complex queries, part2

<table>
<thead>
<tr>
<th>File name</th>
<th>Query description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cq17.qry</td>
<td>A complex query which contains two atomic queries each specifying a LoI (loi3 and loi8), but no temporal properties.</td>
</tr>
<tr>
<td>cq18.qry</td>
<td>A complex query which contains two atomic queries each specifying a LoI (loi3 and loi8) and a tb (5 epochs)</td>
</tr>
<tr>
<td>cq19.qry</td>
<td>A complex query which contains two atomic queries each specifying a LoI (loi3 and loi8), and both tb (epoch 1) and te (epoch 6)</td>
</tr>
</tbody>
</table>

### Table B.5: Complex queries, part3

<table>
<thead>
<tr>
<th>File name</th>
<th>Query description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cq24.qry</td>
<td>A complex query which contains four atomic queries. Each atomic query is related to each other with the consecutiveness relation $\rightarrow$. The query specifies the person should first be in LoI3, then LoI4, then LoI1 and finally LoI2.</td>
</tr>
<tr>
<td>cq27.qry</td>
<td>A complex query which contains four atomic queries. Similar to cq34.qry, but the atomic queries are delta-timed. Each atomic query should last 3 epochs.</td>
</tr>
<tr>
<td>cq34.qry</td>
<td>A complex query which contains four atomic queries. Each atomic query is related to each other with the consecutiveness relation $\rightarrow$. The query specifies the person should first be in LoI3, then LoI4, then LoI1 and finally LoI2.</td>
</tr>
<tr>
<td>cq35.qry</td>
<td>A complex query which contains four atomic queries. Similar to cq34.qry, but the atomic queries are delta-timed. Each atomic query should last 3 epochs.</td>
</tr>
<tr>
<td>cq36.qry</td>
<td>A complex query which contains four atomic queries. Similar to cq34.qry, but the atomic queries are timed. Specific time periods are given and lasts for 2 epochs.</td>
</tr>
<tr>
<td>cq45.qry</td>
<td>A complex query which contains two atomic queries. The first atomic query is a delta-timed query followed by an atomic query which is timed.</td>
</tr>
</tbody>
</table>
Table B.6: Functionality tests part1

<table>
<thead>
<tr>
<th>Test number</th>
<th>Environment</th>
<th>Movement pattern</th>
<th>Complex query</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>e1.env</td>
<td>m1.mov</td>
<td>cq1.qry</td>
</tr>
<tr>
<td>2</td>
<td>e2.env</td>
<td>m2.mov</td>
<td>cq1.qry</td>
</tr>
<tr>
<td>3</td>
<td>e2.env</td>
<td>m3.mov</td>
<td>cq1.qry</td>
</tr>
<tr>
<td>4</td>
<td>e1.env</td>
<td>m1.mov</td>
<td>cq2.qry</td>
</tr>
<tr>
<td>5</td>
<td>e1.env</td>
<td>m2.mov</td>
<td>cq2.qry</td>
</tr>
<tr>
<td>6</td>
<td>e1.env</td>
<td>m4.mov</td>
<td>cq2.qry</td>
</tr>
<tr>
<td>7</td>
<td>e1.env</td>
<td>m5.mov</td>
<td>cq3.qry</td>
</tr>
<tr>
<td>8</td>
<td>e1.env</td>
<td>m6.mov</td>
<td>cq3.qry</td>
</tr>
<tr>
<td>9</td>
<td>e1.env</td>
<td>m7.mov</td>
<td>cq3.qry</td>
</tr>
<tr>
<td>10</td>
<td>e1.env</td>
<td>m5.mov</td>
<td>cq4.qry</td>
</tr>
<tr>
<td>11</td>
<td>e1.env</td>
<td>m6.mov</td>
<td>cq4.qry</td>
</tr>
<tr>
<td>12</td>
<td>e1.env</td>
<td>m7.mov</td>
<td>cq4.qry</td>
</tr>
<tr>
<td>13</td>
<td>e1.env</td>
<td>m3.mov</td>
<td>cq4.qry</td>
</tr>
<tr>
<td>34</td>
<td>e1.env</td>
<td>m1.mov</td>
<td>cq9.qry</td>
</tr>
<tr>
<td>35</td>
<td>e2.env</td>
<td>m2.mov</td>
<td>cq9.qry</td>
</tr>
<tr>
<td>36</td>
<td>e2.env</td>
<td>m3.mov</td>
<td>cq9.qry</td>
</tr>
<tr>
<td>37</td>
<td>e1.env</td>
<td>m1.mov</td>
<td>cq10.qry</td>
</tr>
<tr>
<td>38</td>
<td>e1.env</td>
<td>m2.mov</td>
<td>cq10.qry</td>
</tr>
<tr>
<td>39</td>
<td>e1.env</td>
<td>m4.mov</td>
<td>cq10.qry</td>
</tr>
<tr>
<td>40</td>
<td>e1.env</td>
<td>m5.mov</td>
<td>cq11.qry</td>
</tr>
<tr>
<td>41</td>
<td>e1.env</td>
<td>m6.mov</td>
<td>cq11.qry</td>
</tr>
<tr>
<td>42</td>
<td>e1.env</td>
<td>m7.mov</td>
<td>cq11.qry</td>
</tr>
<tr>
<td>43</td>
<td>e1.env</td>
<td>m5.mov</td>
<td>cq12.qry</td>
</tr>
<tr>
<td>44</td>
<td>e1.env</td>
<td>m6.mov</td>
<td>cq12.qry</td>
</tr>
<tr>
<td>45</td>
<td>e1.env</td>
<td>m7.mov</td>
<td>cq12.qry</td>
</tr>
<tr>
<td>46</td>
<td>e1.env</td>
<td>m3.mov</td>
<td>cq12.qry</td>
</tr>
<tr>
<td>67</td>
<td>e3.env</td>
<td>m1.mov</td>
<td>cq17.qry</td>
</tr>
<tr>
<td>68</td>
<td>e3.env</td>
<td>m2.mov</td>
<td>cq17.qry</td>
</tr>
<tr>
<td>69</td>
<td>e3.env</td>
<td>m4.mov</td>
<td>cq17.qry</td>
</tr>
<tr>
<td>70</td>
<td>e3.env</td>
<td>m5.mov</td>
<td>cq18.qry</td>
</tr>
<tr>
<td>71</td>
<td>e3.env</td>
<td>m6.mov</td>
<td>cq18.qry</td>
</tr>
<tr>
<td>72</td>
<td>e3.env</td>
<td>m7.mov</td>
<td>cq18.qry</td>
</tr>
</tbody>
</table>
Table B.7: Functionality tests part2

<table>
<thead>
<tr>
<th>Test number</th>
<th>Environment</th>
<th>Movement pattern</th>
<th>Complex query</th>
</tr>
</thead>
<tbody>
<tr>
<td>73</td>
<td>e3.env</td>
<td>m5.mov</td>
<td>cq19.qry</td>
</tr>
<tr>
<td>74</td>
<td>e3.env</td>
<td>m6.mov</td>
<td>cq19.qry</td>
</tr>
<tr>
<td>75</td>
<td>e3.env</td>
<td>m7.mov</td>
<td>cq19.qry</td>
</tr>
<tr>
<td>76</td>
<td>e3.env</td>
<td>m3.mov</td>
<td>cq19.qry</td>
</tr>
<tr>
<td>77</td>
<td>e1.env</td>
<td>m9.mov</td>
<td>cq24.qry</td>
</tr>
<tr>
<td>98</td>
<td>e1.env</td>
<td>m10.mov</td>
<td>cq24.qry</td>
</tr>
<tr>
<td>99</td>
<td>e1.env</td>
<td>m11.mov</td>
<td>cq24.qry</td>
</tr>
<tr>
<td>100</td>
<td>e1.env</td>
<td>m12.mov</td>
<td>cq24.qry</td>
</tr>
<tr>
<td>103</td>
<td>e3.env</td>
<td>m5.mov</td>
<td>cq27.qry</td>
</tr>
<tr>
<td>104</td>
<td>e3.env</td>
<td>m6.mov</td>
<td>cq27.qry</td>
</tr>
<tr>
<td>105</td>
<td>e3.env</td>
<td>m7.mov</td>
<td>cq27.qry</td>
</tr>
<tr>
<td>106</td>
<td>e3.env</td>
<td>m8.mov</td>
<td>cq27.qry</td>
</tr>
<tr>
<td>107</td>
<td>e3.env</td>
<td>m13.mov</td>
<td>cq27.qry</td>
</tr>
<tr>
<td>108</td>
<td>e3.env</td>
<td>m14.mov</td>
<td>cq27.qry</td>
</tr>
<tr>
<td>109</td>
<td>e3.env</td>
<td>m15.mov</td>
<td>cq27.qry</td>
</tr>
<tr>
<td>110</td>
<td>e3.env</td>
<td>m16.mov</td>
<td>cq27.qry</td>
</tr>
<tr>
<td>111</td>
<td>e3.env</td>
<td>m3.mov</td>
<td>cq27.qry</td>
</tr>
<tr>
<td>160</td>
<td>e1.env</td>
<td>m19.mov</td>
<td>cq34.qry</td>
</tr>
<tr>
<td>161</td>
<td>e1.env</td>
<td>m20.mov</td>
<td>cq35.qry</td>
</tr>
<tr>
<td>162</td>
<td>e1.env</td>
<td>m21.mov</td>
<td>cq35.qry</td>
</tr>
<tr>
<td>163</td>
<td>e1.env</td>
<td>m20.mov</td>
<td>cq36.qry</td>
</tr>
<tr>
<td>164</td>
<td>e1.env</td>
<td>m21.mov</td>
<td>cq36.qry</td>
</tr>
<tr>
<td>165</td>
<td>e1.env</td>
<td>m22.mov</td>
<td>cq36.qry</td>
</tr>
<tr>
<td>177</td>
<td>e1.env</td>
<td>m12.mov</td>
<td>cq45.qry</td>
</tr>
</tbody>
</table>
## Appendix C

### Results

Table C.1: Functionality tests comparing CommonSens and Esper - part1

<table>
<thead>
<tr>
<th>Test number</th>
<th>CommonSens result</th>
<th>Esper result</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>6</td>
<td>TRUE</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>6</td>
<td>TRUE</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>6</td>
<td>TRUE</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>TRUE</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>3</td>
<td>TRUE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>3</td>
<td>TRUE</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>3</td>
<td>TRUE</td>
</tr>
</tbody>
</table>
Table C.2: Functionality tests comparing CommonSens and Esper - part2

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>6</td>
<td>6</td>
<td>TRUE</td>
</tr>
<tr>
<td>35</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>36</td>
<td>6</td>
<td>6</td>
<td>TRUE</td>
</tr>
<tr>
<td>37</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>38</td>
<td>6</td>
<td>6</td>
<td>TRUE</td>
</tr>
<tr>
<td>39</td>
<td>6</td>
<td>6</td>
<td>TRUE</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>41</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>42</td>
<td>3</td>
<td>3</td>
<td>TRUE</td>
</tr>
<tr>
<td>43</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>44</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>45</td>
<td>3</td>
<td>3</td>
<td>TRUE</td>
</tr>
<tr>
<td>46</td>
<td>3</td>
<td>3</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

Table C.3: Functionality tests comparing CommonSens and Esper - part3

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>67</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>68</td>
<td>6</td>
<td>6</td>
<td>TRUE</td>
</tr>
<tr>
<td>69</td>
<td>6</td>
<td>6</td>
<td>TRUE</td>
</tr>
<tr>
<td>70</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>71</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>72</td>
<td>3</td>
<td>3</td>
<td>TRUE</td>
</tr>
<tr>
<td>73</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>74</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>75</td>
<td>3</td>
<td>3</td>
<td>TRUE</td>
</tr>
<tr>
<td>76</td>
<td>3</td>
<td>3</td>
<td>TRUE</td>
</tr>
</tbody>
</table>
Table C.4: Functionality tests comparing CommonSens and Esper - part 4

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>97</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>98</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>99</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>103</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>104</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>105</td>
<td>3</td>
<td>3</td>
<td>TRUE</td>
</tr>
<tr>
<td>106</td>
<td>3</td>
<td>3</td>
<td>TRUE</td>
</tr>
<tr>
<td>107</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>108</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>109</td>
<td>3</td>
<td>3</td>
<td>TRUE</td>
</tr>
<tr>
<td>110</td>
<td>3</td>
<td>3</td>
<td>TRUE</td>
</tr>
<tr>
<td>111</td>
<td>3</td>
<td>3</td>
<td>TRUE</td>
</tr>
<tr>
<td>160</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>161</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>162</td>
<td>3</td>
<td>3</td>
<td>TRUE</td>
</tr>
<tr>
<td>163</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>164</td>
<td>3</td>
<td>3</td>
<td>TRUE</td>
</tr>
<tr>
<td>165</td>
<td>3</td>
<td>3</td>
<td>TRUE</td>
</tr>
<tr>
<td>177</td>
<td>3</td>
<td>3</td>
<td>TRUE</td>
</tr>
</tbody>
</table>