Karlsruhe Institute of Technology

TECO

Enabling Domain-Specific Rule-Based Automation With Semantic Stream Technology

Master's Thesis
from

Michael Jacoby

Chair of Pervasive Computing Systems/TECO
Institute of Telematics
Department of Informatics

Reviewer #1:	Prof. Dr. Michael Beigl
Reviewer #2:	Prof. Dr. Hannes Hartenstein
Supervisor:	Dr. Till Riedel

Editing Time: 16/06/2014 – 15/12/2014

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe, den 15.12.2014

Zusammenfassung

Ziel dieser Arbeit ist die Zusammenführung von Semantischen Technologien, Streaming Technologien und benutzerzentrierter regelbasierter Heimautomatisierung mit dem Zweck dadurch die Entwicklung von mächtigeren und benuzterfreundlicheren Heimautomatisierungssystemen zu ermöglichen. Um dies zu erreichen verbindet diese Arbeit die relativ neue Technologie des Semantic Streamings mit Heimautomatisierung unter Berücksichtigung der Anforderung und Perspektive der Benutzer.

Ein Beitrag dieser Arbeit ist eine systematische Literaturanalyse existierender Anwendungsfälle der regelbasierten Heimautomatisierung sowie deren Klassifikation anhand neu gefundener Kriterien. Als Teil der Analysephase wurde eine partizipative Designstudie durchgeführt die ergab, dass Benutzer visuelle Programmier- bzw. Abfragesprachen im Bereich der Heimautomatisierung präferieren. Außerdem wurde ein neues Abfragemuster im Bereich des Semantic Streamings entdeckt das als Dynamische Sensorselektion bezeichnet wird.

Als Hauptbeitrag dieser Arbeit wurde ein Heimautomatisierungssystem basierend auf Semantic Streaming entwickelt, genannt ERAS (Event and Rule Automation System) das auf zwei hierarchisch angeordneten visuellen Sprache basiert, genannt Event Language (EL) und Rule Language (RL). ERAS wurde vollständig implementiert und umfasst ebenfalls je einen grafischen Editor für die beiden visuellen Sprachen.

Da sich alle existierenden Semantic Streaming Systeme und Benchmark-Tests als nicht passend für diese Arbeit herausstellten wurde außerdem ein eigenes solches System implementiert, genannt ECQELS (Extended Continuous Query Evaluation over Linked Stream) sowie ein entsprechender Benchmark-Test der die durchschnittliche Antwortzeit als Kriterium für den Vergleich verschiedener Semantic Streaming Systeme einführt.

Das in dieser Arbeit entwickelte System wurde mit Hilfe des ebenfalls in dieser Arbeit entwickelten Benchmark-Tests mit CQELS, dem vermutlich mächtigsten existierenden Semantic Streaming System, verglichen. Dieser Vergleich ergab, dass ECQELS für eher einfache Abfragen ähnlich gute Ergebnisse liefert wie CQELS und es bei der zeitgleichen parallelen Ausführung mehrerer Abfragen sogar übertrifft. Im Weiteren wurde als Teil der Evaluation eine Studie zur Benutzerakzeptanz zum Vergleich der Benutzerfreundlichkeit der grafischen Sprache EL und ihrem textuellen Äquivalent ECQELS durchgeführt. Diese ergab, dass die Benutzer in den meisten Fällen die grafische Notation geringfügig bevorzugten, wobei die nicht gezeigt werden konnte, dass die Unterschiede signifikant sind.

Auf Basis der Ergebnisse dieser Arbeit kann man zu dem Schluss kommen, dass die Verwendung von Streaming Technologien in der Domäne der Heimautomatisierung das Potential bietet große Vorteile für die Entwicklung mächtigerer und zugleich benutzerfreundlicherer Heimautomatisierungssysteme bereitzustellen.

Abstract

The aim of this study is to consolidate semantics, streaming and user-centered rule-based home automation with the goal of thereby enabling more powerful and more user-friendly home automation systems. To reach this goal this thesis incorporates the rather new field of semantic streaming into home automation taking into account the users' needs and perspective.

This thesis contributes a structured literature review on rule-based home automation use cases including their classification by new criteria. As part of the analysis a participatory design study was conducted yielding that users prefer visual languages in the domain of home automation. Furthermore a new pattern in semantic streaming was discovered called *dynamic sensor selection*.

As main contribution of this thesis a home automation system based on semantic streaming called ERAS (Event and Rule Automation System) was developed using two hierarchically-aligned visual languages called Event Language (EL) and Rule Language (RL). ERAS was completely implemented also containing an editor for each of the two visual languages.

As existing semantic streaming engines and benchmarks for semantic streaming engines did proof not suitable a custom semantic streaming engine called ECQELS (Extended Continuous Query Evaluation over Linked Stream) was designed and implemented as well as a corresponding benchmark introducing the average response time as relevant feature for comparison of semantic streaming engines.

The system developed in this thesis was evaluated by performing the implemented benchmark on ECQELS comparing it to CQELS, probably the most powerful existing semantic streaming engine, which yielded that ECQELS can compete with CQELS for rather simple queries. When executing multiple queries simultaneously ECQELS event outperforms CQELS. Furthermore a user acceptance study was conducted as part of the evaluation comparing the visual language EL and the textual language ECQELS regarding usability. As outcome of the study EL was slightly preferred by the users in most cases but differences could not be proven significantly.

On the basis of the results of this research, it can be concluded that the use of semantic streaming technologies in the domain of home automation has the potential to provide great benefits for developing more powerful and nevertheless more user-friendly home automation systems.

Contents

1 Introduction **1**
1.1 Objective . . . 2
1.2 Structure of the Document . . . 2

2 Background & Related Work **5**
2.1 Home Automation . . . 5
2.1.1 Systems using Textual Languages . . . 6
2.1.2 Systems using Visual Languages . . . 8
2.1.3 Analysis and Comparison . . . 9
2.2 Semantic Streaming Technology . . . 13
2.2.1 Data Stream Management Systems . . . 14
2.2.2 Complex Event Processing . . . 15
2.2.3 Semantic Streaming Technologies . . . 15
2.2.4 Analysis and Comparison . . . 17
2.3 Summary . . . 18

3 Analysis **21**
3.1 Smart Home Use Case Classification . . . 21
3.1.1 Use Case Categories . . . 21
3.1.2 Applying the Classification . . . 23
3.2 Participatory Design Study . . . 25
3.2.1 Form of the Experiment . . . 25
3.2.2 The Participants . . . 26
3.2.3 Introduction to Topic . . . 26
3.2.4 Procedure and Presented Use Cases . . . 26
3.2.5 Evaluation and Discussion . . . 27
3.3 Summary . . . 28

4 Design **31**
4.1 System Overview . . . 32
4.1.1 Architecture . . . 34
4.2 ERAS Ontology . . . 35
4.3 Event Language . . . 37
4.3.1 Metamodel . . . 38
4.3.2 Runtime . . . 41
4.4 Rule Language . . . 42
4.4.1 Metamodel . . . 42
4.5 Summary . . . 44

5 Language and Framework Implementation 45
5.1 Development Environment, Tools and Libraries . . . 45
5.2 Event and Rule Automation System . . . 46
5.3 Event Language . . . 47
5.3.1 Language and Editor . . . 48
5.3.2 Runtime and ECQELS . . . 50
5.3.3 Compilation . . . 56
5.4 Rule Language . . . 57
5.4.1 Language and Editor . . . 58
5.4.2 Runtime and Compilation . . . 58
5.5 Summary . . . 59

6 Evaluation 63
6.1 Performance Analysis . . . 63
6.1.1 Benchmark Design and Runtime Environment . . . 63
6.1.2 Results and Analysis . . . 65
6.2 Use Case Expressiveness . . . 69
6.3 User Acceptance Study for the Event Language . . . 72
6.3.1 Study Design . . . 72
6.3.2 Study Outcome . . . 76
6.3.3 Outcome Analysis . . . 78
6.4 Summary . . . 80

7 Conclusion and Future Work 83
7.1 Conclusion . . . 83
7.2 Future Work . . . 84

Bibliography 85

1. Introduction

“A smart home is a residence equipped with a communications network, linking sensors, domestic appliances, and other electronic and electric devices, that can be remotely monitored, accessed or controlled, and which provide services that respond to the needs of its inhabitants”[29]. Over the last years the number of commercially available interconnective home appliances massively increased whereas at the same time the prices dropped significantly. Probably the most well-known products for the smart home are Nest Thermostat[1] taken over by Google just recently, Philips Hue[2], the product line WEMO from Belkin[3] and the company INSTEON[4] which both provide a very wide assortment comprising different sensors and actuators. Due to this rapidly increasing market for smart and interconnective appliances the field of smart homes comes to the center of attention in research and industry. Apart from the hardware-centric aspect also on the software side a lot of movement can be observed as several of the big players are working on or already offering smart home software solutions like Apple's HomeKit[5], Samsung's Smart Home[6], AT&T's Digital Life [7], Honeywell's Evohome[8] and Google's “Works with Nest” project[9].

Surprisingly most of the currently available software solutions rather focus on controlling the home than on automating it whereas research has recognized the need for user-centered home automation systems as shown later in this work in Section 2.1. Furthermore some vendors keep their systems closed to devices produces by themselves (or at least licensed by them) by using proprietary standards like communication protocols instead of openly available and accepted ones. Even though the vendors usually state that this is done to guarantee quality of the provided services it is often perceived as a trick to keep the user buying special hardware which

[1]https://nest.com/
[2]http://www.meethue.com
[3]http://www.wemothat.com/
[4]http://www.insteon.com/
[5]https://developer.apple.com/homekit/
[6]http://www.samsung.com/de/promotions/ifa2014/
[7]https://my-digitallife.att.com/learn/
[8]http://getconnected.honeywell.com/
[9]https://nest.com/works-with-nest/

ensures the company further profit. This is also known as vendor lock-in and usually neither desired by nor beneficial for the user.

The home automation system drafted in this work supports the user in automating his home with the constraints to make this task as easy and intuitive as possible and to be capable of being integrated with any devices and communication protocol. This work uses ontologies and semantics to anticipate any sort of vendor lock-in as ontologies are a known way to enable knowledge sharing and thereby interaction in a specialized domain such as home automation[39].

At the same time using ontologies enable the definition of abstract and therefore powerful rules for home automation as they provide a unified conceptualization of the domain knowledge. Unfortunately the additional power of ontologies does not come for free. Working with ontologies is much slower compared to classical database systems[36]. This is especially the case when dealing with rapidly changing data that needs to be inserted/updated like the data produced by all the sensors in the smart home as semantic database are optimized for bulk updates and inserts [82]. To neither loose performance nor scalability due to the use of semantics this work will make use of the upcoming concept of semantic streaming. Semantic streaming allows to handle vast amounts of rapidly changing semantic data by clustering them into streams and applying so-called window operators to define that parts of the data stream that is of interest to the application or user. Following the concept of semantic streaming a system therefore immediately drops any incoming information if it is not explicitly stated to be of interest at the current time and keeps this information only as long as it can contribute in answering any query. This behavior is also very well suited for the dynamic and error-prone nature of smart homes consisting of multiple partially mobile, battery-driven and/or wireless connected smart devices.

1.1 Objective

The objective of this work is to consolidate semantics, streaming and user-centered rule-based home automation. As shown in Chapter 3 users do want to use a high level of abstraction when formulating rules and therefore semantics are needed in home automation. As timeliness is crucial in home automation scenarios (for example picture a use case where lights are automatically turned on when a presence detector detects movement with a delay in terms of seconds) semantic streaming technology will be used to ensure that timeliness although using semantics which are normally known for rather slow execution compared to relational systems. The system to be developed will also take into account to be easy to use by non-programming experts. This will be ensured by conducting a pre-study.

1.2 Structure of the Document

The rest of the document is structured as follows. Chapter 2 introduces the two main topics of this thesis which are home automation and semantic streaming. Also an overview on related work is given. Furthermore the presented existing solutions are analyzed, categorized and compared. Chapter 3 presents the analysis done in this work. This includes a detailed and systematic review and analysis of use cases for the smart home as well as the study design, realization and evaluation of the pre-study

conducted. From this findings the requirements for the system developed in this thesis are derived. Chapter 4 shows the design process of the system designed in this thesis named Event and Rule Automation System (ERAS) and the rationales behind the design. Also the ERAS ontology is introduced in Section 4.2. In Chapter 5 implementation details and problems are shown. Chapter 6 presents the evaluation of this thesis containing a detailed performance evaluation of the underlying, newly developed semantic streaming engine and a review on use case expressiveness where three sample use cases are implemented using the system. Furthermore it contains a user acceptance study conducted as an online survey. The thesis closes with Chapter 7 showing possible future work and giving an overall conclusion of this thesis.

2. Background & Related Work

This chapter introduces the terms home automation and semantic streaming technology and provides background information on these topics. For each of them related work and existing solutions are presented. These solutions are then analyzed, categorized and compared by suitable criteria.

2.1 Home Automation

Home automation is also known as smart house, smart home, smart living or domotics (lat. *domus* meaning house and *informatics*). It is the extension of building automation realized by Building Automation Systems (BAS) to domestic homes. Building automation describes the concept of centralized management, control and automation of electrical systems deployed in the building such as ventilation, heating, lighting and air conditioning. It is mostly used in institutional and industrial buildings and therefore its objectives are essentially reducing energy consumption and operating costs. A BAS typically consists of multiple controllers, input/output devices (like sensors and actuators) and a user interface all connected via some bus system. One of the biggest challenges is to manage the interconnectivity of these different devices, often produced by different companies, as they use different communication protocols and physical forms of communication.

Home automation takes the concept of building automation to the peoples' homes. Thereby the objectives are advanced and now also cover convenience, comfort, security and increased quality of life. When building automation is applied to domestic homes multiple new challenges arise. As the market for smart home appliances increases it gets more competitive and thus a vast amount of heterogeneous appliances are available for affordable prices today. This intensifies the problem of interconnectivity of different appliances using different physical communication channels and protocols. In this issue, home automation is closely tied to the Internet of Things (IoT) which focuses on interconnection of unique identifiable (embedded) computing devices within the existing internet infrastructure. IoT also introduces the concept of *smart objects* [62] which is defined as objects, that can describe their own possible interactions. These smart objects can be used in home automation to ease the

task of configuration of the smart home. The problem of interconnectivity is further intensified by the use of multiple different physical networks due to retrofitting. As wiring an existing home is not always feasible due to high costs and effort or even that the home is only rented a lot of home automation appliances use wireless networks.

The aspect of applying building automation to the domestic home of most interest for this work is the fact, that home automation systems should be manageable, configurable and useable by end-users. In comparison to technicians and administrators using building automation, end-users using home automation cannot be expected to have a deeper technical knowledge. Thus, the systems must be designed to be easy to manage, easy to configure and easy to use[45]. Given this, it is necessary to abstract from the technical details and find a level of abstraction non-expert users can understand and use. To investigate how such a level could be like we take a look at the interaction between end-user and the home automation system.

The interaction of end-users with smart homes can be divided into configuration/ management, monitoring, control and automation. Monitoring and control are basic functions from building automation and solutions are well known. The challenges of home automation are configuration and management as the deployed hard- and software is much more heterogeneous than in classical building automation and automation, which is much more complex in home automation than in building automation as it deals with objectives like convenience, comfort and increased quality of life which are way harder to express in logical rules then a objectives of building automation like reduction of energy consumption and operating costs. Furthermore, a domestic home needs more specialized and personalized rules than an industrial building as it is used in a more unpredictable and changing way. Because of that, this work focuses on user-defined rule-based home automation.

In the context of this work, two opposed kind of rules are presented: production rules and reaction rules. Production rules, also called inference rules, are of the form *if ... then ...*. The execution of these rules is usually triggered explicitly by a user and it is executed in a stateless manner. The second kind of rules are the reaction rules, also called Event Condition Action (ECA) rules and have the form *when ... if ... do ...*. They are triggered automatically when events occur and are executed in a stateful manner. For home automation reactive rules clearly seem suitable as home automation is a strongly event-centric domain with all the sensor data produced.

2.1.1 Systems using Textual Languages

homeOS[48] The objective of homeOS is to build an operating system for the smart home focusing on protocol-independent integration of network devices to ease development of cross-device applications. The device services are exposed via an API written in C#. It is proposed to distribute applications through some sort of app store as in [49]. Their focus in user-interaction is on management and security, allowing users the define devices as extra sensitive regarding security (e.g., locks and cameras). Rules of simple *if...then...* form can be created through interaction as shown in Figure 2.1a.

SPOK (Simple PrOgramming Kit)[40] The objective of SPOK is to develop an end-user development environment for smart homes. It allows for describing ap-

plications with a dedicated pseudo-natural language which is adapted to the devices connected to the smart home at runtime by using the systems device repository and meta data. The system is based on OSGi and ApAM (Application Abstract Machine)[44] and supports interconnectivity by defining the concept of adapters (based on OSGi containers) realizing the physical integration of devices. The SPOK language is a combination of rule-based and imperative programming and follows the ECA pattern. Conditions can refer to events or states, whereat conditions referring to states are translated to refer to the two corresponding events for entering and leaving the state. It comes with a visual editor supporting the user in combing blocks of text elements to define rules which is connected to the system via HTTP (see Figure 2.1b).

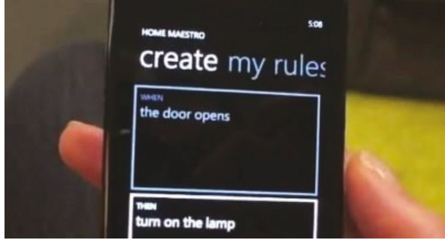
Homer[67] Homer is a fully functioning home automation system focusing on the integration of "any type of device, appliance or home service"[67]. It defines its own abstraction layer for interconnectivity by using Java and OSGi (Open Service Gateway initiative)[1] and also supports self-registering components. Component abstraction is done by treating components as services defining events (called triggers), conditions and actions which are specific per device type. Homer supports monitoring and control of all devices and also rule-based automation using the concept of policies following the pattern *when...do...* as shown in Figure 2.1c. The *when* part can contain multiple *and*, *or* and *then* (which referrers to temporal logic) commands and the *do* part can contain multiple branches in form of *if*s and *event*s which can be seen as commands/actions. Further a time interval is specified per policy in which all conditions must be fulfilled to execute the *do* part. The system does not define any user interface for specifying policies but rather allows them to be remotely created via HTTP and JSON (JavaScript Object Notation)[42].

User-configurable Semantic Home Automation System (USHAS)[59] USHAS combines web services, BPEL, OWL and their self-created markup language SHPL (Semantic Home Process language) into a home automation system that enables user-based semantic definition of processes. Two of the assumptions made are that automation processes are better understandable if semantic concepts are used and if they are defined in a common process pattern. They also use a publish-subscribe approach for distributing generated semantic events. Their introduced language for describing semantic processes SHPL is XML-based and defines the four core elements of home automation processes: preconditions, variables, execution time and flow of invocations. Preconditions can only be combined using conjunction, therefore the flow of invocation supports a *for all* operator. An example rule can be seen in Figure 2.1d. The system is exclusively tailored to home automation.

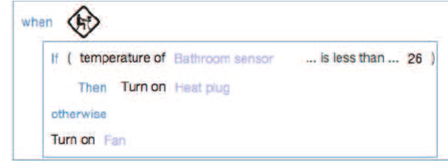
IFTTT (If This Than That)[1] IFTTT is not a classical home automation system but rather a software that allows for very easy end-user automation. Although home automation has not been its purpose it has advanced over the years and now allows to realize multiple typical home automation tasks. It is based upon simple *if...then...* rules called recipes (see Figure 2.1e). As it does not focus on integrating physical devices the basic building blocks of IFTTT are called channels acting as device abstraction. Channels expose triggers which the if-part of a recipe is composed of and actions which form the then-part of a recipe. Furthermore a recipe can contain variables shared between if and then part called ingredients. To this

[1]http://www.osgi.org/

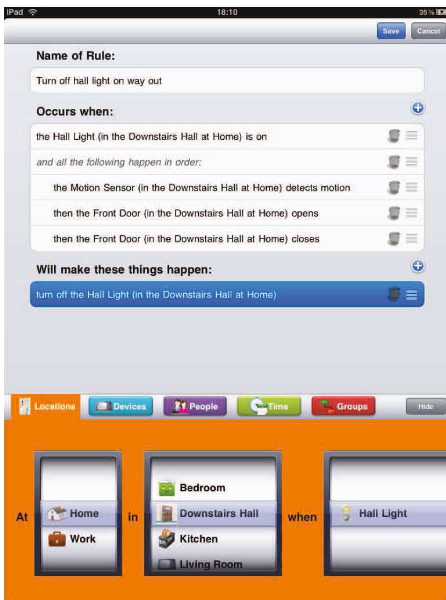
day there exist multiple channels for integrating home automation devices like thermostats (from Nest and Honeywell Evohome[2]) and multiple devices from Belkin's WeMo system like switches, plugs and sensors for motion control, temperature and brightness.



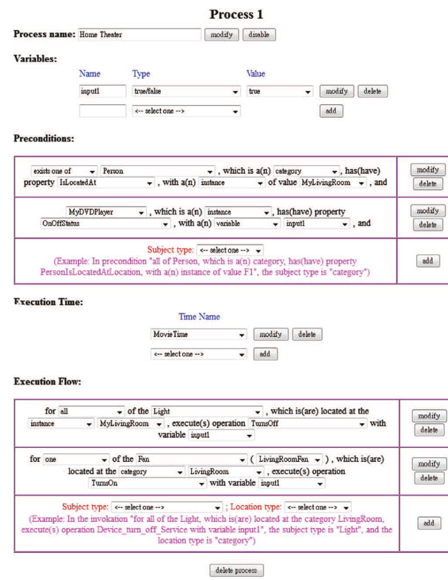
(a) homeOS (screenshot of video from[2])



(b) SPOK (from [40])



(c) Homer (from [67])



(d) USHAS (from [59])

(e) IFTTT (from [1])

Figure 2.1: Overview of the UIs of home automation systems using textual languages.

2.1.2 Systems using Visual Languages

homeBLOX[72] HomeBLOX is a user-centric process-driven home automation system. The user composes sequences by combining events and actions of different devices at different steps using an Android-based tablet app. An example of such a sequence is depicted in Figure 2.2a. These sequences are then translated into Business Process Execution Language (BPEL) and executed using the Apache ODE process engine. The system supports different automation protocols by using its own device abstraction layer.

[2]www.evohome.info/de/

iCAP (Interactive Context-aware Application Prototyper)[46] The goal of iCAP is to "allow[...] end users to visually design a wide variety of context-aware applications, including those based on if-then rules, temporal and spatial relationships and environment personalization"[46]. This is achieved without having the user to write any line of code. Concerning interconnectivity iCAP differs from the system described so far as it does not explicitly deal with multiple communication protocols and channels but rather relies on the Context Toolkit[47] to connect to sensors and actors. Furthermore, a repository approach is used for all user-defined artifacts thus allowing high re-usability by allowing the user to define rules by just combining previously defined artifacts. Figure 2.2b shows the user interface of iCAP including an example rule being defined.

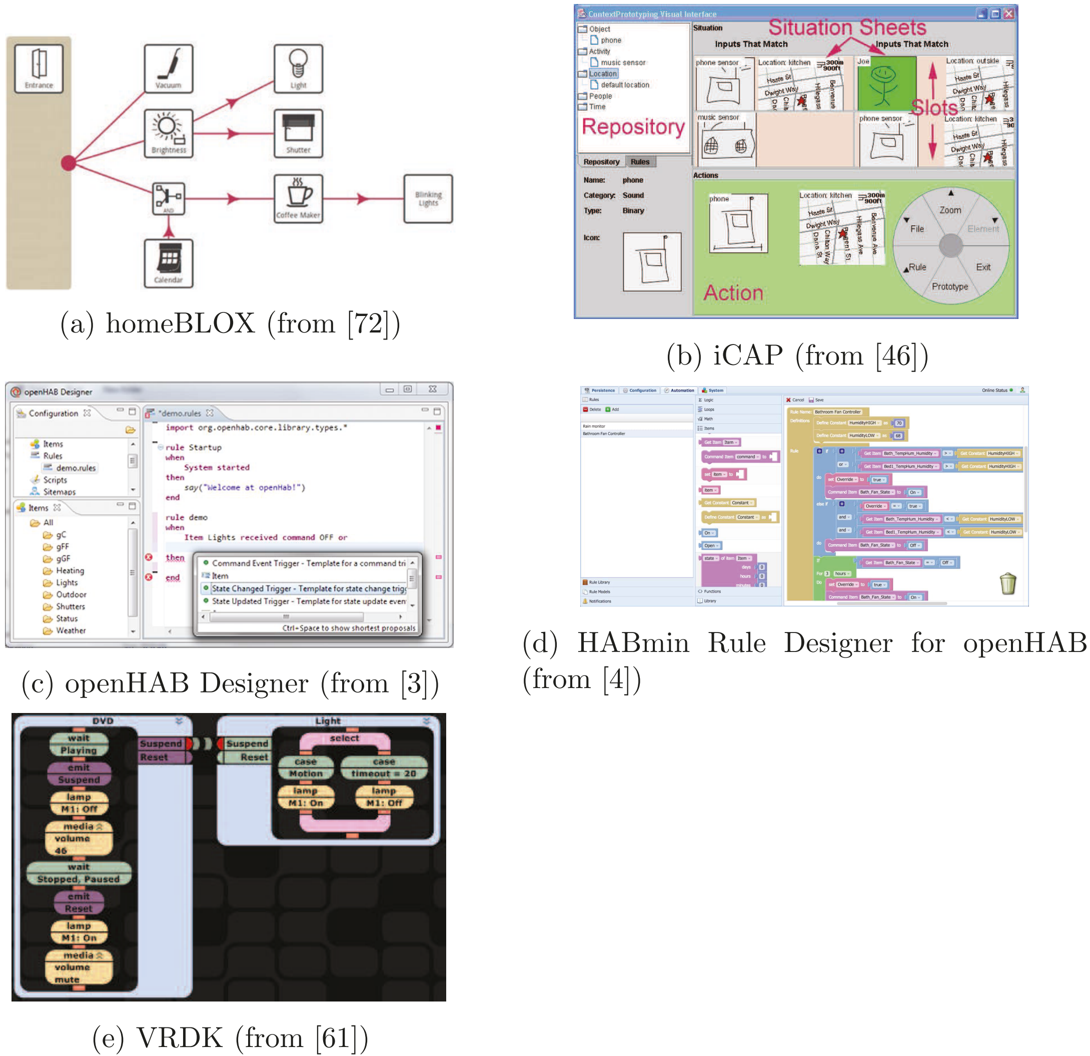
openHAB[3] OpenHAB is a vendor and technology agnostic open source home automation system based on Java and OSGi. It is probably the most wide-spread open source home automation system and won multiple awards (e.g., IoT Challenge 2013[3]). Since November 2013 the core framework of openHAB is part of the Eclipse SmartHome project[4]. It supports monitoring, control and rule-based automation of connected devices. Rules are written in a java-like language following the ECA pattern and are executed using a self-developed framework shown in Figure 2.2c. Furthermore a visual editor for rules is available shown in Figure 2.2d. In the upcoming version 2.0 it supports one-click auto-discovery of new devices though multiple protocols.

VRDK (Visual Robot Development Kit)[61] The approach for home automation described in [61] is not given a special name rather the name of the visual programming language VRDK is mentioned throughout the paper. Therefore this approach will also be referred to as VRDK if not explicitly stated that the language is meant. For context processing it takes a similar approach as iCAP by using an external context server, in this case the Nexus context-server[55]. In all aspects it greatly differs from all system mentioned so far as it works in a decentralized manner. At first a user specifies a script via an application for a tablet or a PC. A script defines which hardware (also referred to as components) it should run on and multiple processes each consisting of one workflow. Each of the previously chosen components has defined a set of commands and events which can now be combined to form the workflow with the support for elements like branches and loops. Components, commands and events are defined by plug-ins which can be loaded at runtime. After the script is ready it is compiled into executable code (currently supported are C# and C) and then remotely deployed to the matching components. Also functionality can be bound to locations, situations, persons or physical objects which defines to which components a script is deployed. Rules are defined using a visual language shown in Figure 2.2e.

2.1.3 Analysis and Comparison

In this section the just introduced home automation systems are analyzed and compared. The result is shown in Table 2.1. Due to the lack of space the information that only to openHAB and homeOS the source code is openly available is not contained in the table. Below all the criteria are explained in detail.

[3]http://iotevent.eu/application-2/announcement-the-winner-of-the-iot-challenge-2013/
[4]https://projects.eclipse.org/projects/technology.smarthome



(a) homeBLOX (from [72])

(b) iCAP (from [46])

(c) openHAB Designer (from [3])

(d) HABmin Rule Designer for openHAB (from [4])

(e) VRDK (from [61])

Figure 2.2: Overview of the UIs of home automation systems using visual languages.

Rule Representation This criteria states how rules are visualized in each system. Most of the systems do visualize the rules in a text-based manner. Even if they use visual elements they are mostly text-based. Only homeBLOX and iCAP use a mostly visual representation as shown in Figure 2.2a and Figure 2.2b.

Rule editing This criteria states in which way the user can create and edit rules. Drag & drop and dialogs are the most common editing types. Only HomeOS has a significantly different editing style. It allows specification of rules in an interactive way. The user interface consists of two blocks called *when* and *then* which can be selected and as soon as the user interacts with the smart home, e.g. by opening a door, the just performed action is added to the selected block (see Figure 2.1a). Homer features three different complexity level for the editor, simple, intermediate and advanced, and thereby accommodates the capabilities and needs of different user groups.

Low-level Rules This means if the system is designed for supporting definition of low-level rules which are rules dealing with handling of raw sensor data. Therefore, low-level rules are mostly data-centered rules doing some data transformation and filtering. In the systems not supporting this the low-level rules are implemented in

the device abstraction by a developer. Not allowing the user to define low-level rules does help him to concentrate on the higher-level logic of his rule but does at the same time limit the expressiveness.

High-level Rules By high-level rules the support of rules with a more complicated structure than simple if...then pattern is understood. In contrast to low-level rules which focus on data transformation high-level rules are used to express the logical structure of detected low-level events and therefore focus on the logical formulation of the automation task.

Event Abstraction This criteria refers to the term *event abstraction* as defined in Section 3.1.1 and specifies if the system allows to defines rules only with concrete sensors/events as triggers or with abstract ones. Only two systems support the use of abstract sensors/events as triggers which are USHAS and openHAB. USHAS can provide this functionality as it stores the meta data of all sensors in a semantic database. OpenHAB however does not use semantics and realizes this functionality by the help of a configuration file containing a hierarchy of devices and groups whereby a group is presented almost just as a regular device to the user.

Action Abstraction Action abstraction just as event abstraction refers to the concept of the same name defined in Section 3.1.1 and therefore specifies if a system only allows the use of concrete actions/actuators or also abstract ones. Again, only USHAS and openHAB support abstract ones.

Event-Action Parametrization Specifies if systems provide support for rules passing parameters between event and actions. To give an example see use case 2 in Section 3.2.4 where the sensed temperature and the room the temperature was sensed in is passed as parameter to the action. This functionality is only supported by openHAB. In some systems it would be possible to integrate without any or with only little costs but this would require the rethink the UI design and concept of user interaction.

Rule Parametrization This criteria specifies if a rule can contain (potentially unbound) variables that can be bound when the rule is instantiated/deployed. This allows to specify parametrizable rules which can be instantiated multiple times with different bindings which results in more re-usability and flexibility when dealing with dynamic adaption of rules. As an example also see use case 2 in Section 3.2.4.

	rule representation	rule editing	low-level rules	high-level rules	event abstraction	action abstraction	event-action parametrization	rule parametrization
homeBLOX	boxes (symbols) and arrows	drag & drop, dialogs	no	yes	no	no	no	no
homeOS	textual (if…then…)	interaction	yes	no	no	no	no	no
USHAS	textual (if…then…)	dialog/form	yes	yes	yes	yes	no	yes
openHAB	textual (when…if…then…) & visual (text-based)	text editor, drag & drop	yes	yes	yes	yes	yes	no
iCAP	symbols with predefined layout	drag & drop, dialogs	no	yes	no	no	no	no
VRDK	boxes and arrows	drag & drop, dialogs	yes	no	no	no	no	no
Homer	textual (if…then…)	dialog/form	no	yes	no	no	no	no
SPOK	textual with symbols (when…if…then…)	text editor, drag & drop	yes	(yes)	no	no	no	no
IFTTT	textual with symbols (if…then…)	dialog/form	yes	(yes)	no	no	no	no

Table 2.1: Classification of home automation systems surveyed in this work.

2.2 Semantic Streaming Technology

As semantics can provide a solution to the above mentioned problem of finding a level of abstraction non-experts can understand and easily use and streaming technology provides solutions for realizing reactive rules, this chapter gives a background on both, semantics and streaming technology, as well as the combination of both known as semantic streaming technology.

The term *semantics* has multiple meaning depending on the context. In this work with semantics is referred to the meaning within the Semantic Web. The Semantic Web is a concept for the further evolution of the Word Wide Web introduced by Tim Berners-Lee[35] and promoted by the World Wide Web (WWW) Consortium (W3C). Its aim is to extend the WWW which is designed for human readability with structured and therefore machine-readable information, so called semantic information or semantic data. It is therefore closely related to the IoT as they share the objective of autonomous machine-to-machine (M2M) communication. As a common data model the Resource Description Format[5] (RDF) is proposed by the W3C. It is a very basic data model consisting of statements about resources in the form *subject - predicate - object* called *triples*. RDF also defines some very basic classes like *rdf:Statement* or *rdf:Property* and also some basic properties like *rdf:type*, *rdf:subject*, *rdf:predicate* and *rdf:object*. For the practical use of RDF there are higher level concepts defined like RDF Schema[6] (RDFS), Web Ontology Language[7] (OWL) and also OWL2[8].

These concepts of the Semantic Web are helpful in finding a suitable level of abstraction for non-expert users of technical systems within a specific domain such as home automation in this work as they can be used to define a domain-specific ontology. An ontology in computer science is a formal definition of types and entities within a specific domain and their interrelationship expressed via classes, individuals (instances of a class), attributes and relations. Such an ontology can act as level of abstraction between a technical system and an end-user as it is able to formally correct represent the technical details of the system and, at the same time, present the user a more understandable view of the system by abstracting from the technical details and providing a human-readable representation (via labeling of the classes, individuals and relations).

The second aspect of semantic streaming technology besides semantics is data stream processing, also known as continuous querying, and complex event processing (CEP) which are summarized as information flow processing (IFP) in [43]. In this work, the term streaming technology is used for technical systems related with IFP.

As described in [43] IFP comes to use when "applications require[...] continuous and timely processing of information"[43]. This obviously is the case in rule-based home automation and in [53] it is explicitly pointed out, that sensor networks, which represent the ears and eyes of a smart home, is a domain for streaming applications. According to [43] the requirement of timeliness can hardly be fulfilled by traditional Database Management Systems (DBMS) and that's where IFP systems have their strengths.

Data stream processing is realized by Data Stream Management Systems (DSMS) of which the typical model is shown in Figure 2.3. Like traditional Data Base Management Systems (DBMS) from which they have evolved DSMSs use a query

language to formalize the users need for information. The difference in execution is, that DSMSs use continuous (or standing) queries which, once registered in the system, are executed periodically or as new stream items arrive and actively notify the user about new results. This is also called *Database-Active Human-Passive* in [20]. Another approach subsumed by IFP is CEP. CEP systems set their focus

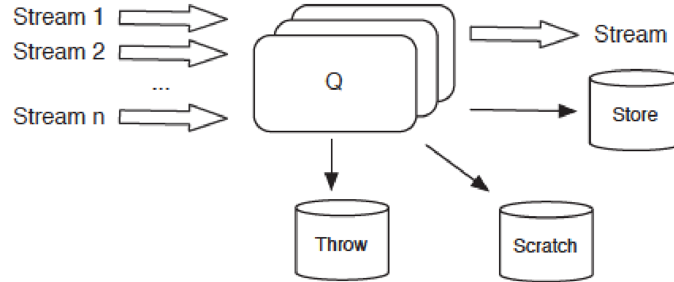


Figure 2.3: The typical model of a DSMS (from [43]).

on filtering, combining and composition of incoming event by using so-called *event patterns* to detect or infer higher-level events. These event patterns are based on casual, temporal and spatial logic[68]. CEP systems take the well-known interaction style of publish-subscribe and extend it by allowing to subscribe to composite events.

Although these two approaches seem very alike they differ in the kind of language they use. DSMS systems usually use a transformation language which can either be a declarative language describing rather what the system should do than how the system should do it or an imperative language describing how the system should act by defining a plan of primitive operators to follow thus describing the data flow. CEP systems however usually use detection languages in an ECA style with patterns as conditions.

Data streaming systems and CEP systems both lack the power of semantics and therefore semantic adaptions of both system types have emerged. They are known as semantic streaming, semantic data streaming, linked data streaming or semantic stream processing and Semantic Complex Event Processing (SCEP). Unfortunately these terms are not clearly separated or get mixed up. So most of the time when it is referred to SCEP in fact semantic streaming systems are mentioned. Combined with the use of background knowledge in the form of ontologies it is also called stream reasoning.

2.2.1 Data Stream Management Systems

CQL (Continuous Query Language)[25] CQL combines the processing of streaming and relation data and is based on SQL. It defines three kind of operators: stream-to-relation, relation-to-relation and relation-to-stream. Stream-to-relation operators allow transforming unbounded streams into bounded sets of tuples and they are implemented as sliding windows which can be time-based, tuple-based or partitioned. Relation-to-relation operators define the execution logic of the query and are directly derived from SQL. Relation-to-stream operators allow for the output of the query to be transformed into streams again. Three different operators are defined: *Istream* only streaming insertions into the underlying relation, *Dstream* analogously only streaming deletions and *Rstream* streaming the whole relation. No

in-order arrival of stream elements for incoming streams is assumed but this leads to the need of a heartbeat function which is realized as a “meta-input”. CQL is implemented in the STREAM prototype (STanford stREam datA Manager)[69].

Aurora[20] Aurora is explicitly designed for realizing monitoring applications. Monitoring applications are defined as “applications for which streams of information, triggers, imprecise data, and real-time requirements are prevalent”[20]. Aurora operates only on streams and uses the SQuAl (Stream Query Algebra) to define workflow networks in a visual editor. SquAl defines seven stream processing operators which more or less correspond to operators know from SQL namely filter (closely related to select in SQL), union, BSort, join, map, aggregate and resample. The last two can be used in combination with user-defined windows. As all data is stream-based it is discarded as soon as it has passed the system unless explicitly instructed to be persistently stored. This functionality can be used to store a history of intermediate results when needed. SQuAl also allows specification of QoS (Quality of Service) parameters for each output. These constraints are taking into account when dynamic continuous query optimization is performed which allows continuous adaption of the execution plan to the changing nature of streams.

2.2.2 Complex Event Processing

SASE [81] SASE has been designed for monitoring of real-time RFID (Radio-Frequency Identification) readings with the aim to provide a compact yet useful event detection language. The introduced language allows formulation of pattern-based detection rules for events having the structure *EVENT ... [WHERE ...] [WITHIN ...]* with the blocks in squared brackets being optional. The *EVENT* part defines which events are of interest as well as their relation and can contain sequences and logic operators. The *WHERE* part defines constraints on and selection of attributes of before select events and the *WITHIN* part allows to specify temporal windows. In [57, 21] an extension called SASE is introduced supporting aggregations and iterations. The system follows the data-flow paradigm and requires the streams of incoming events to by totally-ordered by a discrete time.

TIBCO StreamBase[9] Streambase is a commercial CEP system allowing the integration of information from heterogeneous sources. It is of interest for this work as it claims to provide “the industry’s first and only graphical event-flow language”[10] as they provide an Eclipse-based IDE for their declarative SQL-like event detection language.

Esper[11] Esper is probably the leading open-source CEP provider. They introduced their own declarative language called EPL (Event Processing Language) which allows event pattern to be expressed in two different ways: using EPL pattern defined as nested constraints consisting of conjunctions, disjunctions, negotiations, sequences and iterations as well as flat regular expressions[43]. Users can be notified via listeners of new events or consume them in a pull-based manner using iterators.

2.2.3 Semantic Streaming Technologies

EP-SPARQL[24] EP-SPARQL stands for Event Processing SPARQL and is a language for semantic CEP. It is an extension of SPARQL and adds windows, logical and temporal operators and an interval-based time model for triples. It is

implemented by the ETALIS system[5] which uses event-based backward chaining based on Prolog as execution back-end. EP-SPARQL supports static background knowledge to be integrated with streams and also RDFS reasoning.

C-SPARQL[31, 30, 32] C-SPARQL has been one of the first contributions in this area and was developed within the LarKC project (Large Knowledge Collider)[52]. It follows the intuitive approach to simply combine an existing DSMS (Esper) and an existing SPARQL engine (Apache Jena[6]). To make it work they extended the SPARQL syntax to support streaming and also included physical (triple-based) and logical (time-based) windows. All queries are executed time triggered and return always the complete query result (which corresponds to the concept of an *Rstream* as defined in CQL).

INSTANS[73, 74, 75] INSTANS approaches the problem of semantic CEP by incremental execution of standard-compliant SPARQL 1.1 queries and SPARQL Update using the Rete algorithm[27]. Its functionality can be described that it "shares equivalent parts of queries, caches intermediate matches and provides results immediately, when all the conditions of a query have been matched"[73]. RDF streams have to be fed into the system using SPARQL Insert commands and deleted with SPARQL Delete commands (that is also the way windows are implemented). Detection of missed events is possible through the use of a special "timergraph". Unfortunately it seems, that the system is not designed to work with dynamically changing queries as in setup a Rete-network is build from all queries which is then compiled into executable Lisp code.

CQELS(Continuous Query Evaluation over Linked Streams)[66, 64] As query language CQELS-QL (CQELS Query Language) is introduced which is based on SPARQL and adds constructs for streams as well as windows (tuple-based, time-based, sliding and tumbling). In contrast to all previously shown systems CQELS uses a white-box approach, that means it does not re-use any components as black-boxes but rather integrates them into the own system to allow optimization. It is developed in Java and uses Esper and Jena ARQ[7] which is a SPARQL query engine implemented in Java. It is the fastest of the presented systems which is achieved by only support data-triggered re-evaluation which corresponds to the concept of an *Istream* in CQL. CQELS also supports multiple windows on the same stream and dynamic stream selection by binding the address of the stream to a variable.

SCEPter[83] SCEPter is a semantic CEP system build around the existing CEP engine Siddhi[8] and using Apache Jena for SPARQL processing. Its unique features are that it supports querying for streaming data as well as historical data and that it is fed with streams of typed key-value pairs rather than RDF streams. Therefore it performs a semantic lifting of the incoming stream data with the use of an event ontology, domain-specific ontologies and an annotation file for every stream. For handling historical data it also extends the window concept and splits it into a query window which is used on streams as in all other system and a data window which specifies on data from which interval the query should be executed.

[5] http://code.google.com/p/etalis/
[6] https://jena.apache.org/
[7] http://jena.apache.org/documentation/query/
[8] http://siddhi.sourceforge.net/

2.2.4 Analysis and Comparison

In this section the existing solutions for semantic streaming are analyzed and compared. From the five solutions for semantic streaming only four are analyzed and the fifth, SCEPter, is left out because its source code is not public available.

Table 2.2 shows the four compared languages and all the criteria they have been compared in. In the following, each criteria is described and discussed.

Stream Data Model These criteria describe which data model is used to represent RDF streams. Most of the systems model a data stream as an unordered sequence of pairs where each pair consists of an RDF triple and a timestamp τ.

$$\begin{gathered} \dots \\ (\langle subj_i, pred_i, obj_i\rangle, \tau_i) \\ (\langle subj_{i+1}, pred_{i+1}, obj_{i+1}\rangle, \tau_{i+1}) \\ \dots \end{gathered}$$

EP-SPARQL on the other hand uses an interval-based time model assigning each triple a start and end timestamp. INSTANS is completely flexible which time model is used as it does not define any and instead new data is inserted using SPARQL Update, thus the time model depends on the structure of inserted data. Furthermore it is to distinct between triple-based and graph-based streams. As just mentioned triple-based streams define a stream as sequence of triples and therefore each triple can trigger a new result. This behavior is not always desirable as when it comes to sending events on an RDF stream it hardly can be described using a single triple. Therefore it is in some cases desirable to define a stream element as a pair of a graph consisting of multiple triples and a timestamp. From the four considered systems only INSTANS supports this functionality.

Window Support Windows are used to convert portions of a unbounded to a bounded set which can than be processed with regular relational operators. There are multiple types of windows (see [70]) of which only a subset is used in the four considered systems. It is to notice, that EP-SPARQL and INSTANS do not provide specialized language constructs for handling windows but rather do it with additional queries explicitly using the available time information. Therefore all window logic must be implemented by the user via special queries. Two kinds of windows are supported by some of the four systems. These are physical or tuple-based windows and logical or time-based windows. The criteria *windows in the past* describes if it is possible to define windows that start and end in the past or if only windows starting in the past up to the current time are possible. In C-SPARQL windows in the past are not explicitly supported but as C-SPARQL defines a custom SPARQL function to access the time stamp of a triple it is possible to realize windows in the past by right formulation of the query. Only CQELS allows multiple windows per stream. That is due to INSTANS and EP-SPARQL not having special language construct for streams but rather implicitly addressing them.

Output Type In [26] three different relation-to-stream operators are defined: *Istream*, *Rstream* and *Dstream* which help to classify the possible output formats of the four systems. An Istream only contains triples added since the last result, a Dstream only those deleted since the last result and and Rstream each time contains the whole current result and therefore does not depend on previous results. Since in

EP-SPARQL and INSTANT the user is responsible for inserting and deleting data he has full control and therefore can produce any three types of output stream if wanted. C-SPARQL does always return the whole result due to the black-box approach used whereas CQELS only support Istream output which has been a design decision in favor of very high throughput.

Execution Mode Queries can be triggered to re-evaluate as soon as new data arrives, i.e. data-triggered and also time-triggered which is the case when re-evaluation of a time-based window is triggered.

General All of the four systems can integrate static RDF data while evaluating streams, however only C-SPARQL does support some basic rule-based reasoning. Dynamic stream selection is a functionality normally not listed in any paper probably because it is not needed for the constructed use cases but it is of interest for this work, as it has shown, that it is a powerful tool when it comes to real world applications in a dynamically changing infrastructure. It describes whether streams can only be addressed explicitly via their concrete URI or if they can be addressed implicitly by writing queries that take a variable as stream URI. This functionality is only implemented in CQELS.

2.3 Summary

This chapter provided background information on home automation systems and presented existing solutions with visual as well as textual languages. Furthermore they were compared and classified with the result, that most of them are not well suited to the requirements found in Chapter 3, especially concerning action abstraction and event abstraction. Besides for most of them the source code is not publicly available.

In addition semantics and semantic streaming have been introduced as well as comparable techniques. Existing solutions for semantic streaming have been presented and compared.

	EP-SPARQL	C-SPARQL	INSTANS	CQELS
stream data model				
triples	yes	yes	yes	yes
graphs	no	no	yes	no
time model	interval	timestamp	flexible	timestamp
window support				
physical	yes	yes	yes	yes
logical	yes	yes	yes	yes
sliding	yes	yes	yes	yes
step size	no	yes	yes	yes
tumbling	no	yes	yes	yes
windows in the past	yes	yes	yes	no
multiple per stream	no	no	no	yes
output type				
Istream	yes	no	yes	yes
Rstream	yes	yes	yes	no
Dstream	yes	no	yes	no
execution mode				
data-triggered	yes	yes	yes	yes
time-triggered	no	yes	yes	yes
general				
use static RDF-data	yes	yes	yes	yes
reasoning	no	yes	no	no
dynamic stream selection	no	no	no	yes
detection of missed events	no	no	yes	no
implementation language	Prolog	Java	Lisp	Java

Table 2.2: Classification of semantic streaming languages and engines.

3. Analysis

3.1 Smart Home Use Case Classification

To identify basic needs of the users as well as weaknesses in existing systems that probably point out potential improvements we searched the literature [67, 63, 79, 28, 76, 71, 58, 50, 48, 41, 72, 59] and found 83 uses cases. The aim of this section is to identify few representative use cases to drive this work. To reach this goal, in the next sections we examine different criteria to classify use cases.

It is to mention, that the initial selection of examined papers and use cases is not representative as there exists a vast number of papers containing some sort of home automation use case.

3.1.1 Use Case Categories

By looking at the found use cases it is obvious that they can be intuitively classified by the abstract nature of the task the user wants to achieve. There are simple monitoring tasks like “remotely monitoring cameras from a smartphone”[48], control tasks like turning lights on/off[76] and automation tasks like “when the front door opens, switch on the porch light”[79]. There are also a few work-flow-like use cases (e.g. in [72]) but in this work they are seen as multiple interconnected automation tasks. As this work deals with rule-based home automation only the use cases performing automation are of relevance which make up 66 out of 83 considered use cases.

Taking a closer look at some example use cases classified as automation tasks /listed in Table 3.1) it can be seen that these are not directly applicable to any technical system as they are formulated in natural language which is claimed to be too ambiguous for human-computer interaction [23]. This phenomenon has also been described as “Inference in the Presence of Ambiguity” in [51]. In this paper Edwards and Grinter state that “systems that rely on inference will never be right all of the time, and thus users will necessarily have to have models of how the system arrives at its conclusions. These models must not only concern themselves with the actual rules of inference (“when people gather in the living room, display the television

1.	If the temperature is higher than 30 °C at 6:00 PM, turn on the air conditioner. [59]
2.	When Birgit gets up (detected by a pressure mat in front of the bed) turn on the lights in the bathroom. [72]
3.	When the front door is unlocked and the hall is dark turn on the hall lights. [67]
4.	When the living room is unoccupied for five minutes then turn of then television and speakers. [67]
5.	If presence is detected in any room and the luminosity is too low turn on the lights in the room. [50]
6.	Get notified when a door opens unexpectedly. [48]

Table 3.1: Some example home automation use cases.

schedule"), but also the capabilities of the system's sensors ("how does the system know I'm in the living room in the first place?")" [51]. This makes it harder to find a suitable classification of the use cases as they depend on how the user models or how he would like to model the model closing the gap between raw sensor data and the inferred states and events used in the use cases. Fortunately some further information how the user wants this model to behave can be concluded by looking at requirements, barriers and challenges for the smart home in the literature. There reliability is listed as a requirement [38] and as a challenge [51, 28]. Also (zero) administration is mentioned as a requirement [38, 78] and as a challenge [51, 28] respectively poor manageability and inflexibility as barriers [37]. This indicates the user wants to express this models in a manner that provide reliable results but also are abstract enough to deal with failure, re-arrangement or new installation of some sensor subsystems.

Furthermore, Edwards and Grinter [51] distinct between use cases that are possible with limited (or even no) inference, those which are possible only through inference and those which require an oracle (like the meaning of "unexpectedly" in the last use case in Table 3.1). Applying this to the mentioned sensors and actuators in the use cases and leaving out the ones that require oracles yields two criteria for classification. They both relate to the level of abstraction. The first is named *event abstraction* and distinguishes between the use of precisely named sensors and/or events (e.g. third use case in Table 3.1) and the use of sensors, states or events in an abstract manner (e.g. fourth use case in Table 3.1). The second criteria is the same the as the first one but this time applied to the action-part of the rule and is therefore named *action abstraction*.

Another feature is whether a rule is parametrizable or not. That means the if and the then part have a shared variable like room in the fifth use case in Table 3.1. This indicates that there can coexist multiple instances of that rule with different variable bindings which all have an own state and can be concurrently executed. This feature is named *parametrizability*.

The last feature introduced is named *event type* which can be time-based or event-based (which includes state changes) and also a combination of both.

3.1.2 Applying the Classification

In this section the above found classifications are used to find few distinctive and representative use cases. At first, only the use cases classified as automation tasks are taken into account which leaves 66 use cases from the total of 83 (see Table 3.2). These use cases are evaluated against the found features/classifications as show in Table 3.3.

Nature of Task	**% of use cases**	**# of use cases**
Automation	80%	66/83
Monitoring	12%	10/83
Control	8%	7/83

Table 3.2: Use cases classified by nature of task.

It shows that nearly all use cases use event-based triggers. In fact, there are only three use cases that exclusively use time-based trigger. Nevertheless, time-based triggers can not be neglected as they are used in 20% of the use cases. Looking at Table 3.4 which analyzes the distribution of the different feature vectors in detail, we can see that time-based triggers are mainly used in combination with event-based triggers and event abstraction. It is also to notice that parametrizability seems to be more important than time-based triggers as they are used almost twice as often. This seems somewhat surprisingly as time-based triggers are implemented in virtually every home automation system whereas parametrizability is relatively unused. Furthermore event-abstraction seems to be are very desirable feature (as used in 80% of the use cases) but it is also not that common in openly-available home automation systems.

Feature	**% of usage**	**# of usage**
Event-based	95%	63/66
Event Abstraction	80%	53/66
Action Abstraction	48%	32/66
Parametrizability	36%	24/66
Time-based	20%	13/66

Table 3.3: Automation use cases classification.

Feature	**pattern**									
Event-based	x	x	x	x	x		x	x	x	
Time-based				x		x	x		x	x
Event Abstraction	x	x		x	x				x	x
Action Abstraction	x				x	x		x	x	
Parametrizability	x									
% occurrence	32%	21%	15%	12%	11%	3%	2%	2%	2%	2%
# occurrence	21	14	10	8	7	2	1	1	1	1

Table 3.4: Feature vector analysis of the 66 automation use cases.

As a representative subset, three use cases are chosen in this work. The first one will be a basic one to represent the need to handle the basic functions of home automation as this makes up about 17% use the use cases as shown in Table 3.4. It will have an event-based trigger and no event abstraction, action abstraction or parametrizability.

The second representative use case will be event-based with even abstraction and thus meets the need for event abstraction used in 80% of the use cases. Also one of the first two representative use cases will have a time-based trigger in addition.

As parametrizability is used in over one-third of the use cases and also seems to be very useful to overcome know problems and barriers in current system as mentioned in Section 3.1.1, the third representative use case will be one with parametrizability. As Table 3.4 shows it only occurs in combination with event-based trigger and event and action abstraction, hence also action abstraction is covered.

To not just randomly choose any three use cases that match these specifications but rather choose representative ones we also classify the use cases by the user's need they target as mention in [28]. There, eight categories are listed, namely e-health, security, assisted living, health, entertainment, communication, convenience and comfort, and energy efficiency. In [29] these have been grouped into the three overlapping and interconnected classes energy consumption and management, safety and lifestyle support as shown in Figure 3.1. Table 3.5 shows the uses cases classified by provided service. It is to notice, that some use cases fall into two service categories and are therefore counted in both. The three representative use cases will be taken from the three most used services, i.e. convenience/comfort, energy efficiency and security. For every previously identified feature vector of interest for the representative use cases the service category with the highest matching percentage of use cases is chosen. Therefore will the first representative use case (the easy one) be taken from the service category convenience/comfort, the second one (event-based with event abstraction) will be taken from the security category and the third one (with parametrizability) will be taken from the category energy efficiency. Within these categories the representative use cases are chosen randomly. The ones chosen are listed in Table 3.6.

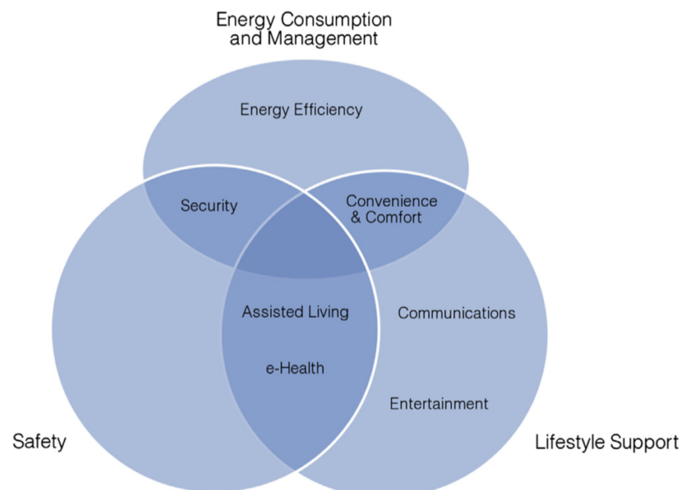


Figure 3.1: Types of smart home services (from [29])

Provided Service	% of use cases	# of use cases
Convenience/Comfort	49%	40/83
Energy Efficiency	22%	18/83
Security	13%	1/83
Communication	12%	10/83
Assisted Living	8%	7/83
Entertainment	6%	5/83
E-Health	4%	3/83

Table 3.5: Use cases classified by provided service.

1.	When the television in the living room is turned off and time is after 9:30pm do turn on the electric blanket in master bedroom and send SMS to Mary saying 'Electric blanket has been turned on!' [67]
2.	When a restricted room is entered by an employee at a prohibited time, a message is sent to the guard [50]
3.	When there is no presence in a room for 5 minutes then switch off the lights in the room [58]

Table 3.6: Chosen representative use cases.

3.2 Participatory Design Study

To get an idea on how users want to program their home a pre-study has been conducted. The goal was to find out what seems to be an easy and intuitive way for non-expert programmers to solve home automation tasks using rules. The question in detail was which kind of notation is easy to use and which components of a programming language or an IDE (Integrated Development Environment) do ease programming in the home automation context.

3.2.1 Form of the Experiment

The study was performed as an expert interview with open discussion in the group moderated by an unbiased computer scientist who has been briefed for this job. Also the whole study was monitored remotely by me with a communication link to the moderator to give instructions if he struggled with the lead through. Also a video including sound of the whole study has been recorded.

The persona of a non-expert programmer has been introduced. He should be depicted as a system administrator or an energy consultant.

The study took about 2 hours and has been split into two logical and two physical parts. It started with a short introduction of about 15 minutes which represents the first logical part followed by a discussion of the first use case which took approximately one hour. These two parts formed the first physical block and were followed by a 10 minutes coffee break. After the break the second use case has been discussed.

3.2.2 The Participants

The participants, five computer scientists, one female, four male, with level of education ranging from B.Sc. to Ph.D have been chosen. They all participated voluntarily and did not receive any reward but free snacks and coffee during the study.

3.2.3 Introduction to Topic

The introduction took about 15 minutes and encompassed seven slides. The first three slides explained the course of the study followed by three slides giving a short example-driven introduction to rule-based home automation. The last of the seven slides did introduce the persona and present the goal of this study. The slides introducing rule-based home automation contained the following two examples: The first was a shortened version of a program written in C reading the value of n temperature sensors, calculating the average of these values and if the average is greater 30 trigger sending a SMS via an API call. The second slide showed a screenshot of the openHAB designer containing some demo rules.

3.2.4 Procedure and Presented Use Cases

As all participants were native speakers of German the study has been carried out in German. After the introduction they were presented (and handed out in paper form) the first use case which is presented here in an English translation:

Bob is a system administrator and would like to monitor the temperature in the server room (named "ServerRoom1"). Therefore he scattered some µPart sensors[34, 33] in the server room. He now wants to receive a notification via SMS as soon as the average temperature of all sensor in ServerRoom1 exceeds 30°C containing the average temperature.
Bob wants to formulate the rule in a way that it still works correctly if a sensor drops out or he adds new sensors.

Afterwards they were given 5 minutes to sketch a possible programming approach each on their own on the given paper. Then randomly one expert was chosen to show his/her sketch on the white board to start a discussion and work out a collective approach. The discussion went on for about 35 minutes and then was ended due to time constraints. It was followed by a coffee break of 15 minutes.

After the break the experts were given further constraints on how to model a possible programming approach. The constraints given were: use a graphical notation, meta data of the sensors and infrastructure is given and sensors publish their data in a streaming manner. As examples for graphical notations a figure of a rule in IFTTT was given (see Figure 2.1e) as well as an example flow diagram modeled with Yahoo! Pipes [1] (see Figure 3.2) was shown. The given meta data and stream data model is depicted in Figure 3.3. After that the experts were again given 5 minutes to rethink and if needed adapt their approach each on their own followed by a discussion on how users probably would like to integrate this information.

After a discussion of approximately 10 minutes the second use case was introduced which was basically a extension of the first. It is presented again in the English translation:

[1] https://pipes.yahoo.com/

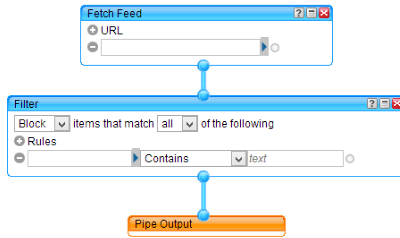


Figure 3.2: Example flow diagram with Yahoo! Pipes used in the study.

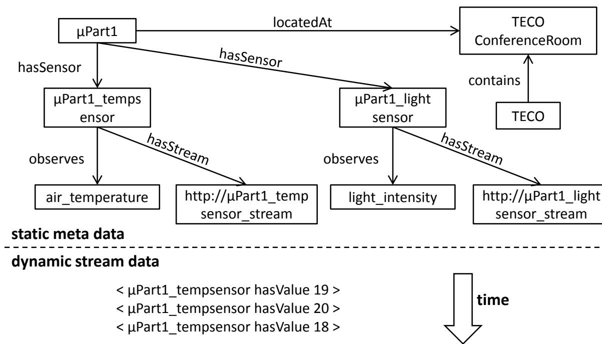
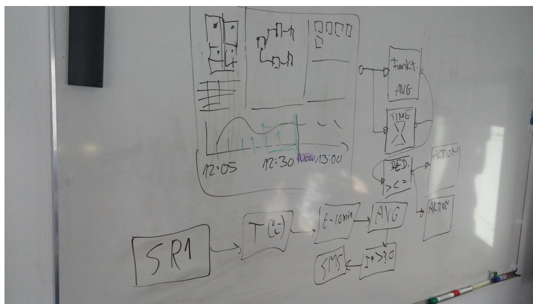


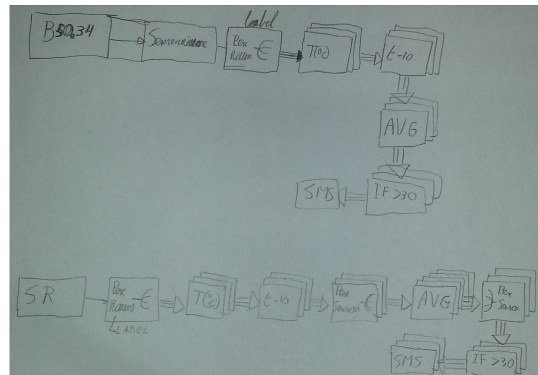
Figure 3.3: Model of meta data and stream data given to experts during study.

Bob is happy with the result of the first use case and now wants to extend is in a way that he can monitor all server rooms in the building. So a notification should be send via SMS every time the average temperature within the last 10 minutes in any server room within the building with the number 50.34 raises above 30° C. The content of the notification should be the average temperature and the room number of the room triggered the notification.
Bob wants, just as is use case one, that the rule works for further extensions like new server rooms being added without further modification.

Again the experts were given 5 minutes to find a solution on their on followed by a discussion and a collective approach developed on the board.



(a) A photograph of the board showing the collective approach for use case 1.



(b) A photograph showing the approach of an expert for use case 2.

Figure 3.4: Two results produced by the experts during the pre-study.

3.2.5 Evaluation and Discussion

As result for the first task where the experts should sketch each on their own how they suppose non-expert programmers would like to model the functionality given in use case 1 all of them came up with a visual language. That was somehow unexpected as they have not been told or shown anything about visual languages but rather have seen two examples using textual languages. As can be seen in Figure 3.4a and Figure 3.4b they all look quite similar and resemble the concept of “boxes and arrows” introduced in [20]. The also share some common operators like

boxes computing the average , filtering by a given condition or extraction a certain portion of data from a stream based on time. The last fact indicates, that the concept of windows on streams seems to be quite intuitive and easy to understand. It has also been agreed that an IDE should provide a palette of such re-usable functional blocks that can be inserted via drag & drop into a rule. This resembles the repository concept used in [46].

An other key aspect for the discussion was the (dynamic) sensor selection, especially after the constraint of accessible meta data has been introduced. It was stated that describing the sensors of interest directly by its meta data would probably overwhelm the user and so there has evolved a discussion on how the user would probably select the sensors. In that point the experts disagreed somewhat. Some proposed selection by sensor type whereas others proposed selection by location. Moreover it has been stated that its probably the most intuitive to define the sensors used by selecting the properties and entities of interest rather than the sensors or their location itself. The most supported idea for an user interface was a tree view with checkboxes to support easy and fast selection of all sensors of interest. This largely matches the concept of faceted search which probably all users will already know from online shops.

All experts furthermore agreed on a workflow they think users would follow when modeling automation rules. It is *sensor selection → time selection (windowing) → data transformation*.

3.3 Summary

In the first part of this chapter a classification of use cases found in a literature review was presented. The use cases have been analyzed and criteria for categorization have been derived. The classification showed that event abstract is a very common pattern used in 80% of the home automation use cases (see Table 3.3). Also action abstraction (used in 48%) and parametrizability (used in 36%) are not to be neglected as they form, together with event abstraction, the most common pattern within all use cases being used in 32% as shown in Table 3.4.

In the second part of this chapter the participatory design study conducted was presented. The study showed that all participants proposed the use of a visual language with a flow-based style like shown in Figure 3.4. Furthermore they agreed that metadata is needed and a repository-style system design in terms of metadata repository and also re-usable user-defined element repository is desirable. Moreover they advocated the use of dynamic sensor selection by metadata and proposed the corresponding workflow *sensor selection → time selection (windowing) → data transformation*.

Therefore the following requirements are derived for the system developed in this thesis:

- Support the three concepts found in the use case analysis and classification: event abstract, action abstraction and parametrizability.
- Design a visual language close to the one shown in Figure 3.4.

- Use semantics to make metadata easily browsable.
- Support re-use of user-defined elements to enable the use of a repository.
- Support dynamic sensor selection and use the found workflow *sensor selection → time selection (windowing) → data transformation*.

4. Design

The goal of this work is to design an easy-to-use rule based home automation system using semantic technology and semantic streaming fulfilling the requirements listed in the previous chapter. Figure 4.1 shows the abstract architecture of such a system. It's mostly characterized by its interfaces. The system contains a semantic database storing the meta data of sensor and actors connected to it as well as user-defined elements modeling the behavior of the system like rules. Such meta data could be for example type, location, measured features and address of the RDF stream the data is sent on for sensors and type, location, supported actions and corresponding parameters for actuators. It consumes the measured sensor values via RDF streams. Based on the sensor data and meta data the user-defined rules are executed and generate actions which are forwarded to the desired actuators as commands.

In general, such a system is composed of three major components: a Domain-Specific Language (DSL) to model rules, an user interface containing an editor for the DSL and a DSL runtime to execute rules defined in that DSL. In Figure 4.1 a component named *Semantic Stream Engine* is modeled as part of the *DSL Runtime* as in this work an existing semantic streaming engine should be re-used to handle RDF streams.

This abstract architecture however does not model a complete automation system as it lacks some functionality like how sensors and actuators are physically connected and how the semantic lifting is done, that means how the relational data and meta data from sensors and actuators is converted into semantic data. This integration of sensors and actuators, mostly connected with different interfaces using different communication protocols, is called physical integration layer in [56]. It is to notice, that this functionality is not part of the system designed in this work. Also the following presumptions are made.

- The meta data of all sensors and actuators connected to the system is stored in the *Semantic Database*.
- Sensor data is published as RDF stream. The address of the stream is stored in the *Semantic Database*.

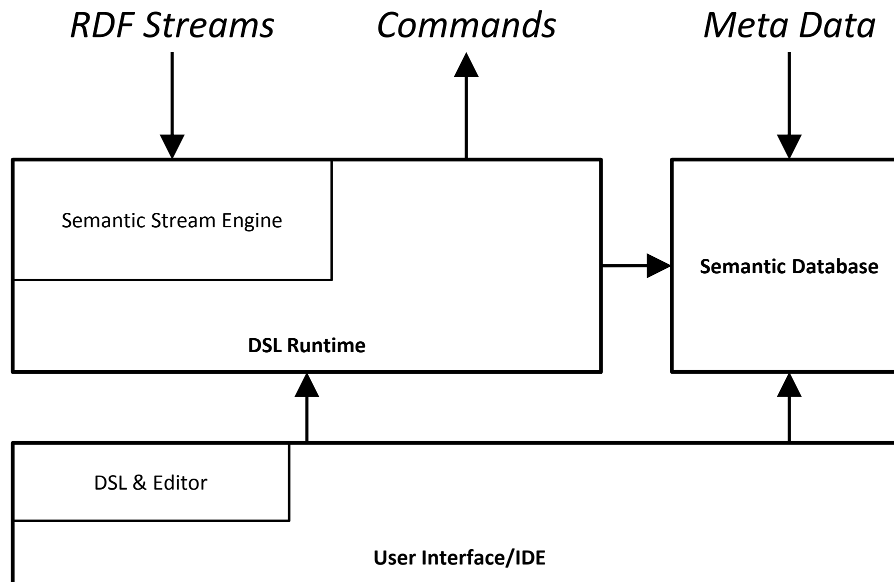


Figure 4.1: Abstract architecture of the system to be designed.

- Actuators can be controlled via uniform commands. The information on how to execute a command on an actuator is stored in the *Semantic Database*.

This chapter describes the design Event and Rule Automation System (ERAS) developed in this work. It therefore points out what design decision were made, why they were made and which alternatives were considered. The following section gives an overview of the designed ERAS and Section 4.2 introduces the basic ontology used. In Section 4.3 and Section 4.4 the two languages that are part of ERAS are introduced. This chapter is completed with a short summary in Section 4.5.

4.1 System Overview

As the use case analysis in Section 3.1.1 shows, users tend to formulate rules rather on a higher semantic level than on the sensor level. In 80% of the reviewed use cases the concept of *Event Abstract* is used as shown in Table 3.3. According to [51] this leads to the need for the user to formulate not only his goals with high-level rules but also to explicitly model how the system's sensors are integrated in this task. Therefore this work proposes to separate these two tasks by adapting the concept of separation of concerns. This will increase to ease of use and also improve re-usability by introducing modularity. To do this, the following concepts are introduced: *Event*, *Action*, *Command*, *Rule*, *Rule Deployment*, *Repository* and *Dynamic Sensor Selection*.

An *Event* is in this work defined as a notification containing the right information at the right time. The term 'right' refers thereby to the user's intention. An *Event* is always driven by incoming sensor data. Therefore it represents the explicit modeling of how sensors are integrated in the task of automation.

An *Action* is considered as some functionality an actuator can perform. It is always triggered by a *Command* send to the actuator describing the *Action* to perform and

the parameters. In this work *Commands* will be sent via REST calls as explained in Section 4.1.1 in detail.

A *Rule* is the key element bringing together *Events* and *Actions* as a sequence of reactions to *Events* and triggering *Actions*. They represent the high-level goals of the user. By allowing to chain *Events* in a sequence a simple temporal logic is provided. This temporal aspect reinforced by the concept of *Rule Deployment* which allows users the control when a *Rule* will be activated.

The concept of a *Repository* is rather optional but will highly increase user acceptance, ease of use and productivity. It stores user-defined *Actions*, *Events* and *Rules* (and *Stream Sources* which are introduced later in this work) and thus allows easy re-usability and combination (like *Events* and *Actions* inside a *Rule*) of elements.

The concept of *Dynamic Sensor Selection* allows to define the data of which sensors are of interest by a query/pattern on their metadata. This will be later on in this thesis also referred to as implicit addressing since in comparison to the classical approach to explicitly write down the sensors of interest here the sensors are chosen by evaluating the query/pattern against the metadata repository. This allows to adapt the sensors used while the query is running without any modification of the query and without stopping and restarting it.

To realize this concepts, two separate languages are designed. The first one is called *Event Language* (EL) and models the concept *Event* and the second one is called *Rule Language* (RL) and models the concept *Rule*. Figure 4.2 shows a mock-up of

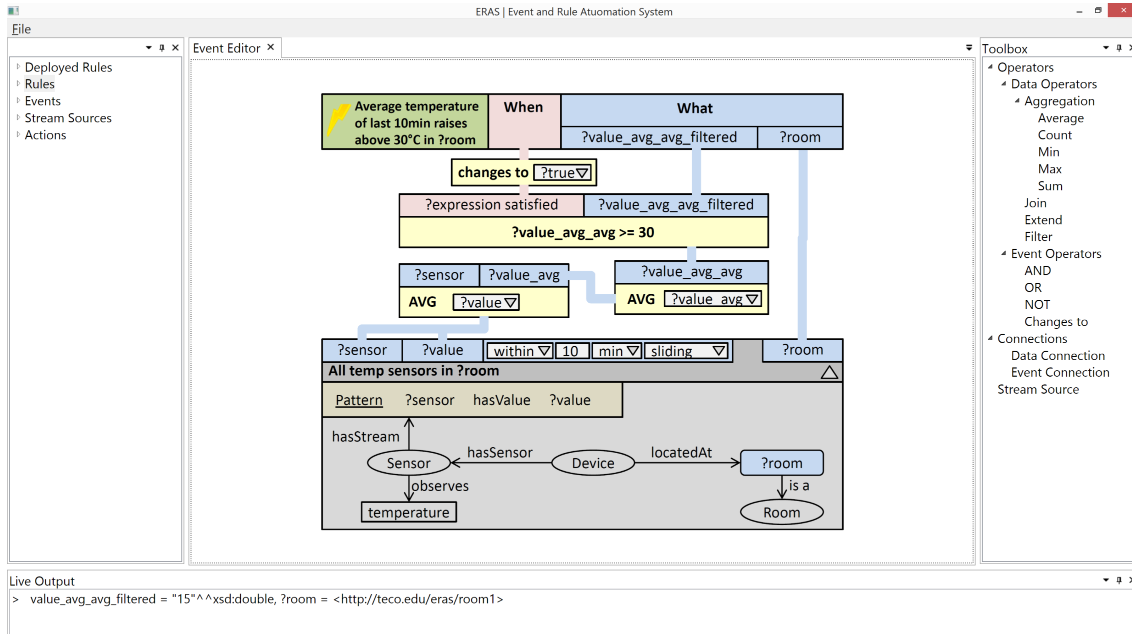


Figure 4.2: Mock-up of a possible ERAS user interface showing an example *Event* modeled with EL.

a possible user interface for ERAS. In the middle an example *Event* modeled with EL is shown which is explained in detail in Section 4.3. The purpose of this figure is to support the reader in envisioning the workflow of how a user would realize an automation task to better get a understanding the concept behind ERAS. Assuming,

that all the needed sensors and actuators are configured and connected, a possible workflow could look like this.

- The user designs an *Event* either from scratch or by reusing an existing one as a template. He starts with the selection of sensors of interest. These can be either explicitly addressed or implicitly by describing the meta data to select sensors by. To simplify the task of formulating proper meta data criteria any kind of wizard could be implemented to support the user (for example faceted search). He then models the data- and event-flow and defines the output data. Before continuing he checks the modeled *Event* via the live output window at the bottom to ensure proper functionality.
- The user has defined all *Event*s needed for modeling his rule. So he created a new *Rule* and places the just created *Event*(s) via drag and drop inside. He also adds the desired actions via drag and drop to the *Rule* and models the behavior he wants to achieve.
- After saving the *Rule* to the repository he schedules his rule to run on every start-up of the system.

4.1.1 Architecture

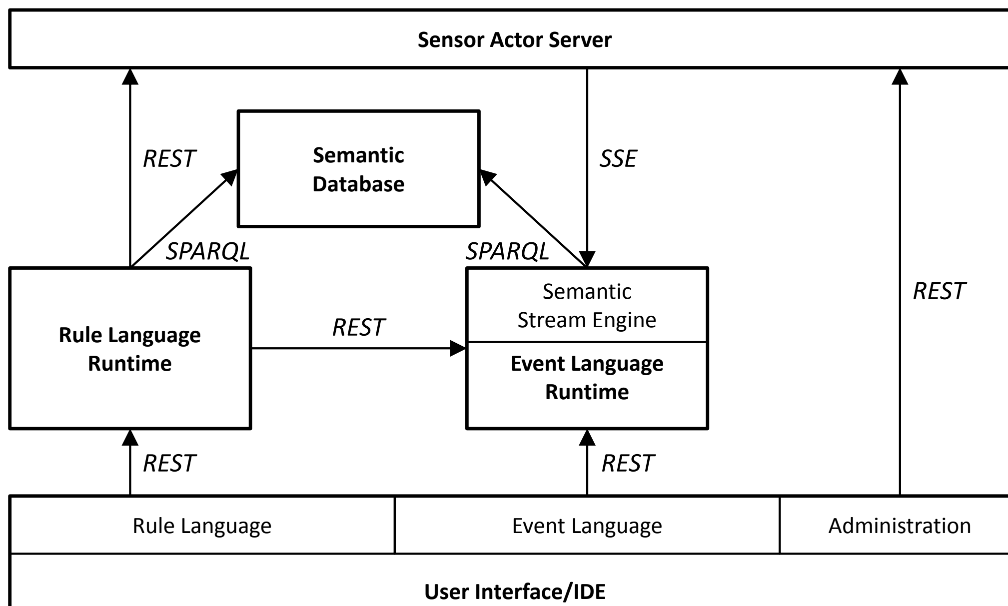


Figure 4.3: System overview.

Figure 4.3 shows the architecture of ERAS. The architecture is based on the abstract architecture shown in Figure 4.1 and adapted to the concepts introduced in the previous section. According to the decision made to have two languages the architecture now shows separate components for the RL and EL as well as separate runtimes. In addition Figure 4.3 is annotated with the protocols used for communication between the components. The communication of both of the runtimes with the *Semantic Database* is realized via SPARQL[12] which stands for SPARQL Protocol And RDF Query Language and is a query language for semantic data. It is an

official W3C (World Wide Web Consortium) recommendation and the de facto standard for querying semantic data. However, more important is the communication between the *Rule Language Runtime* and the *Sensor Actor Server* as it refers to the concept of *Commands* just introduced above. This communication aims on sending commands to an actuator. Here, REST (REpresentational State Transfer) is chosen as it is a lightweight protocol and guarantees loose-coupling and therefore is best suited for resource-constrained, ad-hoc environments like home automation[80]. The data communicated between the *Sensor Actor Server* and the *Event Language Runtime* are the RDF streams containing sensor data. Thereby it is a one-directional communication with the need for high throughput and low latency as the sensor data can arrive at high frequencies. Two protocols have been considered to be used for this communication: Server-Sent Events (SSE) and WebSockets. SSE works on top of HTTP and provides a one-way communication channel whereas WebSockets uses its own protocol based on TCP and provides a two-way communication channel. As SSE is standardized as part of HTML5 by the W3C and only a one-way communication is needed, SSE is chosen as protocol to connect RDF streams to the system. As there are no special requirements on the remaining communication interfaces they will also be realized using REST to keep the communication as simple as possible.

4.2 ERAS Ontology

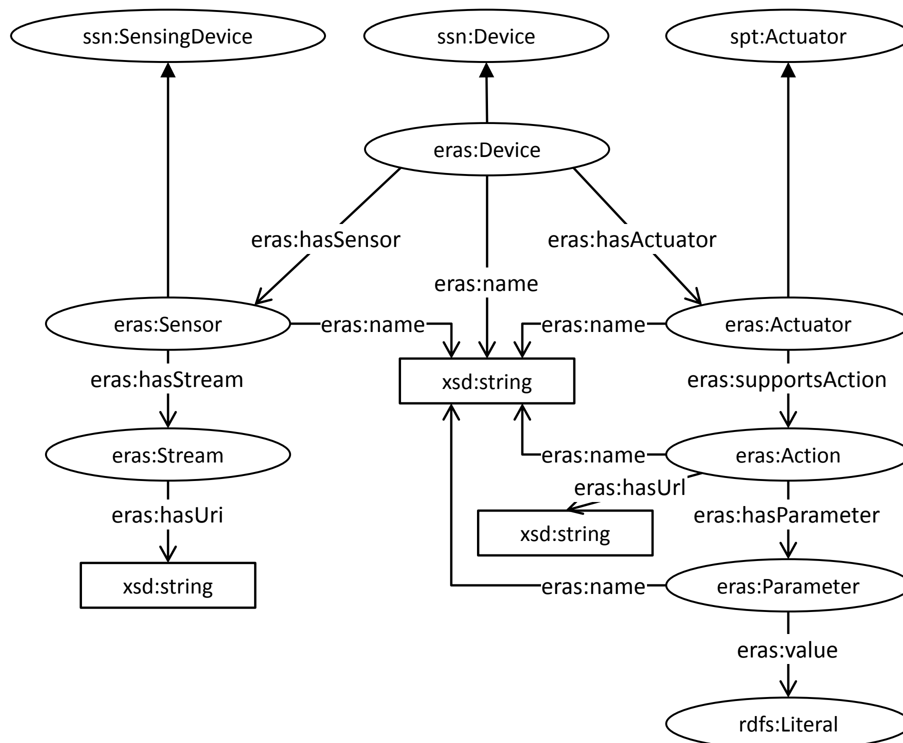


Figure 4.4: Classes and properties of the ERAS ontology.

In this section the ontology designed in this work, called ERAS ontology, is presented. The approach was to design it as simple as possible and as complex as needed. Therefore it is based on the simple RDFS and not the more complex OWL or even OWL2. Also it covers only the very essential concepts and thus leave the possibility to describe everything else needed in additional (custom) ontologies.

Figure 4.4 shows the classes and their properties of the ERAS ontology. The concept of sensors and actuators are aligned with the SSN (Semantic Sensor network) ontology[13] designed by the W3C Semantic Sensor Network Incubator Group which is close to being de facto standard for semantic modeling of sensors respectively the SPITFIRE ontology [14] (for which the prefix *spt* is used in the figure) which has been developed as part of the SPITFIRE project[15] and is itself aligned with the SSN ontology. The most important part of the ontology is the modeling of streams visible on the lower left of Figure 4.4. it is described, that a *eras:Sensor* has a *eras:Stream* which is connected via the *eras:hasStream* property. The property *eras:hasUri* in turn points to the URI of the RDF stream send via SSE. This structure allows dynamic stream selection by meta data pattern. Furthermore, the ontology models actions with the *eras:Action* class having an URL which represents the REST address to send the command to. Also the concept *eras:Device* is introduced which acts as container for sensors and actors and thereby will represent real world thing like for example a fridge in most of the cases.

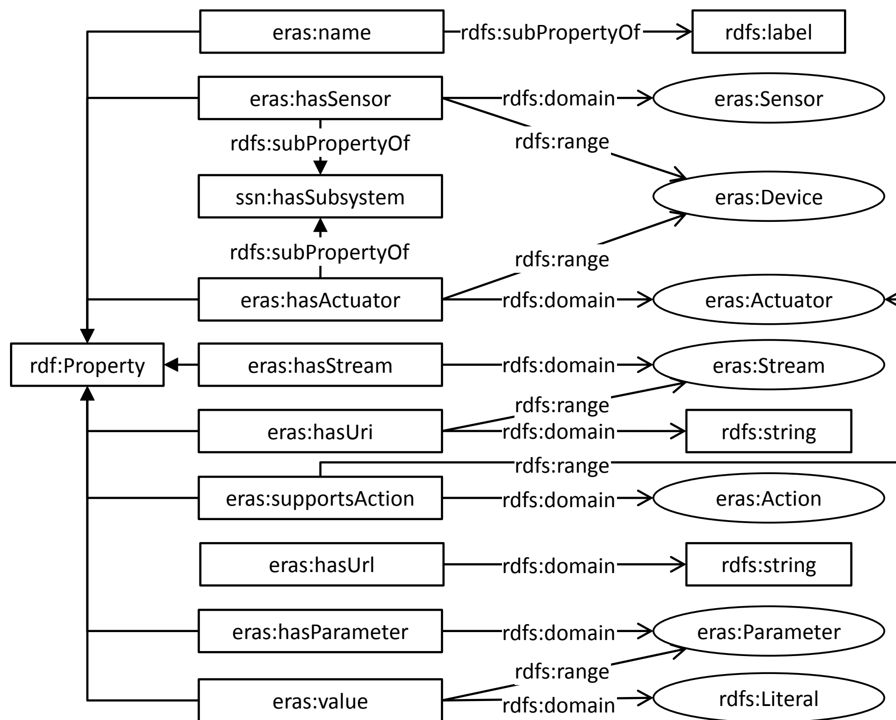


Figure 4.5: Detailed view of the properties defined in the ERAS ontology.

Figure 4.5 gives a detailed view of the properties defined in the ERAS ontology. It shows, that also the properties are aligned with RDFS and the SSN ontology. The property *rdfs:range* ensures that some property can only be applied to an instance of the class referred to by *rdfs:range*. *rdfs:domain* works in the opposite direction and ensures that all instances the referred property points to are instances of the class *rdfs:domain*. So for example *eras:hasStream* can point from any instance (as *rdfs:range* is not defined) but only to instance of *eras:Stream* (or subclasses). This allows not only sensors but every instance in the ontology to state the it is publishing some information on a RDF stream.

4.3 Event Language

In this section the design of EL is illustrated. The first design decision made was to realize it as a visual language rather than a text-based one. This decision was made as all participants in the pre-study intuitively tried to realize the given use case with some sort of visual language. According to the two-folded nature of the definition of the concept *Event* in the Section 4.1, namely a notification containing the right data at the right time, EL is designed to reflect that duality by distinguishing between data-flows, describing what data is of interest, and event-flows describing when it's the right time to trigger a notification. As the data-flow processing is more complicated than the event-flow processing EL is designed like a general query language with a few additional timing and event-handling concepts. Figure 4.6 shows a mockup of an example event modeled with EL. The event notifies when the average temperature of the last 10min in any room (defined by ?room) raises above 30 °C. On top we see one of the core concepts of EL, the event element, decorated with a flash. It defines a name for the event and combines event- and data-flow as it specifies when a notification should be generated and what data it should contain.

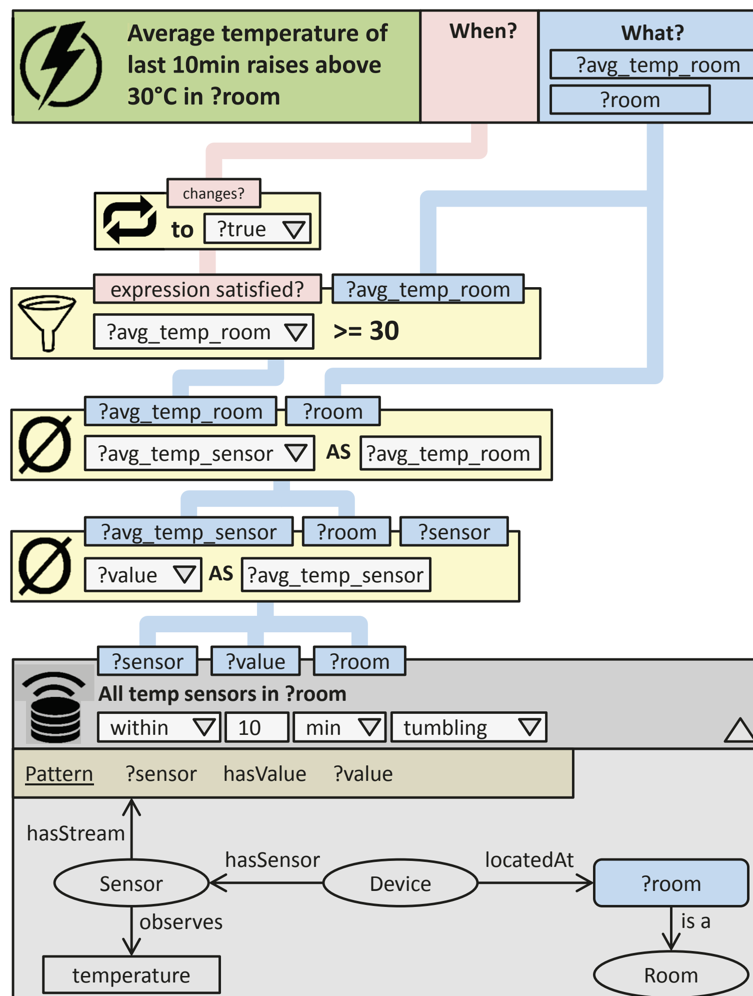


Figure 4.6: A mockup of an example event modeled with EL.

On the bottom of Figure 4.6 we see another core concept: the dynamic stream selection. As seen in Section 2.2.3 dynamic stream selection was a requirement for

the design of this language which has been accounted for with this concept. It allows selection of streams by meta data so that the sensors involved in this event can vary over time as new sensor are added, old ones are deleted or some meta data of a sensor changes. Nevertheless streams can also be addressed explicitly by providing their URI. By this, EL addresses the requirement reliability and the barrier inflexibility shown in Section 3.1.1. The concept of windowing[70] is applied to incoming RDF streams to extract only the data of interest which then can be processed as a set of triples (also known as graph). It is also allowed to introduce variables in the stream selection pattern (which determines which streams are selected). These variable can be used inside the data-flow of the event but also can be used to configure the event when called from outside, thus allowing an event to be parametrized. Furthermore, the formulation of a suitable stream selection pattern should be supported by some wizard helping the user which is not intended to be familiar with writing triple patterns. This could be realized by some sort of faceted search where the user can select the sensors currently deployed in the system he is interested in, probably by filtering by some criteria like sensor type, observed property or location, and then suggested a general pattern like 'Did you mean: all sensors observing temperature with an accuracy of at most $+-1\,^{\circ}\mathrm{C}$ in room xy?'. This kind of sensor selection would enable even users with no knowledge of semantic data to query it to express their intentions. Sadly it is a to complex topic to be covered in this work so it will be listed as a possible extension in future work.

To make it easier to distinguish between data- and event-flow they are highlighted using different colors. The data-flow is shown in blue as the event-flow is depicted in red. In the rest of this section the metamodel of EL is presented and discussed and also the design of a runtime for EL is presented.

4.3.1 Metamodel

Figure 4.7 shows the metamodel of the EL as an UML class diagram. It is as stated above basically designed to represent a regular flow-based query language but with two noticeable differences. The first one is that EL uses two kinds of flows, event-based and data-based flows. This is observable by the separation of the metamodel in two parts. The upper half of the depicted metamodel shows the elements modeling the data-flow whereas the lower half shows the elements modeling the event-flow. Between them in the middle of Figure 4.7 are three classes that are not specialized either to model event- or data-flow: *Element*, *Connection* and *Event*. The first two provide abstract superclasses for an element and a connection whereas the class *Event* is the root element of any model and thus represents any model based on this metamodel. It consists of two attributes: a name and a list of parameter names which will later be used to initially bind to some parameters and thereby making an *Event* reusable. It also contains multiple *StreamSource*s, *Connection*s and *Element*s as well as exactly one *EventSink* which represents the final element in each model. The second difference to a regular flow-based language is that in the data-flow the elements are not directly connected. Rather a *DataElement* contains multiple *DataPort*s which represent the available output parameters and act as sources for *DataConnection*s. *DataPort*s also allow multiple outgoing *DataConnection*s.

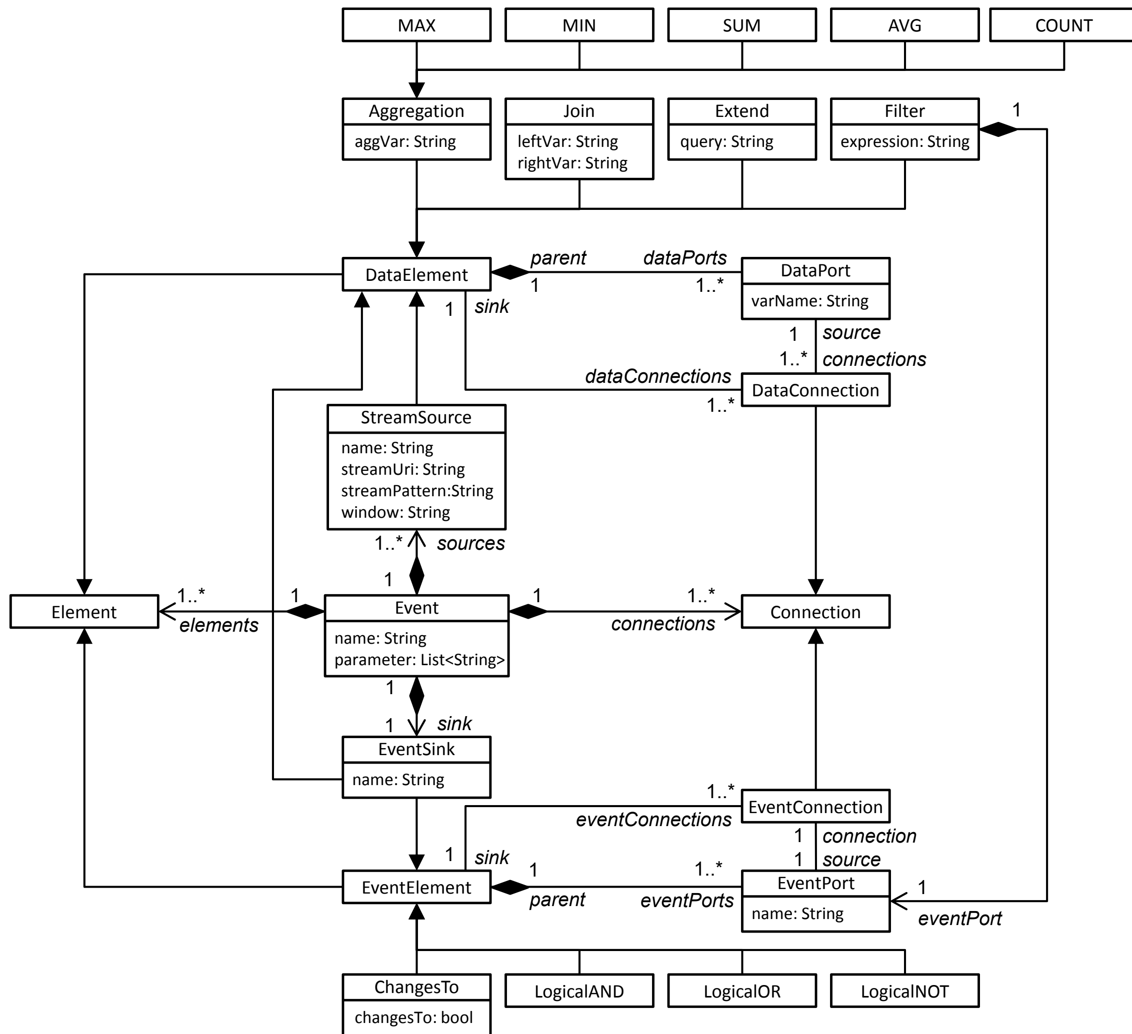


Figure 4.7: UML class diagram showing the EL metamodel.

As not all relevant definitions can be made with UML directly there also exist some constraints on the classes of the metamodel. These constraints are expressed with with the Object Constraint Language (OCL)[1]. In the following the classes of the metamodel and their OCL constraints are briefly described. Each description is followed by the corresponding constraints if there are any.

Event The root class containing all the other elements.

Connection Abstract representation of a connection and superclass of *DataConnection* and *EventConnection*.

Element Abstract superclass of all elements in an *Event*.

DataElement, EventElement Abstract superclasses of all elements for the data-/event-flow. Allow incoming *DataConnection*s/*EventConnection*s and contain any number of *DataPort*s/*EventPort*s, thus allowing outgoing *DataConnection*s/*EventConnection*s.

context DataElement **inv:**
self.dataPorts->forAll(p1, p2 | p1 <> p2 **implies** p1.varName <> p2.varName)

[1]http://www.omg.org/spec/OCL/

context EventElement **inv:**
  self.eventPorts->forAll(p1, p2 | p1 <> p2 **implies** p1.name <> p2.name)

The constraints ensure, that neither a *DataElement* can have multiple *DataPort*s with the same *varName* nor a *EventElement* can have multiple *EventPort*s with the same *name*.

DataConnection, EventConnection Represent a connection between *DataElements*/*EventElements*.

DataPort Represents a data variable which name is given by the property *varName* and allows multiple outgoing *DataConnection*s to be connected.

EventPort In similarity to *DataPort* it represents an event variable but only allows a single outgoing *EventConnection*s to be connected. Thus it is not possible to split up and event-flow like a data-flow in EL.

StreamSource Represents data coming from RDF streams specified by the *streamUri* property, matching the given *streamPattern* within a window specified by the *window* property. It is the only element without any predecessors and therefore the start of any flow. For each variable in *streamUri* and *streamPattern* it contains a *DataPort* instance with the property *varName* set to the name of the variable.

EventSink Acts as final or output element of a flow and accepts *DataConnection*s and *EventConnection*s. It is one of two elements crossing the separation of event- and data-flow.

context EventSink **inv:**
  self.eventConnections->size() <= 1
  self.eventPorts->size() = 0
  self.dataPorts->size() = 0

Constraints the *EventSink* to not have any *DataPort*s and *EventPort*s, i.e. now outgoing connections are allowed. Also only a single incoming *EventConnection* is allowed.

Filter A *Filter* allows incoming data to be filtered by the expression defined by his *expression* property. Furthermore, the *Filter* element is the only element capable of initially generating an event-flow by exposing an *EventPort* representing whether the current data match the *expression*.

Extend Defines an operation that allow to query additional data from a SPARQL endpoint.

Join Allows to merge data-flows by equating two incoming variables whose names are given by the properties *leftVar* and *rightVar*.

context Join **inv:**
  self.dataPorts->size() < self.dataConnections->size()
  self.left <> self.right

The first constraint limits the number of output variables to the number of incoming variables - 1 because as they are joined they have one variable in common. The second constraint prohibits to join a variable with itself.

Aggregation It's an abstract superclass for multiple classical aggregation operators. They accept multiple input variables, execute the aggregation on the variable specified through the *aggVar* property and group by all others inputs.

context Aggregation **inv:**
self.dataPorts->size() <= self.dataConections->size()

The constraint ensures that there are no more output variables than input variables.

LogicalAND, LogicalOR, LogicalNOT Simple logical operators to be used in the event-flow.

context LogicalAND **inv:**
self.dataPorts->size() = 1
context LogicalOR **inv:**
self.dataPorts->size() = 1
context LogicalNOT **inv:**
self.dataPorts->size() = 1
self.dataConnections->size() = 1

These constraints ensure that all three elements only have one output variable and in addition that *LogicalNOT* has only one input variable.

ChangesTo An operator that stores the last received value, compares it with the current value and fires when the input changes to the value specified in the property *changesTo*.

4.3.2 Runtime

To implement the EL runtime an existing semantic streaming engine should be reused to speed up the implementation so there will be more time to focus on ease of use for non-programming experts. Therefore a decision between the four engines compared in Table 2.2 had to been made. After an examination of the engines it became clear that EP-SPARQL and INSTANS are not suited as they both do not implement the concept of a stream. Also there were problems with using the code available due to compilation errors and unclear or missing documentation. This holds especially for INSTANS. Furthermore EP-SPARQL does not support tumbling windows and sliding windows with a step size. Therefore the use of EP-SPARQL and INSTANS has been discarded leaving CQELS and C-SPARQL as possible engines.

The drawbacks of C-SPARQL are that it only supports one window per stream and has no support for dynamic stream selection. Furthermore it has been shown to be remarkable slower than others systems especially than CQELS[65]. In the tests carried out in [65] C-SPARQL yielded execution times about 1000 times lower than CQELS. That C-SPARQL will be slower than CQELS is not surprising because C-SPARQL always yields the complete result (*Rstream*) whereas CQELS only yields the newly added results (*Istream*) but a factor of 1000 seems pretty high for only this.

CQELS does already allow to implicitly address streams by using a variable as address which can then be further specified by additional bindings. Further it is very fast. The downside of CQELS is that it only yields an *Istream* which in some

use cases will not be enough to model the desired functionality but it is assumed that CQELS can be extended yield also *Rstreams*. Because of this detailed analysis CQELS will be used to implement the runtime of EL.

4.4 Rule Language

In this section the design of RL is introduced. It is like the EL language designed as visual language. It realizes the concept *Rule* defined in Section 4.1. Therefore its purpose is to combine events in sequences and trigger actions. Figure 4.8 shows a mock-up of a simple rule modeled with RL. It is no coincidence that its design resembles a finite-state machine. It was modeled like this as rules naturally uses events as triggers and events can be seen as a notification that the (world's) state has changed. Thus it seems naturally that an event can be seen as a transition between to states (of the world). Also this approach allows to model very complex behavior by not making simple rules more complex. Furthermore it allows to break down a complex rule in small simple steps, which may or may not be time-dependent, what seems to suit the human approach in modeling complex behavior. In the following the general functionality of RL is illustrated.

As mentioned, it is designed like a finite-state machine and therefore it has a starting state. States in RL does not have any deeper meaning, they only represent intermediate state of rule execution flow and allow execution of actions when entered. Transitions represent events that can occur. This can be events defined in EL, time-based events or compound events, which means multiple transitions connected with a logical and or logical or. When entering a state the outgoing connections are deployed. If a state has multiple outgoing connections they are all deployed but only the first one to fire is followed. Before exiting a state all deployed events are undeployed. If a state does not have any outgoing transitions the rule will end after that state is reached. If the rule should always be running it should be modeled as a loop. As mentioned above, states can have multiple actions connection which are all executed when the state is entered.

A powerful feature provided is parametrization. It allows to pass values along transitions and also on to actions. A part of it can be seen in Figure 4.8. The event on the transition has two output values, *!room* and *!avg_temp*. It is to notice, that variables are prefixed with a *?* and values with an *!*. Following the transition to the action call it can be observed, that the two output variables from the event are used in the parameter definition of the action depicted in green.

Another very powerful feature is the *foreach* pattern. By placing it before the initial state it allows to specify that this rule shall be deployed multiple times with a different variable binding each time. In the shown example the rule will be deployed once for every room defined in the meta data. The current room is bound to *!room* and accessible inside the rule. In the shown example the event used for the transition has a parameter *?room* defined which is bound to the current value of *!room* before the event is deployed thus taking full advantage of parametrizable events.

4.4.1 Metamodel

In this section the metamodel of RL is introduced. It was basically derived from the example shown in Figure 4.8 and then enriched with features not present in the example.

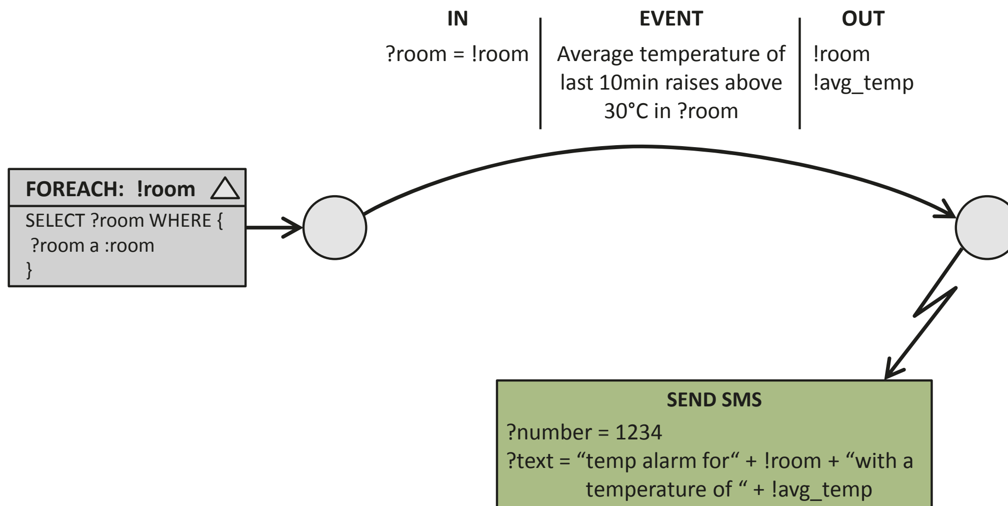


Figure 4.8: A mock-up of an example rule modeled with RL.

Figure 4.9 shows the RL metamodel as UML class diagram. The root class is *Rule* which contains the *State*s and *Transition*s of the rule as well as a reference to the initial state and a *Binding*.A *Transition* always has exactly two *States*, namely *source* and *target* whereas a *State* can have multiple outgoing *Transition*s.

The concept of variable binding is realized by introducing the *Binding* class and modeling relations to this class from all elements taking part in the rule-flow. This allows forwarding the variable binding beginning from the optional *Foreach* to the initial state and through transitions, which can modify the binding with the binding of their condition, up to the actions to be executed.

The metamodel nicely shows the different conditions that can cause a transition to trigger modeled as subclasses of *Condition*. Besides the *EventCondition* used in the example there are two time-based conditions and two compound conditions. The time-based conditions can be absolute in time defined by a time pattern for which it is proposed to use a Cron expression[2]. This is a close de facto standard for defining time patterns and intervals. Thus allows the user to express conditions like 'every Monday at 07:00 AM' or they can be of relative time allowing expression like 'in 5 minutes'. The *CompoundCondition* subclasses *AND* and *OR* allow to combine multiple conditions with a logical and or logical or. Combining some conditions with a logical and only triggers when all conditions are fulfilled and the resulting bindings are merged. Conflicts in merging the bindings are neither resolved nor detected. It is supposed that the result of the execution of any condition is merged into the compound binding right when the execution has finished which results in that results of previously finished sub-conditions returning a value with the same name is overwritten. If multiple conditions are combined with a logical or all sub-conditions are undeployed as soon as the first sub-condition yields a result.

[2]http://en.wikipedia.org/wiki/Cron#CRON_expression

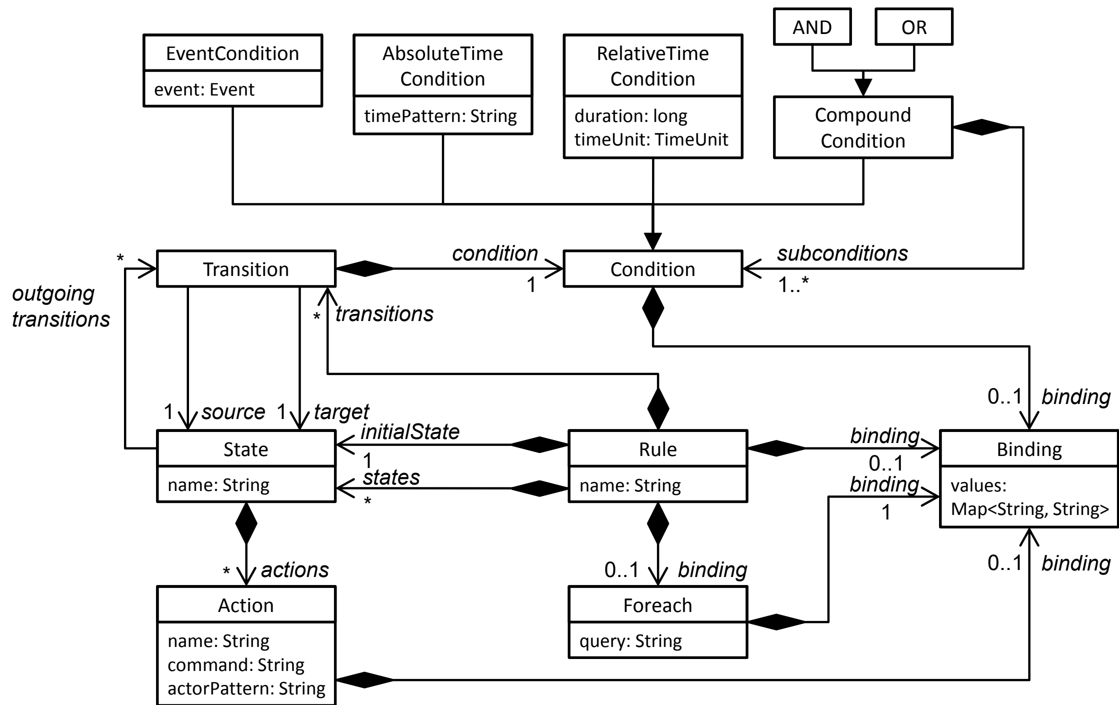


Figure 4.9: UML class diagram showing the RL metamodel.

4.5 Summary

In this chapter ERAS and the underlying concepts were illustrated. It was shown that ERAS tries to improve ease of use by introducing separate languages for rules and events and thus decoupling them to give suitable solutions for each domain. For both languages a formal metamodel was presented as well as a concept for the runtime. It was decided to base the implementation for the EL runtime upon CQELS because it seems best suited for this task.

5. Language and Framework Implementation

In this section the implementation done in this thesis is described. At first, in Section 5.1 the used programming environment and runtime libraries are explained. Section 5.2 gives an overview of the ERAS system implemented and the structure of all of its components. In Section 5.3 and Section 5.4 the implementation of EL respectively RL is described and details on implementation of the languages, editors and runtimes as well as the compilers are presented. The chapter closes with a summary in Section 5.5.

5.1 Development Environment, Tools and Libraries

The system was implemented using Java as it's the most common language for Semantic Web applications. For implementing the visual languages Eclipse Modeling Framework (EMF)[1] was chosen and the corresponding editors were implemented using the Graphical Modeling Framework (GMF)[2]. For easier and a more automated development EuGENia[3] was used, a tool that simplifies the integration of EMF and GMF. The Java Compiler Compiler (JavaCC)[4] was used to generate the compiler for ECQELS (see Section 5.3.3). The following list shows all tools and the version used for development.

- Java 8
- NetBeans IDE 8.0.1
- Eclipse Luna
- Eclipse Modeling Framework (EMF) 4.4.0

[1]http://eclipse.org/modeling/emf/
[2]https://wiki.eclipse.org/Graphical_Modeling_Framework
[3]http://eclipse.org/epsilon/doc/eugenia/
[4]https://javacc.java.net/

- EuGENia 1.2.0
- Graphical Modeling Framework (GMF) Tooling SDK 3.2.0
- JavaCC 6.0.1
- Maven 3.2.1 and SVN

Furthermore some third-party libraries were used. These were Esper[5] an event processing framework, parts of the Apache Jena framework[6] which is a open source framework for building Semantic Web applications, a JSON API, the JavaEE Web API for enabling SSE, JAX-RS[7] for enabling RESTful communication, a library providing a simple HTTP server component and the QUARTZ Job scheduler[8] for time-based scheduling of tasks. The following list shows the third-party libraries including their version used. It is to notice that this list is not complete as libraries not of interest for the reader such as logging libraries are omitted.

- Esper 4.2.0
- Jena ARQ 2.9.3
- Jena TDB 0.9.3
- Jackson JSON processor 2.3.0
- JavaEE Web API 7.0
- JAX-RS 2.7
- Jersey Grizzly2 HTTP Server 2.7
- QUARTZ Job Scheduler 2.2.1

5.2 Event and Rule Automation System

Figure 5.1 gives an overview of the structure of ERAS showing an UML package diagram of the system. It is to notice that each depicted package represents a whole project which is represented by the topmost package. It shows a lot of commonalities to Figure 4.3 which is clear as it is the concrete representation in code of the abstract system design depicted in Figure 4.3. The layout of Figure 5.1 can be considered as split in two almost symmetrical parts by an imaginary horizontal line in the middle. The upper part describes the Event Language (EL) and the lower one the Rule Language (RL). Only two packages in the middle of the left side(*edu.teco.eras.server* and *edu.teco.eras.client*) do not fit into this scheme as they are the main packages of the system representing the ERAS server and the corresponding client API respectively.

As the structure of the two languages is, except for one relation, completely identical the structure will only be explained once. The four most right packages of the form

[5] http://esper.codehaus.org/
[6] https://jena.apache.org/
[7] https://jax-rs-spec.java.net/
[8] http://quartz-scheduler.org/

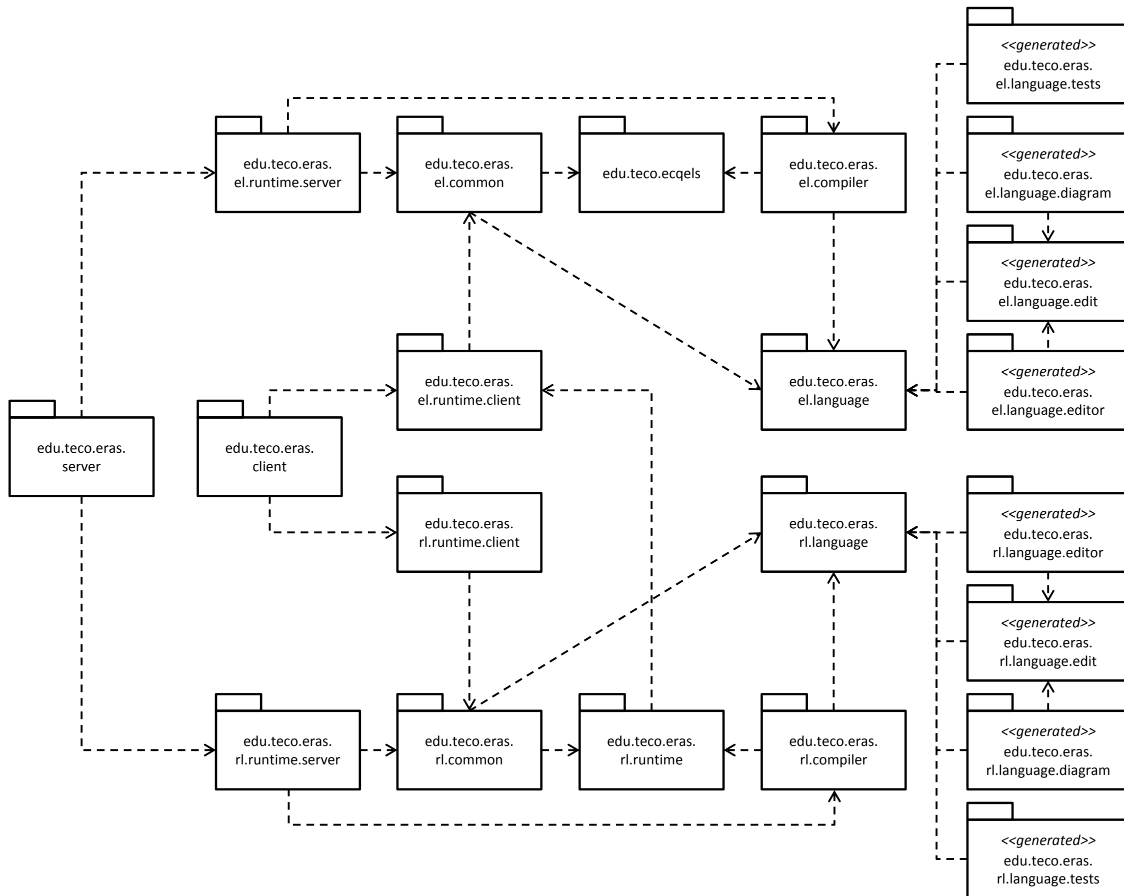


Figure 5.1: UML Package diagram of ERAS.

*edu.teco.eras.X.language.** (where X denotes *el* or *rl* and * is a wildcard) form the visual editor and are auto generated using EMF and GMF. However some parts of them are manually edited to realize custom behavior of the editors. They all import the package defining the language which is *edu.teco.eras.X.language*. Furthermore for each language there is a runtime, realized by the packages *edu.teco.ecqels* and *edu.teco.eras.rl.runtime*, as well as a compiler package named *edu.teco.eras.X.compiler* translating the language into something the runtime can execute. On top there is a server package (*edu.teco.eras.X.runtime.server*) implementing a server and exposing a REST interface as well as a client API to access the server (*edu.teco.eras.X.runtime. client*). Furthermore common functionality and data types shared between the server and client are extracted into a package named *edu.teco.eras.X.common*.

5.3 Event Language

In this section the implementation of EL is shown. As stated above the language was implemented using EMF and the editor using GMF. The language is defined by a file written in the language Emfatic[9] which is language for defining metamodels. GMF allows to define the visual appearance of models created upon this metamodel by adding annotations to the elements inside the Emfatic file. Thereby the language and the corresponding editor can be defined within a single file. Therefore the next

[9]https://wiki.eclipse.org/Emfatic

section does cover the language and the editor rolled into one. This section also shows the implementation of the corresponding runtime and the compiler which closes the gap between the language and the runtime.

5.3.1 Language and Editor

Listing 5.1 shows an excerpt of the Emfatic file defining the EL and shows the abstract superclasses of the metamodel. The metamodel implemented does differ in some parts from the metamodel presented shown in Figure 4.7. This is because the meta model in the design phase has been developed target-platform-agnostic and therefore does not take platform-specific implementation details into account. Therefore the designed metamodel has been slightly adapted to better suit the needs of the platform. Before giving detailed insights on how to metamodel has been adapted some information on how to read Emfatic code and how this code is further processed is given.

Emfatic code defines a metamodel. Therefore at a later step in development Java code is generated out of it. For each class in the metamodel a Java interface with the same name is generated as well as an implementation class with the suffix *Impl*. It is to notice that Emfatic allows multiple inheritance. This is sometimes confusing Java as does not but at the same time Emfatic code can be used to generate Java classes. So multiple inheritance is resolved by only subclassing one of the superclasses and implementing the interface of the other one.

The first three lines of the listing define the external implemented classes as data types by the use of the keyword *datatype* so that they can be used within the Emfatic file. Emfatic allows to express two UML relationships. These are Association using the keyword *ref* and Composition using the keyword *val*. Furthermore multiplicity can be defined using squared brackets. Relations can also be defined as two-way relations called opposite reference in Emfatic and is expressed by using # after the relation definition followed by the name of the opposite relation. The lines staring with *@gmf* are annotations for GMF and specify the visualization in the editor. *@gmf.affixed* can only be defined on a composition and causes the elements to be aligned on the border of the parent element instead of inside of it. Furthermore Emfatic also allows to define method headers with the keyword *op* which have to be implemented in the generated implementation classes.

The adaption of the metamodel designed at runtime manifests for example in the existence of the class *DataEventElement* shown in Listing 5.1 not present in the metamodel at design time. The need for this class arises from the fact that the classes realizing the language model should be designed to also suit the needs of the parser so that models defined in the language can easily be traversed. As typical for parsers the visitor pattern is used that requires all elements to implement a method like *visit(Visitor visitor)*{...}. This is realized as all abstract superclasses does define such a method. As EL has a data-flow and an event-flow the two separate classes *DataElement* and *EventElement* are defined each implementing their own *visit(...)* method thus providing type-safety. As there are some elements that can be part of a data-flow and event-flow the class *DataEventElement* is introduced which uses multiple inheritance to later act both as a *DataElement* and an *EventElement*.

Listing 5.2 shows the definition of the *StreamSource* class as an example for a visual element as well as the definition of *DataConnection* as example for a connection.

Listing 5.1: Excerpt from the EL Emfatic showing the abstract superclasses.

```
datatype EElementVisitor:  edu.teco.eras.el.language.extension.visitor.ElementVisitor;
datatype EDataElementVisitor: edu.teco.eras.el.language.extension.visitor.DataElementVisitor;
datatype EEventElementVisitor:
    edu.teco.eras.el.language.extension.visitor.EventElementVisitor;

abstract class Element{
        op void visit (EElementVisitor visitor);
}

abstract class DataElement extends Element {
        @gmf.affixed
        val DataPort[*]#parent dataPorts;
        ref DataConnection[*]#sink dataConnections;
        op void visit (EDataElementVisitor visitor);
}

abstract class EventElement extends Element {
        @gmf.affixed
        val EventPort[*]#parent eventPorts;
        ref EventConnection#sink eventConnection;
        op void visit (EEventElementVisitor visitor);
}

abstract class DataEventElement extends DataElement, EventElement {

}
```

GMF distinguishes partitions all elements that can later be used in the editor into two classes: nodes defined with the annotation *@gmf.node* and links defined with *@gmf.link*. These annotations can contain further parameters on how the elements should be visualized like for example the attribute used as label or the color. Links furthermore have to define which relation models the source and the target of the link.

It is to notice that the attributes of the *StreamSource* class are all defined as strings even though for most of them dedicated types do exist in the project. This is due to the fact that the generated editor does only support editing basic data types (as can be seen in Figure 5.2 at the bottom) without additional custom implementation which is out of scope of this thesis.

Listing 5.2: Excerpt from the EL Emfatic showing the definition of an example visual element and a connection.

```
@gmf.node(label="name", color="200,200,200")
class StreamSource extends DataEventElement {
        attr String name = "DATA SOURCE";
        attr String StreamPattern = "";
        attr String StreamSelectionPattern = "";
        attr String Window = "";
        attr String Refresh = "";
}

@gmf.link(source="source", target="sink", style="solid", width="5", target.decoration="arrow",
    color="170,210,240")
```

```
class DataConnection extends Connection {
        ref DataElement#dataConnections sink;
        ref DataPort#connections source;
}
```

Figure 5.2 shows the implemented editor modeling the use case already presented in Figure 4.6.

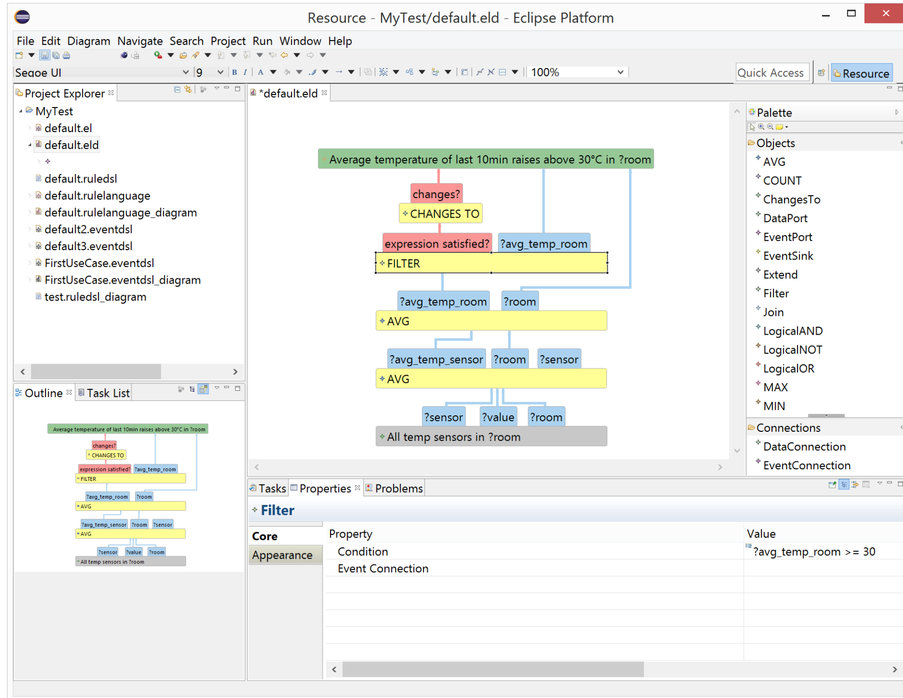


Figure 5.2: A screenshot of the EL editor showing the example use case already shown in Figure 4.6.

5.3.2 Runtime and ECQELS

This section describes the implementation of the runtime of EL. As shown in Section 4.3.2 the runtime will be implemented using CQELS. Listing 5.3 shows the most important methods of the runtime defined in interface-like style. Looking at these methods it is obvious that most of them are for life-cycle management. Unfortunately CQELS does not provide all of these methods so they had to be implemented as part of this thesis.

Listing 5.3: Most important methods of the EL runtime interface.

```
public ContinuousSelect registerSelect (Query query, Map<String, String> initialBinding);
public ContinuousConstruct registerConstruct(Query query, Map<String, String> initialBinding
    );
public void registerStream(RDFStream stream, String metadataGraph, String metadata);
public void unregisterSelect (ContinuousSelect query);
public void unregisterConstruct(ContinuousConstruct query);
public void unregisterStream(RDFStream stream, String metadataGraph, String metadata);
public void send(Node graph, Triple triple);
public void start ();
public void stop();
public void shutdown();
```

Furthermore in the course of the implementation it has been noticed that CQELS does only support a basic and not further defined subset of SPARQL. More precisely the CQELS language (as defined later in Listing 5.4) and Parser does support full SPARQL 1.1 but when executed some undefined behavior occurs which can be the occurrence of an exception without a detailed message or normal program execution without any result showing up. As the concept of compiling EL to CQELS is essential based on using subselects this functionality also needed to be implemented.

Though CQELS does partially support the dynamic stream select out of the box by allowing to bind the address of a stream to a variable like *STREAM ?uri [RANGE 10s]* it is not enough for this thesis as the variable binding is just evaluated once the query is registered. Therefore the concept of refreshable elements was introduced as seen in the next section.

Halfway through the implementation another problem concerning the window semantic arose which in combination with the need for subselects and the overall structure and code quality of CQELS did not allow to further extend the existing CQELS code to fulfill all requirements. In consequence a custom implementation of the CQELS engine was developed from scratch using the original CQELS code as a template. The new implementation of CQELS developed within this work is called ExtendedCQELS or short ECQELS and is presented in the following.

ECQELS

ECQELS defines its own language based on the CQELS language shown in Listing 5.4 and is therefore complete compatible to CQELS (meaning that CQELS queries can be executed with ECQELS, not the other way round), yet yielding different results due to the different window semantic used. Its implementation is largely based on combining Esper and Jena ARQ which is the part of the Apache Jena framework responsible for SPARQL parsing and execution.

Listing 5.4 shows the definition of the CQELS language expressed as a grammar in Extended Backus-Naur Form (EBNF). It is based upon the SPARQL 1.1 grammar as defined by the W3C[16].

Listing 5.4: CQELS language grammar under EBNF notation.

```
GraphPatternNotTriples ::= GroupOrUnionGraphPattern| OptionalGraphPattern|
    MinusGraphPattern|GraphGraphPattern|StreamGraphPattern|ServiceGraphPattern|
    Filter|Bind
StreamGraphPattern ::= 'STREAM' VarOrIRIref '[' Window ']' '{'TriplesTemplate'}'
Window ::= Rangle|Triple|'NOW'|'ALL'
Range ::= 'RANGE' Duration ('SLIDE' Duration |'TUMBLING')?
Triple ::= 'TRIPLES' INTEGER
Duration ::= (INTEGER 'd'|'h'|'m'|'s'|'ms'|'ns')+
```

It links into the SPARQL grammar by adding the *StreamGraphPattern* into the *GraphPatternNotTripples* pattern. Listing 5.5 now shows the extensions introduced by ECQELS highlighted in red. Basically there are only two extension. The first one is the *Refresh* pattern allowing to specify that some parts should be refreshed in a given interval. This pattern is optional and is defined to be used with streams defined by the *StreamGraphPattern* pattern, SPARQL SERVICE blocks defined by the *ServiceGraphPattern* pattern and ordinary SPARQL GRAPH blocks defined by *GraphGraphPattern* pattern. The second extension is the extension of the*Filter* pattern which introduces the *ChangesTo* operator.

Listing 5.5: ECQELS language grammar under EBNF notation. Extensions to CQELS grammar are highlighted.

```
StreamGraphPattern ::= 'STREAM' '[' Window ']' VarOrIRIref Refresh '{' TriplesTemplate
    '}'
Refresh ::= ('[REFRESH' Duration ']')?
GraphGraphPattern ::= 'GRAPH' VarOrIri Refresh GroupGraphPattern
ServiceGraphPattern ::= 'SERVICE' 'SILENT'? VarOrIri Refresh GroupGraphPattern
Filter ::= 'FILTER' Constraint ('CHANGES TO' BooleanLiteral)?
```

In the following a coarse overview of the architecture of Jena ARQ is given and it is explained how ECQELS is build on top of it.

Figure 5.3 shows the classes involved in processing a SPARQL query and the data types passed between them inside Jena ARQ. At first a SPARQL query in form of a string enters the system (depicted on the left) and is passed to the *Tokenizer*. The *Tokenizer* cuts the long query string into smaller ones referred to as tokens. This process is known as lexicographic analysis. These tokens are then passed to the *Parser* which performs a syntactic analysis and builds the abstract syntax tree often only referred to as AST. In Jena ARQ the AST consists of instances of subclasses of the abstract class *Element*. For the AST to become executable it is passed to the *AlgebraGenerator* which compiles the AST into SPARQL algebra. Elements of the algebra are instances of classes implementing the *Op* interface in Jena ARQ. To execute the query the resulting root *Op* instance is passed to the *OpExecutor*. As Jena ARQ extensively uses iterators for lazy execution the *OpExecutor* builds a tree of instances of subclasses of *QueryIterator* which is then return as result. It is to notice that all the classes involved in this processes are designed to be subclassed to implement custom behavior. More precisely, there is already a subclass of *Op*

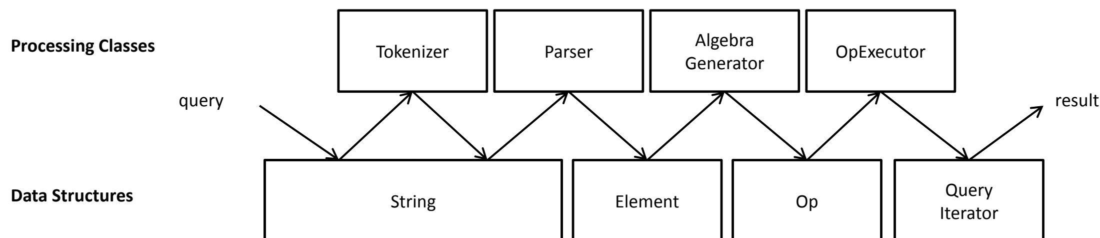


Figure 5.3: Visualization of data structures and processing classes involved in parsing and executing a SPARQL query in Jena ARQ.

defined to link in custom operators. Figure 5.4 shows which classes were added to realize ECQELS and also the corresponding superclasses from Jena ARQ they inherit from (depicted is gray). So five subclasses of *Element* were added extending the classes used for processing SPARQL *ElementNamedGraph*, *ElementService* and *ElementFilter*. Furthermore the just mentioned subclass of *Op*, *OpExt*, is shown which is part of Jena ARQ and designed as entry point to implement custom operators. For each added subclass of *Element* a corresponding subclass of *OpExt* was created. Also the *AlgebraGenerator* and *OpExecutor* classes seen in Figure 5.3 were subclassed. The custom implementations of the *Tokenizer* and *Parser* class are not shown in this figure as they are not derived from any class from Jena ARQ. The class *ECQELSOpVisitorBase* was implemented as the visitor pattern demands so when new subclasses are added and the *QueryIterConditionalFilter* class is needed for run-time execution of the *ChangesTo* operator. In the following the core structure of ECQELS is illustrated and a detailed explanation is given on how ECQELS

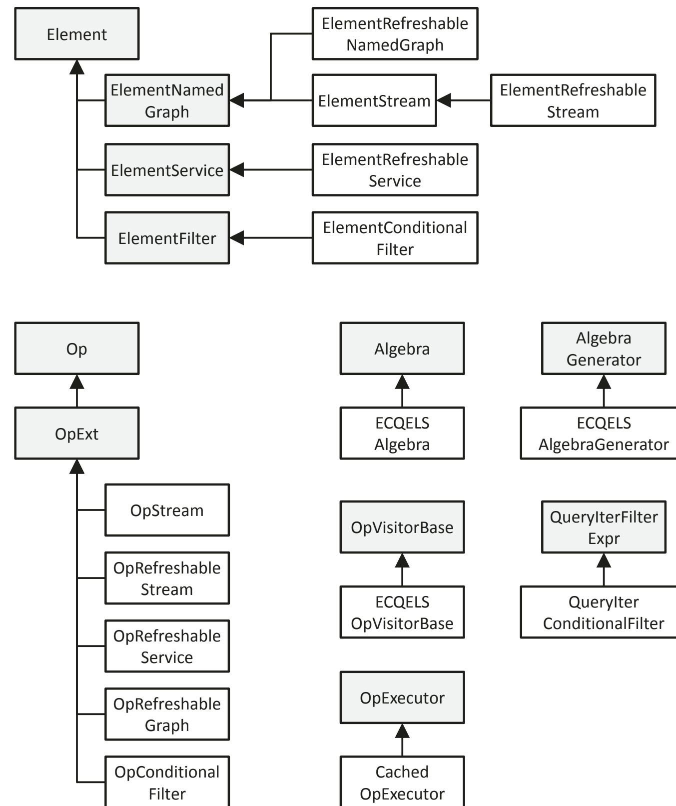


Figure 5.4: An UML class diagram showing the extension points used to integrate into Jena ARQ. Classes belonging to the Jena ARQ framework are highlighted in gray.

works. Figure 5.5 shows an UML class diagram of the most important classes of ECQELS. It is to notice that the diagram is not absolutely complete and correct as simple get and set functions are omitted, uses relationships are not explicitly shown (expect for one) and external classes are only depicted by name and some of them are event shown two times in the diagram to enable a nicer layout.

The main classes when using ECQELS from another project are *ECQELSRuntime* which exposes a nicely to use interface for life-cycle management of queries and stream. It is basically a wrapper for *Engine* adding handling of *RDFStreams*. *RDFStream* is an interface for implementing custom sort of streams. The system already comes with an SSE-based implementation of *RDFStream*. The class *Engine* implements the whole query life-cycle management. As ECQELS internally works with encoded triples where each element is encoded as a *long* most of the time *Engine* provides method to encode and decode. Also methods are provided to add to data and delete data from the internal RDF store used during execution. *Engine* does use *EPServiceProvider* which is an external class from the Esper framework. For every query registered *Engine* creates an instance of *QueryExecutor*.

The *QueryExecutor* class is responsible for executing a query. Each time the query generates a new result the engine is notified via the *NewQueryResultAvailableListener* interface. To execute the query it is divided into static parts which are only evaluated once at registering the query, refreshable parts which are re-evaluated with a fixed interval and streams which can be re-evaluated at a given interval or event-based depending on the type of window used. Therefore the *QueryExecutor*

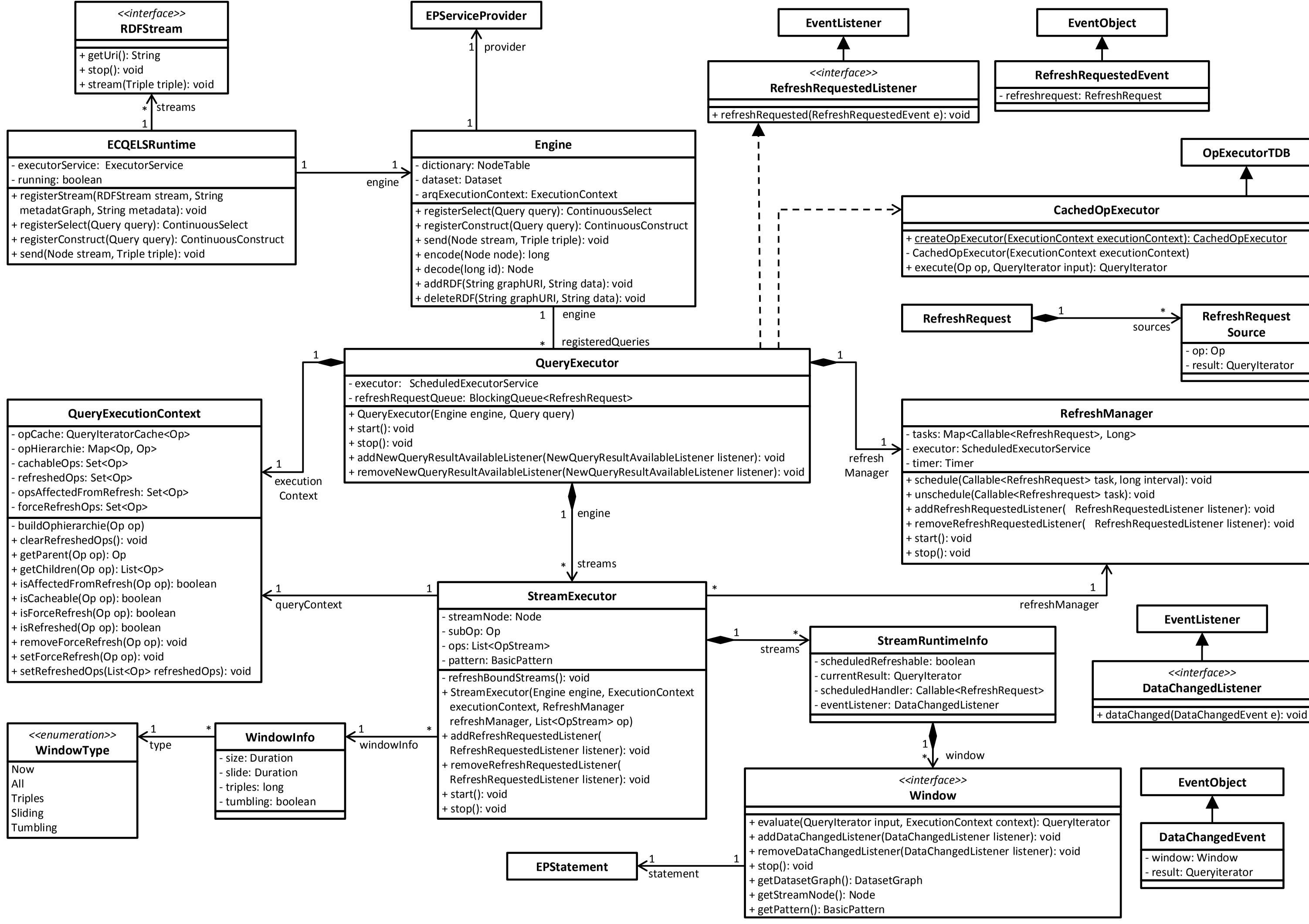


Figure 5.5: An UML diagram showing the main classes of ECQELS.

maintains an instance of *QueryExecutionContext* which on creation builds a tree-based hierarchy of the query. Every time any part, in the later referred to as Op as this is the interface all executable parts implement, is re-evaluated the *QueryExecutionContext* is modified by calling the *setRefreshedOps(...)* method. As an Op gets re-evaluated all Ops on the path from that Op up to the root needs to be re-evaluated. *QueryExecutionContext* does allow to query if an Op is affected and needs to be re-evaluated. It also provides methods to check whether the so generated new result of an Op needs to be cached or not. This is extensively used by *CachedOpExecutor* which inherits from the Jena ARQ class *OpExecutorTDB* and maintains a cache for all cacheable intermediate results. It also decides which Ops need the be re-evaluated using the provided methods from *QueryExecutionContext*.

All Ops that can be re-evaluated over time and therefore can cause the whole query to be re-evaluated and produce a new result cause the firing of the *refreshRequested(RefreshRequestedEvent e)* method from the interface *RefreshRequestedListener*. The *RefreshRequestedEvent* contains an instance of *RefreshRequest* which itself consists of a list of instances of *RefreshRequestSource*. Each instance of *RefreshRequestedSource* contains the information which Op has been refreshed and what the new result is. This structure allows multiple refresh actions taking place at the same time to only trigger one re-evaluation of the whole query and therefore only produce a single new query result. This is desirable as for example a query could contain multiple streams all with a time-based window of the same size. Without a centralized scheduling component each window would trigger one re-evaluation of the query and therefore multiple new query results would be issued each time the windows are triggered. As this is not desirable and only one new query result should be generated in such cases ECQELS contains a component for central scheduling interval-based re-evaluation tasks realized as the class *RefreshManager*. It exposes the method *schedule(Callable<RefreshRequest> task, long interval)* which allows to register an action returning a *RefreshRequest* and an interval in which that task should be executed. All registered tasks are then aligned so that at every time step all registered tasks that need to be executed are executed and than all results are merged together into one *RefreshRequest* which is than passed back to the *QueryExecutor* to trigger re-evaluation of the rest of the query with the new results. Tasks scheduled at the *RefreshManager* can be refreshable graphs or services as defined in Listing 5.5 or time-based windows on streams.

The execution of streams is handled by the *StreamExecutor* class. It is to notice that *QueryExecutor* does not create one instance of *StreamExecutor* per stream but rather one per "different" stream. "Different" in that case means that one of the following properties of two streams are different: URI, window, pattern and if they use a variable as URI the subselect must be identical to not be treated as different. This means that for all streams with the exact same definition used multiple times across a query only one instance of *StreamExecutor* is created. When a *StreamExecutor* signals a new result for a stream used multiple times the result is duplicated to be valid for all occurrences of that stream.

In the following the behavior for implicitly addressed streams is depicted which is quite complex, especially when the subselect binding the variable used for addressing of the stream is refreshable. This means that the stream can actually bind to multiple changing RDF streams and thus needs to maintain which streams he is

actually bound to and also keep a single window for each of them. Therefore the *StreamExecutor* subscribes with the *RefreshManager* to get notified the subselect binding his stream variable is re-evaluated and then calls his private method *refreshBoundStreams()* shown in the diagram. This method checks if new streams are available, if yes, sets them up by creating a new window based on the *WindowInfo* and creates an instance of *StreamRuntimeInfo* used for maintenance and life-cycle management of bound streams. The handling of incoming stream data is done by classes implementing the *Window* interface not shown to not clutter up the diagram.

Windows are implemented using an in-memory graph realized by the *DatasetGraph* class of Jena ARQ. They receive their data by subscribing to a single RDF stream. This is done using Esper and the *EPStatement* class. If they are time-based then they schedule themselves with the *RefreshManager* to be in sync with all other time-based tasks, otherwise they propagate their new results to the *StreamExecutor* using the *DataChangedListener* interface.

5.3.3 Compilation

In this section it is described how EL is compiled into ECQELS to be executed. This functionality is implemented in the project *edu.teco.eras.rl.compiler* as shown in Figure 5.1. The compilation works as follows.

An event modeled with EL has the form of a tree but with possible multiple edges between two vertexes. Also a vertex can have multiple parents. This structure if converted into a real tree by following rules. Multiple edges between two vertexes are combined into one edge containing the variable names from all edges. Vertexes with multiple parents are duplicated for each parent. Applying these rules we get a tree with an *Event* as root node and *StreamSource*s as leaves. Now a depth-first search is done using the visitor pattern. Thereby each vertex is converted into its ECQELS counterpart based on a static pattern. Listing 5.6 shows some of these patterns which will be discussed in details within a short time. Vertexes with multiple children do a union on them by wrapping them all together in the *WHERE* block of their query.

In the following four example patterns used to compile EL to ECQELS are shown. The are depicted as ECQELS queries with variables highlighted in italic. Two variables are query independent and are explained in the following. These are *projected vars* and *incoming elements*. The variable *projected vars* is not part of the element compiled itself but rather is context-sensitive and therefore replace by additional information from the parent more precisely from the edge between element and its parent. It therefore refers to all variables that are passed from that element to the parent. The second one is *incoming elements* which is also context-sensitive and refers to the ECQELS representation of all children.

Listing 5.3.0a shows the pattern used to compile a *StreamSource* element of EL. Actually it shows how *StreamSource*s with implicit stream addressing are compiled. The variables *window*, *stream pattern*, *refresh* and *uri pattern* are replaced as defined in the EL element. The value of *metadatagraph* could also be passed via the EL model but at the moment is passed as a parameter for the compiler.

Listing 5.3.0b shows the pattern used to compile an aggregation operator from EL. Therefore *operator* is replaced by the SPARQL aggregation command suited like *MIN*, *MAX*, *AVG*, *SUM* or *COUNT*. *aggVar* describes the variable to be aggregated

by and *newVar* the name the result should be projected as defined in the underlying EL model. *projected vars w/o aggVar* is replaced with all projected vars except for *aggVar*. Also it is inserted a *GROUP BY* at the and containing all projected vars that are not aggregated.

Listing 5.3.0c shows the pattern used to compile an *Extend* element of EL. An *Extend* element simple fetches additional data and therefore the variable *query* is replaced by the subselect as defined in the underlying EL model. Putting multiple elements side-by-side in a *WHERE* block in SPARQL does represent a union of the results.

Listing 5.3.0d shows the pattern used to compile a *Filter* element in EL. It maps to the SPARQL function *FILTER* which is passed the given *expression* to filter by. If to the *EventPort* of a filter a *ChangesTo* operator is connected then that logic is compiled into the filter using the *CHANGES TO* operator defined in ECQELS (see Listing 5.5). The squared brackets mean the contained block is optional (as it depends on the occurrence of a *ChangesTo* operator).

```
SELECT projected vars
WHERE
{
    STREAM ?uri window
    {
        stream pattern
    }
    GRAPH metadata graph refresh
    {
        uri pattern
    }
}
```

(a) StreamSource pattern

```
SELECT (operator(aggVar) AS newVar)
  projected vars w/o aggVar
WHERE
{
    incoming elements
}
GROUP BY projected vars w/o aggVar
```

(b) Aggregation pattern

```
SELECT projected vars
WHERE
{
    incoming elements
    query
}
```

(c) Extend pattern

```
SELECT projected vars
WHERE
{
    incoming elements
    FILTER ( expression ) [CHANGES TO
          changesTo]
}
```

(d) Filter pattern

Listing 5.6: ECQELS templates used for compilation of EL.

5.4 Rule Language

This section presents the implementation of the Rule Language RL. It is, like EL, implemented using EMF and GMF. Also the corresponding runtime and the compiler are presented.

5.4.1 Language and Editor

As mentioned RL was also implemented using EMF and GMF. Unfortunately GMF does not support a nice looking visual representation of attributes of a *link* as drafted in the design phase (see Figure 4.6). Because of this a *Transition* in RL has been realized using two GMF *link*s namely a *BeginTransition* from a *State* to a *Condition* and an *EndTransition* pointing in the opposite direction as shown in an excerpt of the Emfatic file defining RL in Listing 5.7. The listing shows the two *link* classes *BeginTransition* and *EndTransition* as well as the class *State* and an abstract definition of a *Condition*. The code is pretty straight forward as it maps to the RL metamodel shown in Figure 4.7 except the just mentioned split of the *Transition* class.

Listing 5.7: Excerpt from the RL Emfatic file.

```
abstract interface Condition {
        ref BeginTransition#target incomingTransition;
        ref EndTransition#source outgoingTransition;
        @gmf.compartment
        val Binding binding;
}

@gmf.link(source="source", target="target", style="solid", width="2", target.decoration="
    arrow")
class BeginTransition {
        ref State#outgoingTransitions source;
        ref Condition#incomingTransition target;
}

@gmf.link(source="source", target="target", style="solid", width="2", target.decoration="
    arrow")
class EndTransition {
        ref Condition#outgoingTransition source;
        ref State#incomingTransition target;
}

@gmf.node(label = "label")
class State {
        attr String label = "STATE";
        ref BeginTransition[*]#source outgoingTransitions;
        ref EndTransition#target incomingTransition;
        @gmf.compartment
        val Action[*] actions;
}
```

Figure 5.2 shows a screenshot from the RL editor implemented. It could have been made much nicer looking and easier to use but as a complete custom implementation of CQELS had to be implemented there was no time left in this thesis for improvement.

5.4.2 Runtime and Compilation

Because transitions had to be modeled as two classes to be compatible with GMF (as explained in the previous section) the runtime contains an own set of classes

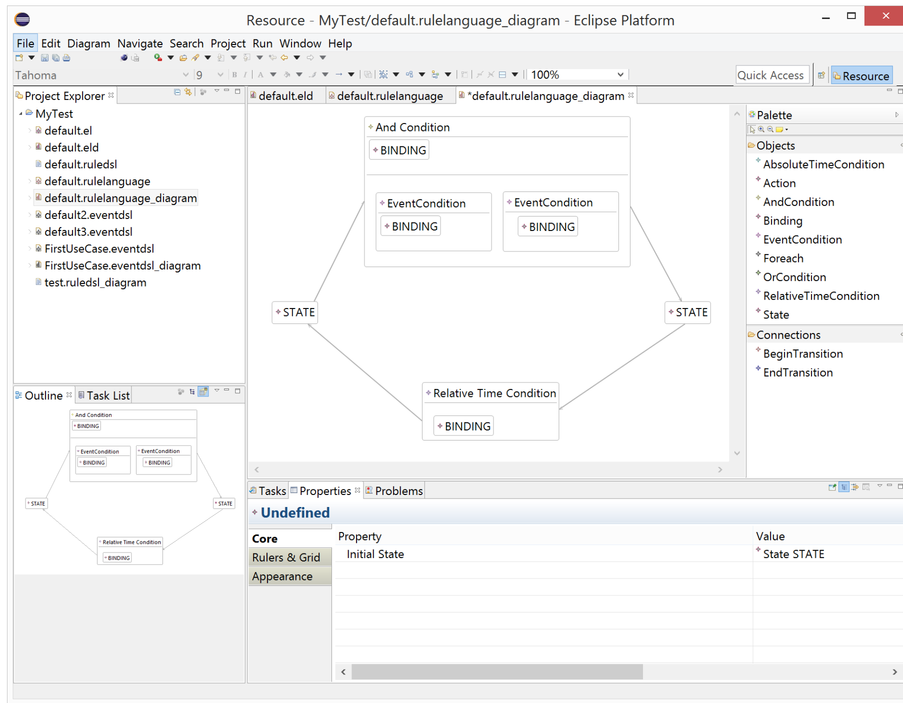


Figure 5.6: A screenshot of the RL editor showing an example Rule.

modeling RL. They represent the metamodel shown in Figure 4.9 but at the same time implement execution logic. According to the package overview in Figure 5.1 these classes are located in the project *edu.teco.eras.rl.runtime*. Figure 5.7 shows them as UML class diagram. Again simple getter and setter methods to attributes shown are omitted. The main difference is that *Rules* and *Actions* now expose an *execute(...)* method. Compilation simple maps the Emfatic-based model back on this model and the runtime executes *Rules* by invoking *execute()* respectively *executeAsync()* on them. A *Rule* is executed with the code snippet shown in Listing 5.8. Beginning with the initial state it is looped until the current state does not have any outgoing connections. In every loop the method *doTransition(...)* is called which executes all outgoing transitions in a separate thread by invoking *transition.getCondition().execute(binding)*. As soon as the first *Condition* returns all others threads are canceled an the method returns an instance of *TransitionResult* which contains the new *Binding* an also the information which transition has triggered. After that, the current binding and the one returned are merged and the transition is taken. After that all actions in the new current state are invoked and the loop is repeated (if the new current state has any outgoing transitions). It is to notice that instances of the *EventCondition* class use an instance of *ELRuntimeClient* to schedule their event on an EL server.

5.5 Summary

This chapter showed the implementation of ERAS and illustrated the implementation details for the languages and editors which were realized using EMF respectively GMF. Also the compilation of the languages into a form executable by the corresponding runtime was shown. For RL a runtime based on a finite state machine was implemented. For EL it was planned to use CQELS (with some small extensions) as runtime. Unfortunately CQELS did not prove suited for being used as runtime.

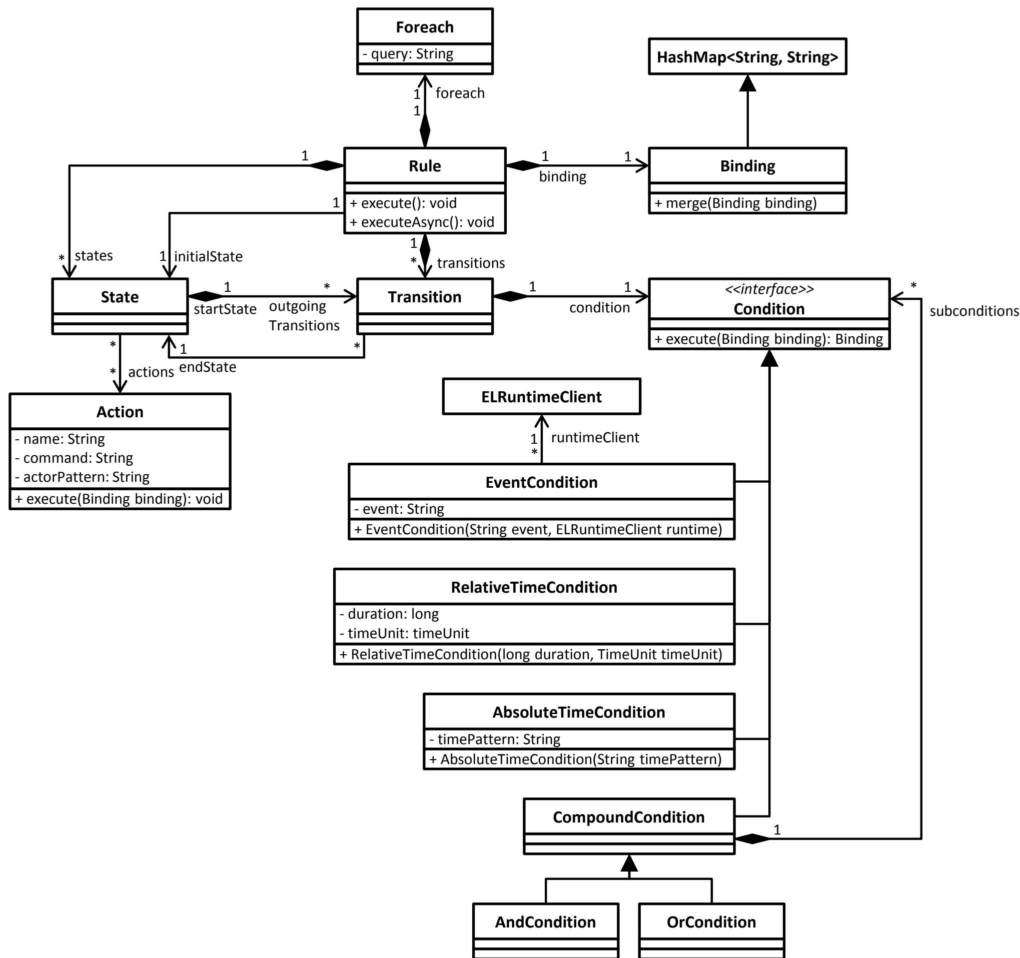


Figure 5.7: UML class diagram of the classes used in RL runtime.

Listing 5.8: The *execute()* method of the class *Rule*

```
public void execute() {
    Binding currentBinding = binding;
    State currentState = initialState;
    while(!currentState.getOutgoingTransitions().isEmpty()) {
        TransitionResult result = doTransition(currentState);
        currentBinding.merge(result.getBinding());
        currentState = result.getTransition().getEndState();
        currentState.getActions().forEach(action -> action.execute(currentBinding));
    }
}
```

The reasons therefore were elaborated in Section 5.3.2 in detail. Because of this an own complete semantic streaming engine, named ECQELS, had to be implemented as part of this thesis to be used as runtime for EL. The implementation of ECQELS was much additional effort not scheduled in this thesis (at least not to this extent). Because of this reason a big part of this chapter was dedicated to the implementation of ECQELS. Furthermore the implementation did costs a lot of time and because of this there was less time than planned to implement the editors for EL and RL in a nicely-looking and more user-friendly way which was one objective in this thesis.

Despite all challenges and problems the ERAS system has been implemented and is working.

6. Evaluation

6.1 Performance Analysis

In this section the performance and correctness of ECQELS will be analyzed and compared to CQELS. It was planned to use one of the two existing evaluation frameworks that CQELS has been tested on. These are LSBench (Linked Stream Benchmark) [17] used in [65] and the benchmark CQELS has initially been evaluated against in [64] which is also available online [18]. Unfortunately it was not possible to get one of these benchmarks to work even with two days of work as they are poorly documented and the provided jar files could not be executed without errors. Therefore an own benchmark system to evaluate query performance and correctness of ECQELS against CQELS was implemented as part of this thesis. In the following the developed benchmark is introduced and the results are presented and analyzed.

6.1.1 Benchmark Design and Runtime Environment

The benchmark was developed based on the one used in [64, 18] as the other one seemed to complex to adapt in the short time remaining in this work due to the unforeseen implementation of ECQELS. The benchmark is originally based on a Live Semantic Experiment presented in [22] and is basically composed of static background knowledge on the adjacency of rooms within a floor as well as on persons, their possible role as authors of scientific papers and their correlations and stream information that represent the movement of persons through that floor. The information on persons, papers and their correlation is presented as an excerpt of the DBLP dataset [19] which contains these information in RDF format and is public available. For the benchmark the five queries used in the original benchmark were used but with some small changes which will be discussed later. The queries used are shown in Listing 6.1. It is to notice that these queries were used for both CQELS and ECQELS evaluation with two exceptions: For query 3 and 5 the ECQELS version has the streams written as a subquery and furthermore the *FILTER* operation moved up into that subquery. This was done for performance tuning as ECQELS supports all SPARQL 1.1 features like subqueries which CQELS does not.

```
PREFIX lv: <http://deri.org/floorplan/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?person ?location ?locName
WHERE
{
   STREAM <http://deri.org/streams/rfid> [NOW] { ?person lv:detectedAt ?location. }
   GRAPH <http://deri.org/floorplan/> { ?location lv:name ?locName. }
}
```

(a) Query 1

```
PREFIX lv: <http://deri.org/floorplan/>

SELECT  ?person1 ?location1 ?person2 ?location2
WHERE
{
   GRAPH <http://deri.org/floorplan/> { ?location1 lv:connected ?location2. }
   STREAM <http://deri.org/streams/rfid> [NOW] { ?person1 lv:detectedAt ?location1. }
   STREAM <http://deri.org/streams/rfid> [TRIPLES 5] { ?person2 lv:detectedAt ?location2. }
}
```

(b) Query 2

```
PREFIX lv: <http://deri.org/floorplan/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?auth ?coAuth ?location ?paper ?locName
WHERE
{
   GRAPH <http://deri.org/floorplan/> {
      ?paper dc:creator ?auth.
      ?paper dc:creator ?coAuth. }
   STREAM <http://deri.org/streams/rfid> [NOW] { ?auth lv:detectedAt ?location. }
   STREAM <http://deri.org/streams/rfid> [TRIPLES 5] { ?coAuth lv:detectedAt ?location. }
   GRAPH <http://deri.org/floorplan/> { ?location lv:name ?locName. }
   FILTER(?auth!=?coAuth)
}
```

(c) Query 3

```
PREFIX lv: <http://deri.org/floorplan/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX swrc: <http://swrc.ontoware.org/ontology#>

SELECT ?auth ?editor ?loc1 ?loc2 ?paper ?proceeding ?editorName
WHERE
{
   GRAPH <http://deri.org/floorplan/> { ?loc1 lv:connected ?loc2. }
   STREAM <http://deri.org/streams/rfid> [NOW] { ?auth lv:detectedAt ?loc1. }
   STREAM <http://deri.org/streams/rfid> [TRIPLES 5] { ?editor lv:detectedAt ?loc2. }
   GRAPH <http://deri.org/floorplan/> {
      ?paper dc:creator ?auth.
      ?paper dcterms:partOf ?proceeding.
      ?proceeding swrc:editor ?editor.
      ?editor foaf:name ?editorName. }
    FILTER(?auth!=?editor)
}
```

(d) Query 4

```
PREFIX lv: <http://deri.org/floorplan/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?auth ?location2 ?locationName (COUNT(DISTINCT ?coAuth) AS ?noCoAuth)
WHERE
{
   GRAPH <http://deri.org/floorplan/> {
      ?paper dc:creator ?auth.
      ?paper dc:creator ?coAuth. }
   STREAM <http://deri.org/streams/rfid> [NOW] { ?auth lv:detectedAt ?location1. }
   STREAM <http://deri.org/streams/rfid> [TRIPLES 5] {   ?coAuth lv:detectedAt ?location2. }
   GRAPH <http://deri.org/floorplan/> {
      ?location2 lv:connected ?location1.
      ?location2 lv:name ?locationName. }
   FILTER(?auth!=?coAuth)
}
GROUP BY ?auth ?location2 ?locationName
```

(e) Query 5

Listing 6.1: The queries used for evaluation.

When designing a benchmark for semantic streaming engines there is always the question what features are relevant to compare the tested engines against. Unfortunately there is no feature that is commonly agreed to be relevant as there are no semantic stream benchmarks commonly agreed on at all. In [64] the average query execution time in ms is used and [65] uses the execution throughput defined as executions per second whereby a execution is defined as insertion of a triple on a stream. Both features do neglect the output of a query probably because it would require having a mapping between input triples and the output expected when they are inserted. Therefore in this work the average response time (in ms) is proposed as feature. It is defined as the delay between the insertion of a triple and the reception of the expected result. This also fits very well the requirements of home automation to be quickly notified when a new result becomes available. On the other hand this feature also has a downside. It only works well with queries with triple-based windows as queries with time-based windows tend to not yield deterministic results at the latest when the insert frequency increases and/or the time window is very small. Therefore all time-based windows in the original queries have been replaced by triple-based windows. In general the benchmark works with variable window sizes but this evaluation was done only using a window size of 5.

To compute the average response time it must be known which triple causes which results when inserted. Therefore a data generator has been implemented which generates input triples of the form *person :detectedAt room* just as in the original benchmark but on the fly does simulate the query and thereby calculates the expected outcome. As CQELS and ECQELS use different output types (Istream and Rstream) it was decided to only use the Istream result as expected result so that both engines can be compared. Another difficulty encountered was that when inserting a triple causes a result with multiple entries for every entry an separate event is triggered in CQELS which made it hard to determine which results were triggered by which input triple. It also could happen that an engine does not yield an expected result. Therefore the number of expected results, received results and results hit (means expected and received) were kept track of. Only the results hit were used to calculate the average response time.

The benchmark was executed using a stream interval of 10ms which means that the engines were fed one triple every 10ms if they processed the previous triple within this time, otherwise the next triple was inserted as soon as the engine finished processing the previous one. The queries were execute against variable sized streams, 1,000, 2,000, 10,000 and 50,000 triples, and a background knowledge of varying size of the DBLP excerpt. It was planned to use four sizes of background knowledge, 10,000 (10k), 100,000 (100k), 1,000,000 (1M) and 2,000,000 (2M) triples, but as ECQELS performed very poorly with very big background data the tests with 2M triples background knowledge were rejected.

The benchmark was executed on a desktop machine with an Intel i7-2600k, a quad-core processor running at 3.40GHz, and 8GB RAM using Windows 8.1 The JVM was started with the arguments *-Xms1024m -Xmx4g* which ensures a minimum heap size of 1GB and a maximum heap size of 4GB.

6.1.2 Results and Analysis

In this section the results of the benchmark are presented and discussed. Figure 6.1 shows the result for all five queries with a stream size of 1,000 triples and three

different sizes of background knowledge. It is to notice that all test have been run ten times and the values listed in the following are average of these ten runs.

Figure 6.1a shows that ECQELS slightly outperforms CQELS for query 1 which is the most simple one with only one time-based window of size 1 and no reference to the background knowledge. For query 2 shown in Figure 6.1b it is the other way round and CQELS slightly outperforms ECQELS. For both, query 1 and query 2, the results can be interpreted to be in $\mathcal{O}(c * n)$ with n representing the size of background knowledge and c a very small constant factor. If the change in average response time is interpreted as jitter which seems plausible as the average response time for ECQELS slightly decreases from 100k to 1M background knowledge for both queries it could even be interpreted to be in $\mathcal{O}(1)$ as expected when looking at the queries which do not make use of any background knowledge.

In query 3 and 4 CQELS can show its power with handling big amounts of background knowledge and therefore performs in $\mathcal{O}(1)$ for query 3 and $\mathcal{O}(n)$ for query 4. It is to notice that in Figure 6.1c, Figure 6.1d and Figure 6.1e a log-scale y-axis is used. ECQELS seems to perform in $\mathcal{O}(n^2)$. Looking at the code and how CQELS and ECQELS work this outcome is quite obvious as CQELS is designed to only support Istream output streams and therefore only the new incoming triple are processed. Furthermore CQELS has a very efficient caching system based on Berkeley DB [1]. ECQELS on the other hand supports Rstream output streams and reprocesses the whole current window each time it changes. Furthermore it uses the default query execution mechanism of Jena ARQ and only adds caching for parts of the execution plan. Therefore ECQELS has to recalculate the time-consuming joins between the cached static data and the window each time a new triple is inserted whereas CQELS only has to do a table lookup for the one new triple. Knowing this the question is still why does ECQELS not scale linearly but rather exponential? This is due to the fact that in SPARQL, contrary to relational query languages, self joins are very common due to the pattern matching style. From this it follows that how much effect the size of the dataset has strongly depends on the triple pattern on the dataset.

The results of query 5 shown in Figure 6.1e again show that ECQELS slightly outperforms CQELS. This is probably due to the aggregation used in query 5 which seems to break the caching logic auf CQELS.

Table 6.1 shows detailed results of the benchmark for varying stream size and a background knowledge of 10,000 triples. Besides the arithmetic mean of the average response time it also shows the corresponding standard deviation. The first thing to notice is that ECQELS gets slower as the stream size increases. This means that the longer the engine is running and receiving stream data the slower the execution of the query gets. This is a major issue especially when being used in the domain of home automation where very long running queries are not unusual. This indicates that there might be some sort of memory leak in ECQELS but unfortunately it could not be found and fixed within the limited time of this thesis. A possible cause could be that Jena ARQ extensively uses the iterator pattern to support lazy evaluation of queries which is in most of the cases desirable but not when being used inside a semantic streaming engine as all results have to be materialized during

[1] https://oss.oracle.com/berkeley-db.html

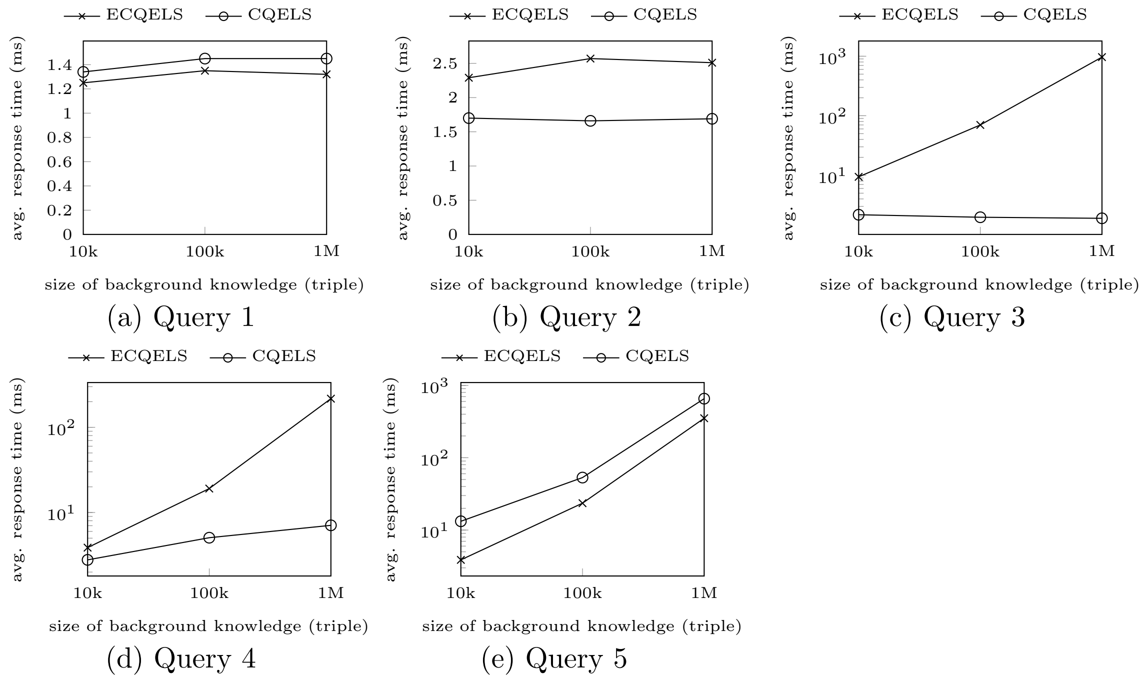


Figure 6.1: Result of the performance evaluation for different sizes of background knowledge with a stream size of 1000.

execution. ECQELS uses these cached version of these iterators to store intermediate results and as iterators can only be used to visit every element once (at least in the implementation of Jena ARQ) every time a cached value is used the iterator is copied into a new one. This could be one possible cause for this potential memory leak and should be fixed in a later version as it also would allow to implement a caching logic similar to CQELS allowing to massively boost performance in large background knowledge when done right.

Corresponding to the the findings above CQELS scales very well for the first three queries and also seems to improve average response time and its standard deviation over time which might also be reason to the caching logic. CQELS struggles again with query 5 and is outperformed by ECQELS event with increasing average response time for ECQELS.

As the benchmarks presented in [64, 65] evaluate the performance for multiple simultaneous queries this is also done in this benchmark. To get a comparable result this test is only performed with a stream size of 1000 triples and a background knowledge of 10,000 triples so that ECQELS is neither penalized for its potential memory leak nor for its structural design and therefore weaker caching logic. Figure 6.2 shows the results of the test. Surprisingly ECQELS outperforms CQELS clearly for all four queries as none of both do any between-query optimization or share any information between queries. One possible reason for this could be that ECQELS uses one thread per query to process new triples for all queries in parallel wheres CQELS supposedly does not use multithreading.

Concerning the correctness of the results produced by CQELS and ECQELS it was observed that ECQELS in all cases returned the exact same results as expected. CQELS however did often return additional results that were not expected. This was the case when a query contained multiple windows over the same stream which

	stream size (triples)	ECQELS avg. response time (ms)	ECQELS standard deviation	CQELS avg. response time (ms)	CQELS standard deviation
Query 1	1000	1.25	1.88	1.34	1.18
	2000	1.34	1.75	1.35	0.92
	10000	1.66	1.00	1.01	0.52
	50000	2.88	0.99	0.99	0.36
Query 2	1000	2.29	1.32	1.70	1.18
	2000	3.04	1.00	1.62	0.70
	10000	4.31	1.76	1.43	0.82
	50000	14.36	7.49	1.34	0.73
Query 3	1000	9.41	4.53	2.17	3.01
	2000	15.42	7.42	1.85	2.22
	10000	69.68	36.18	1.84	1.95
	50000	433.76	289.61	1.61	1.49
Query 4	1000	3.87	1.77	2.78	3.40
	2000	4.33	2.82	3.29	4.49
	10000	6.96	2.75	3.07	4.43
	50000	22.36	11.04	2.96	4.09
Query 5	1000	3.94	1.37	13.22	10.66
	2000	4.17	1.38	12.13	8.81
	10000	4.59	1.27	11.39	8.73
	50000	9.98	3.49	23.35	18.16

Table 6.1: Result of the performance evaluation with a background knowledge of 10k triples and varying stream size for all queries.

triggered one execution per window whereby for all executions except the last one some old triples actually no longer part of the window(s) were used as the window caches are only refreshed one by one. ECQELS avoids this problem by synchronizing refreshs and refreshing all triple-based windows over one stream at the same time. Nevertheless CQELS yielded besides the additional invalid results all expected results at least for the single query evaluation. When processing multiple queries simultaneously CQELS very rarely (meaning at most 0,1%) missed some expected results.

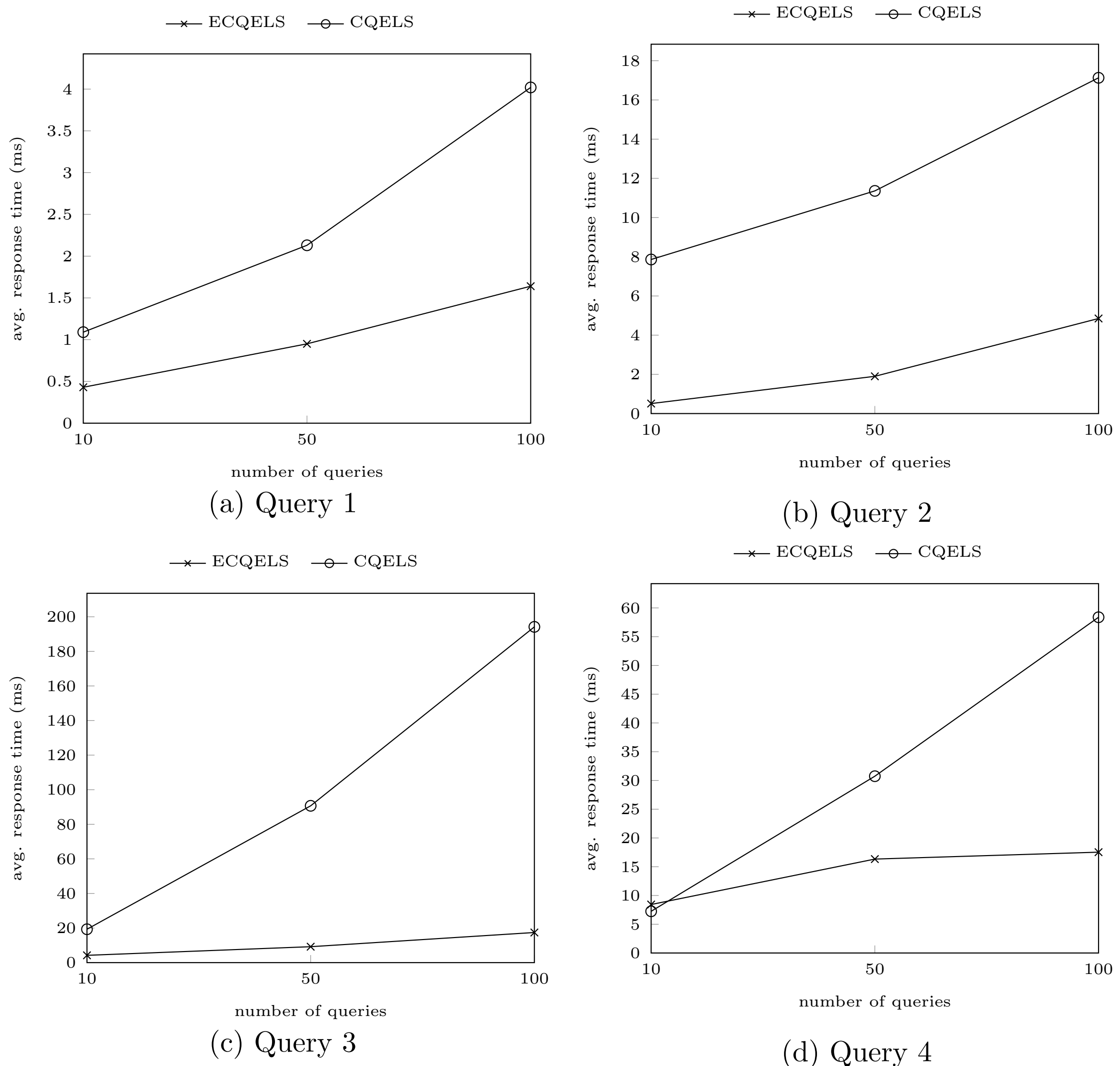


Figure 6.2: Result of the performance evaluation for multiple queries with a background knowledge of 10k triples and a stream size of 1000 triples.

6.2 Use Case Expressiveness

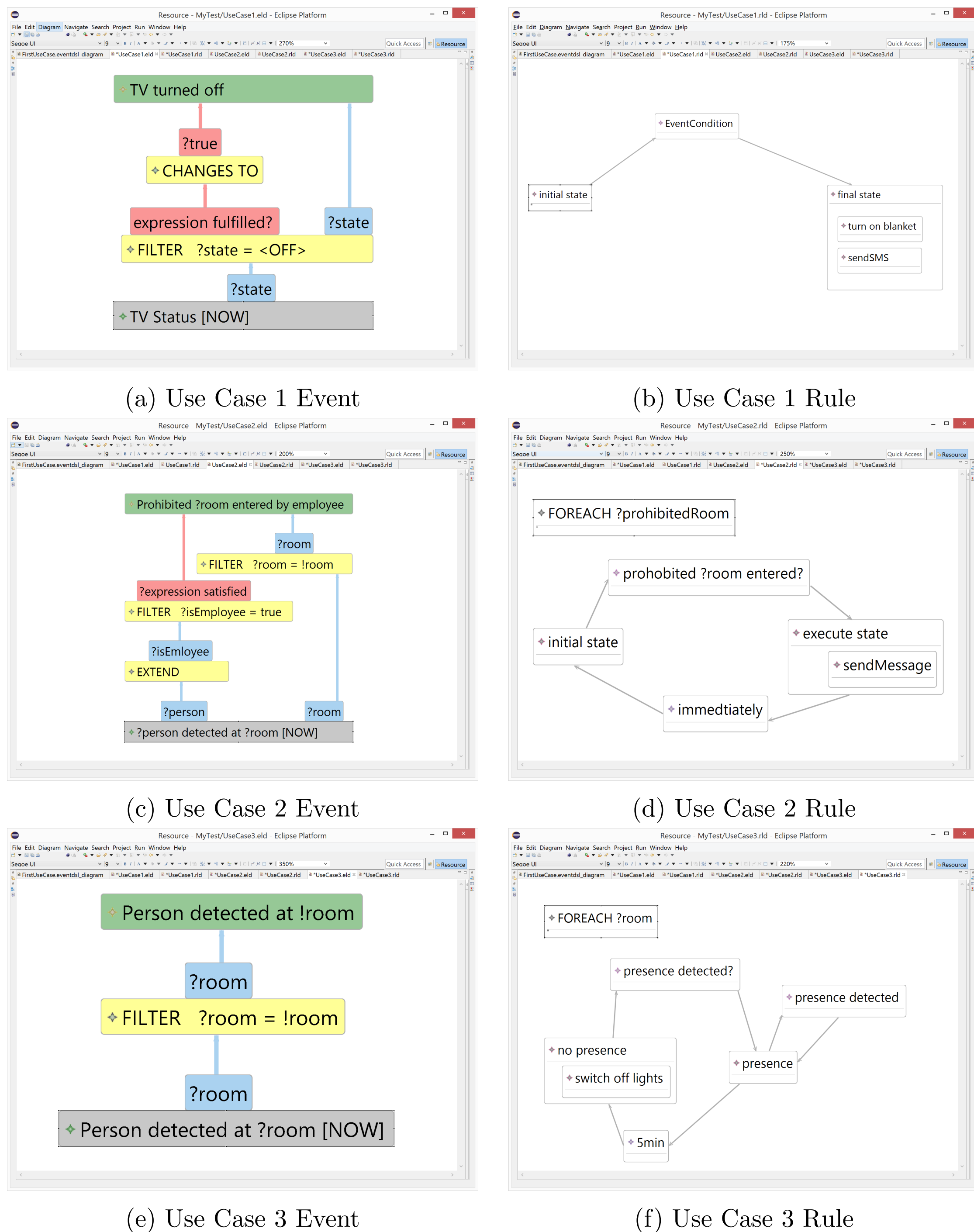
In this section the implementation of the three representative use cases chosen in Section 3.1.2 listed in Table 3.6 is presented. The use cases are modeled with the Event Language EL and the Rule Language RL and were executed using the Event and Rule Automation System ERAS developed as part of this thesis. For testing them a simple sensor simulation framework was developed and some example background knowledge data needed for the execution of the use cases was generated.

Figure 6.3 shows the EL and RL diagrams modeled to represent the use case functionality. It is to notice that in the RL editor the arrow heads are unfortunately very small and hard to see. Nevertheless all edges in diagrams modeled with RL are directed edges.

Figure 6.3a shows the event modeled with EL for use case 1 which registers explicitly to the stream coming from the TV. It uses only the last triple on the stream, extracts the state and generates the event named *TV turned off* ever time the state changes to off. Figure 6.3b shows the corresponding rule using the just seen event as an event-based condition. It starts in an initial state, waits for the event to occur and then changes into its final state where the two given actions, turning on the blanket and sending a SMS, are executed. The time constraint that this rule should only apply every evening after 9:30pm is satisfied by a time-based scheduling of the shown rule everyday between 9:30pm and some not further specified point somewhere in the night. Once the rule fired it will no longer be active until next day at 9:30pm.

Figure 6.3c and Figure 6.3d show the event respectively the rule used for use case 2. The event registers to the streams of all sensors that can detect the presence of a person via the dynamic stream select pattern introduced in this work in a room which could be for example a bluetooth beacon inside the room or the server of a interconnected CCTV system supporting facial recognition. From that stream every time a person is detected at a room the information of person and room are extracted and a lookup is done if the person is an employee via the extend operator. Furthermore this query is explicitly designed as parametrizable and expects the input parameter *!room*. For this use case it is not necessary to explicitly introduce an input parameter as any variable could be bound from outside when instantiating the event rather it actually makes the query unnecessarily a bit more complex. Nevertheless it was chosen to model it this way to better show the powerful capabilities of ERAS within this three simple use cases. The corresponding rule uses the *foreach* operator in RL which allows to schedule a rule multiple times for each result of a SPARQL query. In this case the query lists all prohibited rules of interest. The rule itself starts again in an initial state and waits for the event to be fired. As soon as that happens the state is changed and the message is send to the guard. From that state the rule return immediately to the initial state and waits for another unauthorized employee entering. The timing constraints that this rule should only be active during a certain time is again realized through timed scheduling of the rule. Another possibility using explicit event parameters would have been to not only select the prohibited room in the foreach operator but also the prohibited time (if it is different per room). This information could be passed to the event and than within the event be compared to the time the person entered the room. There an explicit event parameter would be a good choice as the event would not be useable without passing it a timestamp or interval.

Use case 3 has a very simple event shown in Figure 6.3e which again takes *!room* as explicit parameter. It registers to all streams of sensors that can detect presence within a room regardless of their capability to identify the person or not again using the dynamic stream selection pattern. It then only filters by the room and forward the information that a person was detected at that room. The rule shown in Figure 6.3f start in the state named *no presence* and listens for the event. When the event is received and it changes its state and again registers to the event. When



(a) Use Case 1 Event (b) Use Case 1 Rule

(c) Use Case 2 Event (d) Use Case 2 Rule

(e) Use Case 3 Event (f) Use Case 3 Rule

Figure 6.3: The three example use cases listed in Table 3.6 implemented with EL and RL.

the event is detected in this state it loops to itself and thereby causing all current executed conditions, which are those located an all the outgoing edges of the state and in this case the event and a relative time condition, to be canceled when exiting the state and immediately being rescheduled when again entering the state. This means that every time the loop from and to the state *presence* is take the five minute timer for the relative time condition is reset. This is a very similar logic to how presence detector based lighting works and therefore fulfills the use case. This rule also uses the *foreach* operator to apply this rule to all rooms of interest.

6.3 User Acceptance Study for the Event Language

For evaluating user acceptance of ERAS a good way would be to do a user study where subjects are introduced to the system and afterwards presented some scenarios which they would have to solve using ERAS. Unfortunately no end-user-friendly IDE for ERAS could be implemented due to the unforeseen need to implement an own semantic streaming engine. Therefore such a study is not possible within this thesis. As an alternative the design of EL has been evaluated against the textual equivalent ECQELS. The objectives of the study are to find out which of both representations the users prefer and which they achieve better results with. In addition EL will be classified by some *cognitive dimensions* as introduced in[54].

In the following the type and structure of the study is shown. Afterwards the outcome of the study is presented which is then analyzed.

6.3.1 Study Design

The study has been conducted as an online survey. The overall structure was as follows: four queries (further referred to as query 1-4) have been modeled with ECQELS (further referred to as textual representation) and EL (further referred to as visual representation), two of them were rather simple (query 1 and query 2) and the other two were rather complex. The subjects were presented the queries along with the question "What does this query do?" and had to choose between three possible answers in form of a textual description of the result. After each query the subjects were shown if their answer was correct and asked to rate the difficulty by being asked "How hard did you find the previous task?". After completing the queries for one kind of representation they were asked to answer three question on how they think this language relates to three given cognitive dimensions. After finishing the question for both kind of representations they were asked for each query "Which language would you prefer to use?" followed by a possibility to give feedback as free text.

To compensate any possible learning effect the actual study design is more complex than just stated. Figure 6.4 shows the actual structure of the study. The study starts with some questions on sex, age and experience with programming languages whereas the answers to questions on sex and age were optional. The questions on experience programming languages covered textual programming languages, visual programming languages and query languages and were to answer on a five-point Likert scale with the minimum being "used once or twice" and the maximum "daily use" with "never used one" as alternative option. The next page in the survey contained some general information on streaming. Up to that point all subjects were presented the same questions in the same order. After this question the subjects were split into four equal-sized groups to compensate learning and order effects. Therefore the subjects are first split into two groups changing the order of the kind of representation they are presented first. Each of those groups is then further split into two groups which differ in which queries were presented for which kind of representation. The four groups are depicted in Figure 6.4 as horizontally separated columns. It is also depicted that before each group is asked question on a queries in a representation they have not seen before they are presented a short introduction

to that representation. To gain a higher completion rate of the survey every subject is only presented each query once either in visual or in textual representation as the completion rate in online surveys tend to drop if the survey takes to long to complete. After having completed the questions on both representation types all subjects were presented the same questions comparing the representation types per query. In the following the details of the survey are presented such as details on questions and answers and also some screenshots.

The type specific introduction to the visual and textual representation were kept as short as possible to better represent how intuitive the representation is to use. So for the textual representation it only contained a very simple example query with a textual description of the result of that query. For the visual representation a mapping between the icon and their meaning were added as the visual representation used icons instead of keywords. The actual language queries are designed like the introduction but with three possible answers to choose from. A screenshot for a textual representation question is shown in Figure 6.5a and for a visual representation question in Figure 6.5b. After each query question the subject was asked to rate how hard he found the previous task as shown in Figure 6.5c. As a scale the SMEQ (Subjective Mental Effort Questionnaire)[77] was chosen as it is well suited for measuring mental effort for a single task, especially in online surveys citeSauro2009.

For evaluation of a query type three cognitive dimension have been chosen from [54]: Consistency, Role-Expressiveness and Closeness of Mapping. The first two do reflect the intention to design an intuitive language and the third one checks if EL is non-domain-specific even though it has been developed in the context of home automation. As the dimensions are not known to the subjects they are asked for by posing a question understandable for everybody where the subjects can answer on a five-point Likert scale with “strongly disagree” as minimum and “strongly agree” as maximum as shown in Figure 6.5d. Language comparison is also done using a five-point Likert scale where on the one side of the scale the query in textual representation is shown and on the other side of the scale the query in visual representation. The memorization questions were designed as follows: The solution of the query just seen in the question before is given as text and four possible queries in the same type of representation are shown of which one is the query shown in the last question. The feedback page contains three input fields for free test answers, one for comments on the visual language, one for comments on the textual language and one for general comments.

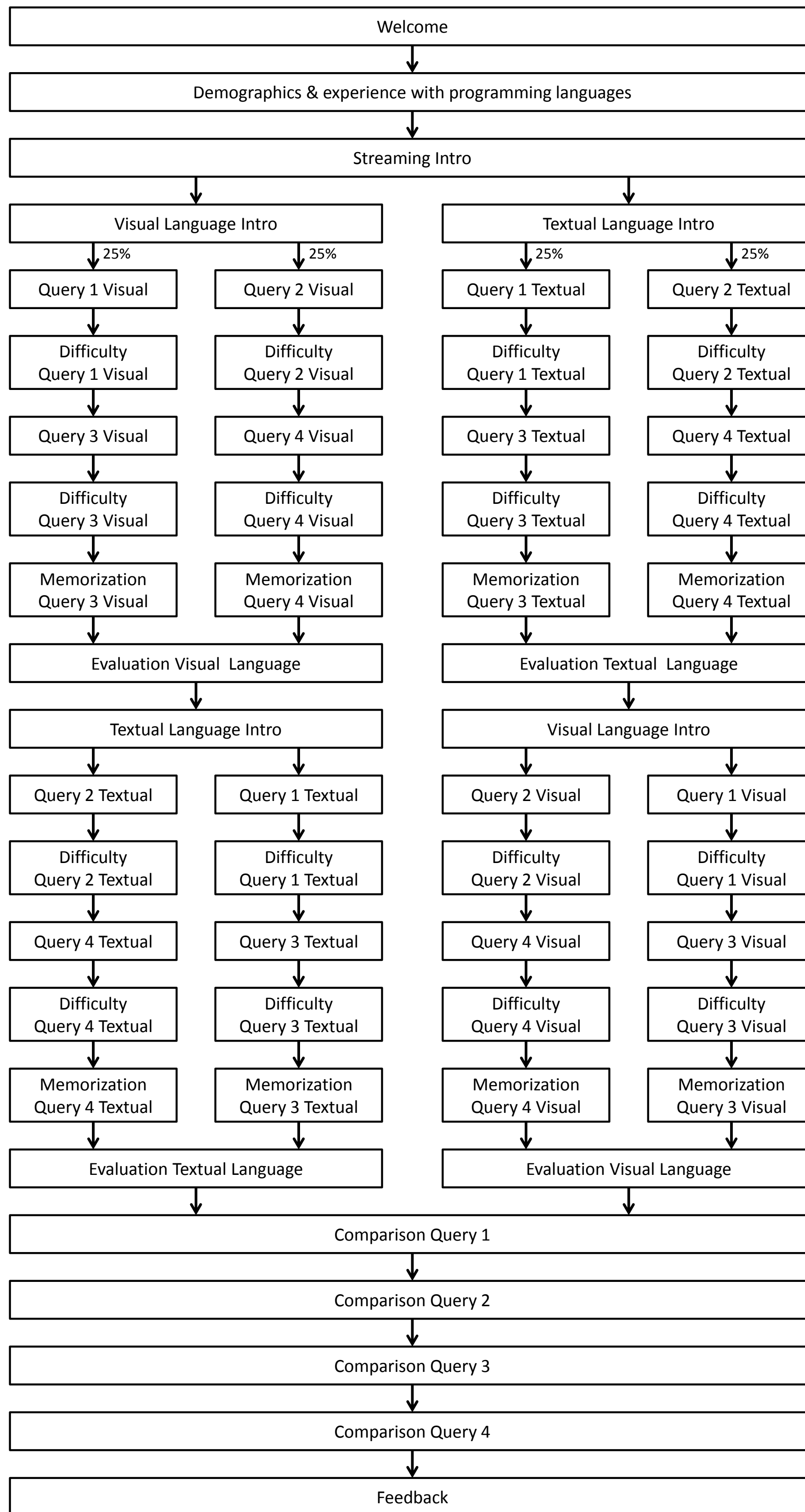


Figure 6.4: A schema showing the structure of the online survey.

TECO — KIT Karlsruher Institut für Technologie

4. What does this query do?

```
SELECT ?room
WHERE
{
   {
      SELECT ?sensor ?temp
      FROM [List of all temperature sensors with their last sensed value,
            refreshed on data change]
   }
   {
      SELECT ?sensor2 ?room
      FROM [List of all sensors with the room they are located in,
            refreshed on data change]
   }
   FILTER(?sensor = ?sensor2)
   FILTER(?temp > 30)
}
```

- ○ It yields a list with all rooms with a temperature above 30°C every 30 minutes.
- ○ It yields a list with all rooms that contain a sensor whose last sensed value was above 30° C every time a new sensor reading becomes available.
- ○ It yields a random room that contains a sensor whose last sensed value was above 30° C every time a new sensor reading becomes available.

Next

Michael Jacoby, Karlsruhe Institute of Technology – 2014 — 19% completed

(a)

TECO — KIT Karlsruher Institut für Technologie

10. What does this query do?

EVENT | When? | What? ?avg_temp

changes? to ?true

expression satisfied? ?avg_temp — ?avg_temp > 30

?avg_temp — ?temp AS ?avg_temp

?sensor ?temp — List of all temperature sensors in a room with their last sensed value, refreshed when a new value arrives

- ○ It yields the average temperature every time it changes.
- ○ It yields the average temperature every time it raises above 30°C.
- ○ It yields the average temperature every time it changes and it's above 30°C.

Next

Michael Jacoby, Karlsruhe Institute of Technology – 2014 — 51% completed

(b)

TECO — KIT Karlsruher Institut für Technologie

5. How hard did you find the previous task?

Scale	Label
150	
140	
130	
120	
	Tremendously hard to do
110	
	Very, very hard to do
100	
90	
	Very hard to do
80	
70	Pretty hard to do
60	
	Rather hard to do
50	
40	
	Fairly hard to do
30	A bit hard to do
20	
10	Not very hard to do
0	Not at all hard to do

Next

Michael Jacoby, Karlsruhe Institute of Technology – 2014 — 24% completed

(c)

TECO — KIT Karlsruher Institut für Technologie

15. To what degree do you agree with the following statements concerning the just seen visual programming language?

Below the questions an example of the visual language is shown.

	strongly disagree				strongly agree
When a person knows some of the language structure, the rest can be successfully guessed.	○	○	○	○	○
It is easy to answer the question 'What is this bit for?'.	○	○	○	○	○
The language is rather general than specific to a certain domain.	○	○	○	○	○

EVENT | When? | What? ?avg_temp

changes? to ?true

expression satisfied? ?avg_temp — ?avg_temp > 30

?avg_temp — ?temp AS ?avg_temp

?sensor ?temp — List of all temperature sensors in a room with their last sensed value, refreshed when a new value arrives

Next

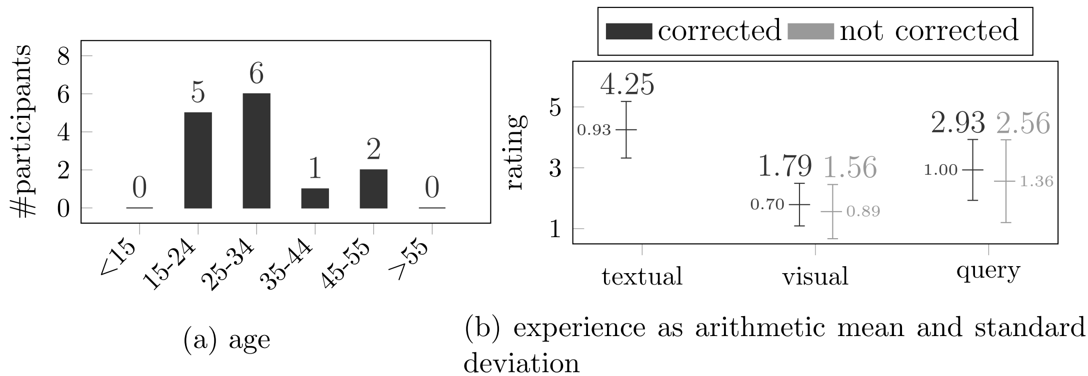
Michael Jacoby, Karlsruhe Institute of Technology – 2014 — 73% completed

(d)

Figure 6.5: Screenshots from the online survey.

6.3.2 Study Outcome

In this section the outcome of the online survey is presented. The total number of subjects was 20 whereof 4 did not complete the survey, leaving a total of 16 complete records. The subjects were equally distributed within the groups so that each group consisted of 4 subjects. Among these 16 subjects 13 were male, one female and two did not answer that question. The subjects were between 15 and 55 years old as shown in detail in Figure 6.6a. In Figure 6.6b the self-assessment of the subjects regarding their programming experience is shown as the arithmetic mean and the standard deviation both rounded to two decimal places. All subjects were experienced using textual programming languages whereas for visual and query languages two subjects each had no experience with. Therefore it is to notice that Figure 6.6b shows the corrected values where all subjects with no experience with the given language type are not taken into account as well es the not corrected values which take subjects which had no experience with a given language type into account with a score of 0.



(a) age

(b) experience as arithmetic mean and standard deviation

Figure 6.6: Age and experience of subject within study.

Figure 6.7 shows the number of correct answers to the query questions for each query separated by query representation type. It shows that for query 1 all eight subjects asked that question did answer it correctly when presented the visual representation whereas for the textual representation two out of eight subjects did answer false. For query two five out of the eight subjects asked answered correctly when presented the visual representation whereas when presented the textual representation seven answered correctly. For query 3 again five out of eight subjects answered correctly when presented the visual representation and for the textual representation four out of eight subjects did answer correctly and the other four false. For query 4 seven out of eight subject answered correctly regardless of the representation shown in the query.

Figure 6.8 shows the arithmetic mean and standard deviation rounded to the nearest integer of the difficulty of the questions according to the subjects per query and representation type. The difficulty was measured on a SMEQ scale between 0 and 150. It also shows that the mean difficulty is very close for the visual and textual representation for each query and that the standard deviation for the textual representation is always greater or equal to the standard deviation of the visual representation.

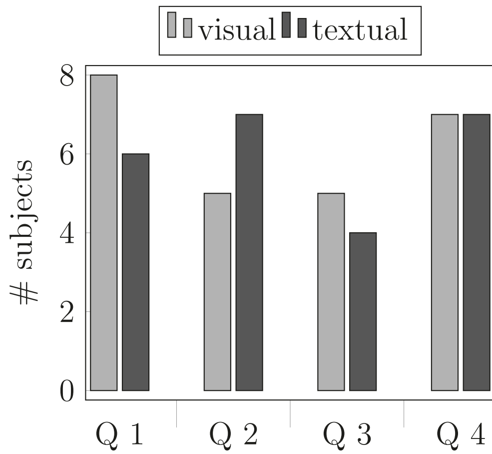


Figure 6.7: Number of query questions answered correct for all queries separated by query representation type.

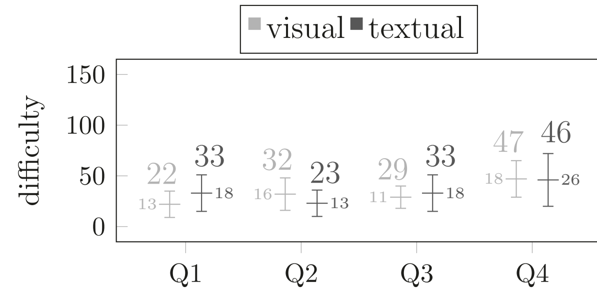


Figure 6.8: Subjective difficulty of the language questions for all queries by query representation type displayed as arithmetic mean and standard deviation.

Figure 6.9 shows the outcome of the question on the three cognitive dimensions Consistency, Role-Expressiveness and Closeness of Mapping. As the answer was in form of a five-point Likert scale with the minimum of 1 meaning "strongly disagree" and the maximum of 5 meaning "strongly agree" the outcome is depicted again with arithmetic mean and standard deviation rounded to two decimal places.

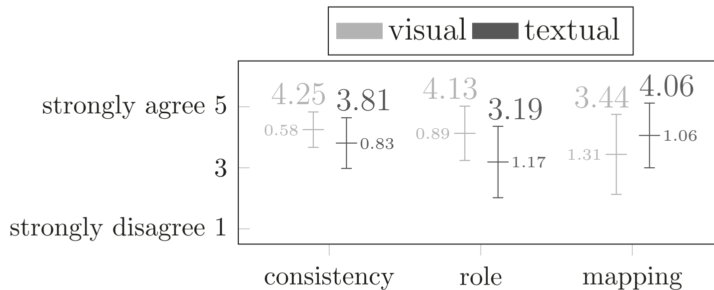


Figure 6.9: Outcome of the language evaluation questions on cognitive dimensions displayed as arithmetic mean and standard deviation.

Figure 6.10 shows the outcome of the final comparison of both languages where the subjects were shown each query in both representations side-by-side and were asked "Which language would you prefer to use?". The answer was again in form of a five-point Likert scale with the visual representation on the one end and the textual representation on the other representing the preference for one of the two representation.

Figure 6.11 shows how the subjects scored in the memorization questions. For query 3 there was no difference in the representation type as with both seven out of 8 subjects did answer correctly. For query 4 all eight subjects answered correctly when presented the visual representation but when presented the textual representation only six answered correctly.

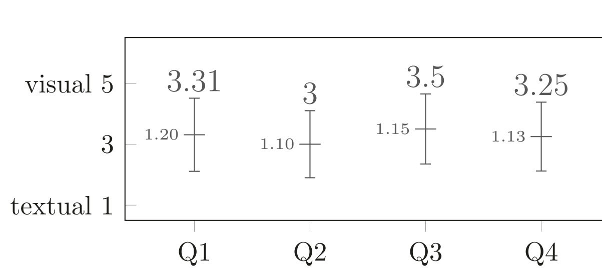


Figure 6.10: Outcome of the questions on language comparison between the visual and the textual representation for all queries displayed as arithmetic mean and standard deviation..

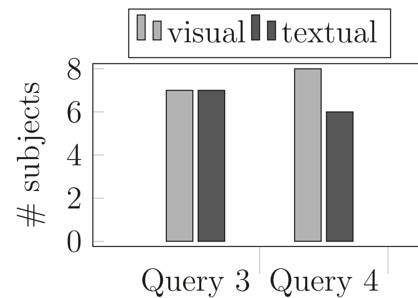


Figure 6.11: Number of memorization questions answered correctly for both queries asked and both representation types.

6.3.3 Outcome Analysis

In this section the outcome of the online survey presented in the previous section is analyzed and discussed. Starting with the subjects Figure 6.6b shows that the average subject uses textual programming languages nearly on a daily basis and is rather inexperienced with visual programming languages. In fact two subjects have never used a visual programming language before and no subject has rated their experience with visual languages above medium (3). The subjects' experience with query languages is also medium in average whereby there are also two subjects that never used a query language before. Unfortunately no clear user groups could be identified. Looking at the outcome of the query question in Figure 6.7 the visual representation scores better for query 1 and query 3. For query 2 the textual representation scores better and for query 4 the are indifferent. Overall the visual representation scores slightly better than the textual representation in correctness as in sum 25 of 32 answers were correct whereas for the textual representation only 24 of 32 questions were answered correctly. To better draw a conclusion out of this the the subjective difficulty of the question as stated by the subjects shown in Figure 6.8 should be taken into account. Again for questions 1 and 3 the visual representation scored better (meaning a lower subjective difficulty). For query 2 and 4 the outcome is nearly the same as for the query question correctness; in query 2 the textual representation scores better and in query 4 both representations are indifferent. It is noticeable that the standard deviation for the visual representation is always less or equal to the standard deviation for the textual representation. This is probably due to the fact that the experience with visual languages is more uniform than the experience with textual languages. Nonetheless it is hard to say why query 2 yields worse results in query question correctness when using the visual representation.

Regarding the outcome of the memorization questions shown in Figure 6.11 it can be stated that the visual representation scores slightly better and therefore seems to be more easy to memorize. This is probably because shapes, icons and colors are more easy to remember than plain text or even text with syntax highlighting as in the study.

Analyzing the outcome of the language comparison shown in Figure 6.10 it is again shown that there are no great differences between the representation styles. In fact the visual representation scores a little better with all means being greater or equal

to 3 but as the standard deviation is always greater 1 this result is not significant. Furthermore no significant difference between the outcomes for the different groups could be seen. It rather seems that the preference for a representation is personal preference as some subjects preferred the same representation for all queries. A reason for this personal preference could be that the subject is very experienced with programming languages of the one representation type and very inexperienced with the other and therefor tends to prefer the representation he is skilled in. This could be the case for one subject that stated to use textual programming languages on a daily basis and never had used a visual programming language and then slightly preferred the textual representation for all queries. Comparing Figure 6.8 and Figure 6.10 it is surprising that although the subjects found the visual representation of query 2 more difficult than the textual representation they are completely indifferent which representation they would like to use for this query.

Feedback on visual representation
- clear structure (3x)
- easy to learn/understand for non-programmers (3x)
- bulky, uses a lot of space (3x)
- easier for complex queries (2x)
- symbols are intuitive
- no editor known
- data flow partially confusing
- structured like textual languages
Feedback on textual representation
- more compact (3x)
- hard for non-programmers (3x)
- hard to read (3x)
- easy for experienced programmers used to SQL (2x)
- did not understand the curly braces / data flow (2x)
General feedback
- maybe more easy visual language by reducing universality
- do not want to use visual language for huge queries
- comparison is hard without suitable editor

Table 6.2: Summary of the free text feedback of the online survey.

Table 6.2 shows the feedback given in free text form as bullet points annotated with their frequency of occurrence. As positive for the visual representation it was stated that is has a clear structure, uses intuitive symbols and is easy to learn and understand for non-programmers. Also two subjects found it easier to use when dealing with complex queries. This is contrasted with the feedback for the textual representation which is stated to be hard to read and generally hard for non-programmers. It also is stated to be more compact whereas the visual representation is stated to be bulky. For both representations a few subject did find the data flow confusing. Furthermore the textual representation is considered to be easy for experienced programmers that are used to SQL as it strongly resembles SQL.

In the general feedback a subject stated that it would probably be a good idea to make the visual representation even more easy to understand by reducing its universality. This is a good idea and should be done when applying it to a specific domain. Therefore the most used patterns could be investigated by a suitable study and special controls or wizards could be implemented. Actually this has been done in this work for the dynamic sensor selection. Another subject stated that he does not want to use the visual representation for huge/complex queries which is of course related to its bulkiness. The third general feedback is cited here as it confirm exactly was has been proposed in the introduction of this section: “It is hard to compare textual languages and visual languages only on images. It would be way more realistic, if this would be done in a real environment, where I can click, and check the parentheses and so on. It would allow me to move my mouse over certain elements to get more information, so it would be faster to get the meaning of it, in my opinion.” This statement indicates that usability of a language is very closely connected to its editor/IDE. With a proper IDE the problem of bulkiness could probably be overcome for example with some mechanism to change the level of detail by collapsing certain areas.

Figure 6.9 shows the outcome of the questions on the cognitive dimensions. The visual representation is rated higher in consistency, meaning it is easier for someone who knows some of the language structures to successfully guess the rest, and in role-expressiveness, meaning that it is more easy to answer the question “What is this bit for?” for the visual representation. Considering this the visual representation can be seen as slightly more intuitive than the textual representation. Interestingly the textual representation is conceived as more general than the visual one although they both have the same expressiveness.

Putting all the results together the visual representation seems to yield slightly more correct results and is also preferred by the users. Furthermore it seems slightly more intuitive and easy and is therefore better suited for inexperienced or even non-programmers. It is to notice that these differences are not significant and therefor are only a personal interpretation. Furthermore these differences do not hold for all types of queries as the results for query 2 show the contrary. The reason to this could not be discovered. It is also highlighted that the visual representation is only easy-to-use when there is a suitable editor.

6.4 Summary

In this chapter the evaluation of the ERAS system and especially ECQELS was shown. In Section 6.1 a new benchmark for streaming engines was introduced using the feature of average response time which seems of more interest for users of such systems then the formerly used average query execution time or executions per seconds whereby an execution is defined as insertion of a triple.

The evaluation has shown that ERAS is working. It was shown that ECQELS can compete with CQELS for queries without or only small background knowledge and is superior when processing multiple queries a once. On the other hand it seems that ECQELS gets very slow for queries using large background knowledge due to its architecture based on Jena ARQ. Furthermore there seems to be a memory leak in ECQELS causing the execution to getting slower and slower over time.

In Section 6.3.2 it was shown how the three representative use cases introduced in Section 3.1.2 were implemented using ERAS which shows that the developed system is working.

Finally in Section 6.3 an online user study was introduced in which ECQELS and EL, the textual and visual representation, are compared with regard to usability. The study structure and outcome were presented and the results were discussed. The outcome was that the visual representation, EL, scored slightly better in most cases and was slightly more favored by the subjects but these differences could not be shown to be significant.

7. Conclusion and Future Work

7.1 Conclusion

In this thesis the rule-based home automation system called ERAS (Event and Rule Automation System) has been developed. To the knowledge of the author this is the first work using semantic stream technologies in the context of home automation. Using semantic streaming in home automation provides the gains of using semantics but with far less performance loss in comparison to normal query-based semantic systems. Therefore it seems perfectly suited for the home automation domain and in this thesis it has been shown that it can be beneficial.

ERAS has been realized using two hierarchically visual languages, the event language EL and the Rule Language RL, following a two-folded approach to adapt the different needs for different domains covered. EL focuses on detecting meaningful events based on processing sensor data streams and offers re-usability by supporting parametrization. This thesis also contributes a new pattern for semantic stream processing especially useful in dynamic infrastructures called dynamic stream selection (or also dynamic sensor selection) presented in Chapter 3.1 which is also supported by EL. RL was designed as a visual modeling language for rules based on time triggers or event triggers modeled with EL and resembles a finite state machine. The use of events modeled in EL as triggers in RL in combination with native support for parametrization of these events allows extensive re-usability of user-defined elements and supports a repository-style usage. Thereby all requirements specified in Section 3.3 have been addressed.

As already mentioned in Section 4.5 the editors for EL and RL could not be implemented as user-friendly as desired due to the fact that CQELS which was planned to use as execution runtime for EL has proven not suitable enough for this work. This fact required to implement a custom semantic streaming engine from scratch which cost much time and has not been scheduled. In spite of everything a running version of ERAS has been implemented.

As a further contribution of this thesis the structured literature review on and analysis and classification of use cases in the domain of home automation presented in Section 3.1 should be mentioned.

Another contribution of this work is the developed semantic streaming engine ECQELS along with the corresponding benchmark for semantic streaming engines.

7.2 Future Work

As the editors for EL and RL could only be implemented in a very basic version the list of possible future work to improve ERAS is long. The first thing of course would be to implement them in a nicer looking and more user-friendly way and to integrate them in a common IDE. Such an IDE could also support further additional features like a repository of present devices or previously user-defined elements to support re-usability. Therefore the elements of EL and RL could be represented as RDF and stored together with the metadata of the deployed devices. Furthermore an existing system for physical integration of actuators could be included so that ERAS becomes are complete infrastructure solution. On top an online event and rule repository could be possible allowing to transfer events and rules between different homes which should be possible if they use a common domain vocabulary or a mapping is provided.

Also ECQELS does offer possibilities for future work. For example performance could be optimized and memory consumption could be reduced. Especially the potential memory leak needs further investigation. In addition an better caching logic for ECQELS could be implemented to better perform with large background data. Furthermore ECQELS could be extended to support graph-based RDF streams as discussed in [60]. This probably will be more complex as it requires to re-think the window semantic on graph-based streams.

A question more general and not only focused on the work presented in this thesis is how users can be supported in using dynamic sensor selection. This topic came up in the pre-study conducted and has been intensively discussed. There were multiple proposals such as floor plans, tree-views with check boxes and faceted search. From the author's view the most promising proposal is the use of a faceted search in combination with an auto-generated abstract representation, probably with a textual description close to natural language.

Furthermore there seems to be a need for further studies on use cases for rule-based home automation. This could be done probably by interviewing non-expert programmers that are already interested or even experienced in home automation. As soon as an IDE for ERAS suitable for non-expert programmers is available there should be a study whether the concept improves ease of use or even better the IDE should be designed iteratively including the feedback of non-expert programmer users more early in the development.views

Bibliography

[1] https://ifttt.com/. Last accessed: 2014-12-15.

[2] http://research.microsoft.com/en-us/um/redmond/projects/homeos/homeos-demos.htm. Last accessed: 2014-12-15.

[3] http://www.openhab.org/. Last accessed: 2014-12-15.

[4] https://github.com/cdjackson/HABmin/wiki/Rule-Designer:-Overview. Last accessed: 2014-12-15.

[5] http://www.w3.org/TR/rdf11-concepts/. Last accessed: 2014-12-15.

[6] http://www.w3.org/TR/rdf-schema/. Last accessed: 2014-12-15.

[7] http://www.w3.org/TR/owl-ref/. Last accessed: 2014-12-15.

[8] http://www.w3.org/TR/owl2-overview/. Last accessed: 2014-12-15.

[9] http://www.streambase.com/. Last accessed: 2014-12-15.

[10] http://www.tibco.com/assets/blt0e3d0c71656918c5/ds-tibco-streambase-overview_tcm8-19262.pdf. Last accessed: 2014-12-15.

[11] http://www.espertech.com/. Last accessed: 2014-12-15.

[12] http://www.w3.org/TR/rdf-sparql-query/. Last accessed: 2014-12-15.

[13] http://www.w3.org/2005/Incubator/ssn/. Last accessed: 2014-12-15.

[14] http://spitfire-project.eu/incontextsensing/ontology.php. Last accessed: 2014-12-15.

[15] http://www.spitfire-project.eu. Last accessed: 2014-12-15.

[16] http://www.w3.org/TR/sparql11-query/#grammar. Last accessed: 2014-12-15.

[17] http://code.google.com/p/lsbench/. Last accessed: 2014-12-15.

[18] http://code.google.com/p/cqels/wiki/Experiments. Last accessed: 2014-12-15.

[19] http://dblp.uni-trier.de/. Last accessed: 2014-12-15.

[20] DJ Abadi and Don Carney. "Aurora: a new model and architecture for data stream management". In: *The VLDB Journal—The International Journal on Very Large Data Bases* 12.2 (Aug. 2003), pp. 120–139. ISSN: 1066-8888.

[21] Jagrati Agrawal et al. "Efficient pattern matching over event streams". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08* (2008), p. 147.

[22] Alain Alani, Harith and Szomszor, Martin and Cattuto, Ciro and Van den Broeck, Wouter and Correndo, Gianluca and Barrat. *Live social semantics*. Ed. by Abraham Bernstein et al. Vol. 5823. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. ISBN: 978-3-642-04929-3.

[23] I Androutsopoulos. “Natural language interfaces to databases–an introduction”. In: *Natural language engineering* 1.01.709 (1995), pp. 1–50.

[24] Darko Anicic and Paul Fodor. “EP-SPARQL: a unified language for event processing and stream reasoning”. In: *Proceedings of the 20th international conference on World wide web* (2011), pp. 635–644.

[25] Arvind Arasu, S Babu, and Jennifer Widom. “An abstract semantics and concrete language for continuous queries over streams and relations”. In: (2002).

[26] Arvind Arasu, Shivnath Babu, and Jennifer Widom. “The CQL continuous query language: semantic foundations and query execution”. In: *The VLDB Journal* 15.2 (July 2005), pp. 121–142. ISSN: 1066-8888.

[27] Mostafa M Aref and Mohammed a Tayyib. “Lana–Match algorithm: a parallel version of the Rete–Match algorithm”. In: *Parallel Computing* 24.5-6 (June 1998), pp. 763–775. ISSN: 01678191.

[28] Nazmiye Balta-Ozkan et al. “Social barriers to the adoption of smart homes”. In: *Energy Policy* 63 (Dec. 2013), pp. 363–374. ISSN: 03014215.

[29] Nazmiye Balta-Ozkan et al. “The development of smart homes market in the UK”. In: *Energy* 60 (Oct. 2013), pp. 361–372. ISSN: 03605442.

[30] DF Barbieri and D Braga. “An execution environment for C-SPARQL queries”. In: *Proceedings of the 13th International Conference on Extending Database Technology* (2010).

[31] DF Barbieri, D Braga, and S Ceri. “C-SPARQL: SPARQL for continuous querying”. In: *Proceedings of the 18th international conference on World wide web* c (2009).

[32] DF Barbieri, Daniele Braga, and S Ceri. “Querying RDF Streams with C-SPARQL”. In: *ACM SIGMOD Record* 39.1 (2010), pp. 20–26.

[33] Michael Beigl, Albert Krohn, and Till Riedel. “The uPart experience: Building a wireless sensor network”. In: *Information Processing in Sensor Networks, 2006. IPSN 2006. The Fifth International Conference on* (2006), pp. 366–373.

[34] Michael Beigl et al. “µparts: Low cost sensor networks at scale”. In: *Ubicomp 2005* (2005).

[35] T Berners-Lee. “The semantic web”. In: *Scientific american* 284.5 (2001), pp. 28–37.

[36] Peter Boncz, Orri Erling, and Minh-duc Pham B. “Advances in Large-Scale RDF Data Management”. In: Lecture Notes in Computer Science 8661 (2014). Ed. by Sören Auer, Volha Bryl, and Sebastian Tramp, pp. 21–44.

[37] AJ Brush, Bongshin Lee, and Ratul Mahajan. “Home automation in the wild: challenges and opportunities”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2011).

[38] Marie Chan et al. “A review of smart homes- present state and future challenges.” In: *Computer methods and programs in biomedicine* 91.1 (July 2008), pp. 55–81. ISSN: 0169-2607.

[39] B. Chandrasekaran, J.R. Josephson, and V.R. Benjamins. “What are ontologies, and why do we need them?” In: *IEEE Intelligent Systems* 14.1 (Jan. 1999), pp. 20–26. ISSN: 1094-7167.

[40] Joëlle Coutaz and Sybille Caffiau. “Early lessons from the development of SPOK, an end-user development environment for smart homes”. In: *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication* (2014), pp. 895–902.

[41] Joëlle Coutaz and Emeric Fontaine. “DisQo: A user needs analysis method for smart home”. In: *Proceedings of the 6th Nordic Conference on Human-Computer Interaction: Extending Boundaries* (2010).

[42] Douglas Crockford. *The application/json media type for javascript object notation (json)*. 2006.

[43] Gianpaolo Cugola and Alessandro Margara. “Processing flows of information: From data stream to complex event processing”. In: *ACM Computing Surveys (CSUR)* V.i (2012), pp. 1–70.

[44] Elmehdi Damou. “ApAM: Un environnement pour le développement et l’exécution d’applications ubiquitaires”. Thèse de doctorat. Universit´ e de Grenoble, 2013.

[45] Scott Davidoff, MK Lee, and Charles Yiu. “Principles of smart home control”. In: *UbiComp 2006: Ubiquitous Computing* (2006), pp. 19–34.

[46] AK Dey et al. “iCAP: Interactive prototyping of context-aware applications”. In: *Pervasive Computing* (2006), pp. 254–271.

[47] Anind Dey, Gregory Abowd, and Daniel Salber. “A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications”. In: *Human-Computer Interaction* 16.2 (Dec. 2001), pp. 97–166. ISSN: 0737-0024.

[48] Colin Dixon et al. “An operating system for the home”. In: *NSDI* (2012).

[49] Colin Dixon et al. “The home needs an operating system (and an app store)”. In: *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks - Hotnets ’10* (2010), pp. 1–6.

[50] Z Drey, Julien Mercadal, and Charles Consel. “A taxonomy-driven approach to visually prototyping pervasive computing applications”. In: *Domain-Specific Languages* (2009).

[51] WK Edwards and RE Grinter. “At home with ubiquitous computing: seven challenges”. In: *Ubicomp 2001: Ubiquitous Computing* (2001).

[52] Dieter Fensel et al. “Towards LarKC: A Platform for Web-Scale Reasoning”. In: *2008 IEEE International Conference on Semantic Computing* (Aug. 2008), pp. 524–529.

[53] Lukasz Golab and MT Özsu. “Issues in data stream management”. In: *ACM Sigmod Record* 32.2 (2003), pp. 5–14.

[54] TRG Green and M Petre. “Usability analysis of visual programming environments: a ’cognitive dimensions’ framework”. In: *Journal of Visual Languages & Computing* January (1996), pp. 1–51.

[55] Matthias Grossmann and Martin Bauer. “Efficiently managing context information for large-scale scenarios”. In: *Pervasive Computing and Communications, 2005. PerCom 2005. Third IEEE International Conference on* PerCom (2005).

[56] Bin Guo, Daqing Zhang, and Michita Imai. “Enabling user-oriented management for ubiquitous computing: The meta-design approach”. In: *Computer Networks* 54.16 (Nov. 2010), pp. 2840–2855. ISSN: 13891286.

[57] Daniel Gyllstrom et al. “On Supporting Kleene Closure over Event Streams”. In: *2008 IEEE 24th International Conference on Data Engineering* (Apr. 2008), pp. 1391–1393.

[58] M Jimenez, Francisca Rosique, and P Sanchez. “Habitation: a domain-specific language for home automation”. In: *Software, IEEE* 26.4 (2009), pp. 30–38.

[59] Yung-Wei Kao and Shyan-Ming Yuan. “User-configurable semantic home automation”. In: *Computer Standards & Interfaces* 34.1 (Jan. 2012), pp. 171–188. ISSN: 09205489.

[60] R Keskisärkkä and E Blomqvist. “Event Object Boundaries in RDF Streams–A Position Paper”. In: ().

[61] Mirko Knoll et al. “Scripting your home”. In: *Location-and Context-Awareness* (2006), pp. 274–288.

[62] G. Kortuem et al. “Smart objects as building blocks for the Internet of things”. In: *IEEE Internet Computing* 14.1 (Jan. 2010), pp. 44–51. ISSN: 1089-7801.

[63] Othmar Kyas. *How To Smart Home*. ISBN: 9783944980003.

[64] D Le-Phuoc and M Dao-Tran. “A native and adaptive approach for unified processing of linked streams and linked data”. In: *The Semantic Web–ISWC 2011* (2011), pp. 370–388.

[65] D Le-Phuoc, M Dao-Tran, and MD Pham. “Linked stream data processing engines: Facts and figures”. In: *The Semantic Web–ISWC 2012* (2012), pp. 300–312.

[66] D Le-Phuoc et al. *Continuous query optimization and evaluation over unified linked stream data and linked open data*. Tech. rep. Galway, Ireland: DERI, IDA Business Park, 2010.

[67] Claire Maternaghan. *The homer home automation system*. December. Department of Computing Science and Mathematics, University of Stirling, 2010.

[68] BM Michelson. “Event-driven architecture overview”. In: *Patricia Seybold Group* (2006).

[69] Rajeev Motwani et al. “Query processing, resource management, and approximation in a data stream management system”. In: *CIDR* (2003).

[70] Kostas Patroumpas and Timos Sellis. “Maintaining consistent results of continuous queries under diverse window specifications”. In: *Information Systems* 36.March (2011), pp. 42–61.

[71] Vincent Ricquebourg and David Durand. “Context inferring in the Smart Home: An SWRL approach”. In: *Advanced Information Networking and Applications Workshops, 2007, AINAW’07. 21st International Conference on* 2 (2007), pp. 290–295.

[72] Michael Rietzler, J Greim, and M Walch. “homeBLOX: introducing process-driven home automation”. In: *Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication* (2013), pp. 801–808.

[73] Mikko Rinne, Esko Nuutila, and S Törmä. “INSTANS: High-Performance Event Processing with Standard RDF and SPARQL”. In: *11th International Semantic Web Conference ISWC 2012* (2012), pp. 6–9.

[74] Mikko Rinne, S Törmä, and E Nuutila. “SPARQL-Based Applications for RDF-Encoded Sensor Data”. In: *SSN* 904 (2012), pp. 81–96.

[75] Mikko Rinne et al. “Processing heterogeneous rdf events with standing sparql update rules”. In: *On the Move to Meaningful Internet Systems: OTM 2012* (2012), pp. 797–806.

[76] Pedro Sánchez et al. “A framework for developing home automation systems: From requirements to code”. In: *Journal of Systems and Software* 84.6 (June 2011), pp. 1008–1021. ISSN: 01641212.

[77] Jeff Sauro and JS Dumas. “Comparison of three one-question, post-task usability questionnaires”. In: *Proceedings of the SIGCHI Conference on Human …* (2009), pp. 1599–1608.

[78] Leila Takayama, Caroline Pantofaru, and David Robson. “Making technology homey: finding sources of satisfaction and meaning in home automation”. In: *Proceedings of the 2012 ACM Conference on Ubiquitous Computing* (2012), pp. 511–520.

[79] KJ Turner. “Flexible management of smart homes”. In: *Journal of Ambient Intelligence and Smart Environments* 3.2 (2011), pp. 83–109.

[80] EU Warriach. “State of the Art: Embedded Middleware Platform for A Smart Home.” In: *International Journal of Smart Home* 7 (2013), pp. 1–20.

[81] Eugene Wu, Yanlei Diao, and Shariq Rizvi. “High-performance complex event processing over streams”. In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data* (2006), pp. 407–418.

[82] Marcin Wylot and J Pont. “dipLODocus - Short and Long-Tail RDF Analytics for Massive Webs of Data”. In: *The Semantic Web–ISWC 2011* (2011), pp. 778–793.

[83] Qunzhi Zhou, Yogesh Simmhan, and Viktor Prasanna. *SCEPter: Semantic complex event processing over end-to-end data flows.* Tech. rep. April. Computer Science Department, University of Southern California, 2012.