



# MONASH University

## **CDQL: A Generic Context Definition and Querying Language for Internet of Things Applications**

Alireza Hassani

Faculty of Information Technology  
Monash University

Supervisor

Dr. Chris Ling

Dr. Pari Delir Haghighi

Professor Arkady Zaslavsky

Dr. Prem Prakash Jayaraman

Dissertation submitted for partial fulfillment of the Requirements for the  
Degree of

Doctor of Philosophy

May- 2019  
Monash University

## **Copyright notice**

©copyright  
by  
Alireza Hassani

2019

## **Declaration**

This thesis is an original work of my research and contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Signature:

Print Name: Alireza Hassani

Date: 30/06/2019

**To my wife Sahar**



# Acknowledgements

I would like to take the opportunity to appreciate and acknowledge the people whose contributions and support realised this dissertation.

First and foremost, I want to thank my supervisors Prof. Arkady Zaslavsky, Dr. Chris Ling, Dr. Pari Delir Haghighi, and Dr. Prem Jayaraman for their supervision, friendship, support and advice during my PhD candidature.

Arkady, thank you for believing in me and giving me the opportunity to work under your supervision. I am thankful for your insights and immense knowledge that enlighten my path and enabled me to complete this PhD dissertation. Working with you truly thought me a lot and helped me to grow as a research scientist.

Pari, thank you for all your support and help during my candidature. You have been more than a supervisor for me. Your continuous academic and spiritual supports and incredible encouragement accompanied me throughout the years of my candidature. I would also like to thank you for providing me with the opportunity to join your teaching team.

Prem, I want to thank you for all your invaluable advice, sharp intellect, and brilliant ideas. Your bright vision has been of great value for my research and considerably contributed to the development of the context definition and query language in this dissertation.

Chris, thank you for your kindness, understanding, and support. Your valuable feedback helped me a lot during my candidature. I truly appreciate the time you took to review my dissertation and finalise this work.

I would like to thank Allison Mitchell and Helen Cridland for providing me with administrative support during my research.

I would express my gratitude to Rob Gray for helping me to proofread my dissertation.

I would like to thank my fellow PhD student, Alexey Medvedev for his friendship and fruitful discussions.

Last but not least, I would like to thank my family, especially my beloved wife, my parents, and my brother for their endless support, patience, encouragement, and love.

## **Publications during enrolment**

This PhD research has resulted in nine peer-reviewed publications. These publications include one journal article, and eight international conference papers.

### Journal

A. Hassani, A. Medvedev, P.D. Haghighi, S. Ling, A. Zaslavsky, & P. P. Jayaraman, “Context Definition and Query Language: Conceptual Specification, Implementation, and Evaluation,” *Sensors*, 19(6), 1478.

A. Hassani, A. Medvedev, P.D. Haghighi, S. Ling, A. Zaslavsky, & P. P. Jayaraman, P. “Execution of Context queries in dynamic IoT environments,” *IEEE Internet of Things (IoT) Journal*, 2019 [planned].

### International Conferences

#### 2019

A. Medvedev, A. Hassani, A. Zaslavsky, P. D. Haghighi, S. Ling, P. P. Jayaraman, and N. Kolbe, “Benchmarking IoT Context Management Platforms: High-level Queries Matter,” in *GIoTS 2019 - Global Internet of Things Summit, Proceedings*, 2019. [Accepted]

#### 2018

A. Hassani, P. D. Haghighi, P. P. Jayaraman, A. Zaslavsky, and S. Ling, “Querying IoT Services: A Smart Carpark Recommender Use Case,” in *4th IEEE World Forum on Internet of Things WF-IoT 2018*, 2018.

A. Hassani, A. Medvedev, A. Zaslavsky, P. Delir Haghighi, S. Ling, and M. Indrawan-Santiago, “Context-as-a-Service Platform: Exchange and Share Context in an IoT Ecosystem,” in *Percom 2018*, 2018.

A. Medvedev, A. Hassani, A. Zaslavsky, P. D. Haghighi, S. Ling, M. I. Santiago, P. P. Jayaraman, and N. Kolbe, “Situation Modelling , Representation , and Querying in Context-as-a-Service IoT Platform,” in *GIoTS 2018 - Global Internet of Things Summit, Proceedings*, 2018.

#### 2017

A. Medvedev, A. Hassani, P. Delir Haghighi, A. Zaslavsky, and P. P. Jayaraman, “Architecting IoT context storage management for context-as-a-service

platform,” in GIoTTS 2017 - Global Internet of Things Summit, Proceedings, 2017.

A. Medvedev, A. Hassani, A. Zaslavsky, P. P. Jayaraman, M. Indrawan-Santiago, P. D. Haghighi, and S. Ling, “Data ingestion and storage performance of IoT platforms: Study of OpenIoT,” in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2017, vol. 10218 LNCS, pp. 141–157.

## 2016

A. Hassani, A. Haghighi, P. D. Jayaraman, P. P. Zaslavsky, S. Ling, and A. Medvedev, “CDQL: A Generic Context Representation and Querying Approach for Internet of Things Applications,” 14th Int. Conf. Adv. Mob. Comput. Multimed., 2016.

A. Medvedev, A. Zaslavsky, M. I. Santiago, P. D. Haghighi, and A. Hassani, “Storing and indexing IoT context for smart city applications,” in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2016, vol. 9870 LNCS, pp. 115–128.

# Abstract

As Internet of Things (IoT) grows at a staggering pace, the need for contextual intelligence is a fundamental and critical factor for delivering IoT intelligence, efficiency, effectiveness, performance, and sustainability. Contextual intelligence enables intelligent interactions between IoT devices such as sensors/actuators, mobile smart phones, smart vehicles to name a few. Context management platforms (CMP) are emerging as a promising solution to deliver the contextual intelligence for IoT. However, a generic solution that allows IoT devices and services to publish, consume, monitor, and share context is still in its infancy.

In this dissertation, we propose, develop, implement, and evaluate a solution that enables IoT devices and services to seamlessly publish and query context. The first component of the solution is two novel languages, namely Context Service Description Language (CSDL) that facilitates publishing context by providing means to describe and register the IoT devices and services that produce context (i.e. context services); and Context Definition and Query Language (CDQL) that allows IoT devices and applications to query and consume the context data produced by context service. The second component of this solution includes two novel mechanisms, namely Context Query Engine (CQE) and Situation Monitoring Engine (SME). CQE is responsible for parsing incoming queries, generating and orchestrating the query execution plan, and producing the final query result. CQE has a sub-component called Context Service Discovery (CSD) which allows dynamic discovery of context services based on incoming queries. Situation Monitoring Engine (SME) enables the execution of complex context queries and monitoring context for IoT. SME is designed to support continuous monitoring of incoming context from IoT devices and services, infer situations from available context, detect changes in situations and provide notification of detected changes. The proposed solution facilitates the development of context-aware IoT applications by providing a generic yet tailorable mechanism to query and publish context.

We exemplify the usage of CDQL on three different smart city use-cases to highlight how the proposed solution can be utilised to deliver contextual intelligence to IoT devices and services. We have implemented and conducted extensive experimental validation of the proposed solutions. Performance evaluation has demonstrated and

validated the scalability and efficiency of the proposed solution against the current state-of-the-art in handling and servicing a significantly large number of concurrent context queries originating from IoT devices and services.

The outcomes of this dissertation have resulted in one journal article and nine international conference papers. Furthermore, the proposed solution has been integrated with a pioneering CMP called Context-as-a-Service (CoaaS). The proposed query language namely, CDQL is currently being considered by ETSI CIM group to be included as part of their specification for context information exchange in smart cities.

## List of Abbreviations

CASM	Communication and Security manager
CC	Context Consumer
CCM	Contextual Characteristics Matching
CDL	Context Definition Language
CDQL	Context Definition and Query Language
CEP	Complex Event Processing
CMP	Context Management Platform
CoaaS	Context as a Service
CoC	Cost of Context
CP	Context Provider
CQA	Context Query Aggregator
CQC	Context Query Coordinator
CQE	Context Query Engine
CQL	Context Query Language
CQP	Context Query parser
CR	Context Request
CRE	Context Reasoning Engine
CS	Context Service
CSC	Context Similarity Calculator
CSD	Context Service Discovery
CSDL	Context Service Description Language
CSI	Context Service Invoker
CSMS	Context Storage Management System
CSR	Context Service Repository

CST	Context Space Theory
NM	Notification Manager
PSM	Preliminary Service Matching
SDS	Situation Description Statement
SIM	Situation Inference Manager
SME	Situation Monitoring Engine
QoC	Quality of Context
SO	Situation Orchestrator

# Table of Contents

<b>Chapter 1:</b>	<b>Introduction .....</b>	<b>1</b>
1.2	<i>INTRODUCTION.....</i>	<i>1</i>
1.3	<i>MOTIVATING USE CASES .....</i>	<i>4</i>
1.3.1	USE CASE 1: SCHOOL SAFETY .....	4
1.3.2	USE CASE 2: SMART PARKING RECOMMENDER.....	5
1.3.3	USE CASE 3: VEHICLE PRE-CONDITIONING .....	5
1.3.4	SUMMARY OF USE-CASES .....	6
1.4	<i>RESEARCH AIM AND QUESTIONS.....</i>	<i>6</i>
1.5	<i>RESEARCH CONTRIBUTIONS AND IMPACT.....</i>	<i>7</i>
1.6	<i>DISSERTATION STRUCTURE.....</i>	<i>8</i>
<b>Chapter 2:</b>	<b>Literature review .....</b>	<b>9</b>
2.1	<i>CONTEXT .....</i>	<i>10</i>
2.1.1	DEFINITIONS OF CONTEXT .....	10
2.1.2	CHARACTERISTICS OF CONTEXT .....	12
2.1.3	DEFINITIONS OF CONTEXT-AWARENESS.....	14
2.1.4	CONTEXT MODELLING AND REPRESENTATION .....	15
2.2	<i>THE INTERNET OF THINGS PARADIGM .....</i>	<i>19</i>
2.2.1	WHAT IS THE INTERNET OF THINGS?.....	20
2.2.2	CHARACTERISTICS OF THE IOT .....	22
2.3	<i>CONTEXT IN IOT.....</i>	<i>24</i>
2.4	<i>CONTEXT MANAGEMENT PLATFORM (CMP) FOR IOT.....</i>	<i>25</i>
2.5	<i>CONTEXT SERVICE REGISTRATION AND DISCOVERY.....</i>	<i>27</i>
2.6	<i>CONTEXT QUERY LANGUAGES.....</i>	<i>28</i>
2.6.1	SQL-BASED .....	29
2.6.2	RDF-BASED.....	32
2.6.3	XML-BASED.....	33
2.6.4	API BASED CONTEXT QUERY / OR JSON BASED.....	34
2.6.5	DISCUSSION.....	35
2.7	<i>SUMMARY .....</i>	<i>37</i>
<b>Chapter 3:</b>	<b>Context Definition and Query Language .....</b>	<b>39</b>
3.1	<i>CONTEXT-AS-A-SERVICE DEFINITIONS AND BLUEPRINT ARCHITECTURE .....</i>	<i>40</i>
3.1.1	CONTEX-AS-A-SERVICE: OVERVIEW AND DEFINITIONS .....	40
3.1.2	COAAS PLATFORM BLUEPRINT ARCHITECTURE.....	45
3.2	<i>CONTEXT SERVICE DESCRIPTION AND CONTEXT QUERY LANGUAGE .....</i>	<i>48</i>
3.2.1	CONTEXT MODEL .....	49
3.3	<i>CONTEXT SERVICE DESCRIPTION LANGUAGE (CSDL).....</i>	<i>52</i>
3.4	<i>CONTEXT DEFINITION AND QUERY LANGUAGE (CDQL) .....</i>	<i>54</i>
3.4.1	CONTEXT QUERY LANGUAGE (CQL).....	54
3.4.2	CONTEXT DEFINITION LANGUAGE (CDL).....	67



3.5	<i>SUMMARY</i> .....	82
<b>Chapter 4:</b>	<b>Context Query and Situation Monitoring Engines: Design and Implementation</b> .....	<b>83</b>
4.1	<i>CONTEXT QUERY ENGINE</i> .....	83
4.2	<i>CONTEXT QUERY PARSER AND EXECUTION PLAN S GENERATION</i> .....	85
4.3	<i>CONTEXT QUERY COORDINATOR</i> .....	92
4.3.1	PULL-BASED CDQL QUERY .....	92
4.3.2	PUSH-BASED CDQL QUERY .....	98
4.4	<i>CONTEXT SERVICE DISCOVERY</i> .....	102
4.4.1	CONTEXT SIMILARITY CALCULATOR (CSC) .....	104
4.4.2	PRELIMINARY SERVICE MATCHING (PSM) .....	106
4.4.3	CONTEXTUAL CHARACTERISTICS MATCHING (CCM) .....	107
4.4.4	EXAMPLE .....	110
4.5	<i>SITUATION MONITORING ENGINE (SME)</i> .....	116
4.6	<i>IMPLEMENTATION</i> .....	120
4.7	<i>SUMMARY</i> .....	129
<b>Chapter 5:</b>	<b>CDQL, CQE and SME: Evaluations</b> .....	<b>131</b>
5.1	<i>CDQL QUERY DEMONSTRATION</i> .....	131
5.1.1	USE CASE 1: SCHOOL SAFETY .....	132
5.1.2	USE CASE 2: SMART PARKING RECOMMENDER .....	133
5.1.3	USE CASE 3: VEHICLE PRECONDITIONING .....	140
5.2	<i>COMPARISON OF CDQL WITH NGSI</i> .....	143
5.3	<i>PERFORMANCE EVALUATION</i> .....	147
5.3.1	EXPERIMENT ENVIRONMENT AND METRICS .....	147
5.3.2	EXPERIMENT 1: CONTEXT QUERY ENGINE - PULL-BASED QUERIES .....	150
5.3.3	EXPERIMENT 2: SITUATION MONITORING ENGINE .....	154
5.3.4	<i>EVALUATION OF CONTEXT SERVICE DISCOVERY ALGORITHM</i> .....	158
5.4	<i>SUMMARY</i> .....	161
<b>Chapter 6:</b>	<b>Conclusion</b> .....	<b>162</b>
6.1	<i>SUMMARY OF CONTRIBUTIONS</i> .....	162
6.2	<i>LIMITATIONS AND FURTHER RESEARCH</i> .....	165
	<b>Glossary</b> .....	<b>167</b>
	<b>References</b> .....	<b>168</b>
	<b>Appendices</b> .....	<b>183</b>

## List of Figures

<b>FIGURE 1.1 - SCHOOL SAFETY USE-CASE .....</b>	<b>4</b>
<b>FIGURE 1.2 - DISSERTATION STRUCTURE.....</b>	<b>8</b>
<b>FIGURE 2.1 - STRUCTURE OF CHAPTER 2 .....</b>	<b>9</b>
<b>FIGURE 3.1 - OVERVIEW OF CONTEXT-AS-A-SERVICE PLATFORM IN IoT .....</b>	<b>41</b>
<b>FIGURE 3.2 - COAAS BLUEPRINT ARCHITECTURE .....</b>	<b>45</b>
<b>FIGURE 3.3 - ENTITY DATA MODEL .....</b>	<b>50</b>
<b>FIGURE 3.4 - STRUCTURE OF CSDL .....</b>	<b>53</b>
<b>FIGURE 3.5 - AN EXAMPLE OF SERVICE DESCRIPTION IN CSDL .....</b>	<b>54</b>
<b>FIGURE 3.6 - CQL PRODUCTION RULE .....</b>	<b>55</b>
<b>FIGURE 3.7 - PREFIX CLAUSE PRODUCTION RULE .....</b>	<b>55</b>
<b>FIGURE 3.8 - SELECT CLAUSE PRODUCTION RULE .....</b>	<b>56</b>
<b>FIGURE 3.9 - DEFINE CLAUSE PRODUCTION RULE.....</b>	<b>57</b>
<b>FIGURE 3.10 - CONDITION CLAUSE PRODUCTION RULE .....</b>	<b>58</b>
<b>FIGURE 3.11 - SORT-BY CLAUSE PRODUCTION RULE.....</b>	<b>59</b>
<b>FIGURE 3.12 - SUBSCRIPTION CLAUSE PRODUCTION RULE .....</b>	<b>62</b>
<b>FIGURE 3.13 - SET CLAUSE PRODUCTION RULE.....</b>	<b>64</b>
<b>FIGURE 3.14 - OUTPUT-CONFIG CLAUSE PRODUCTION RULE.....</b>	<b>66</b>
<b>FIGURE 3.15 - CDL PRODUCTION RULE .....</b>	<b>69</b>
<b>FIGURE 3.16 - CREATE FUNCTION PRODUCTION RULE.....</b>	<b>70</b>
<b>FIGURE 3.17 - API-BASED AGGREGATION FUNCTIONS .....</b>	<b>72</b>
<b>FIGURE 3.18 - CST-SITUATION STATEMENT PRODUCTION RULE .....</b>	<b>76</b>
<b>FIGURE 3.19 - CST-ATTRIBUTE-DEFINITION.....</b>	<b>77</b>
<b>FIGURE 3.20 - HIGH-LEVEL-SITUATION STATEMENT PRODUCTION RULE .....</b>	<b>79</b>
<b>FIGURE 3.21 - ALLEN’S ALGEBRA GRAPHICAL REPRESENTATION (I STANDS FOR INVERSE) ..</b>	<b>80</b>
<b>FIGURE 3.22 - ISVALID FUNCTION PRODUCTION RULE .....</b>	<b>81</b>
<b>FIGURE 4.1 - CONTEXT QUERY ENGINE ARCHITECTURE.....</b>	<b>84</b>
<b>FIGURE 4.2 - CDQL EXECUTION PLAN GENERATOR.....</b>	<b>87</b>
<b>FIGURE 4.3 - QUERY EXECUTION PLAN GRAPH.....</b>	<b>91</b>
<b>FIGURE 4.4 - PUSH-BASED CDQL EXECUTION WORKFLOW .....</b>	<b>93</b>
<b>FIGURE 4.5 - PUSH-BASED CDQL EXECUTION WORKFLOW .....</b>	<b>99</b>
<b>FIGURE 4.6 - AN EXAMPLE OF SUBSCRIPTION DATA MODEL.....</b>	<b>100</b>
<b>FIGURE 4.7 - CONTEXT SERVICE DISCOVERY ARCHITECTURE .....</b>	<b>103</b>
<b>FIGURE 4.8 - SEMANTIC HIERARCHY EXAMPLE BASED ON SCHEMA.ORG .....</b>	<b>105</b>
<b>FIGURE 4.9 - CONTEXTUAL CHARACTERISTICS MATCHMAKING ALGORITHM .....</b>	<b>109</b>
<b>FIGURE 4.10 - VISUALISATION OF EXAMPLE FOR CONTEXT SERVICE DISCOVERY PROCESS .</b>	<b>109</b>

<b>FIGURE 4.11 - SITUATION MONITORING ENGINE.....</b>	<b>116</b>
<b>FIGURE 4.12 - SME WORKFLOW .....</b>	<b>117</b>
<b>FIGURE 4.13 - SITUATION INFERENCE ALGORITHM.....</b>	<b>119</b>
<b>FIGURE 4.14 - ARCHITECTURE OF PROTOTYPE IMPLEMENTATION OF COAAS PLATFORM ...</b>	<b>120</b>
<b>FIGURE 4.15 - AUTHENTICATION AND AUTHORISATION MECHANISM.....</b>	<b>121</b>
<b>FIGURE 4.16 - COAAS IDE .....</b>	<b>127</b>
<b>FIGURE 4.17 - SITUATION MONITORING INTERFACE .....</b>	<b>127</b>
<b>FIGURE 4.18 - NODE-RED BASED EXAMPLE .....</b>	<b>128</b>
<b>FIGURE 5.1 - SMART PARKING SUGGESTION APPLICATION SCREENSHOT .....</b>	<b>134</b>
<b>FIGURE 5.2 - PoC PARKING APPLICATION SCREENSHOT .....</b>	<b>137</b>
<b>FIGURE 5.3 - EXECUTION AND INTERACTION PROCESS OF SMART PARKING SUGGESTION ..</b>	<b>138</b>
<b>FIGURE 5.4 - PRE-CONDITIONING SCENARIO WORKFLOW.....</b>	<b>140</b>
<b>FIGURE 5.5 - ACTIVATION OF BMW I3 CLIMATE CONTROL SYSTEM.....</b>	<b>141</b>
<b>FIGURE 5.6 - QUERY RESPONSE TIME VS INPUT RATE .....</b>	<b>153</b>
<b>FIGURE 5.7 - QUERY RESPONSE TIME VS NUMBER OF REGISTERED ENTITIES .....</b>	<b>154</b>
<b>FIGURE 5.8 - THROUGHPUT VS INPUT RATE .....</b>	<b>156</b>
<b>FIGURE 5.9 - RESOURCE UTILIZATION.....</b>	<b>156</b>
<b>FIGURE 5.10 - PROCESSING TIME VS THROUGHPUT .....</b>	<b>157</b>
<b>FIGURE 5.11 - PROCESSING TIME VS NUMBER OF SUBSCRIPTIONS.....</b>	<b>157</b>
<b>FIGURE 5.12 - CPU UTILISATION VS NUMBER OF SUBSCRIPTIONS.....</b>	<b>158</b>
<b>FIGURE 5.13 - PERFORMANCE OF CSD .....</b>	<b>159</b>
<b>FIGURE 5.14 - PRECISION VS RECALL OF CSD.....</b>	<b>160</b>

## List of Tables

<b>TABLE 2.1</b> - IoT DEFINITIONS .....	21
<b>TABLE 2.3</b> - EVALUATION OF EXISTING CQLS.....	36
<b>TABLE 3.1</b> - COAAS MAJOR COMPONENTS .....	46
<b>TABLE 3.2</b> - CONTEXT METADATA .....	51
<b>TABLE 3.3</b> - CQL BUILT-IN FUNCTIONS.....	67
<b>TABLE 4.1</b> - RPNCONDITION REFORMULATION STRATEGIES.....	95
<b>TABLE 4.2</b> - REGISTERED PARKING FACILITIES' CONTEXT SERVICES .....	111
<b>TABLE 4.3</b> - RESULT OF PSM.....	112
<b>TABLE 4.4</b> - ASSIGNED IDS FOR EACH PREDICATE.....	113
<b>TABLE 4.5</b> - OUTCOME OF CCM FOR THE FIRST CONJUNCTION.....	115
<b>TABLE 4.6</b> - OUTCOME OF CCM FOR THE SECOND CONJUNCTION IN .....	115
<b>TABLE 4.7</b> - COAAS INTERFACE ENDPOINTS .....	122
<b>TABLE 5.1</b> - CDQL QUERIES FOR PERFORMANCE EVALUATION .....	151

# List of Code Blocks

<b>CODE BLOCK 2.1</b> - A BASIC CONDITION EXPRESSED IN SPARQL .....	32
<b>CODE BLOCK 3.1</b> - EXAMPLE OF PREFIX CLAUSE .....	56
<b>CODE BLOCK 3.2</b> - EXAMPLE OF SELECT CLAUSE .....	57
<b>CODE BLOCK 3.3</b> - EXAMPLE OF DEFINE CLAUSE.....	60
<b>CODE BLOCK 3.4</b> - EXAMPLE OF A PULL-BASED QUERY .....	61
<b>CODE BLOCK 3.5</b> - EXAMPLE OF A BASIC PUSH-BASED .....	63
<b>CODE BLOCK 3.6</b> - EXAMPLE OF USING WHEN CLAUSE IN A CDQL QUERY .....	63
<b>CODE BLOCK 3.7</b> - EXAMPLE OF PUSH-BASED QUERY WITH CALLBACK CLAUSE.....	65
<b>CODE BLOCK 3.8</b> - EXAMPLE OF QUERYING THE RESULTS OF SUBSCRIPTIONS .....	65
<b>CODE BLOCK 3.9</b> - EXAMPLE OF META CLAUSE.....	66
<b>CODE BLOCK 3.10</b> - EXAMPLE OF CREATE-FUNCTION CLAUSE.....	72
<b>CODE BLOCK 3.11</b> - EXAMPLE OF CREATING A CUSTOM AGGREGATION FUNCTION USING JAVASCRIPT.....	73
<b>CODE BLOCK 3.12</b> - EXAMPLE OF CST-BASED SITUATION FUNCTION DEFINITION .....	78
<b>CODE BLOCK 3.13</b> - EXAMPLE OF ISVALID FUNCTION .....	81
<b>CODE BLOCK 4.1</b> - CDQL FOR FINDING TRAFFIC INCIDENT NEAR A SPECIFIC VEHICLE.....	86
<b>CODE BLOCK 4.2</b> - AN EXAMPLE OF PARSED CDQL QUERY .....	89
<b>CODE BLOCK 4.3</b> - EXTENDED PARKING AND TRAFFIC ELEMENTS QUERY .....	90
<b>CODE BLOCK 4.4</b> - CDQL QUERY FOR FINDING VEHICLES DRIVING FASTER THAN 60 KM/H NEAR A SCHOOL IN MELBOURNE.....	96
<b>CODE BLOCK 4.5</b> - REFORMULATED WHERE CLAUSE.....	97
<b>CODE BLOCK 4.6</b> - AN EXAMPLE OF SIDDHI APPLICATION GENERATED BY THE CQC .....	101
<b>CODE BLOCK 4.7</b> - CDQL QUERY FOR FINDING AVAILABLE PARKING OPTIONS .....	110
<b>CODE BLOCK 4.8</b> - EXAMPLE OF AUTHENTICATION REQUEST .....	122
<b>CODE BLOCK 4.9</b> - EXAMPLE OF ISSUING CDQL QUERY .....	123
<b>CODE BLOCK 4.10</b> - EXAMPLE OF SENDING CONTEXT UPDATE .....	124
<b>CODE BLOCK 5.1</b> - CDQL QUERY FOR SCHOOL SAFETY USE-CASE. ....	133
<b>CODE BLOCK 5.2</b> - CDQL QUERY FOR SMART PARKING RECOMMENDER USE-CASE.....	135
<b>CODE BLOCK 5.3</b> -CDQL QUERY FOR VEHICLE PRECONDITIONING USE-CASE.....	142
<b>CODE BLOCK 5.4</b> - NGSi QUERIES FOR SMART PARKING RECOMMENDER USE-CASE.....	144
<b>CODE BLOCK 5.5</b> - NGSi SUBSCRIPTION.....	145



# Chapter 1: Introduction

---

## 1.2 INTRODUCTION

Nowadays, the advancements in hardware and software technologies have made it possible to embed sensing, computation, and communication capabilities in everyday objects, from a coffee mug to an autonomous car, and turn them into smart connected devices. These devices can form a worldwide network of interconnected objects, where each device can collect and distribute enormous amounts of data about its environment. This network is known as the Internet of Things (IoT). IoT is a fast-evolving trend and expected overall spending on IoT will reach US \$1.3 trillion by 2020 from US \$696 billion in 2015 (Meulen, 2017).

Due to the proliferation of smart connected devices (known as IoT devices or IoT things), which is expected to reach 20 to 30 billion in 2020 (Meulen, 2017), it is possible to build services that can share rich, useful and relevant information to users about an entity of interest (e.g. the environment, a car, a building to name a few). These services, which are referred to as IoT services, enable that development of many applications in various domains, such as smart cities, smart environment, smart agriculture, and eHealth.

A key requirement for IoT to be able to deliver the smartness is the ability to extract context from the data produced by IoT devices. Context as defined by Dey, is “any information that can be used to characterise the situation of an entity, where an entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves” (Dey, 2001, p. 5). The greater benefit is in being able to share this context extracted/reasoned from data produced by the IoT devices with other IoT applications that can use this context to support decision making, actuation, analysis etc. For example, consider a smart home scenario where a smart washing machine is tasked to wash a piece of clothing tagged with information (e.g. using RFID) regarding fabric care instructions. Using this information, the smart washing machine can automatically choose the right setting for washing the clothes. Moreover, this information can be used by a smart tumble dryer to decide what temperature and Revolution Per Minute (RPM) should be used for

drying the clothes. Assuming the delicate clothing material is not suitable for tumble drying, without context, the smart dryer will dry the clothes unaware of this fact. Augmenting IoT application with context that stem from IoT devices will enable the application (e.g. an application running on the smart dryer) to reason about the data and arrive at the right decision, in this case, not to tumble dry the delicate clothes.

Such IoT applications that utilise context data and adapt their behaviours accordingly are known as context-aware IoT applications. Context-awareness enables intelligent adaptation of IoT applications such that they can perform their tasks in an efficient, proactive and autonomous manner (Perera, Zaslavsky, Christen, & Georgakopoulos, 2014). Further, context can have different levels of abstraction. Context can be low-level information such as a Celsius temperature value of 35 or high-level context, which is inferred from low-level context such as ‘a fire threat’. High-level context is also known as ‘situation’. While context-driven intelligence is a fundamental factor for IoT sustainability, growth, interoperability and acceptance, IoT’s characteristics, such as scalability, big data, heterogeneity and dynamism, will make the development of context-aware IoT applications and services a very challenging task.

In general, three typical approaches exist for the development of context-aware applications (Li, Eckert, Martinez, & Rubio, 2015). In the first approach, context-aware applications acquire, process and use their context of interest themselves. In the second approach, context-aware applications are developed by using some libraries/toolkits that facilitates obtaining and processing context. In the third approach, the context-aware applications are developed on the basis of context-aware middleware that enables context management (i.e. acquire, process, store, and publish). The third approach, which is referred to as Context Management Platform (CMP), is superior to the first and second approaches as it can reduce the complexity of developing context-aware IoT applications (Li et al., 2015).

A fundamental requirement of a context management platform is to be able to provide support for publishing, querying, monitoring, and sharing contextual information. Such a platform will manage interaction between sources of context; in our case context provided and reasoned from IoT devices, and offers contextual information to context-aware IoT applications. A notable number of CMPs have been



proposed; surveys of which have been published for instance in (Baldauf, Dustdar, & Rosenberg, 2007; Hong, Suh, & Kim, 2009; Knappmeyer, Kiani, Reetz, Baker, & Tonjes, 2013; Truong & Dustdar, 2009). However, the existing CMPs suffer from one common shortcoming, which is the lack of a generic and expressive interface that allows IoT devices, applications, and services to publish, consume, monitor, and share context data seamlessly.

In this dissertation, we propose, develop, implement, and evaluate a comprehensive solution for publishing, querying, monitoring, and sharing context. The proposed solution will facilitate the development of smarter and context-aware IoT applications. The proposed solution consists of two specially designed languages, namely Context Service Description Language (CSDL) that facilitates publishing context by providing the means to describe and register the IoT devices and services that produce context (i.e. context services); and Context Definition and Query Language (CDQL) that allows IoT devices and applications to query, monitor, and consume the context data produced by IoT devices and services.

Based on the aforementioned languages, we propose, develop, and implement two engines, namely Context Query Engine (CQE) and Situation Monitoring Engine (SME), that enable execution of complex context queries and monitoring context in IoT ecosystem. CQE is mainly responsible for parsing the incoming queries, generating and orchestrating the query execution plan, and producing the final query result. Furthermore, CQE has a sub-component called Context Service Discovery (CSD) which allows dynamic discovery of context services based on incoming queries. The Situation Monitoring Engine (SME) is designed to support continuous monitoring of incoming context, infer situations from available context, detect changes in situations and provide notification of detected changes.

The solution proposed in this dissertation has been integrated and is an underpinning component of a pioneering CMP called Context-as-a-Service (CoaaS). CoaaS is part of EU Horizon-2020 project called bIoTope<sup>1</sup> – Building IoT OPEN Innovation Ecosystem for connected smart objects.

---

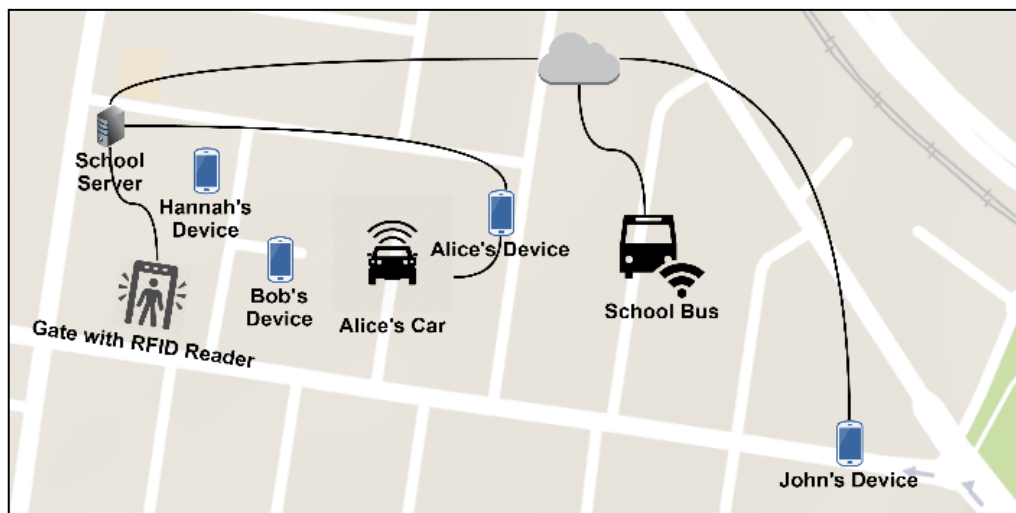
<sup>1</sup> [www.biotope-h2020.eu](http://www.biotope-h2020.eu)

### 1.3 MOTIVATING USE CASES

In this section, we present three motivating smart city use cases that highlight the need for a solution to publish, monitor, and query context in IoT environment.

#### 1.3.1 USE CASE 1: SCHOOL SAFETY

The first use case is called school safety and is depicted in **Figure 1.1**. Consider a user John who wants to pick up his daughter, Hannah, from school. On his way to school, due to unexpected traffic, he realises that he cannot arrive at the school on time. Realising this, a smart IoT system begins to determine alternatives to achieve the goal “pick up Hannah”.



**Figure 1.1** - School safety use-case

An option could be to request another trusted parent to pick up Hannah from school on John's behalf. In order to represent this context request, several factors should be considered, namely:

- The selected parent(s) for picking up Hannah should be trusted by John;
- The selected parent(s) should have a car with an extra seat for Hanna;
- The selected parent(s) should be close enough to the school;
- The child of the selected parent(s) should finish school at the same time as Hannah;
- The child of the selected parent(s) should be currently at school.

- Additionally, this process needs to be automated, so John’s device can automatically trigger the same query, “pick up Hannah” whenever he is running late.

### **1.3.2 USE CASE 2: SMART PARKING RECOMMENDER**

The second use case we consider in this paper focuses on facilitating the development of a context-aware IoT application that suggests parking facilities to drivers. Such an application needs to: 1) have access to live data regarding the availability of different parking facilities owned by different providers (e.g., city administrators, building owners, and organizations), 2) provide personalised recommendations to users, considering factors such as user preferences, car specifications, and related environmental conditions such as weather and 3) continuously monitor relevant context and notify the driver about any changes in situations that can affect his/her experience, e.g. notify the driver if the suggested parking becomes unavailable or another parking place with better conditions (such as cheaper or closer to the destination) becomes available.

### **1.3.3 USE CASE 3: VEHICLE PRE-CONDITIONING**

The third use case under consideration is a smart connected electric vehicle pre-conditioning use-case. Pre-conditioning allows the drivers to begin their journey with a properly heated or cooled cabin. The pre-conditioning use case requires continuous monitoring of several situations (computed from context of various IoT smart things and applications) such as the car’s location (provided by the connected car), the driver’s location, the driver’s calendar (provided by the driver’s smart mobile device), and weather conditions (obtained from nearby IoT weather stations) to name a few. Moreover, such a use-case also requires specific reasoning to infer the likelihood of the driver commencing a journey, e.g. walking past the car is different from walking towards the car to begin a journey. Finally, based on inferred situations, an actuation signal to start the pre-conditioning process will be sent to the car’s onboard computer.

### **1.3.4 SUMMARY OF USE-CASES**

Developing context-aware IoT applications for the abovementioned use cases, which utilise context to provide better services to the end users, is a complicated task. This complexity is formed by the need to discover heterogeneous sources of context (silos) that can provide data about the entities of interest for each use case. Moreover, the raw data produced by these sources will not be of any use unless it is analysed and interpreted. For example, in order to implement an application for the smart parking recommender use case, several challenges need to be addressed. First of all, it is essential to have access to live data regarding the availability of different parking facilities. The fact that these facilities are owned by different providers (e.g. city administrators, building owners, and organizations) makes the process of data retrieval even more complex. Further, to be able to provide personalised suggestions to the users, it is necessary to consider additional factors, such as user preference, user calendar, car specifications, and weather condition. Moreover, some of this data needs to be reasoned before being used. Lastly, the application should be capable of processing streams of data in order to continuously monitor relevant context to this use case (i.e. the suggested parking becomes unavailable). Addressing all of these challenges needs a considerable amount of effort even for an expert team of software developers.

One possible solution for tackling these challenges is to develop a context management platform that enables applications to publish and consume context about their entities of interest seamlessly, without requiring manual integration of IoT silos. However, since all these use cases have different requirements, it is essential for a context management platform to support an expressive language that makes it possible to query context according to the needs of a consumer. As a result, utilising such a platform can free developers from the concern of managing context and allow them to focus on designing desired application functions and business logic.

## **1.4 RESEARCH AIM AND QUESTIONS**

The aim of this dissertation is to investigate, propose, design, implement and validate a generic approach to define, represent, monitor, and query context. We have formulated the following research question to address this aim:

RQ1- How can context be shared, exchanged, monitored, and queried by a feature-rich query language in IoT environments?

In order to address the research question (RQ1), the following sub-questions need to be addressed.

- RQ1.1- What formal methods can be used to represent, model and reason about context?
- RQ1.2- How can context queries and services be defined and represented in formal language constructs?
- RQ1.3- How can IoT entities (context consumers and providers) communicate to advertise, monitor, discover and invoke context?

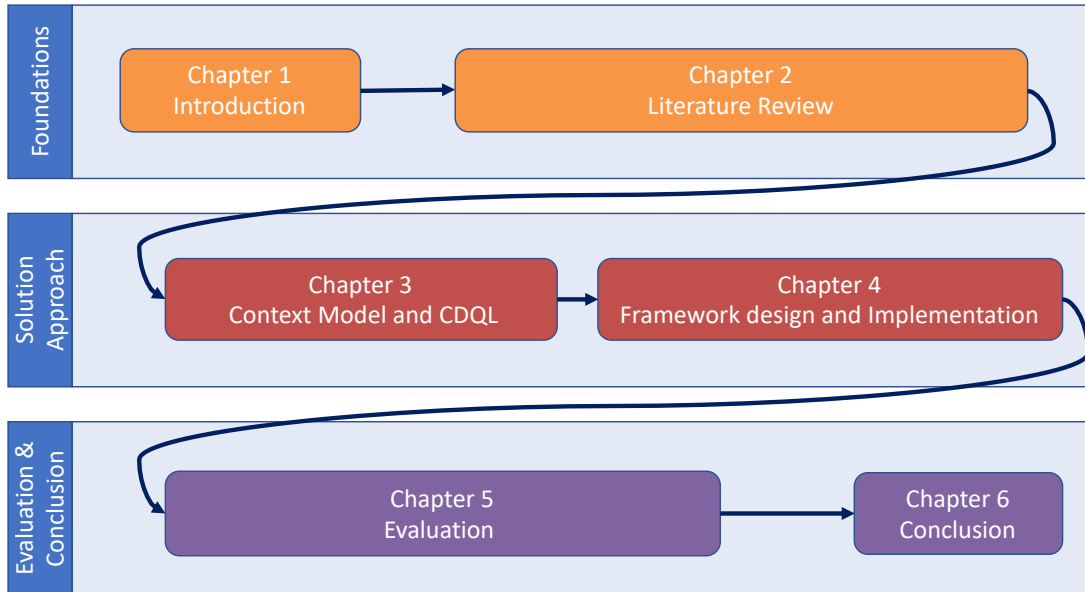
## **1.5 RESEARCH CONTRIBUTIONS AND IMPACT**

This section lists the main contributions and impact of this dissertation to the current body of knowledge. We have:

- Designed and developed a novel language for describing and registering context services.
- Designed and developed a novel context query language, which is under review by ETSI CIM group as complementary to its current proposed draft of NGSI-LD especially in addressing high-level context- and situation-awareness.
- Designed and developed a mechanism (i.e. Context Query Engine) that allows execution of complex context queries.
- Designed and developed a service discovery technique that can discover eligible context services based on the query requirements.
- Designed and developed a situation monitoring engine that supports continuous monitoring of incoming context, infers situations from available context using a well-established situation inference method, detects changes in situations and provide notification of detected changes.
- Implemented a prototype of the proposed solution and validated it by conducting a comprehensive evaluation, using real-world and synthetic datasets.

## 1.6 DISSERTATION STRUCTURE

The dissertation is arranged in succession in terms of background, theoretical contributions, architectural approach, evaluation, and conclusion. It progressively presents different facets of our proposed context service description and query language and builds upon these as the basis for developing a context management platform. Roadmap for the dissertation layout is presented in Figure 1.2. The dissertation comprises a background chapter (Chapter 2) followed by two theoretical chapters (Chapters 3 and 4), presenting the research theoretical contributions. Moreover, Chapter 4 also presents the design and implementation of the proposed solution. The case-studies and evaluation of the proposed solution are presented in Chapter 5. Lastly, Chapter 6 concludes the entire dissertation.



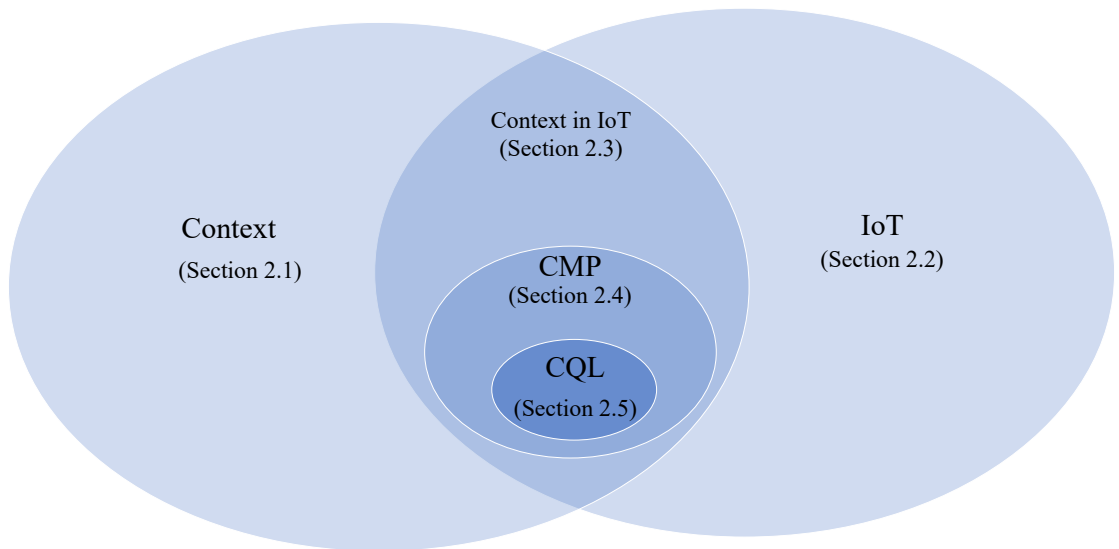
**Figure 1.2** - Dissertation structure

## Chapter 2: Literature review

---

Chapter 1 described the research problem under consideration in this PhD dissertation. It presented an overview of the structure of this dissertation and described the research aims and contributions. This chapter presents a review of the relevant literature on context management and provisioning for the IoT ecosystem and identifies gaps in the body of knowledge.

We first provide a background on context-aware computing that includes definitions of context, and context modelling. Then we introduce IoT and briefly describe and explore the importance of context in IoT. We explain the main challenges that need to be addressed in order to successfully utilise the context produced by IoT in context-aware IoT applications. We provide a background on context management and provisioning platforms (CMP) for IoT with specific focus on their ability to query contextual information. On this basis, we provide detailed descriptions of current state-of-the-art context query languages. We identify six main requirements for a context query language (CQL) that considers the characteristics of context-aware IoT applications (such as the motivating use-cases presented in Chapter 1). Finally, we conduct a comparative analysis of current state-of-the-art CQLs based on the identified requirements. The structure of this chapter is represented in Figure 2.1.



**Figure 2.1** - Structure of Chapter 2

## **2.1 CONTEXT**

### **2.1.1 DEFINITIONS OF CONTEXT**

The term context (from Latin *contextus*, from *con-* together + *texere* to weave.) is defined in the Oxford dictionary as “The circumstances that form the setting for an event, statement, or idea, and in terms of which it can be fully understood”. While this definition is understandable for most people, it is not clear enough to be used as a formal definition. Therefore, a considerable number of attempts have been made by many researchers to develop a generic and standard definition for the term context. In this section, in order to find a formal definition that meet the requirements of this research, we will look at the existing context definitions used in the literature. These works can be classified into three main categories: defining context by example, defining context by synonyms, and defining context by concepts. The latter is a more formal approach and concentrates on the relationships and structure of contextual information (Kofod-petersen & Mikalsen, 2005).

#### **Defining Context by Example**

In general, this category of context definitions refers to those works that try to determine context by using examples. In the rest of this section, an overview of some of the well-known context definitions that fall into this category is provided.

The term context was introduced for the first time by Theimer and Schilit (1994). They define context as ‘where you are, who you are with, and what resources are nearby’. In this definition, location is considered as the core element of the context. However, Theimer and Schilit (1994) partially include contextual information about nearby people and objects in their definition as well. Abowd and Mynatt (2000) also proposed a similar definition and identified the five W’s (Who, What, Where, When, Why) as the minimum information that is necessary to understand context.

These definitions that define context by example are hard to use. The main shortcoming of this type of context definition is their inability to determine whether a potential new type of information is context or not. For example, none of the previous definitions helps decide whether a user’s preferences or interests are context information or not.



## **Defining Context by Synonyms**

Another sub-class of context definition describes context by simply providing synonyms for context, referring to context as the environment or situation (Brown, Bovey, & Chen, 1997; Franklin & Flachsbart, 1998; Hull, Neaves, & Bedford-Roberts, 1997; Rodden, Cheverst, Davies, & Dix, 1998; Ryan, Pascoe, & Morse, 1999; Ward, Jones, & Hopper, 1997).

Brown et al. (1997) defined context as location, identity of nearby people, and time of day. Ryan et al. (1999) reported on a fieldwork where they viewed context as location, environment, identity, and time. Franklin and Flaschbart (1998) saw it as the situation of the user. Ward et al. (1997) viewed context as the state of the application's surroundings and Rodden et al. (1998) defined it as the application's setting. Hull et al. (1997) included the entire environment by defining context to be aspects of the current situation. These definitions are clearly more general than enumerations, but this generality is also a limitation. These definitions provide little guidance to analyse the constituent elements of context, much less identify them. Furthermore, these definitions are also inadequate to identify new context (Dey, 2001).

## **Defining Context by Concepts**

Some other researchers try to formally define context. Schmidt et al. (1999) defined context as “knowledge about the user's and IT device's state, including surroundings, situation, and to a less extent, location” (p. 90). Another formal definition is provided by Chen and Kotz (2000). They defined context as “set of environmental states and settings that either determines an application's behaviour or in which an application event occurs and is interesting to the user.” (G. Chen & Kotz, 2000, p. 3)

Dey (2001) defines context as “any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves” (Dey, 2001, p. 5). We adopt this definition and define context as follows:

**Definition 2.1 (Context).** Context is the information that can be used to characterise the state of an entity. Entities are persons, locations, or objects that affects the behaviour of an application.

### 2.1.2 CHARACTERISTICS OF CONTEXT

Context information has a set of unique characteristics that makes it different from raw data. In this section, we review the existing characteristics defined for the context information in the literature.

Henricksen, Indulska, and Rakotonirainy (2002) considered four characteristics for context information which are listed below:

- “*Context Information Exhibits a Range of Temporal Characteristics.*” Contextual information can be classified into two groups, static context and dynamic context. The static context is referred to context information that is invariant, such as date of birth of a person. However, since pervasive systems are typically characterised by frequent changes, most of the context information falls into the second category, dynamic context. The change frequency of dynamic context information does not follow a fixed pattern. For instance, the occupation of a person typically remains unchanged for years, while a person’s location and activity often change from one minute to the next.
- “*Context Information is Imperfect.*” Contextual information might be incorrect (reflecting the wrong state of the world), inconsistent (containing contradictory information), or incomplete (missing some aspects of the context).
- “*Context has Many Alternative Representations.*” In other words, different applications are interested in different aspects of the same contextual value based on their requirements. For example, a location sensor may supply raw coordinates, whereas one application might be interested in the identity of the building, and the other application might be interested in the identity of the suburb. Therefore, a context model must support multiple representations of the same context in different forms and at different levels of abstraction, and must also be able to capture the relationships that exist between alternative representations.
- “Context Information is Highly Interrelated.”

Another important aspect of context information is Quality of Context (QoC). As Henricksen et al. (2002) discussed, context information is imperfect. Krause and Hochstatter (2005) stated some of the main reasons for unreliable or error-prone context information:

- Unavailability of required context information (or context sources).
- Out-dated context information which is no longer reflecting the correct state of the world.
- Inaccurate and malfunctioning sensors due to physical constraints.
- Possible issues in the inference and reasoning mechanism.
- Existence of malicious external context sources which can provide wrong context that is not real.

Due to these reasons, QoC plays a vital role in context-aware systems. Buchholz, Küpper, and Schiffers (2003) present a set of parameters to determine QoC. These parameters are precision, probability of correctness, trust-worthiness, resolution and up-to-dateness. They also proposed a formal definition for QoC:

*“Quality of Context (QoC) is any information that describes the quality of information that is used as context information. Thus, QoC refers to information and not to the process nor the hardware component that possibly provide the information.”* (Buchholz et al., 2003, p. 5).

Some other researchers also provide their own definitions of QoC and identify a set of parameters to determine the quality of context (Manzoor, Truong, & Dustdar, 2008; Sheikh, Wegdam, & Van Sinderen, 2007). Since analysing all of these works is out of scope for this dissertation, we only provide the key QoC parameters discussed in these papers and their definitions.

- Precision describes how exactly the provided context information mirrors reality (Buchholz et al., 2003).
- Probability of correctness is defined as the probability that an instance of context accurately represents the corresponding real world situation, as assessed by the context source, at the time it was determined (Sheikh et al., 2007).

- Trust-worthiness describes how likely it is that the provided information is correct (Buchholz et al., 2003).
- Resolution denotes the granularity of information (Buchholz et al., 2003).
- Up-to-datedness/freshness describes the age of context information (Buchholz et al., 2003). This parameter can be used to identify the degree of rationalism to use a context object for a specific application at a given time (Manzoor et al., 2008).
- Temporal resolution determines the period to which a single instance of context information is applicable (Sheikh et al., 2007).

The last important aspect of context we want to mention in this section is Cost of Context (CoC). CoC can be defined as the cost for acquiring the context information. This does not necessarily need to be monetary but can also be interpreted as for example power consumption of the sensors for acquiring the information. Villalonga et al. (2009) define Cost of Context (CoC) as a parameter associated to the context that indicates the resource consumption used to measure or calculate the piece of context information.” (Villalonga, Roggen, Lombriser, Zappi, & Tröster, 2009).

### **2.1.3 DEFINITIONS OF CONTEXT-AWARENESS**

As we mentioned earlier, the term “*context-aware*” was introduced for the first time by Theimer and Schilit (1994). Based on their definition, a software is context-aware if it “adapts according to the location of the user, the collection of the nearby people, hosts, and accessible devices, as well as to changes to such things over time.” (Theimer & Schilit, 1994, p. 22). Later, a similar definition was stated by Ryan et al. (1999).

Abowd et al. (1999) showed that those definitions are too specific to be used as yardsticks to identify whether a given application is context-aware or not. Therefore, to solve this problem, Abowd et al. (1999) provide their definition of context-awareness as follows: “A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user’s task.” (Abowd et al., 1999).

Later, Chen, Finin, and Joshi (2004) defined context-awareness as “a computer system’s ability to provide relevant services and information to users based on their situational conditions.” (p. 1) .

The last two definitions, proposed by Abowd et al. (1999) and Chen et al. (2004), focus on the provisioning of information and/or services to the user. However, some other researchers have another point of view and proposed a more general definition for context-awareness. For instance, in the work done by Razzaque, Dobson and Nixon (2005), context-awareness is defined as “a term from computer science, which is used for devices that have information about the circumstances under which they operate and can react accordingly” (p. 2).

Becker and Nicklas (2004) state that “an application is context-aware if it adapts its behaviour depending on the context” (p. 2). Baldauf et al. (2007) introduced a new aspect of context-aware systems (i.e. self-adaptiveness) and defined it as a system that is “able to adapt their operations to the current context without explicit user intervention and thus aim at increasing usability and effectiveness by taking environmental context into account.” (p. 263). Similarly, Huebscher and McCann (2004) defined context-awareness as “the ability of an application to adapt itself to the context of its user(s).” (p. 111). Both definitions mentioned above highlight the context of the user.

In our opinion, both – the context of the user of the application and the context of the application itself – are important. The following definition will be used in this dissertation:

**Definition 2.2 (Context-Awareness).** An application is context-aware, if it adapts its behaviour to the context of itself, its users, or its surrounding environment.

## **2.1.4 CONTEXT MODELLING AND REPRESENTATION**

The main motivation behind this research work is to enable IoT entities (e.g. machines and smart devices) to share and exchange their context with each other. To achieve this goal (context sharing and interoperability between different context-aware applications), it is essential to have a uniform and machine understandable representation scheme for context information. Otherwise, it is not possible for different systems to communicate with each other without a common understanding among

them. We refer to this common understanding as a Context Model. It is essential to have a sophisticated context model. In this section, we first provide a definition for context modelling and then review the most relevant context modelling approaches.

Knappmeyer et al. (2013) defined context modelling as “the process of designing a model of real world entities, their properties, state of their environment and situations that can be used as a reference for acquiring, interpreting and reasoning contextual information”. We accept this definition to be used in this research work.

Strang and Linnhoff-Popien (2004) and Bettini et al (2010) surveyed the most popular context modelling techniques. Strang and Linnhoff-Popien (2004) identify a set of generic requirements for context modelling. They claim the modelling approach should:

- Be able to cope with high dynamics and distributed processing and composition.
- Allow for partial validation independent of complex interrelationships.
- Enable rich expressiveness and formalism for a shared understanding.
- Indicate richness and quality of information (QoI).
- Not assume completeness and unambiguousness.
- Be applicable to existing infrastructures and frameworks.

Context modelling techniques can be categorised into six different classes, namely key-value models, mark-up scheme models, graphical models, object oriented models, logic-based models and ontology based models (Baldauf et al., 2007). More recently, Ikram, Baker, Knappmeyer, Reetz, and Tonjesy (2011) introduced a new class of context modelling, chemistry inspired models. In the rest of this section, a brief overview and pros and cons of each of these modelling approaches is presented.

### **Key-value**

Key-value approach is the simplest form of context modelling. In this approach, contextual information is modelled as key-value pairs and is represented in different formats (e.g. text files, and binary files). This approach is only applicable when we are dealing with small amounts of data. However, in more complex systems, key-value modelling is not a good option since it is not scalable and cannot handle complex data structures. Moreover, this technique lacks the ability to model hierarchical structures or

relationships. Therefore, it cannot efficiently extract the modelled information. Another disadvantage of key-value modelling is its inability to attach meta information.

### **Mark-up Scheme Modelling**

Mark-up Scheme Modelling uses hierarchical data structure consisting of mark-up tags with attributes and content to model and represent contextual information. This technique has some advantages over key-value modelling. The first advantage is that it allows efficient data retrieval (Perera et al., 2014). Further, it supports validation and range checking. Most of the works done in this category use the most well-established mark-up language, XML (Extensible Markup Language) which provides sophisticated validation tools. However, the concept of mark-up languages is not limited only to XML. Any language or mechanism (e.g. JSON) that supports tag-based storage allows mark-up scheme modelling.

Mark-up scheme modelling suffers from two major drawbacks. Firstly, this modelling technique does not support expressive capabilities which allow reasoning. Further, due to the lack of design specifications, context modelling, context retrieval, context interoperability, and context re-usability over different mark-up schemes can be difficult.

Examples include the User Agent Profile and the Composite Capabilities/Preference Profile (CC/PP) (Klyne et al., 2004), which are based on XML and standardised by the World Wide Web Consortium (W3C). The Context Meta Language (ContextML) (Knappmeyer, Kiani, Frà, Moltchanov, & Baker, 2010) is another mark-up based scheme that represents not only context information but also context metadata as well.

### **Graphical models**

Graphical models (e.g. based on the Unified Modelling Language) allow for a pictorial description of a context model (Sheng & Benatallah, 2005) and for deriving an Entity-Relationship model as required in relational databases. Graphical context models are readable by both machines and humans. Further, it is a great tool for identifying relations between model components.

An extension is proposed by Henricksen and Indulska (2004), introducing Object-Role Modelling (ORM). This approach also has a number of disadvantages, namely complex querying, and poor support of interoperability due to the existence of different implementation.

### **Object-Oriented Models**

Object oriented concepts are used to model data using class hierarchies and relationships. Object oriented paradigm promotes encapsulation and reusability. Further, object-oriented models offer powerful capabilities of inheritance. As most of the high-level programming languages support object-oriented concepts, modelling can be integrated into context-aware systems easily. Access of contextual information is provided by well-defined interfaces (Hofer et al., 2002). Therefore, object-based modelling is suitable to be used as internal, non-shared, code based, run-time context modelling, manipulation, and storage mechanism. However, it does not provide inbuilt reasoning capabilities. Validation of object-oriented designs is also difficult due to the lack of standards and specifications.

### **Logic Based Models**

Logic Based Models offer a high degree of formalism and typically comprise facts, expressions and rules. The first logic based context modelling approach has been introduced by McCarthy (1986), which introduced context as abstract mathematical entities in artificial intelligence.

Logic Based Models enable formal inference, e.g. by means of general probabilistic logic, description logic, functional logic or first-order predicate logic. Rules are primarily used to express policies, constraints, and preferences. It provides much more expressive richness compared to the other models discussed previously. Therefore, reasoning is possible up to a certain level. The specific structures and languages that can be used to model context using rules are varied.

The main shortcoming of the Logic Based context modelling is lack of standardisation that reduces the re-usability and applicability of this approach. Furthermore, highly sophisticated and interactive graphical techniques can be employed to develop logic based or rule-based representations. As a result, even non-



technical users can add rules and logic to the systems during run time. Logic based modelling allows new high-level context information to be extracted using low-level context. Therefore, it has the capability to enhance other context modelling techniques by acting as a supplement.

## **Ontology Models**

Ontological modelling refers to an abstract conceptual vision of the world. The relations within could also be described by object-oriented methods. However, an ontology is commonly described by using languages standardised by the W3C in the context of the semantic web. Most relevant are the Resource Description Framework Schema (RDF-S) (Brickley & Guha, 2004) and the Web Ontology Language (OWL) (Deborah L. McGuinness, 2004).

Korpipää and Mäntyjärvi (2003) enumerate the following goals for designing a context ontology: simplicity, flexibility, extensibility, generality and expressiveness. Many researchers have come to the conclusion that ontologies are theoretically the best way to represent and model context due to their extendibility and unambiguousness (Baldauf et al., 2007; Wang, Da Qing Zhang, Tao Gu, & Pung, 2004). However, there may be certain drawbacks as ontology engineering is a challenging and interminable matter. With the size of the ontology, querying and processing the information embedded within becomes slow, in particular if performed on resource constrained mobile devices. The context model can be arranged in layers to cushion this effect. Wang et al. (2004) propose ontology modularization, i.e. a generic upper ontology on top and domain specific ontologies below. Fully featured ontological representations tend to decrease the inference performance and are not suitable for highly dynamic systems. If resource constrained mobile devices are envisaged as the main source and consumer of context, an appropriate alternative must be chosen. Another argument for not applying ontological representation is its limited support for modelling uncertain and unavailable data.

## **2.2 THE INTERNET OF THINGS PARADIGM**

In the previous section, we formally defined context and context-awareness. We also described the main characteristics of context and reviewed the existing context modelling approaches.

In this dissertation we are focusing on context-awareness in the IoT ecosystem. Hence, it is essential to define what is IoT and explain its main properties. As a result, in this section, we will focus on providing a basic overview of the IoT paradigm. In the remainder of this section, we first formally define the IoT paradigm and provide some preliminary knowledge about it. Then, we identify the main aspects of IoT and explain them briefly.

### **2.2.1 WHAT IS THE INTERNET OF THINGS?**

Internet of Things (IoT) is a paradigm that considers pervasive presence of a variety of things or objects around us (e.g. smart wearable devices, mobile phones, smart home appliances, sensors, Radio Frequency Identification (RFID) tags, actuators, etc.) that can communicate with each other through unique addressing schemes to reach common goals (Atzori, Iera, & Morabito, 2010). The ultimate goal of the IoT paradigm is to build a world where everything around us is interconnected through the Internet and interact with each other automatically without human intervention (Le-Phuoc, Polleres, Hauswirth, Tummarello, & Morbidoni, 2009). In other words, IoT envisions a world where our surrounding objects are aware of “what we like, what we want, and what we need” and automatically take action according to our needs (Dohr, Modre-Osprian, Drobics, Hayn, & Schreier, 2010).

The concept of IoT was first coined by Kevin Ashton in a presentation in 1998. He has mentioned “The Internet of Things has the potential to change the world, just as the Internet did. Maybe even more so.” Later in 2001, the MIT Auto-ID centre introduced their vision on IoT (Brock, 2001). Subsequently, the International Telecommunication Union (ITU) published a report (i.e. ITU Internet report) in 2005 that formally defined IoT (Union, 2005).

In recent years, IoT has become more relevant to the practical world (Patel & Patel, 2016) due to the proliferation of mobile devices, embedded and ubiquitous communication, cloud computing and data analytics. Nowadays, thanks to the availability of low-cost sensors, processors, and wireless networks, it is possible to turn any physical object, from a coffee mug to an autonomous car into an IoT device. However, the research into IoT is still in its infancy. As a result, there are no standard definitions for IoT.

Table 2.1 presents some of the existing definitions found in the literature. Among these definitions, we adopted the last definition provided by Vermesan et al. (2011) as it provides a broader vision for IoT.

**Table 2.1 - IoT definitions**

<b>Authors</b>	<b>DEFINITION</b>
<b>(Tan &amp; Wang, 2010, p. 376)</b>	“Things have identities and virtual personalities operating in smart spaces using intelligent interfaces to connect and communicate within social, environment, and user contexts.”
<b>(Bassi &amp; Horn, 2008, p. 4)</b>	“The semantic origin of the expression is composed by two words and concepts: Internet and Thing, where Internet can be defined as the world-wide network of interconnected computer networks, based on a standard communication protocol, the Internet suite (TCP/IP), while Thing is an object not precisely identifiable Therefore, semantically, Internet of Things means a world-wide network of interconnected objects uniquely addressable, based on standard communication protocols.”
<b>(Davies, 2015, p. 1)</b>	“The Internet of Things (IoT) refers to a distributed network connecting physical objects that are capable of sensing or acting on their environment and able to communicate with each other, other machines or computers. The data these devices report can be collected and analysed in order to reveal insights and suggest actions that will produce cost savings, increase efficiency or improve products and services.”
<b>(Vermesan et al., 2011)</b>	“The Internet of Things allows people and things to be connected Anytime, Anyplace, with Anything and Anyone, ideally using Any path/network and Any service.”

### 2.2.2 CHARACTERISTICS OF THE IOT

In this section we will briefly discuss the main characteristics of the IoT paradigm that is obtained from the current state-of-the-art (Miorandi, Sicari, De Pellegrini, & Chlamtac, 2012; Perera et al., 2014). IoT has several unique characteristics. However, in this section, we only focus on seven IoT characteristics which are relevant to the research problem under consideration in this dissertation. These characteristics are connectivity, heterogeneity, interoperability, real-time consideration, scalability, dynamicity, and security and privacy. A short description of each of these characteristics are provided below:

- **Connectivity:** Connectivity refers to the ability to transmit and receive data and has two main aspects: network accessibility and compatibility. Network accessibility means IoT things should have access to a global network (i.e. the Internet). Compatibility refers to the fact that the data produced by an IoT thing should be consumable by another IoT thing (Patel & Patel, 2016).
- **Heterogeneity:** The IoT contains a large number of heterogeneous devices that interact with each other autonomously. These devices have different hardware platforms, different operating systems, and different networking technologies. Moreover, the IoT devices have different capabilities that can affect the way they interact. Some devices may have very limited capabilities, for example an IoT device might have very limited storage capacity with no processing capability. On the other hand, some IoT devices can have a large memory and strong processing unit that is capable of performing various processes such as data mining and context reasoning.
- **Interoperability:** Interoperability is defined by IEEE as “*the ability of two or more systems or components to exchange information and to use the information that has been exchanged*” (IEEE, 1990). In the realm of IoT, interoperability can be seen from different perspectives such as (i) device interoperability, (ii) networking interoperability, (iii) syntactic interoperability, (iv) semantic interoperability, and (v) platform interoperability (Noura, Atiquzzaman, & Gaedke, 2019). Interoperability plays an integral role in enabling seamless interaction of IoT devices.

- **Real-time consideration:** An IoT platform should be able to deal with billions of parallel requests coming from IoT devices simultaneous in (near) real-time. Hence, it is essential for an IoT platform to offer real time data processing and analytic capabilities.
- **Scalability:** The number of IoT devices connected to the Internet is predicted to reach 50-100 billion by 2020 (Sundmaeker, Guillemin, Friess, & Woelfflé, 2010). On top of this, due to the advancement in sensing, computation, and networking technology, the IoT devices are becoming more sophisticated and will be able to collect and share more information. As a result, the number of interactions that needs to be handled by IoT also increase significantly. Therefore, IoT solutions should have a scalable design to be able to deal with the billions of parallel interactions.
- **Dynamicity:** Due to the volatile nature of IoT environments, the state of IoT devices can frequently change, for example, the state of an IoT device might vary from the connected state to the disconnected state or vice versa (Youn, 2018). Therefore, the number of IoT devices can change dynamically as a new IoT device might become available or an existing one can disappear. Therefore, the number of IoT devices can change dynamically as a new IoT device might become available, or an existing one can disappear. On top of that, the context of IoT devices can change as well. One typical example is changes in the location of mobile IoT devices such as smart-phones, smart vehicle, and wearable devices.
- **Security and Privacy:** IoT collects and produces an enormous amount of sensitive information about us and our environment. As a result, this information should be kept private and secure. Moreover, the IoT devices can be used by hackers to launch Distributed Denial of Service (DDoS) attacks. In a DDoS attack, a hacker enslaves several IoT devices into an arrangement (i.e. botnet) and sends a huge number of parallel requests to a server. This attack disrupts the normal behaviour of the server and makes it inaccessible for end users. Therefore, IoT must have a scalable and comprehensive security mechanism that protects the endpoints, the networks, and the data moving across.

These characteristics are all essential and should be taken into account when developing solutions for IoT during all the stages from design, implementation and assessment.

## **2.3 CONTEXT IN IOT**

Context-awareness has become a hot trend in the last decade, especially in the realm of the Internet of Things (IoT). As IoT evolves, the need for accessing contextual information in real time is becoming a crucial factor for the improvement of IoT services. Since the early 1990's, a large body of research has been conducted on context/context-awareness in pervasive computing to enable intelligent adaptation of applications allowing them to perform their tasks in an efficient, proactive and autonomous manner (Perera et al., 2014), according to the context of its users or other involved entities.

IoT things, which include sensors, mobile devices, connected cars, smart meters and other smart devices, are rich sources of data that is fundamental for reasoning about context of users, applications, and environment. In most cases, IoT-based smart services and applications are responsible for converting raw data coming from IoT data sources to higher-level context. However, most of these applications and services are designed to provide context within closed loop systems (silos). They do not provide standard mechanisms or approaches to discover, share and distribute context across multiple IoT applications and services, especially when these services are developed and operated by different organisations/vendors. In other words, if context generated by one IoT device is required by another IoT application, current systems lack the capability to share this context without manual integration. A key factor that will underpin the success of future IoT applications and services, in order to provide greater benefits to customers, is the ability of applications and devices (machines) to exchange context seamlessly.

To overcome this problem and ease the development of context-aware applications, which use the maximum capacity of IoT paradigm (augmenting it with context-awareness), the developer should be able to acquire contextual data from external context providers independently from the underlying structure of context providers. Therefore, it is essential to provide an easy and standard approach to define, advertise, discover/acquire, store, and query context. A promising solution to address the aforementioned problem is to build a middleware platform that manages interaction

with sources of context and offers contextual information to context-aware applications. As a result, in the next section, we will review the current state of the art in this area.

## 2.4 CONTEXT MANAGEMENT PLATFORM (CMP) FOR IOT

The management and provisioning of context information are essential elements for realising context-aware services and applications in the realm of IoT. A notable number of context management platforms (CMP) have been presented; surveys of which have been published for instance in (Baldauf et al., 2007; Hong et al., 2009; Knappmeyer et al., 2013; Truong & Dustdar, 2009). In this section, we first review the main aspects and functionalities of a CMP. Then, a brief overview of some of the most recognised CMPs is presented.

Knappmeyer et al. (2013) subdivide the major functionalities of context management platforms into six classes, which are below:

- **Sensor Data Acquisition.** This function is responsible for fetching raw context related data from multiple sources. In the context-aware system, it is essential that the system can support a variety of heterogeneous context sources. Based on the computational capability of context sources, pre-processing and data cleaning might be executed locally (on the context source) or externally as part of the CMPs functionality.
- **Context Storage.** This function refers to the mechanism of persisting contextual information in the platform. Two crucial aspects of context storage systems are caching and storing historical context. Caching improves the performance of CMPs in answering incoming queries by omitting the process of fetching repeated context. Moreover, a CMP should be capable of storing and indexing historical context. Historical context can be utilised by CMPs to produce valuable insights about IoT entities. For example, the historical data can be used to learn the habits of IoT entities and predict their future states.
- **Context Service registration and Discovery.** A CMP should provide a mechanism that allows sources of context (i.e. IoT devices and services) to describe and register their offered contextual information. Moreover, it is vital for a CMP to be able to search and find the matching sources of context for an incoming query.

- **Privacy, Security & Access Control.** This feature is considered as a vital function in CMPs as they might expose sensitive information about IoT devices and their owners to unauthorised third-parties. As a result, it is essential for a CMP to have a sophisticated authentication and authorisation mechanism to guarantee the privacy and security of users' contextual information.
- **Context Processing & Reasoning.** Sources of context (e.g. sensors) mostly offer raw sensory data to CMPs. Hence, a CMP is required to perform some pre-processing to infer context information from raw sensory data. Moreover, in many use-cases, it is essential to infer high-level context/situation from multiple existing low-level context. Therefore, a CMP should be capable of performing different context inference and situation reasoning techniques such as feature extraction, description logic, rule-based reasoning or probabilistic inference.
- **Context Querying (Context Diffusion & Distribution).** The ultimate objective of a CMP is to facilitate the development of context-aware applications. Each context-aware application has unique contextual requirements. As a result, a CMP should provide a generic approach that allows context-aware applications to request for contextual data based on their unique requirements. This approach should define a comprehensive and tailorable query language that allows context-aware applications to query for the context of their entities of interest. Moreover, it should support different communication's mode, namely push-based queries and pull-based queries. Push-based queries refer to event-driven asynchronous queries (i.e. publish/subscribe) that allows context-aware applications to subscribe for changes in the context of their entities of interest and get notified about context changes. Pull-based queries refer to synchronous on-demand queries.

Existing context management platforms can be classified in three main generations. The earliest generation, such as the Active Badge System (Want, Hopper, Falcão, & Gibbons, 1992) only focused on utilising location data. The second generation includes systems such as Context Toolkit (Dey, 2001), SOCAM (Gu, Pung, & Zhang, 2005), and Cobra (H. L. Chen, 2004). These platforms tried to achieve a higher level of generality, supporting more varieties of context. However, these platforms suffer from a number of common constraints that makes them inefficient to be used in real world context-aware systems. These constraints include lack of fault tolerance and scalability, poor interoperability support and naïve reasoning just to name



a few, which lead to low market penetration of these platforms. The effort of the research community to address these limitations lead to the development of third generation context management platforms, such as CA4IoT (Perera, Zaslavsky, Christen, & Georgakopoulos, 2012) and CAMPUS (Wei & Chan, 2013). While they successfully addressed some of the mentioned limitations, they failed to evolve to an industry standard level.

We believe the main shortcoming of these CMPs is the lack of a comprehensive and flexible context query language (CQL) that allows context-aware applications to repurpose existing contextual data based on their specific requirements.

## **2.5 CONTEXT SERVICE REGISTRATION AND DISCOVERY**

As mentioned in the previous section, one of the main functions of a CMP is context service registration and discovery. A similar concept was raised and studied in the realm of Semantic Web Service (SWS) (McIlraith, San, & Zeng, 2001) to add automation and dynamics to traditional web services. SWS aims at providing formal descriptions of requests and web services that can be exploited to automate several tasks in the web service usage process, including dynamic discovery of services.

During the last two decades, a large body of research has been conducted on definition and composition of semantic service in the domain of Semantic Web Service (SWS). These efforts led to the development of several web service description languages, such as Semantic Markup for Web Services (OWL-S) (W3C, 2004), Web Service Modelling Ontology (WSMO) (Domingue, Roman, & Stollberg, 2005), and Semantic Annotation for WSDL and XML Schema (SAWSDL) (Kopecký, Vitvar, Bournez, & Farrell, 2007).

Most of the abovementioned languages to some extent allow specifying services in terms of their signature (i.e., inputs and outputs of the service), behavioural specification (i.e., preconditions and effects), and the non-functional properties (NFPs). However, all of these languages suffer from the same limitation that makes them insufficient to describe IoT services and their context-related aspects. To overcome these shortcomings, a number of different approaches have been proposed (Fujii & Suda, 2009; Guinard, Trifa, Karnouskos, Spiess, & Savio, 2010; Hossain, Parra, Atrey,

& El Saddik, 2009). However, they do not fully support different types and aspects of context and lack an expressive language to represent them.

In our research, we adopted OWL-S as the basis of our context service description model by taking advantage of its flexibility and dynamicity in service composition. OWL-S is the most dominant approach compared to similar approaches, such as WSMO, SAWSDL, and WSDL-S (Ngan, Kir, & Kanagasabai, 2010). OWL-S is more mature in many aspects such as the definition of the process model and the grounding of services (Polleres et al., 2005). These reasons made us chose OWL-S as the most appropriate ontology for describing IoT context services.

## **2.6 CONTEXT QUERY LANGUAGES**

In this section, we will review the existing context query languages. Query languages are pivotal for querying context and determining the way queries are expressed and what information needs to be obtained. There are a variety of query languages that have been employed in CMPs to allow context-aware IoT applications retrieve contextual information. Some CMPs have used existing query languages (e.g. SQL and SPARQL) to access information or extended them such that they are tailored to context query needs. On the other hand, other CMPs have introduced a specially designed language for querying context .

CQLs can be categorized into five subclasses: SQL-based, RDF-based, XML-based, API-based and Graph-based CQLs. In the work done by Haghighi et al. (Delir Haghighi, Zaslavsky, & Krishnaswamy, 2006), an evaluation of different CQLs is presented. They compared different CQLs and demonstrated that SQL-based, XML-based, RDF-based, and API-based CQLs are more effective and powerful compared to the other subclasses.

Therefore, in the rest of this section, we provide a critical review and comparison of well-known existing context query languages that fall into these four subclasses of context query languages. Furthermore, in order to accurately identify to what extent existing CQLs can support the needed requirements for a CMP, we try to illustrate the applicability of each approach by considering the smart parking recommender use-case described in Section 1.2.

This use-case focus on developing a context-aware mobile application that uses the contextual information produced by IoT devices and services to suggest parking to smart vehicles. The main function of this application is to find the best available car parks based on the vehicle specification (i.e. width and height of the vehicle) and driver's profile (i.e. preferred price). Moreover, the application should reason about the weather conditions near available parking options to find if the weather is good for walking or not. If the weather conditions are not suitable for walking, it should suggest a parking facility that is less than 500 meters to the destination. Otherwise, the walking distance can be up to 1km. On top of this, after the driver selected a parking facility, the application should continuously monitor relevant context and notify the driver about any changes in situations that can affect the driver experience, i.e. if the parking facility is not available anymore.

### **2.6.1 SQL-BASED**

SQL is the most well-known declarative query language which is designed for accessing data from relational databases. However, directly utilising SQL as a CMP context query language is not possible as context data has its own characteristics that are different from relational database data. Compared to traditional database data, context data has its own special characteristics. According to Haghighi et al. (2006), context:

- Can be dynamic or static.
- Can be continuous data streams.
- Can be temporal, erroneous, ambiguous, unavailable or incomplete.
- Can be spatial.
- Can be unstructured.
- Can be a situation that is derived and reasoned from other context.

Considering the aforementioned scenario, it is not possible to implement a query for such a sophisticated use case by only using native SQL. For instance, SQL cannot be used to express queries for monitoring context of IoT entities as it does not support continuous queries over data streams. Moreover, since SQL is a generic query language, it does not satisfy the specific requirements needed in a CMP such as defining situations and high-level context. Besides, SQL does not deal with semantic annotations and does not incorporate the concept of ontologies used for establishing a common understanding of the context information. Therefore, some researchers extend traditional SQL by adding optional instructions to support querying context data.

Henricksen and Indulska (2004) developed a context management system on top of the Context Modelling Language (CML). CML is a powerful modelling approach for describing information's type, their classification, and quality of context. In their proposed system, a simple API is designed for accessing the context information. The context management framework for CML (Henricksen & Indulska, 2004) is based on the Object Relational Mapping (ORM) concept and maps its models to relational data schemes. Therefore, SQL can be used to retrieve contextual information. In other words, context queries are internally mapped to SQL (McFadden, Henricksen, & Indulska, 2004). While SQL supports some of the required functionalities, it cannot be used for context retrieval due to several weaknesses as stated before. For instance, when extracting context from multiple tables, queries become complex since a number of joins might be necessary. Furthermore, the programming API of CML does not address the retrieval of context information with heterogeneous representations. Lastly, this approach does not fully support complex reasoning and aggregation functions.

Another SQL-based query language that uses a relational database is presented by Feng (Feng, 2010). They designed a query language for an ambient intelligent environment, which utilises contextual data to identify data retrieval conditions in a relational database. Since this approach is based on the relational database, it suffers from similar drawbacks identified for CML. In general, works that use native SQL for context retrieval are not suitable for context data management since they are limited to relational databases.

Riva et al. (2006) proposed a SQL-based CQL to provide contextual information for mobile applications, which is called Contory. They proposed context query

language consists of three fundamental clauses, namely SELECT, FROM, and Where. The SELECT clause identifies the type of the required context item (e.g. location, light, temperature, and activity). FROM clause specifies type and characteristics of the sources from which desired context data should be collected. Lastly, the WHERE clause filters context values according to specific requirements on their associated context metadata. Furthermore, they defined four more attributes to provide a better filtering functionality. The first attribute is ‘freshness’ which identifies how recent the context data must be. The other three attributes (i.e. DURATION, EVERY, and EVENT) are responsible for supporting event-based and continues/periodic queries.

The main shortcoming of Contory that makes it inappropriate to be used as the main interface of a CMP is its simplified data model, which does not have a mechanism to indicate the entity of interest in a query. For example, while it allows querying temperature, it does not support querying temperature of a specific oven. Furthermore, another shortcoming of this approach is the lack of supporting context processing operations. On top of these, Contory is not interoperable with different external infrastructures and sensor devices. Last but not least, this language does not support querying multiple sources of context simultaneously (in one query).

Schreiber and Camplani (2012) proposed a framework to configure and manage pervasive systems, called PerLa. PerLa also adopts the database metaphor and uses an SQL-like query language for context retrieval.

PerLa queries support both data acquisition and context retrieval by providing three types of queries: Low Level Queries (LLQ), which describe the behaviour of nodes, and determine the data selection criteria, the sampling frequency and the computation to be performed on sampled data; High Level Queries (HLQ), which determine the high-level elaboration involving data streams coming from multiple nodes, and Actuation Queries (AQ), which can modify devices’ parameters. Similar to Contory, the main shortcoming of PerLa is lack of support for expressing and distinguish the entity of interest in a query. Furthermore, PerLa does not support domain-based standards and has a very limited support for processing context data.

The most recent work in the area of SQL-based CQL is presented in (P. Chen, Sen, Pung, & Wong, 2014). Chen et al. (2014) proposed a new SQL-based CQL that

supports both pull-based and push-based queries. This work introduces some useful ideas and concepts. Their work supports continuous queries with compound conditions for accessing contextual information from various context entities. Furthermore, they claim that their work also supports contextual functions, however, they did not mention how this contextual function can be represented.

## 2.6.2 RDF-BASED

As it is demonstrated by Haghighi et al. (2006), another powerful type of CQLs is RDF-based. The most well-established RDF query language is SPARQL. SPARQL has been used in many IoT platforms, such as OpenIoT (Soldatos et al., 2015), for querying contextual information. SPARQL (Prud'hommeaux & Seaborne, 2008) is a W3C standard proposal for an RDF query language whose syntax is inspired by SQL. It incorporates semantic concepts and ontologies into a SQL-inspired query language. SPARQL facilitates querying concepts of an entity, but it is not intended to be used for querying complex data constructs with several levels of nesting (Reichle et al., 2008), which is commonly used in context-aware IoT applications. To clarify, consider the basic SPARQL condition presented in Code block 2.1. This example presents an equality expression, which can be used to find all the parking facilities that have a parking space with fast charging points. In this example, the parking facility is defined based on mobivoc semantic vocabulary.

```
{
?parkingFacility a mv:ParkingFacility;
                  mv:parkingSpace ?parkingSpace .
?parkingSpace mv:charger ?charger .
?charger mv:isFastChargeCapable ?isFastCharger .
}

FILTER (?isFastCharger = 'true')
```

**Code block 2.1** - A basic condition expressed in SPARQL

As it is shown in the code block above, five lines of code with several variables are required to express this basic condition in SPARQL. As a result, queries easily become quite long and complicated which increases developers' cognitive load. However, the same condition can be easily represented with only one line of code: `parkingFacility.parkingSpace.charger.isFastCharger = 'true'`. Another drawback of SPARQL is its lack of support for defining custom aggregation functions. Furthermore, SPARQL does not provide a mechanism to define and query high-level context (i.e., situation). While it is possible to assume that context consumers can implement custom aggregation and situation reasoning functions as an additional layer of software, it contradicts with one of the main motivations behind developing a CMP which is providing a fast and easy way to query context and hide the complexity of low-level programming.

The MUSIC CQL proposed by Reichle et al. (2008) is another well-known RDF-based CQL. Their work has a good support for querying contextual information. However, since MUSIC CQL can only represent context request from a single entity, it cannot express complex context queries (e.g., the query for the school safety scenario).

SOCAM (Service-oriented Context-Aware Middleware) framework (Gu et al., 2005) also provide a RDF-based CQL (based on OWL) for context retrieval. This language is capable of providing contextual data about context entities and the relationships among them by using ontology technology. However, the main shortcoming of this work is its restriction on supporting complex queries.

### **2.6.3 XML-BASED**

The other category of context query languages that we review in this section is XML-based CQLs.

A simple XML-based context description and query language was developed in the MobiLife project (Floreen et al., 2005). This CQL provides a good set of simple relational operators and also string-based operators. There are some operators to combine simple filters to more complex ones as well. A query can be expressed with regard to a value of a parameter, the timestamp of a parameter and on associated meta-data, as for example the accuracy or the confidence of a parameter (probability of

context information to be correct). There is also support for including the position of a parameter in an array of context elements, which allows the selection of a specific parameter in the array. The concept of placeholders is also supported. However, there is lack of aggregation functions and the need for ontologies and semantic reasoning is not sufficiently addressed. Furthermore, another important limitation is that the application must know beforehand the provider of the context information and then query the provider. Thus, support for specifying queries involving sub-queries for different context providers is not provided.

Another work that uses XML-based language for querying context is the Nexus architecture (Bauer, Becker, & Rothermel, 2002; Hönle, Käppeler, Nicklas, Schwarz, & Grossmann, 2005). Nexus is an open platform that facilitates developing location-aware applications and enables integration of and interaction between the applications. The Nexus platform is based on a common augmented world model that is described by AWML (Augmented World Modelling Language) and can be queried using AWQL (Augmented World Querying Language). The augmented world model represents the world as data objects with attributes and all the objects produced by a context provider belong to an Augmented Area. Context providers register their Augmented Areas and their object types with the Area Service Register that will assist the system with queries. An extension to AWML has integrated metadata into the model to facilitate resource finding, context selection, context quality and data processing (Hönle et al., 2005). Some of the strengths of AWQL queries are their support for generalization and aggregation rules, nearest neighbour queries and spatial relationships (Grossmann et al., 2005). Other advantage of AWQL queries is that they can be mapped into SQL queries using multiple joins. Despite expressiveness of XML, this language does not provide sufficient flexibility to support complex queries and expression of different aspects of context.

#### **2.6.4 API BASED CONTEXT QUERY / OR JSON BASED**

Another significant context query language is NGSI language (“NSGIV2 API Walkthrough - Fiware-Orion,” n.d.). NGSI is the main interface of FIWARE project (“FIWARE,” n.d.), which is one of the most advanced CMPs in terms of consistent development and market penetration. Further, NGSI was recently used as a base for the development of an ETSI NGSI-LD standard for context information management



(“ETSI - ETSI ISG CIM group releases first specification for context exchange in smart cities,” n.d.; Sophia Antipolis, 2017). However, the NGSI language (“NSGIV2 API Walkthrough - Fiware-Orion,” n.d.), suffers from a number of drawbacks. NGSI supports only one entity per query, which limits the expressivity, flexibility, and query performance, and it also adds network overhead. Moreover, NGSI has limited support for situation reasoning and monitoring. To address this, FIWARE has integrated the Esper Complex Event Processing (CEP) engine (“Esper,” n.d.), which uses Esper EPL (“Esper,” n.d.) to represent monitored situations. However, NGSI and Esper EPL are two disjoint technologies, and this increases the development and maintenance efforts. Such an approach also adds conceptual complexities as Esper EPL is a more generic technology and is not designed to support IoT context-aware environments.

### **2.6.5 DISCUSSION**

This section presents a qualitative evaluation of existing context query languages. Based on the existing works and other aforementioned considerations, we have identified six requirements for a context query language.

1. Support for complex context queries concerning various context entities and constraints;
2. Support for interoperability. In other words, provide a context model that can be converted into different data models as required;
3. Support for both pull-based and push-based queries;
4. Support for aggregating and reasoning functions to query both low level and infer high-level context;
5. Support for continuous and situation/event-based queries;
6. Support for different aspects of context such as imperfectness, QoC, and CoC;

Table 2.2 reports a summary of the comparative evaluation of current CQLs with respect to these requirements.

**Table 2.2** - Evaluation of Existing CQLs

Title	CQL Type	Requirements					
		#1	#2	#3	#4	#5	#6
Contory (Riva & Di Flora, 2006)	SQL-based	✗	✗	✓	✗	✗	✓
CML (Henricksen & Indulska, 2004)	SQL-based	↗	✗	✗	↘	↘	✓
PerLa (Schreiber & Camplani, 2012)	SQL-based	↗	✗	✓	↘	↗	↘
SPARQL (Prud'hommeaux & Seaborne, 2008)	RDF-based	✓	↗	✗	✗	✗	✗
MUSIC-CQL (Reichle et al., 2008)	RDF-based	✗	✓	✓	↗	↗	↗
SOCAM (Gu et al., 2005)	RDF-based	✗	✓	✓	↘	↗	↘
Nexus (Bauer et al., 2002)	XML-based	↗	↘	✓	✗	✓	✓
MobiLife (Floreen et al., 2005)	XML-based	↘	✓	✓	↘	✗	↗
ContextML (Knappmeyer et al., 2010)	XML-based	↗	✓	✓	↘	✗	↗
NGSI-9/10 (P. Chen et al., 2014)	API-based	✗	↗	✓	↘	✓	↗

✓ full support; ↗ partially supported; ↘ limited support ✗ not supported;

In our view, meeting all of these requirements is essential for a CQL. For example, as it is illustrated in Table 2.3, more than half of the existing CQLs (six out of ten) only support context queries concerning a single entity. However, in real-life scenarios (e.g. school safety scenario), the contextual information is coming from different context sources (e.g. a smart bus, a smart car, mobile devices, a school server, and a smart gate). Therefore, those CQLs that does not fully support this criterion are not a good candidate for our objective.

Furthermore, another important aspect which needs to be addressed properly in designing a CQL is supporting interoperability. More precisely, without a common understanding (i.e. context model), smart entities (context providers and consumers) cannot communicate and exchange context with each other. Therefore, it is vital for a CQL to provide a mechanism to query for context data presented in heterogeneous formats. As depicted in Table 2.3, only ContextML (Knappmeyer et al., 2010) and SPARQL support both criteria 1 and 2. However, both of these CQLs fail to meet requirements 4 (i.e. 4. support for aggregating and reasoning functions) and 5 (i.e. support for continuous and situation/event-based queries). It can be seen that none of the existing CQLs fulfils all the requirements, whereas most of the approaches failed to meet the first two requirements. Moreover, to the best of our knowledge, none of these languages are known outside the research community and are not used in real environments. Furthermore, none of these languages have become a widely adopted standard, while such a standard is fundamental nowadays (Sophia Antipolis, 2017).

## 2.7 SUMMARY

In this chapter we presented the state of the art in three areas of research concerning context in IoT. We started with providing an exhaustive background of context and context-awareness. In this section, we first reviewed the formal definitions for context and context-awareness. We then discussed the main characteristics of context and reviewed existing context modelling approaches.

Afterwards, we briefly described the IoT paradigm and discussed its main characteristics. Further, we argued about the correlation between context and IoT and explained the context's lifecycle in IoT ecosystem.

The second part of the chapter introduced context management platforms and discussed the state of the art in this area of research by reviewing the existing CMPs. The discussions reveal the one of the main drawbacks of CMPs, which is the lack of a comprehensive context query language. As a result, we focused on this topic, and reviewed existing context query languages. Moreover, we identified six main requirements for a CQL for IoT ecosystem, which had been used for qualitative evaluation of the presented context querying approaches. We evaluate strengths and weaknesses of specific CQL and examine the limitations of each language. Resulting

from our evaluation, we identify a theoretical gap which currently exists in querying context.

## Chapter 3: Context Definition and Query Language

---

The rapid development and penetration of the Internet of Things (IoT) into daily life leads to an enormous increase in the number of IoT-based smart services and devices. These IoT devices and services, which include sensors, mobile devices, connected cars, smart meters and other smart devices, produce rich, useful and relevant context data about the state of various physical (e.g. a car, a carpark, a building) and conceptual (e.g. a meeting, an accident, a traffic jam) entities. In this dissertation, all these sources that can generate contextual information are abstracted as context services. The context data produced by context services can be shared and consumed by IoT applications that may reuse and repurpose it. Such a paradigm will enable the realisation of IoT vision, i.e. smart devices and objects to become active participants in business, and social life by autonomously interacting among themselves and exchange information about the entities they monitor.

However, managing and utilising the large volume of data generated by various context services is a challenging task. Most of the context services (in the current IoT ecosystems) are designed to work within closed loop systems (silos). They do not provide standard mechanisms or approaches to discover, share and distribute context across multiple IoT applications, especially when the services are developed and operated by different organisations/vendors. In other words, if a context service owned and managed by a service provider is required by an external IoT application (owned by another service provider), current systems lack the capability to easily share the context produced by heterogeneous context services with the context-aware IoT application, without manual integration. Therefore, to underpin the success of future IoT applications that can provide greater benefits to customers, it is essential to find an efficient solution which allows applications and IoT devices (machines) to advertise, query, discover, combine and consume context seamlessly.

As discussed in the preceding chapter, a unified, reliable and flexible approach for advertising, querying and discovering context services that incorporates high-level context is still an open research problem. As a result, the current chapter addresses this

open issue by introducing a novel context management platform (CMP) called Context-as-a-Service (CoaaS), which is enhanced with a generic yet tailorable mechanism to query and publish context.

This chapter consists of two main parts. The first part presents the vision of CoaaS, its blueprint architecture, and the fundamental concepts and definitions, which will be frequently accessed in the rest of this dissertation. The second part is dedicated to introducing the CoaaS pioneering mechanism for publishing and querying context. To achieve this goal, two novel languages have been designed and implemented, namely Context Service Description Language (CSDL) that is used to describe and register context services (i.e. publish context), and Context Definition and Query Language (CDQL) that allows IoT devices and applications to query and consume the data produced by context services.

### **3.1 CONTEXT-AS-A-SERVICE OVERVIEW, DEFINITIONS, AND BLUEPRINT ARCHITECTURE**

This section describes the overview of CoaaS platform and its role in the IoT ecosystem. Furthermore, the formal definitions for the underlying concepts of CoaaS will be presented in this section. Lastly, we present the blueprint architecture of CoaaS platform and briefly explain its main components.

#### **3.1.1 CONTEX-AS-A-SERVICE: OVERVIEW AND DEFINITIONS**

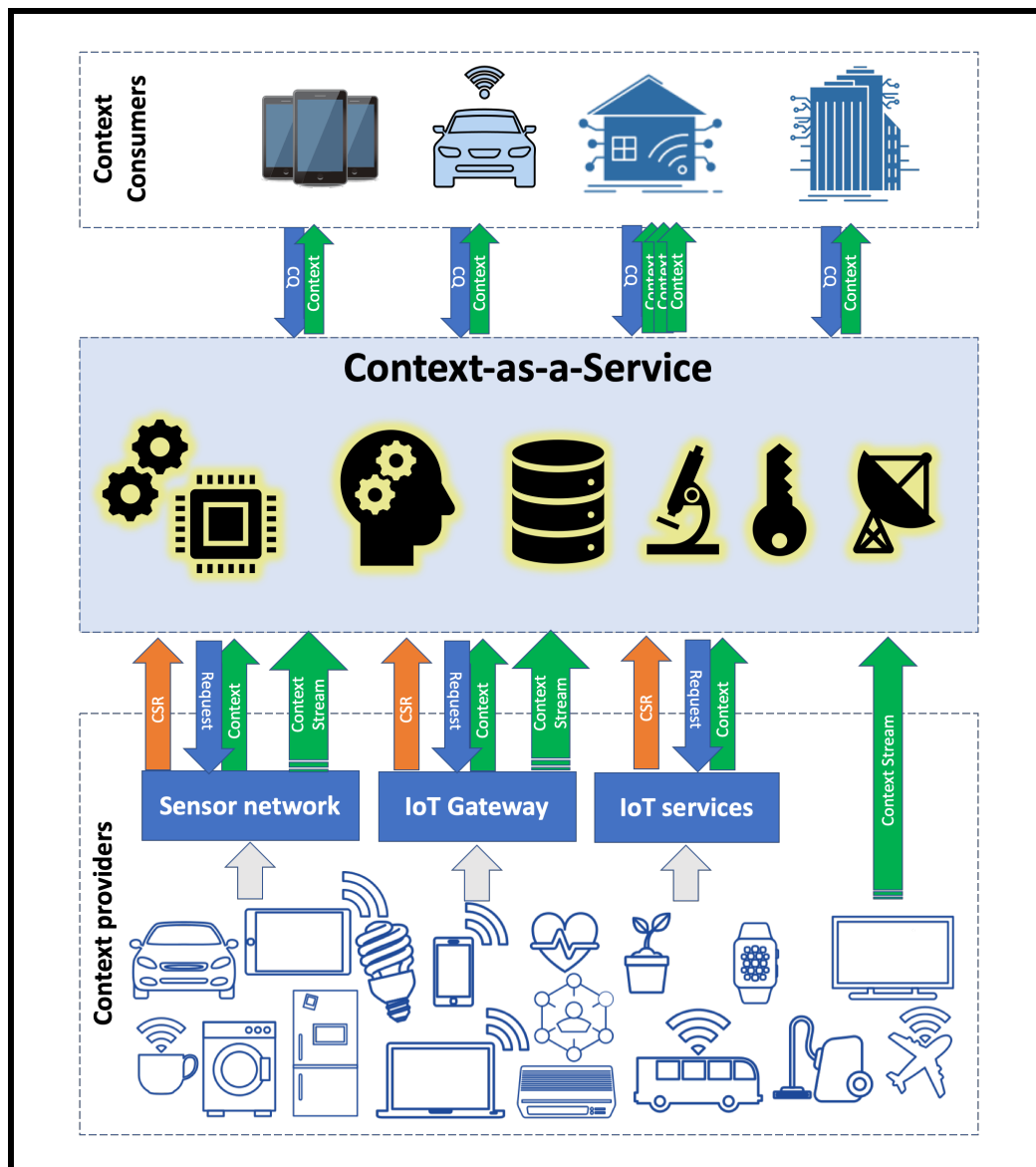
In this section we introduce the fundamentals and definitions of context-as-a-service (CoaaS) platform in IoT. CoaaS is a context management platform, which has been designed to facilitate the development of context-aware IoT applications by providing a generic yet tailorable mechanism to query and publish context. In other words, CoaaS enables applications to provide and consume context about their entities of interest seamlessly, without requiring manual integration of IoT silos.

As mentioned earlier, context is the information that can be used to characterise the situation of an entity (Dey, 2001). Entities can be persons, locations, or objects which are considered to be relevant for the behaviour of an application. An entity can be characterised by a set of parameters, known as context attributes.

**Definition 3.1 (Entity and Context Attribute).** In context-aware systems, an entity (denoted by  $E$ ) accounts for a physical or virtual object (such as a person, a car, an electronic device, or an event) that can be associated with one or more context attributes (denoted by  $ca$ , which can be any type of data that characterises this entity).

For example, a ‘car’ entity can have a location, speed, fuel level, the number of available seats, model, and manufacturer as its context attributes.

The big picture view of Context-as-a-Service platform in the IoT ecosystem is represented in Figure 3.1, which consists of three layers of Context Consumers, Context Providers, and the context management platform (CMP).



**Figure 3.1** - Overview of Context-as-a-Service platform in IoT

The top layer is a collection of context-aware IoT applications in various domains that require contextual information in order to perform their task. These applications are interested in collecting contextual information about a particular entity with specific characteristics. They are defined as context consumers.

**Definition 3.2 (Context Consumer).** Context Consumer (CC) refers to any device or system that queries and receives context about one or several entities.

The bottom layer, in Figure 3.1 shows the sources of context, which consists of sensors, smart connected devices, and systems that can produce context about entities. They are the context providers.

**Definition 3.3 (Context Provider).** Context Provider (CP) refers to any device, application or system that provides context or data that can be used to infer context about one or several entities.

In our system, we distinguish between different classes of CPs based on the type of context they produce. At the most basic level, a context provider can be a standalone sensor that is connected to the Internet and is capable of transmitting raw sensory data about a particular attribute of an entity. For example, a temperature sensor connected to a Wi-Fi microchip such as ESP8266 (“Espressif Systems - Wi-Fi and Bluetooth chipsets and solutions,” n.d.) can act as a CP. However, CPs can be more sophisticated and provide either low-level or high-level context about characteristics of several IoT entities. For example, IoT gateways and middleware, sensor networks, or even a mobile application can play the role of a CP and supply context. Lastly, some web-based services such as Google Maps APIs, or weather forecast APIs can also act as context providers as they can produce useful information.

As a result, based on the CPs’ type, each context provider can have one or more services, which produce context about an entity. We refer to these services as Context Services.

**Definition 3.4 (Context Service).** A Context Service (denoted by  $cs_{j,j} \in \mathbb{N}$ ) provides contextual information about a particular entity. Context service can be represented as a triple:  $\langle E, CA, P \rangle$  where  $E$  denotes the related entity,  $CA$  is a set of



provided context attributes, and Predicates (denoted by  $P$ ) form a composite logical expression defined over CA.

For example, a smart garage (which is a context provider) can provide a context service to deliver values of context attributes such as cost, available facilities, and time limit (contextual information) about available car parks (entity) in a specific location. Further, the working hours of this garage are from 8 am to 8 pm during weekdays, and 10 am to 10 pm on weekends (complex context attribute). This context service description can be represented as:

$$cs_1: \langle E_1, CA_1, P_1 \rangle$$

where:

$$\left\{ \begin{array}{l} E_1: \text{carpark} \\ CA_1: \{\text{cost, location, available facilities, number of available parking spots, working hours}\} \\ P_1: \\ \quad \text{location} = \text{LocA} \wedge \\ \quad ((\text{workingHours between 8:00 and 20:00} \wedge \text{weekdays}) \vee \\ \quad (\text{workingHours between 10:00 and 22:00} \wedge \text{weekends})) \end{array} \right.$$

On the basis of the presented definition for context services, we have designed a high-level language for describing context services, which will be described in Section 3.3.

The middle layer of Figure 3.1 shows the actual CoaaS platform, which enables global standardisation and interworking among context providers and consumers.

CoaaS can interact with CPs in two ways, either by fetching context on-demand or through receiving context/data streams. In the first case, the CPs must have registered the description of their services first by sending a context service registration (CSR) request. Then, CoaaS can retrieve data about IoT entities by sending requests to corresponding providers on-demand. As mentioned above, CoaaS can also process streams of context updates, which CPs are sending to the platform. Context updates contain updates of the entities' states and are processed by CoaaS to monitor situations. The blueprint architecture of CoaaS platform is presented in Section 3.1.2.

On the other hand, context consumers can retrieve context information from the middleware by issuing context queries (CQ).

**Definition 3.5 (Context Query).** Context query is a request for contextual information (either context attributes or high-level context inferred from context attributes) from one or many entities.

For example, a smart vehicle can issue a context query to retrieve the cost, location, and number of available spaces (contextual information) of the best parking facilities (entity of interest) near the driver's meeting location based on his/her preferences. This query contains three main entities, namely parking facility, smart vehicle, and driver.

Each context query can be split into several sub-requests, where the final result of the query will be computed based on the contextual information retrieved from the results of these sub-requests by aggregating the results or using the results to infer a higher-level context.

**Definition 3.6 (Context Request).** A context request (denoted by  $cr_{i,i \in \mathbb{N}}$ ) represents a request for contextual information about a particular entity. Context request can be represented as a triple:  $\langle E, CA, P \rangle$  where  $E$  denotes the entity of interest,  $CA$  is a set of requested context attributes, and  $P$  is a set of predicates, which are defined over  $CA$  using logical expressions.

Based on Definitions 3.5 and 3.6, we have designed a novel context query language that supports complex context queries concerning various entities. This language will be presented in Section 3.4.

The aforementioned context query for finding car parks can be broken down into three context requests, one for each entity. The first request is issued to retrieve context about the driver, the second request is issued to identify the smart vehicle, and the last context request is issued to retrieve information about available parking. These context requests are represented as below:

$$cr_1: \langle person, \{meeting, parking\ preferences\}, \{driver\ id = 101\} \rangle$$

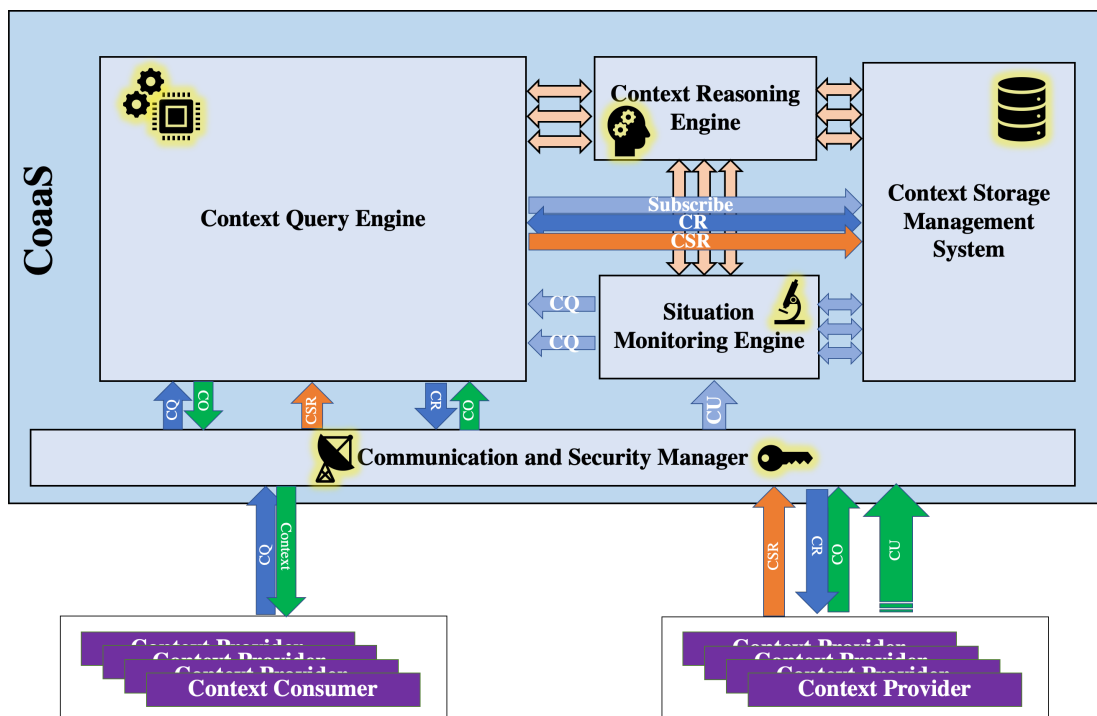
$$cr_2: \langle car, \{location, width, height, length\}, \{VIN = 202\} \rangle$$

$$cr_3: \langle \text{parking facility}, \{\text{location}, \text{cost}, \text{\#available spots}\}, \{\text{distance}(\text{meeting.location}, \text{parking.location}) < 500\} \rangle$$

After defining the underlying concepts in this section, we present in the next section the blueprint architecture of CoaaS platform and introduce its main components to illustrate how CoaaS platform works.

### 3.1.2 COAAS PLATFORM BLUEPRINT ARCHITECTURE

This section presents the blueprint architecture of CoaaS platform and discusses its main components. As mentioned in Section 2.4, CMPs have six major functionalities, namely (i) sensor data acquisition, (ii) context storage, (iii) context lookup and discovery, (iv) privacy, security and access control, (v) context processing and reasoning, and (vi) context diffusion and distribution. Aligned with these functionalities, we designed the blueprint architecture of CoaaS platform accordingly, which can be seen in Figure 3.2.



**Figure 3.2 - CoaaS Blueprint Architecture**

As this figure shows, the CoaaS platform has five main components: Communication and Security Manager, Context Query Engine (CQE), Situation Monitoring Engine (SME), Context Storage Management System (CSMS), and

Context Reasoning Engine (CRE). Table 3.1 provides a mapping between the CoaaS components and the aforementioned CMP functionalities. In the rest of this section, a brief description of each of these main enabling components is presented.

**Table 3.1** - CoaaS major components

<b>Component</b>	<b>Responsibilities</b>
Communication and Security Manager	(iv) Privacy, security and access control
Context Query Engine	(i) Sensor data acquisition  (iii) Context service registration and discovery  (vi) Context querying (Context diffusion and distribution)
Situation Monitoring Engine	(i) Sensor data acquisition  (v) Context processing and reasoning
Context Storage Management System	(i) Sensor data acquisition  (ii) Context Storage  (iii) Context service registration and discovery
Context Reasoning Engine	(v) Context processing and reasoning

The **Communication Manager** is responsible for the initial handling of all incoming and outgoing messages, namely context services registration (CSR), context queries (CQ), context updates (CU), and context responses. This module acts as a proxy and distributes all the incoming messages from CPs and CCs to the corresponding components. To guarantee the privacy and security of CoaaS, this component is linked to the **Security Manager**. The **Security Manager** module firstly checks the validity of

incoming messages and authenticates requests. Moreover, the Security Manager checks whether the context consumer has access to the requested context service or not (authorization). Lastly, it is also responsible for monitoring all the incoming messages to identify any suspicious patterns, such as distributed denial-of-service (DDoS) attacks.

**Context Query Engine (CQE)** is mainly responsible for parsing the incoming queries, generating and orchestrating the query execution plan, and producing the final query result. Furthermore, this component also takes care of fetching required data from context providers on demand. This component will be discussed in more detail in Chapter 4 (See Section 4.1).

**Situation Monitoring Engine (SME)** is designed to support the continuous monitoring of incoming context, infer situations from available context, detect changes in situations and provide notification of detected changes. This component monitors the real-time context of the IoT entities and reason about their situations. It also initiates the actuation procedure by notifying context consumers when their situation of interest is detected. The architecture and workflow of this component will be presented in Chapter 4 (See Section 4.5).

**Context Storage Management System (CSMS)**, which is described in detail in (Medvedev, Indrawan-Santiago, et al., 2017), has two main objectives. First of all, it stores descriptions of context services and facilitates service discovery. Secondly, it caches contextual information to ensure reasonable query response time and deals with problems like network latencies and potential unavailability of context sources.

The main task of the **Context Reasoning Engine (CRE)** is to infer situations from raw sensory data or existing primitive low-level context. It is a common need in many context-aware IoT applications to query about the situation of a context entity or trigger a query when a specific situation is detected. A situation can be seen as a high-level context that is inferred from multiple low-level context (Delir Haghighi, Krishnaswamy, Zaslavsky, & Gaber, 2008).

So far in this chapter, we have provided an overview of CoaaS platform and presented its blueprint architecture. Moreover, we have identified the main components of CoaaS and explained their roles. However, in this dissertation, we will only focus on

two components of CoaaS platform, namely CQE and SME, that deal with context monitoring, discovery and querying. These components will be discussed in detail in Chapter 4.

In the remainder of this chapter, we will focus on the main aim of this dissertation, which is designing formal language constructs for describing and querying context services. Aligned with characteristics of the IoT ecosystem, requirements of context-aware IoT applications, and the architecture of CoaaS platform, we have designed two high-level languages, one for representing context services and one for modelling context queries. The details of each language will be presented in the next section.

### **3.2 CONTEXT SERVICE DESCRIPTION AND CONTEXT QUERY LANGUAGE**

As the standardisation efforts for IoT are fast progressing, efforts in standardising context management platforms led by the European Telecommunications Standards Institute (ETSI) are gaining more attention from both academic and industrial research organisations. These standardisation endeavours will enable intelligent interactions between ‘things’, where things could be devices, software components, web-services, or sensing/actuating systems. Therefore, having a generic approach to describe and query context is crucial for the success of IoT applications. In this section, we focus on addressing such an approach by proposing two specially designed high-level languages to enable IoT things to exchange, reuse and share context between each other.

The first proposed language is designed for describing context services and called Context Service Description Language (CSDL). CSDL is an abstract service description language, which allows context providers to describe and register their services.

The second language called Context Definition and Query Language (CDQL), which provides a generic and flexible approach to defining, representing, inferring, monitoring, and querying context. CDQL consists of two main parts, namely: Context Query Language (CQL), which is a powerful and flexible query language to express contextual information requirements without considering the details of the underlying data structures; and Context Definition Language (CDL), which is designed to describe situations and high-level context. An important feature of the proposed query language

is its ability to query entities in IoT environments based on their situation in a fully dynamic manner where, users can define situations and context entities as part of the query.

In the rest of this section, we will first introduce our context model. The context model and the corresponding data descriptions provide the foundation for all other components of our work. Then, we will present CSDL and CDQL in detail in the following sections.

### **3.2.1 CONTEXT MODEL**

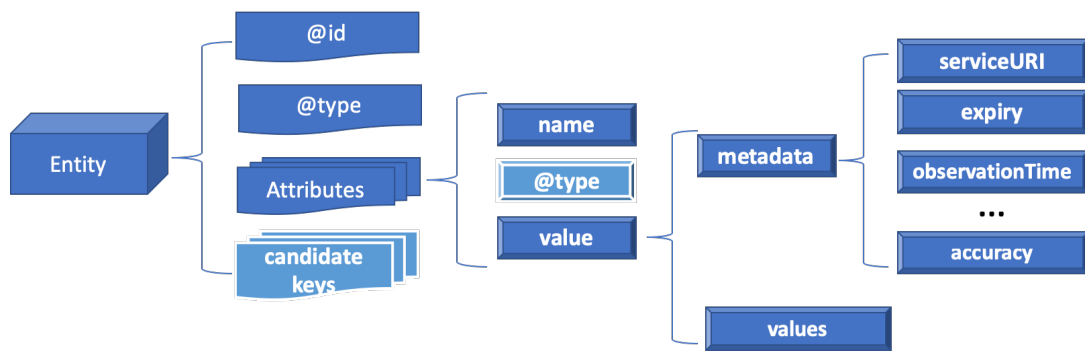
The main objective of this dissertation is to enable heterogeneous IoT entities to share and exchange context. For example, consider an entity that wants to know about the level of light at night on a certain bike path. To answer this query, first, we need to find those entities (e.g. humans carrying mobile devices, fixed sensors etc. that are part of an IoT application for environmental monitoring) located in that area. Then, we need to filter the retrieved list based on the entity's context, e.g. in case of a smartphone, its owner activity (context) to determine relevance, the smart device capabilities such as equipped with a light sensor etc. As the first step towards supporting such a scenario, there is a need to capture and model different types of contextual information and the corresponding characteristics and capabilities using a generic and standard approach, known as context model.

In order to design a generic context model for IoT environment, several challenges are needed to be considered and addressed. First, context information is distributed on an arbitrary number of devices; these devices are unreliable and can appear and disappear. On top of this, IoT ecosystems consists of heterogeneous devices providing different sets of context artefacts in different representations and under different names. Furthermore, the context model should take the general characteristics of context data into account, like ambiguity, impreciseness or incompleteness (see Section 2.2.2). Based on these considerations, we list three key requirements in designing context model in the IoT ecosystem:

- (1) The context model should provide a common vocabulary to achieve interoperability between heterogeneous context services and consumer;

- (2) The context model should support the integration of domain-specific vocabularies and ontologies.
- (3) The context model should define and capture the different aspects of context, e.g., cost, quality, accuracy, and freshness of context;

By considering these requirements, we have designed a context model that consists of two layers: a cross-domain layer and a domain-specific layer. The cross-domain layer provides a common structure and vocabulary to achieve interoperability between heterogeneous CPs and CCs. Further, the cross-domain layer can be extended by various application-specific ontologies, which is referred to as domain-specific layer. The domain-specific layer introduces the particular entity types required for a particular domain.



**Figure 3.3** - Entity Data Model

Figure 3.3 represents the structure of the cross-domain layer. The centre of gravity in the proposed model is the notion of context entity. As defined above, each context entity represents the state of a physical (e.g., a sensor or a person) or logical object (e.g., an event, a traffic accident). In the proposed context model, JSON-LD is used to provide representations for context entities and associate them with semantics defined by the domain-specific ontologies.

In our context model, context entities are uniquely represented by the combination of two attributes, namely @id and entity @type. The @id assign a unique identifier (i.e. URI) to each entity in order to distinguish it from any other entity. This ID can be used by CPs and CCs to easily interact with a specific entity.



Entity types are intended to describe the type of thing represented by the entity. Each entity type corresponds to a semantic class of entities, which is defined in the domain-specific layer. For example, a context entity with id ‘parkingFacility-101’ could have the type ParkingFacility (i.e. <http://schema.mobivoc.org/ParkingFacility>), which is defined by the MobiVoc (Brümmer & Weilandt, 2018) domain-specific ontology.

Further, as mentioned earlier in this chapter, each context entity can have several context attributes. In our proposed model, attributes have an attribute name, an attribute type, an attribute value. The attribute name describes what kind of property the attribute value represents for the entity, for example the available number of parking spaces in a parking facility. The attribute type represents the value type of the attribute value. The attribute value finally contains the actual data, and an optional metadata describing the properties of the attribute value.

Metadata provides important information about the actual context information, which facilitates the management of context data. Each metadata consists of a key-value pair, where the key represents the role of the metadata and the value contains the actual value of metadata. In our model, we have considered ten main metadata for context attributes, which are presented in Table 3.2.

**Table 3.2 - Context metadata**

<b>Name</b>	<b>Description</b>
<b>Accuracy</b>	“Describes how exactly the provided context information mirrors reality” (Buchholz et al., 2003, p. 5).
<b>Precision</b>	“Denotes the probability that a piece of context information is correct” (Buchholz et al., 2003, p. 6).
<b>Trust-worthiness</b>	“Describes how likely it is that the provided information is correct” (Buchholz et al., 2003, p. 6).

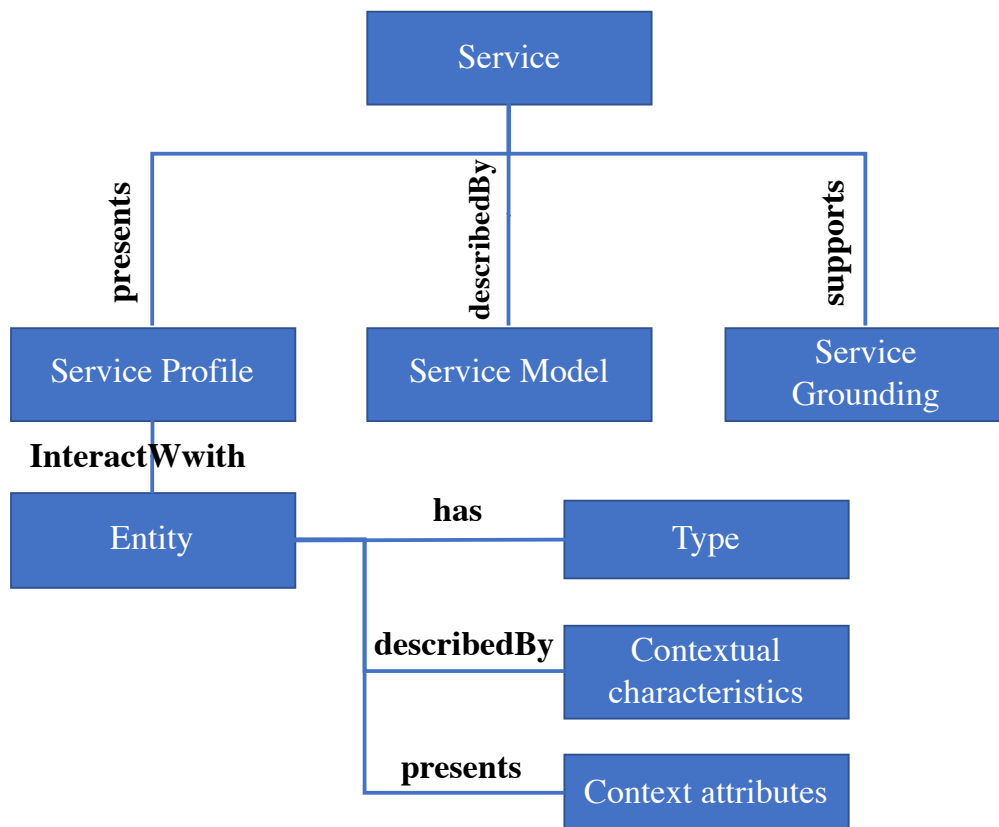
<b>Resolution</b>	“Denotes the granularity of information” (Buchholz et al., 2003, p. 6).
<b>Freshness</b>	“Indicates the time that elapses between the determination of context information and its delivery to a requester” (Sheikh, Wegdam, & van Sinderen, 2008).
<b>Cost of context</b>	Indicates the cost associated with accessing and processing context information.
<b>Observation timestamp</b>	Indicates the exact time the context value is sensed. In the case of high-level context, which is inferred from several low-level contexts, the observation timestamp of the oldest involved context will be considered.
<b>Expire timestamp</b>	Indicates the exact timestamp when the context data is no longer valid.
<b>Average update interval</b>	Indicates how frequently a context data will be updated.
<b>Service endpoint URI</b>	Represents the endpoint of a context service that can be invoked in order to fetch the context data.

### 3.3 CONTEXT SERVICE DESCRIPTION LANGUAGE (CSDL)

In this Section, we describe our proposed Context Service Description Language (CSDL) (Hassani, Haghighi, Jayaraman, Zaslavsky, & Ling, 2018). CSDL is a JSON-LD-based language that enables developers of context services to describe their services in terms of semantic signature and contextual behavioural specification; where the semantic signature defines the service name, number and types of its parameters, and the type of its output, and the contextual behavioural presents the context of the entities provided by the service.

Further, CSDL allows developers to describe their services using a standard language. CSDL enables the fast development of IoT applications that can discover and consume context services owned and operated by different individuals and

organisations. For describing the semantics of context services, we adopted Web Ontology Language for Services (OWL-S) (W3C, 2004) which is a W3C recommendation, as the basis of CSDL. OWL-S is an ontology language, which is developed based on the Web Ontology Language (OWL) to enable automatic discovery, invocation, and composition of web services. However, as OWL-S was initially designed for describing web services and does not support the semantic description of context, we extended the OWL-S by adding the context description of the entities associated with context services.



**Figure 3.4** - Structure of CSDL

As shown in Figure 3.4, CSDL consists of three main components: (i) Service Profile, (ii) Service Grounding, and (iii) Service Model. Service Model gives a detailed description of a service signature, namely its input and output, and identifies the semantic vocabularies that are supported by the given service. Service Grounding provides details on how to interact with a service. This component identifies which type of communication needs to be used to call the service (e.g., HTTP get, XMPP, Google Cloud Messaging). Further, based on the type of communication, it will provide other required information to make the service invocation possible (e.g., URI in the case of

HTTP get). Lastly, Service Profile is used to make service advertising and discovery possible. This component indicates the type of the entity that a service interacts with. Further, it defines the context-aware behaviour of the service. Figure 3.5 shows an example of a service description in CSDL. This context service provides information about parking facilities located in Monash University.



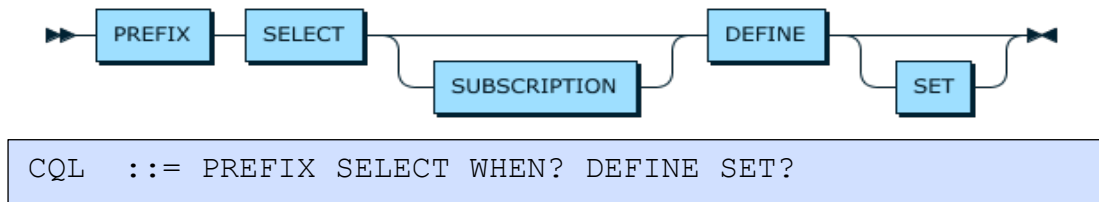
**Figure 3.5** - An example of service description in CSDL

### 3.4 CONTEXT DEFINITION AND QUERY LANGUAGE (CDQL)

To fulfil all the discussed requirements for querying and sharing context (as discussed in section 1.2 and 2.4) between entities in the IoT environment, we propose a novel query language called CDQL. As mentioned earlier, CDQL consists of two main parts, Context Query Language (CQL) and Context Definition Language (CDL) that will be described in the rest of this section.

#### 3.4.1 CONTEXT QUERY LANGUAGE (CQL)

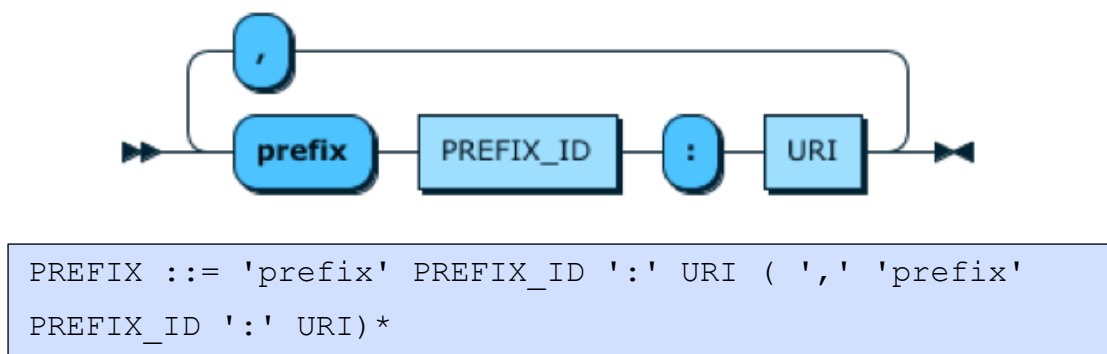
In this section we will present the conceptual model and syntax of our proposed Context Query Language (CQL). Figure 3.6 presents the production rule and highlights the core elements of this language.



**Figure 3.6** - CQL production rule

As the figure shows, CQL has three mandatory clauses, which are PREFIX, SELECT, and DEFINE; and two optional clauses, namely SUBSCRIPTION and SET. In the rest of this section, the details of each of these elements will be discussed. We will use an example to explain the syntax of CQL. The example under consideration expresses a query to find parking facilities with certain characteristics near a specific location.

A CQL query starts with a prefix clause. The prefix clause is responsible for identifying the semantic vocabularies that are used in a query to facilitate interoperability (Requirement 2). Using semantic vocabularies provides an easy and unambiguous way for a CQL developer to present their context queries. Further, it helps CMPs to understand the information requested in a query and provide richer results.



**Figure 3.7** - PREFIX clause production rule

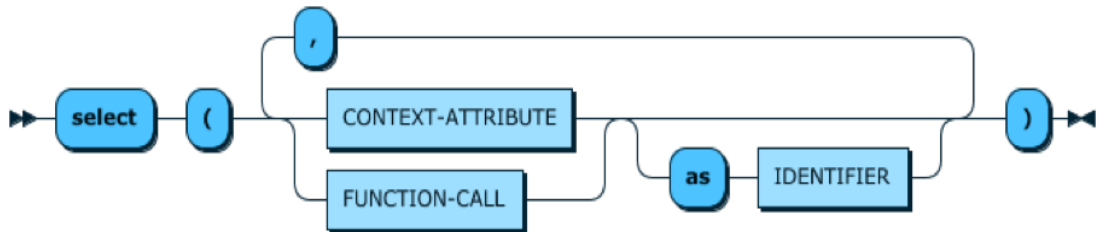
As it is illustrated in **Figure 1.1**, a prefix clause consists of two parts, a prefix id and a URI, which are separated by a colon. The prefix id assigns an identifier to a semantic vocabulary that will be used when it is needed to refer to it, and the URI refers to a semantic vocabulary. A CQL query can contain several semantic vocabularies separated by a comma. The following code block represents an example of PREFIX clause for the aforementioned parking query.

```
prefix mv:http://mobivoc.org,
prefix schema:http://schema.org
```

**Code block 3.1** - Example of PREFIX clause

The second mandatory clause of CQL is SELECT. This clause determines the query response structure. As shown in Figure 3.8 - , each context query can return a set of values as the query result, where each value can be represented as either a CONTEXT-ATTRIBUTE or a FUNCTION-CALL.

A CONTEXT-ATTRIBUTE represents a feature of an entity. This element consists of two parts: CONTEXT-ENTITY-ID and IDENTIFIER. The CONTEXT-ENTITY-ID identifies the entity which the context attributes will be queried from. The value for this element can be any of the entities known to the IoT ecosystem. We provide a mechanism to define such entities through the DEFINE clause, which is explained later in this section. The IDENTIFIER determines the type of context we are interested in, such as temperature, noise level, or any other type. Furthermore, it is possible to retrieve all the available attributes of an entity by using an asterisk (\*) wildcard.



```
SELECT ::= 'select' '(' ( CONTEXT-ATTRIBUTE |
CONTEXT-ENTITY | FUNCTION-CALL ) ( 'as' IDENTIFIER )? (
', ' ( CONTEXT-ATTRIBUTE | CONTEXT-ENTITY | FUNCTION-
CALL ) ( 'as' IDENTIFIER )? )* '
```

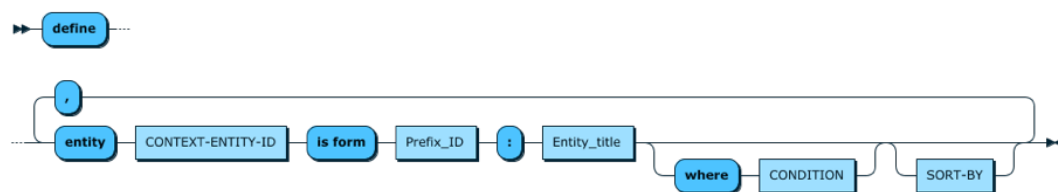
**Figure 3.8** - SELECT clause production rule

The second possible element in the SELECT clause is a FUNCTION-CALL. This element allows querying high-level context, which is one of the requirements (Requirement 4) of a context query language. In CQL, reasoning and aggregation

techniques are encapsulated as functions, referred to as CONTEXT-FUNCTION. A detailed explanation of CONTEXT-FUNCTIONs is provided in the next section. CONTEXT-FUNCTIONs can be easily integrated into a query using the FUNCTION-CALL statement. The FUNCTION-CALL has four components: PACKAGE-TITLE, FUNCTION-NAME, ARGUMENT, and IDENTIFIER. A PACKAGE-TITLE is an optional element that will only be used when the user wants to access a function defined inside a package. In this case, it is required to identify the namespace that the function belongs to. On the other hand, a FUNCTION-NAME is a mandatory module and determines the context function that needs to be applied to a set of arguments. The function's argument can be a CONTEXT-ATTRIBUTE, a CONTEXT-ENTITY, or a FUNCTION-CALL. Code block 3.2 represents an example of a PREFIX clause for the parking query. The first argument in this example is targetCarpark.\*, which represents all the available attributes of an entity with 'id' equals to targetCarpark. The second argument is a FUNCTION-CALL that is used to calculate the walking distance between the selected car parks and the driver's destination.

```
select (targetCarpark.*, distance(targetCarpark,
destinationLocation.geo , 'walking'))
```

**Code block 3.2** - Example of SELECT clause



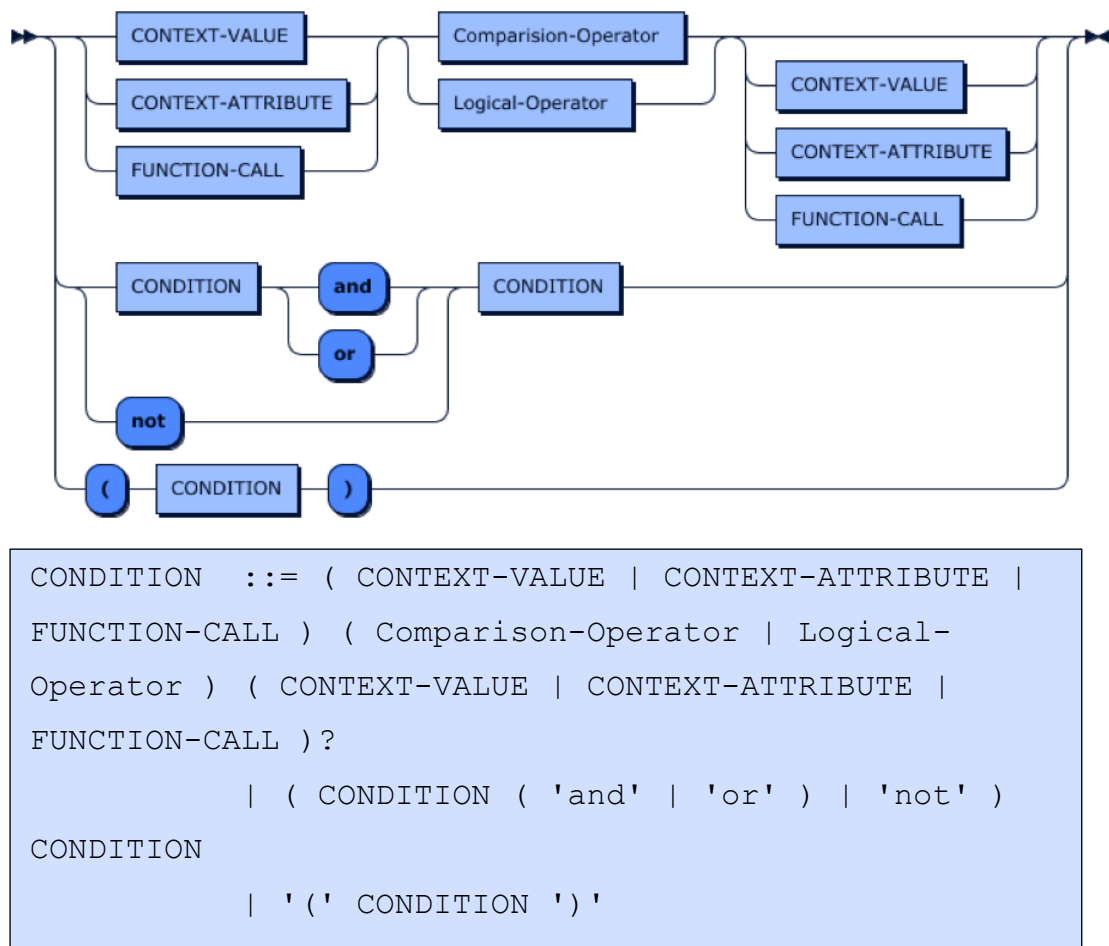
```
DEFINE ::= 'define' 'entity' CONTEXT-ENTITY-ID 'is
from' Prefix_ID ':' Entity_title ( 'where' CONDITION
)? SORT-BY? ( ',' 'entity' CONTEXT-ENTITY-ID 'is from'
Prefix_ID ':' Entity_title ( 'where' CONDITION )?
SORT-BY? ) *
```

**Figure 3.9** - DEFINE clause production rule

The last mandatory element of CQL is the DEFINE clause, which is represented in Figure 3.9. This clause allows querying contextual information from multiple entities (Requirement 1) by identifying the entities (one or several) that are involved in a query. In CQL, each entity is represented using four elements, CONTEXT-ENTITY-ID, ENTITY-TYPE, CONDITION, and SORT-BY.

The CONTEXT-ENTITY-ID assigns a name to an entity, which will be used when referring to the entity (e.g. in the SELECT clause).

The ENTITY-TYPE defines the type of an entity (e.g. car, parking facility, or a smart home) and consists of two parts, the PREFIX-ID that refers to a semantic vocabulary defined in PREFIX section, and a title, which represents the exact entity.

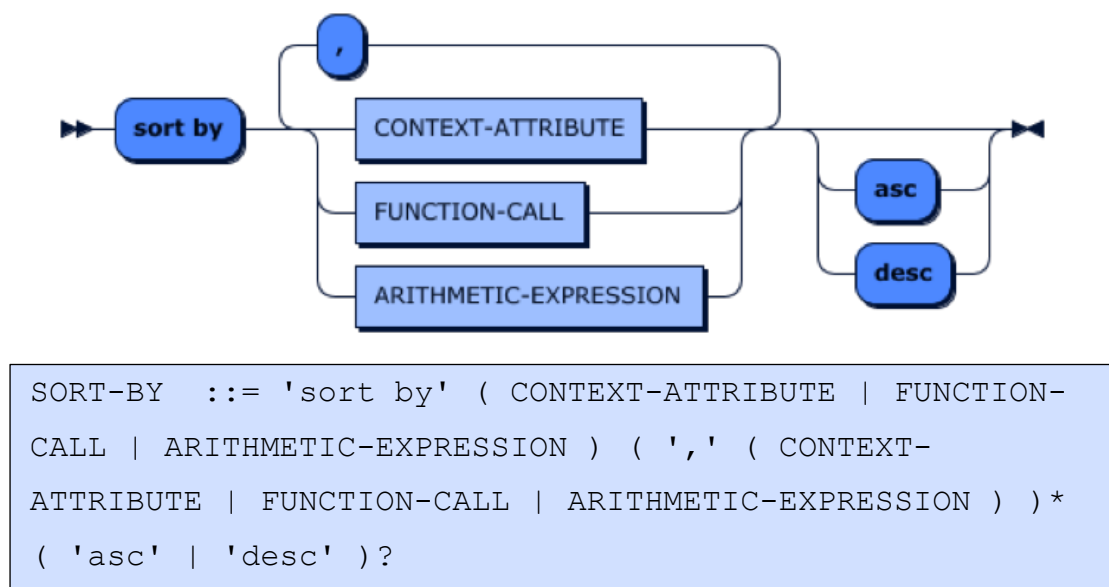


**Figure 3.10** - Condition clause production rule

The **CONDITION** clause provides a guideline on how to filter out unwanted context entities from a large number of available entities. The **CONDITION** allows



representing compound predicates that consist of several constraints connected by logical operators (AND/OR). These constraints define characteristics of the entity of interest. A constraint can be applied either to low-level context (CONTEXT-ATTRIBUTE), high-level context (FUNCTION-CALL), meta-data about context (e.g. freshness), or a simple value represented as a string or number. Furthermore, it is possible to combine multiple conditions into a compound condition by using the AND and OR operators. Figure 3.10 shows the production rule of the CONDITION clause. Please note self-referencing is used in this figure to represent compound conditions.



**Figure 3.11** - SORT-BY clause production rule

Lastly, the SORT-BY clause is used to sort the retrieved entities in ascending or descending order. The syntax of this clause is presented in Figure 3.11. As this figure shows, this clause allows users to sort the result of each context request based on one or more values, where values can be either a CONTEXT-ATTRIBUTE, a FUNCTION-CALL, or an ARITHMETIC-EXPRESSION.

An example of DEFINE clause based on the parking query is shown in the Code block 3.4. This example consists of two entities, “destinationLocation” that identifies the destination location of the driver and “targetCarpark” that represents parking facilities with specific characteristics based on user preferences. As this example shows, attributes of one entity can be used in the definition of another entity.

```

define
entity destinationLocation is from schema:place
where
destinationLocation.address = "Monash University
Clayton Campus, 40 Exhibition Walk, Clayton VIC 3800",
entity targetCarpark is from mv:ParkingGarage
where
distance(targetCarpark, destinationLocation.geo ,
"walking") < {"@type":"shema:QuantitativeValue",
"value": 500, "unitCode":"m"}
and
targetCarpark.chargingPoint.charger.powerInkW > 10
and
targetCarpark.chargingPoint.charger.threePhasedCurrentA
available = true
and
targetCarpark.chargingPoint.charger.plug.plugType
containsAny ["EUDomesticPlug", "CHAdemo", "ShukoPlug"]
sort by
distance(targetCarpark, destinationLocation.geo,
"walking")

```

### Code block 3.3 - Example of DEFINE clause

So far, we introduced all the mandatory clauses of CQL. Using these clauses, a context consumer can issue complex context queries concerning various context entities and constraints, which will be executed only once immediately after the query has been issued. We refer to these type of queries as pull-based queries. Code block 3.4 presents the full example of a pull-based query, which will be issued to retrieve all the available parking with specific characteristics close to a specific location.

```

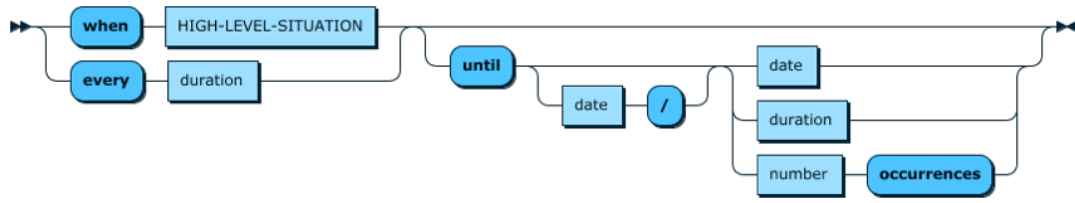
prefix mv:http://mobivoc.org , prefix
schema:http://schema.org
select (targetCarpark.*, distance(targetCarpark,
destinationLocation.geo , 'walking'))
define
entity destinationLocation is from schema:place where
destinationLocation.address = "Monash University
Clayton Campus, 40 Exhibition Walk, Clayton VIC 3800",
entity targetCarpark is from mv:ParkingGarage where
distance(targetCarpark, destinationLocation.geo ,
"walking") < {"@type":"schema:QuantitativeValue",
"value": 500, "unitCode":"m"}
and targetCarpark.chargingPoint.charger.powerInkW > 10
and
targetCarpark.chargingPoint.charger.threePhasedCurrentA
vailabile = true
and targetCarpark.chargingPoint.charger.plug.plugType
containsAny ["EUDomesticPlug", "CHAdemo", "ShukoPlug"]
sort by distance(targetCarpark, destinationLocation.geo
, "walking")

```

### Code block 3.4 - Example of a pull-based query

As mentioned earlier, a common requirement in many context-aware IoT applications is to monitor IoT entities, discover situation changes, and adjust to them automatically. Therefore, we introduced the SUBSCRIPTION clause to address this requirement (Requirement 5). The SUBSCRIPTION clause supports the representation of periodic (e.g. check the temperature of a room every 10 minutes) and event/situation-based (e.g. when the temperature is more than 10 °C) context queries. Using this clause, a context consumer can receive periodic updates about the real-time state of an entity or subscribe to a specific situation. The result of the query will be sent back to the consumer asynchronously when the defined situation is detected. We refer to such queries as PUSH-based queries. In CDQL, to represent situations, we designed a specific syntax that supports rule-based reasoning, uncertainty handling, temporal

relations, and windowing functionality. The syntax will be explained in the next section.



```

SUBSCRIPTION      ::= ( 'when' HIGH-LEVEL-SITUATION |
                        'every' duration ) ( 'until' date '/'? ( date |
                        duration | number 'occurrences' ) )?

```

**Figure 3.12** - SUBSCRIPTION clause production rule

The syntax of the SUBSCRIPTION clause is depicted in Figure 3.12. As this figure shows, the SUBSCRIPTION clause consists of either a WHEN or EVERY statement. Furthermore, it has an optional statement that is called UNTIL.

The EVERY statement is designed to represent periodic queries by identifying the sampling interval for a context query. This statement starts with the ‘every’ keyword followed by a string which represents the sampling interval. To represent sampling intervals (i.e. duration) in CQL, we adopted ISO 8601 standard that provides a standard way to specify the amount of intervening time in a time interval in the format P[n]Y[n]M[n]DT[n]H[n]M[n]S[n]MS. In this format, [n] is replaced by the value for each of the date and time elements that follow the [n]. The capital letters P, Y, M, W, D, T, H, M, S and MS are designators for each of the date and time elements. For example, "P1Y2M6DT8H7M15S20MS" represents a duration of "one year, two months, six days, eight hours, seven minutes, fifteen seconds, and twenty milliseconds". Date and time elements including their designator may be omitted if their value is zero. Lower order elements may also be omitted for reduced precision. An example of a basic push-based query with an EVERY statement is provided in the following code snippet. By issuing this query, the subscribed context consumer will receive updates (i.e. every five minutes) about the temperature of a specific location.

```

prefix schema:http://schema.org
select (destinationLocation.weather.airTemperature)
every pT5M
define
entity destinationLocation is from schema:place where
destinationLocation.address = "Monash University
Clayton Campus, 40 Exhibition Walk, Clayton VIC 3800"

```

### Code block 3.5 - Example of a basic push-based

The WHEN statement is the enabling element for situation-based queries. This statement starts with the ‘when’ keyword followed by a situation definition, which is expressed in a HIGH-LEVEL-SITUATION statement. Using this element, an IoT application can define and monitor their situations of interest. The HIGH-LEVEL-SITUATION statement is fully discussed in the next section. The following query is an example of a CQL query with a WHEN clause. This query expresses a request for monitoring a specific parking spot that a car is driving to and suggests alternative car parks as soon as the situation “isFull” for the given carpark becomes true.

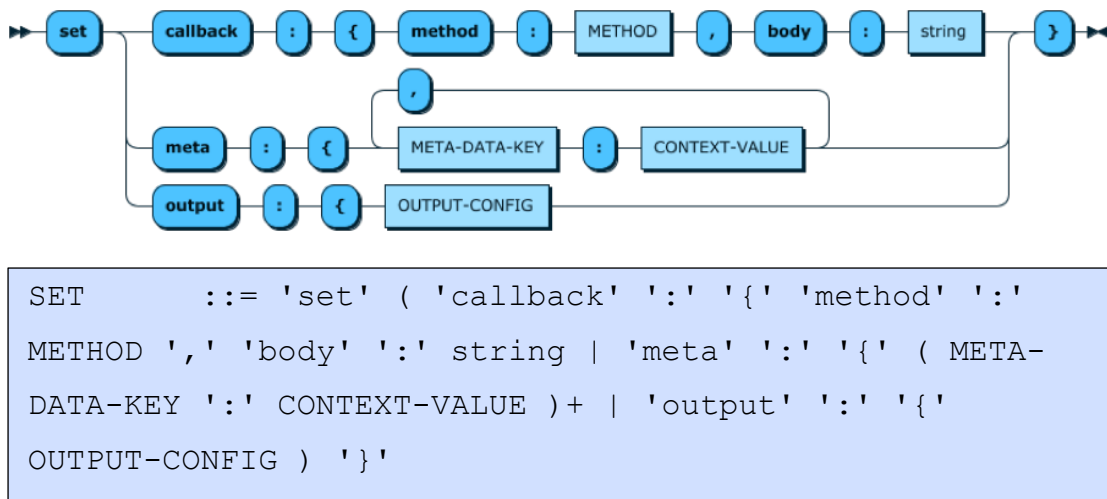
```

prefix mv:http://schema.mobivoc.org
select (targetCarpark.*)
when isFull(selectedParking, car, event) > 0.80
define
entity selectedParking is from mv:ParkingFacility
where
selectedParking.id = 'parking 1',
entity destinationLocation is from schema:place
where
destinationLocation.address = "Monash University
Clayton Campus, 40 Exhibition Walk, Clayton VIC 3800",
entity targetCarpark is from mv:ParkingGarage where
...

```

### Code block 3.6 - Example of using WHEN clause in a CDQL query

Lastly, the UNTIL statement indicates the timespan of the context retrieval by defining queries' lifetime. As Figure 3.12 shows, the UNTIL statement provides three options to determine the query lifetime: the first option is to provide a DateTime struct to indicate the expiry date and time of a query, the second option is to provide the duration of subscription, and the last option is to provide the number of occurrences of query executions before it becomes deactivated. Furthermore, this statement can express the activation date and time of a subscription. In CQL, the DateTime struct is based on ISO 8061 standard and represented as "yyyy-mm-ddThh:mm:ss[.mmm]" (e.g. "2019-06-15T08:28:38").



**Figure 3.13** - SET clause production rule

The last clause of CQL is the SET clause, which is illustrated in Figure 3.13. This clause consists of three elements, namely CALLBACK, META, and OUTPUT.

The CALLBACK clause identifies how the result of queries should be sent back to the context consumers. This clause describes the callback method (e.g. HTTP Post) and other required fields (e.g. Callback URL and headers). Further, this clause provides a mechanism to define the body of the message that will be sent back to the subscribed context consumer. As it is shown in Figure 3.13, the value for the 'body' attribute is a string, which can represent any custom messages in any format (e.g. JSON, XML, plain text, or others). Moreover, it is possible to include any of the retrieved contextual information in the body string by using the '\$' prefix, i.e. "\$CONTEXT-

ATTRIBUTE". If the 'body' attribute is not provided, all the entities and attributes defined in the select clause will be used as the message's body. An example of using the CALLBACK clause is provided in Code block 3.7.

```
prefix schema:http://schema.org
select (events.*) when
timeDifference(events.startDate,currentTime("Australia
/Melbourne")) -
distance(car.geo,events.geo,"DRIVING").duration <
{"value":"30","unit":"minutes"}
define entity events is from schema:event where
events.attendee.email="biotope2018.au@gmail.com",
entity car is from schema:Vehicle where
car.vehicleIdentificationNumber = "9d791e4d-8181",
set callback : {"method":"post",
url:"http://138.194.106.20","headers":{"ContentType":
"application/json" }}
```

**Code block 3.7 - Example of push-based query with CALLBACK clause**

The CALLBACK clause can be used for both push-based and pull-based queries. In the case of pull-based queries, it will allow context consumers to issue non-blocking queries and receive the result as soon as the execution of a query is finished. Regarding push-based queries, when the callback clause is presented, the result of the query will be pushed back into the subscribed entity as soon as the related situation is detected. When the callback is not provided, the result of the query will be temporarily stored, and the context consumer can pull the data by issuing a query similar to the following code snippet, which indicates the subscription id.

```
prefix coaas:http://coaas.csiro.au/schema
select (subs.*)
define
entity subs is from coaas:subscription where subs.id =
'subscription1'
```

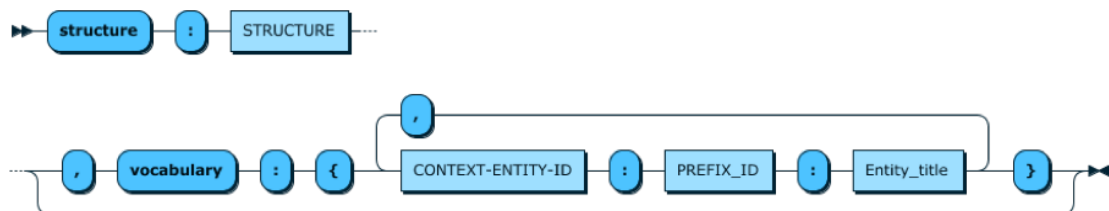
**Code block 3.8 - Example of querying the results of subscriptions**

The META clause enables another essential requirement for a context query language, which is expressing different aspects of context, such as imperfectness, uncertainty, QoC, and CoC (Requirement 6). In other words, this clause allows users to set the minimum acceptable (or default) value for each metadata. For example, the following code block indicates that the minimum acceptable freshness for each context attribute is 100ms and the total cost of query should be less than 50 cents.

```
Set meta : {
    "freshness" : "T100ms",
    "cost" : {"value":0.50, "unit":"aud"}
}
```

**Code block 3.9** - Example of META clause

Lastly, CQL allows developers of context query to define their preferred structure of output through the OUTPUT clause. The production rule of the OUTPUT clause is depicted in Figure 3.14. As it is shown in this figure, the output clause consists of two main elements, a STRUCTURE that identifies the output data structure (e.g. XML, JSON, or ODF), and a vocabulary that specifies which semantic vocabulary should be used for each context-entity.



```
OUTPUT-CONFIG ::= 'structure' ':' STRUCTURE ( ','
'vocabulary' ':' '{' CONTEXT-ENTITY-ID ':' PREFIX_ID
':' Entity_title ( ',' CONTEXT-ENTITY-ID ':' PREFIX_ID
':' Entity_title ) * '}' ) ?
```

**Figure 3.14** - OUTPUT-CONFIG clause production rule

In order to express the grammar of CQL, we used Extended Backus–Naur Form (Wirth, 1996)(EBNF). The full grammar of CQL is represented in Appendix A.



### 3.4.2 CONTEXT DEFINITION LANGUAGE (CDL)

As mentioned earlier, the reasoning and aggregation functionalities are supported in CQL through the notion of function. CDQL offers a rich set of built-in context-functions that can be easily integrated into context queries through a FUNCTION-CALL. Some of the most important CQL built-in functions are presented in Table 3.3.

**Table 3.3** - CQL built-in functions

Function Title	Details
Max(argument, [window <sup>2</sup> ])	Returns the maximum value of a given argument. If the window is provided, the value will be calculated during the provided window.
Min(argument, [window])	Returns the minimum value of a given argument. If the window is provided, the value will be calculated during the provided window.
Sum(argument, [window])	Returns the total sum of a given argument. If the window is provided, the value will be calculated during the provided window.
Average(argument, [window])	Returns the average of a given argument. If the window is provided, the value will be calculated during the provided window.
SD(Ca, [window])	Returns the standard deviation of a given argument during the provided window.
Count(argument, [window])	Returns the number of times the value of a given argument has been updated. If the window is provided, the value will be calculated during the provided window.
Increased(argument, window)	Returns true when the value of a given attribute increased during the provided window.

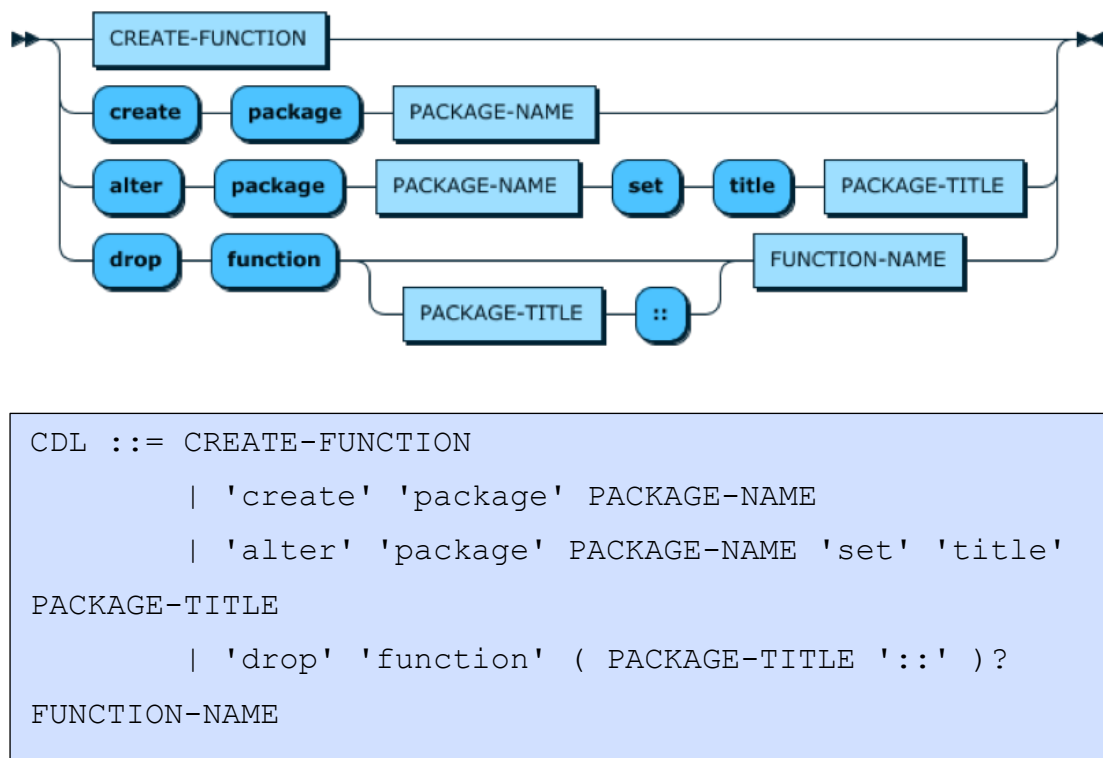
---

<sup>2</sup> Window identifies a limited subset of context attributes which the function will be applied to. Refer to Section 3.4.2.2 for more details.

Decreased(argument, window)	Returns true when the value of a given attribute decreased during the provided window.
isValid(argument, window)	Returns true when the value of a given attribute is unchanged during the provided window.
change(argument, [value],[window])	<p>Returns true when the value of a given attributes changes. If the value is provided, returns true only if the value of the given attribute changes to the provided value. In all the other cases returns False.</p> <p>If the window is provided, the value will be calculated during the provided window.</p>
Distance(origin, destination,[transport_type])	Returns a JSON result which contains the Euclidean distance between the origin and destination. If the transport_type is provided, returns the travel distance and time for a given origin and destination, based on the recommended route between start and end points considering the travel mode. The following travel modes are supported: driving, walking, bicycling, and transit.
Intersect(Geo-shape*, Geo-shape*)	Allows you to compare two geospatial types to see if they intersect or overlap each other.
SpatioTemporalIntersect(Route*, Route*)	Returns true if the provided routes have an intersection considering both location and time.
Within(Geo-shape*, Geo-shape*)	Returns true if the first geo-shape is inside the second Geo-shape.

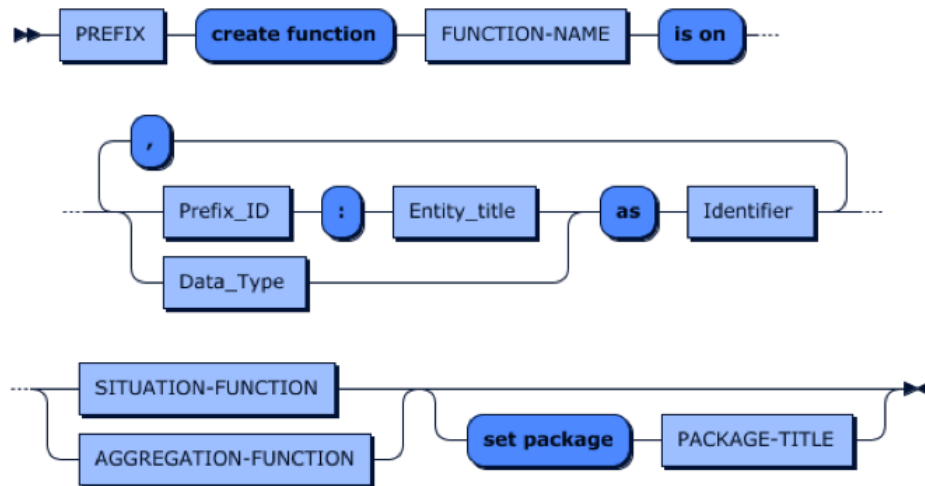
\* All the Geo-shapes can be represented either by GeoJSON format or Well-known text markup language.

While built-in functions are sufficient for most common use cases, we believe it is mandatory for a CQL to support the definition of custom functions (Requirement 5), as these functions are usually application dependent and predefining a comprehensive list of them is not possible. As a result, we introduce the CREATE-FUNCTION clause in CDL to define aggregation and reasoning functions dynamically as part of the CDQL language.



**Figure 3.15** - CDL production rule

Figure 3.15 shows the CDL production rule. As depicted in this figure, CDL allows context query developers to create and remove CONTEXT-FUNCTIONS. Further, it has three statements to create, alter, and drop packages. In general, packages in CDL are designed to organise functions and prevent function name collisions. Since the syntax of most statements in CDL are quite self-explanatory, except for CREATE-FUNCTION. Hence, in the rest of this section, we will focus on explaining the details of the CREATE-FUNCTION statement.



```

CREATE-FUNCTION ::= PREFIX 'create function'
FUNCTION-NAME 'is on'
(Prefix_ID ':' Entity_title | Data_Type ) 'as'
Identifier
( ',' ( Prefix_ID ':' Entity_title | Data_Type )
'as' Identifier ) *
( SITUATION-FUNCTION | AGGREGATION-FUNCTION )
( 'set package' PACKAGE-TITLE ) ?

```

**Figure 3.16 - Create function production rule**

Figure 3.16 highlights the syntax of the CREATE-FUNCTION statement. As this figure shows, the CREATE-FUNCTION statement starts with a PREFIX clause, which identifies the semantic vocabularies used in the definition of the function's parameters. It is followed by the 'create function' keyword and the FUNCTION-NAME construct that assigns a title to a context function and makes it accessible via this title.

The next keyword in the CREATE-FUNCTION statement is 'is on', which together with the PARAMETER-DEFINITION construct specifies the input parameters of a context function. This construct supports the definition of two types of parameters, which are CONTEXT-ENTITY and data type. The supported data types in CDL are Number, Date, Time, DateTime, String, Array, and Object. Further, the PARAMETER-DEFINITION construct assigns an id to each parameter using the 'as' keyword. In the FUNCTION-CALL statement, these parameters can be a CONTEXT-ENTITY, a CONTEXT-ATTRIBUTE, a FUNCTION-CALL, a literal value, or an

expression, for example, it could be the arithmetic expression like '5\*8' or 'parking.priceSpecification.price \* meeting.duration' where 'parking' and 'meeting' are context entities.

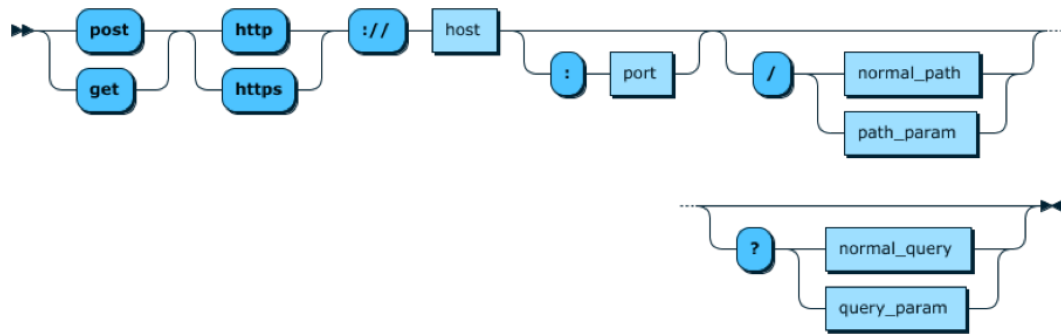
After defining the signature of a function, the body of context function is constructed using either the SITUATION-FUNCTION construct or the AGGREGATION-FUNCTION construct. The details and syntax of these constructs is discussed in the rest of this section.

The last construct in the CREATE-FUNCTION statement is SET-PACKAGE. SET-PACKAGE is an optional construct and allows specifying the package to contain the function. If SET-PACKAGE is omitted, the context function will be placed into a default package, which has no name.

#### **3.4.2.1 Aggregation Function**

As mentioned earlier, aggregation functions are usually application dependent, and it is not feasible to define all possible functions for all domains in advance. As a result, CDQL supports definition of custom aggregation functions. In CDL, aggregation functions can be expressed in two different approaches.

The first approach is to provide aggregation functions through Restful API calls. This approach allows CDQL developers to register custom RESTful methods and use them in their context queries. The syntax of API-based aggregation functions construct can be divided into two sections. The first section of this construct expresses the endpoint of a Restful method by indicating the method type (i.e. get or post), the protocol (i.e. http or https), host address, and port number (if required). The second section, which consists of path parameters and query parameters, specifies the method of interest and its parameters. The production rule of the API-based function is presented in Figure 3.17. As this figure shows, functions can have several paths and query parameters, where each of them might be either a literal or one of the parameters defined in the PARAMETER-DEFINITION section. To distinguish parameters from literal, parameters are indicated by the dollar sign and curly braces ( $\{\text{car.speed}\}$ ).



```

API-AGGREGATION-FUNCTION ::= ( 'post' | 'get' ) (
  'http' | 'https' ) '://' host ( ':' port )? ( '/' (
    normal_path | path_param ) )? ( '?' ( normal_query |
    query_param ) )?

```

**Figure 3.17** - API-based aggregation functions

It is worth mentioning that if a method of an API-based aggregation function is set to ‘post’, all the parameters defined in the `PARAMETER-DEFINITION` section will be sent to the provided URI as a JSON object.

The following example shows a `CREATE-FUNCTION` statement that registers one of the Google maps’ APIs. This API takes up to 100 GPS points collected along a route and returns a similar set of data with the points snapped to the most likely roads the vehicle was travelling along.

```

create function snap2Roads
is on
string as path,
Boolean as interpolate
get
https://roads.googleapis.com/v1/snapToRoads?path=${path}
&interpolate=${interpolate}
set package google

```

**Code block 3.10** - Example of `CREATE-FUNCTION` clause

The two main advantages of defining custom aggregation function as APIs are high reusability and ease of development. However, this approach might lead to a performance issue during query execution, especially when the volume of data that needs to be passed to the third-party APIs becomes large. Hence, to mitigate the performance issue in this type of use-cases, we introduced the second approach of defining custom aggregation functions. In this approach, CDQL developers can implement their custom aggregation functions using a scripting language, such as JavaScript or Python. This approach potentially has better performance compared to the first approach since the script will be executed locally (in the CMP) and there will be no communication overhead. The code snippet below shows the implementation of the VARIANCE aggregation functions using the JavaScript language.

```
create function variance
is on
array as values
{
    var squared_Diff = 0;
    var total = 0;
    for(var i = 0; i < values.length; i++) {
        total += values[i];
    }
    var mean = total / values.length;
    for(var i = 0; i < values.length; i++)
    {
        var deviation = values[i] - mean;
        squared_Diff += deviation * deviation;
    }
    var variance = squared_Diff/(values.length);
    return variance;
} set package math
```

**Code block 3.11** - Example of creating a custom aggregation function using JavaScript

### 3.4.2.2 Situation Function

In this section, we illustrate how SITUATION-FUNCTIONs are represented in CDL. First, we describe the situation model that serves as a basis for the definition of situations in CDL. Then, we explain the syntax of SITUATION-FUNCTION statement.

In CDQL, the situation representation and modelling are based on the Context Spaces Theory (CST) model (Padovitz, Loke, & Zaslavsky, 2004) with some modifications and extensions to tailor our requirements.

The central notion in CST is the concept of situations. The CST model represents situations as geometrical objects in multidimensional space (Padovitz et al., 2004). Such a geometrical object is called a situation space. A situation space is a tuple of regions of attribute values related to a situation. Each region is a set of accepted values for an attribute based on a predefined predicate. For example, consider a situation labelled as ‘Good for Walking’ which indicates that the walking path from a suggested carpark location to the driver’s destination is good for walking or not. This situation space can be characterised using several context attributes such as temperature, rain intensity, snow intensity, time of the day, the safety of the area, health status of a driver, age, etc. Further, the acceptable regions of values for each context attribute should be defined, e.g., the lower and upper bounds of temperature.

In addition to basic concepts and techniques for situation modelling and reasoning, the CST model provides heuristics developed specifically for addressing context-awareness under uncertainty. These heuristics are integrated into reasoning techniques to compute the confidence level of the occurrence of a situation (Padovitz, Loke, Zaslavsky, Burg, & Bartolini, 2005). One of the main heuristics of the CST model is considering individual significance (weight) of each attribute. Weights are values from 0 to 1 assigned to every context attribute, and they represent the importance of each attribute in a situation, with a total sum of one per situation. In a simplified version of the example, only considering temperature, rain intensity, and safety of the area, the values 0.1, 0.3, and 0.6 can be assigned to these attributes respectively.



Moreover, CST assigns a contribution value to each region that indicates its level of participation in the occurrence of the situation. Back to our previous example, the regions and their confidence for the temperature attribute could include:

$$Contribution_{temp} = \begin{cases} 0.05 & \text{Less than } (-)5C \\ 0.6 & \text{Between } (-)5C \text{ and } 6C \\ 1 & \text{Between } 6C \text{ and } 26C \\ 0.6 & \text{Between } 26C \text{ and } 36C \\ 0.05 & \text{More than } 36C \end{cases}$$

Based on the discussion above, in CST, the confidence in the occurrence of a whole situation is defined as:

$$Confidence = \sum_{i=1}^n w_i * C_i$$

Where  $w_i$  represents the weight of a particular context attribute and  $C_i$  stands for the contribution of the range to which the value of attribute ‘ $i$ ’ belongs to.

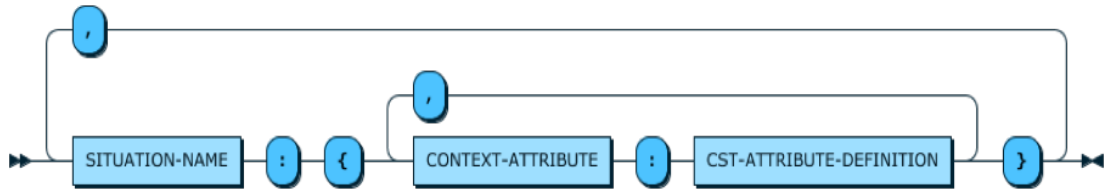
Another way to represent situations is to combine several already inferred situations. However, the sequence of occurrences of such situations might play a role in situation inference. For example, a situation ‘S’ can be considered to be happening if the situation ‘A’ happens before the situation ‘B’, but not if ‘B’ happens before ‘A’. This type of dependence is called ‘temporal relation’, and it is essential to include this feature in the situation description model. Since this feature is not directly supported in CST, we adopt Allen’s interval algebra (Allen, 2013; Mavrommatis, Artikis, Skarlatidis, & Paliouras, 2016) to enable the representation of such relations.

Furthermore, a situation can be defined as a generalisation of similar events over a certain period of time. In other words, situation A can be described as: “Situation A is happening if a particular sensor reading was in the range between X and Y during the last 30 minutes”. In this example, the “during the last 30 minutes” is an implicit usage of a common technique for data stream processing - a sliding window. A window can be defined as “a mechanism for adjusting flexible bounds on the unbounded stream in order to fetch a finite, yet ever-changing set of tuple”(Patroumpas & Sellis, 2006).

Similar to the temporal relationships, the windowing is not supported in CST. Therefore, in order to support this functionality, we integrated four types of windows into the situation description model, namely (i) sliding window, (ii) tumbling window, (iii) hopping window, and (iv) eviction window. Until now, we covered the core concepts that form the foundation of situation description in CDL. In the rest of this section, we will present the syntax of Situation Description Statement (SDS). SDS provides two statements for describing situations, namely the CST-SITUATION statement and the HIGH-LEVEL-SITUATION statement.

The CST-SITUATION statement is based on Context Spaces Theory (CST) and describes situations in terms of their related context attributes combined with acceptable regions of values for each attribute.

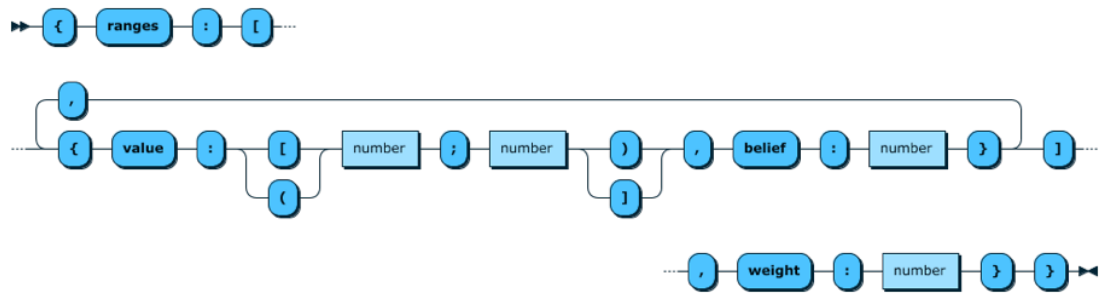
Figure 3.18 shows the syntax of CST-based situation description. As illustrated in the figure, a CST-SITUATION statement can have several situations, where each situation starts by assigning a name to it. In the next part, all the involved CONTEXT-ATTRIBUTEs and their corresponding values, which define the characteristics of the situation, should be listed.



```
CST-SITUATION ::= SITUATION-NAME ':' '{' CONTEXT-
ATTRIBUTE ':' CST-ATTRIBUTE-DEFINITION ( ',' CONTEXT-
ATTRIBUTE ':' CST-ATTRIBUTE-DEFINITION ) * '}' ( ','
SITUATION-NAME ':' '{' CONTEXT-ATTRIBUTE ':' CST-
ATTRIBUTE-DEFINITION ( ',' CONTEXT-ATTRIBUTE ':' CST-
ATTRIBUTE-DEFINITION ) * '}' ) *
```

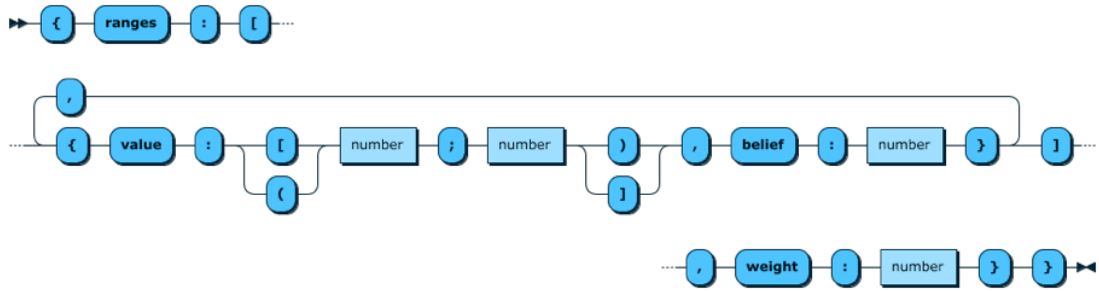
**Figure 3.18** - CST-SITUATION statement production rule

The value of CONTEXT-ATTRIBUTES is represented by the CST-ATTRIBUTE-DEFINITION construct that can be seen in Figure



```
CST-ATTRIBUTE-DEFINITION ::= '{' 'ranges' ':' '[' '{'
'value' ':' ( '[' | '(' ) number ';' number ( ')' | ']'
) ',' 'belief' ':' number '}' ( ',' '{' 'value' ':' (
 '[' | '(' ) number ';' number ( ')' | ']' ) ','
'belief' ':' number '}' )* ']' ',' 'weight' ':' number
'}' '}'
```

**Figure 3.193.19.** This construct has two elements, ‘ranges’ and ‘weight’. The ‘ranges’ defines the acceptable regions for an attribute by indicating the exact range, and the value of ‘belief’ that indicates the level of participation of an attribute in the occurrence of a situation, when its value is within the indicated range. The ‘weight’ construct identifies the importance of an attribute in a situation by providing a numeric value between 0 and 1.



```

CST-ATTRIBUTE-DEFINITION ::= '{' 'ranges' ':' '[' '{'
'value' ':' ( '[' | '(' ) number ';' number ( ')' | ']'
) ',' 'belief' ':' number '}' ( ',' '{' 'value' ':' (
 '[' | '(' ) number ';' number ( ')' | ']' ) ','
'belief' ':' number '}' )* ']' ',' 'weight' ':' number
'}' '}'

```

**Figure 3.19** - CST-ATTRIBUTE-DEFINITION

The code snippet in Code block 3.12 shows an example of a situation function definition in CDL. This example expresses the aforementioned goodForWalking situation.

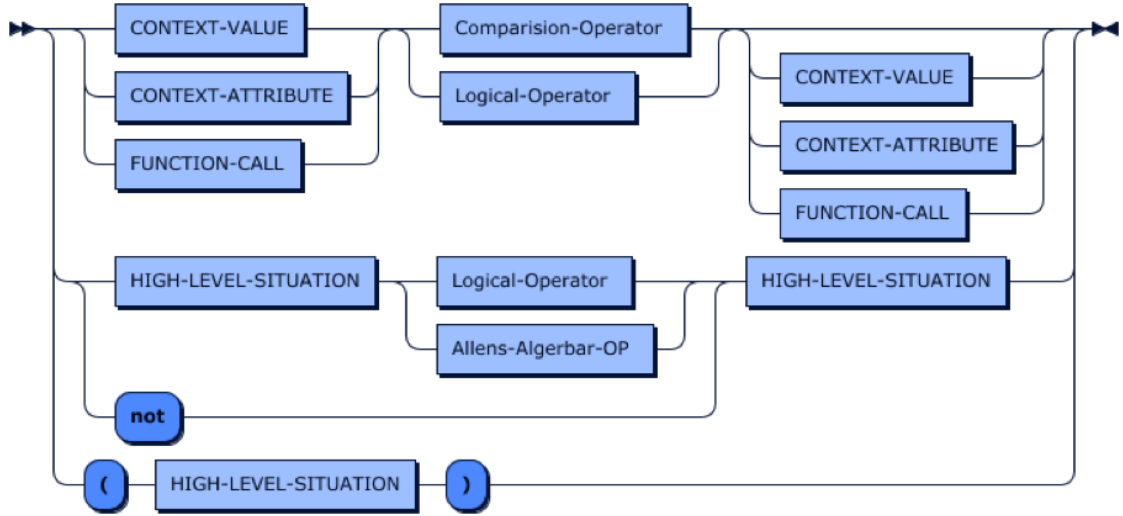
```

prefix schema:http://schema.org
create function weatherSituation is on
schema:weather as r1 {
    "goodForWalking" : {
        r1.airTemperature : {
            ranges : [
                { value:(0;6], belief : 20 } ,
                { value:(6;13], belief : 50 },
                { value:(13;28], belief : 100 } ,
                { value:(28;38], belief : 20 }
            ],
            weight : 10
        } ,
        r1.windSpeed : {
            ranges : [
                { value:(0;8], belief : 100 } ,
                { value:(8;20], belief : 50 },
                { value:(30;40], belief : 10 }
            ] ,
            weight : 5
        }
    }
}

```

**Code block 3.12** - Example of CST-based situation function definition

As mentioned earlier, the CST model does not support expressing situations that contain temporal relationships or window functions. Therefore, to express this kind of situations, we introduced the HIGH-LEVEL-SITUATION statement. This statement supports description of higher-level situations by describing the correlation of situations via temporal relationships and logical operators.





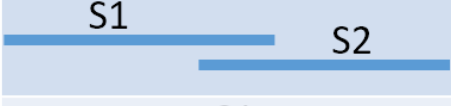
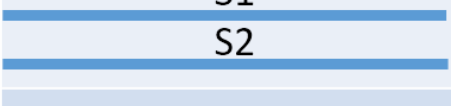

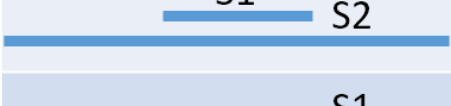
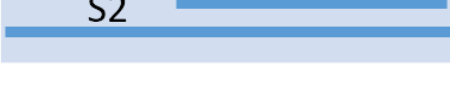
```

HIGH-LEVEL-SITUATION ::= ( CONTEXT-VALUE | CONTEXT-
ATTRIBUTE | FUNCTION-CALL ) ( Comparison-Operator |
Logical-Operator )
( CONTEXT-VALUE | CONTEXT-ATTRIBUTE | FUNCTION-CALL )?
    | ( HIGH-LEVEL-SITUATION ( Logical-Operator |
Allens-Algerbar-OP ) | 'not' ) HIGH-LEVEL-
SITUATION
    | ' ( ' HIGH-LEVEL-SITUATION ' ) '

```

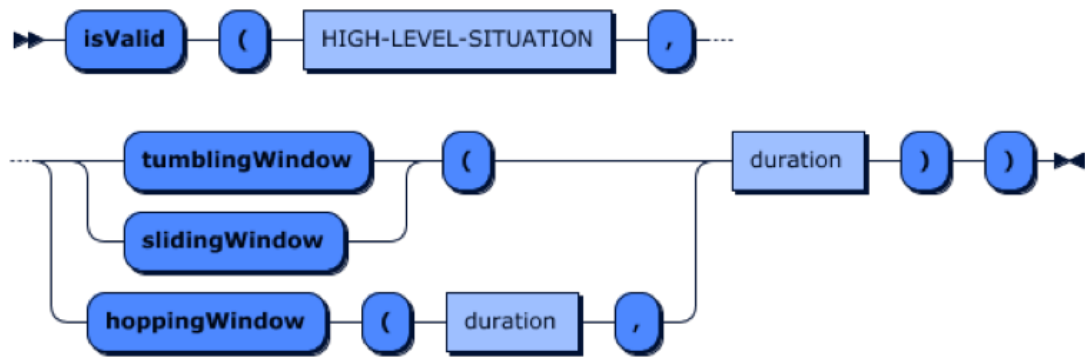
**Figure 3.20** - HIGH-LEVEL-SITUATION statement production rule

The production rule of this statement is presented in Figure 3.20. As shown in the figure, the syntax of the HIGH-LEVEL-SITUATION statement is very similar to the CONDITION clause, with the only difference that the former allows connecting two high-level-situations with temporal relationships operators. In CDL, we adopted seven operators from Allen's interval algebra, namely Before, Meets, Overlaps, Starts, During, Finishes, and Equals. The graphical representation of temporal relations between events is presented in Figure 3.21.

CDQL operator	Graphical representation
S1 BEFORE(e) S2 S2 AFTER(e) S1	
S1 MEETS(e) S2 S2 <u>iMEETS</u> (e) S1	
S1 OVERLAPS(e) S2 S2 <u>iOVERLAPS</u> (e) S1	
S1 EQUALS(e) S2	
S1 STARTS(e) S2	
S1 DURING(e) S2	
S1 FINISHES(e) S2	

**Figure 3.21** - Allen's algebra graphical representation (i stands for inverse)

Another concept that was mentioned earlier in this section is windowing. To enable a query to express the validity of a situation over time, we introduced a new built-in function – ‘isValid’. This function accepts a situation and a period of time as its inputs and returns the average confidence of occurrence of the given situation over a defined period. It enables both the possibility to access historical trajectory of the situation, and, also, a sliding, hopping, tumbling and eviction window functionality. The formal representation of using the ‘isValid’ operator is presented in Figure 3.22.



```
IS-VALID ::= 'isValid' '(' HIGH-LEVEL-SITUATION ',' ( (
'tumblingWindow' | 'slidingWindow' ) '(' |
'hoppingWindow' '(' duration ',' ) duration ')' )'
```

**Figure 3.22** - isValid function production rule

An example of using the ‘isValid’ operator for a real situation’s description is shown in Code block 3.13 line 5. This SDS describes a situation when a period of parking exceeds the allowed maximum duration.

```
prefix mv:schema.mobivoc.org
create function parkingTimeEnded is on
mv:parking as p1,
mv:car as c1
isValid(charIsParked(c1,p1),slidingWindow(p1.maxDurati
on))= true
```

**Code block 3.13** - Example of isValid function

Further, as shown in Table 3.3, CDQL is enhanced with a rich set of statistical functions that can be used to improve the expressiveness of the situation description of the language. These functions accept a context attribute and a window as its input and return statistical information as the output.



### 3.5 SUMMARY

In this chapter, we tackled the fundamental challenges in designing context management platforms, which is the need for a generic, flexible, and easy to use approach for publishing and querying context. To achieve this goal, we presented Context Service Description Language (CSDL) and Context Definition and Query Language (CDQL), which are used to describe the information provided by context providers and required by context consumers respectively.

The CSDL is an abstract service description language and supports the definition of context services in terms of their semantic signature, service characteristics, and contextual behaviour specification. Using this language, context providers can describe the semantics and structure of their context services and register them in CoaaS.

The CDQL aims to define and represent context entities and context requests for IoT applications, services, and systems. CDQL consists of two main parts namely: Context Query Language (CQL), which is a powerful and flexible query language to express contextual information requirements without considering the details of the underlying data structure, and Context Definition Language (CDL), which facilitates the description of high-level context and situations. CQL supports both pull- and push-based queries. One of the main features of this language is its ability to support and represent contextual functions, namely situation (high-level context) and aggregation functions, using CDL facility.

The CSDL and CDQL are key components of CoaaS that facilitate sharing context among heterogeneous IoT entities, namely context providers and context consumers.

On top of that, in this chapter, we presented the blueprint architecture of CoaaS platform, identified its major components, and briefly explained them. In the next chapter, we will focus on two of these components, Context Query Engine (CQE) and Situation Monitoring Engine (SME), and will show how they utilise CDQL and CSDL to allow context consumers to query and monitor the context published by context providers.

# Chapter 4: Context Query and Situation Monitoring Engines: Design and Implementation

---

In the preceding chapter, we presented the overview architecture of CoaaS platform, introduced its main components, and discussed their responsibilities. Furthermore, we proposed a novel mechanism for publishing and querying context, accomplished by two specially designed high-level languages, namely Contest Service Description Language (CSDL) and Context Definition and Query Language (CDQL). Using these languages, context consumers can query the information offered by the context providers.

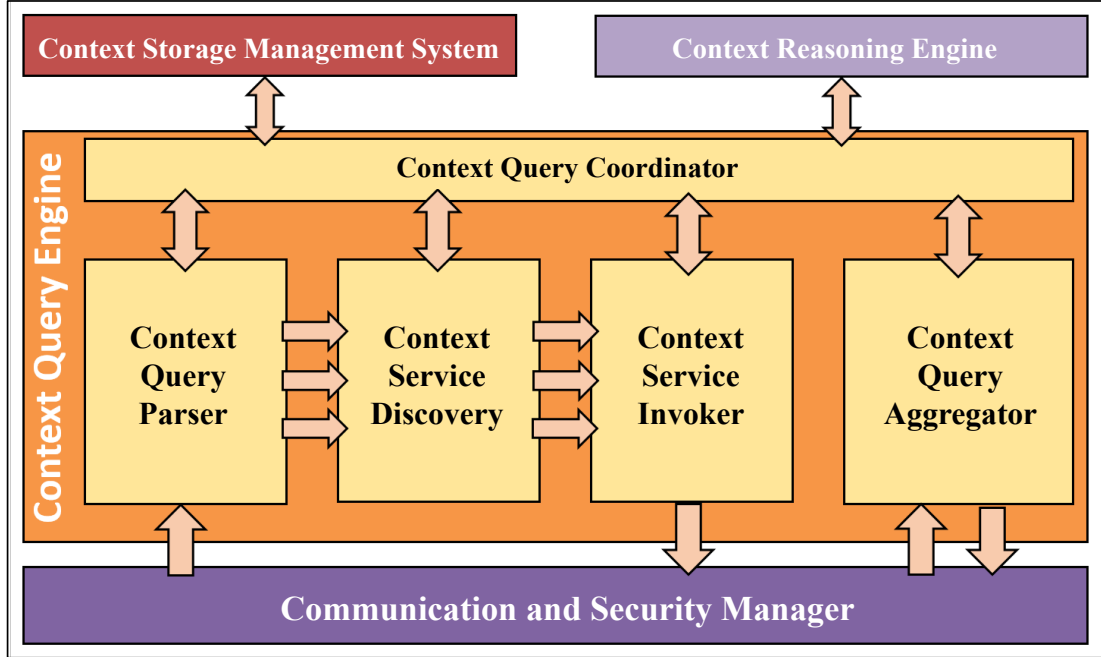
In this chapter, to demonstrate how CoaaS utilises CSDL and CDQL, we will focus on two enabling components of CoaaS platform that directly interact with the proposed context service description and context query language to enable context sharing in IoT ecosystem. These components are Context Query Engine (CQE), which parse the incoming CDQL queries and manage their execution, and Situation Monitoring Engine (SME), which enables continuous monitoring of IoT entities' situations.

Moreover, after introducing the aforementioned components and providing details about their underlying sub-components and algorithms, as a proof of concept, we will present a prototype implementation of CoaaS platform and will explain its general infrastructure and execution environment.

## 4.1 CONTEXT QUERY ENGINE

The architecture of Context Query Engine (CQE) is illustrated in Figure 4.1. As mentioned earlier, this module is mainly responsible for parsing the incoming queries, generating query execution plan, orchestrating the execution of queries, and producing the final query result. Furthermore, CQE also takes care of fetching required data from context providers on demand.

As shown in Figure 4.1, within CQE there are five main components, namely (i) Context Query parser (CQP), (ii) Context Query Coordinator (CQC), (iii) Context Service Discovery (CSD), (iv) Context Service Invoker (CSI), and (v) Context Query Aggregator (CQA). A detailed description of each of these components will be presented in the remainder of this section.



**Figure 4.1 - Context Query Engine Architecture**

When a query is issued to CoaaS, after passing the security checks, it will be sent to the Context Query Parser (CQP) by Communication and Security Manager. The CQP has three main responsibilities, namely parse the incoming queries, break them into several sub-queries (i.e. context requests), and determine the query's execution plan. The details of generating the execution plan for CDQL queries are discussed in Section 4.2.

Then, the parsed query plus the execution plan will be sent to the Context Query Coordinator (CQC). The CQC plays an orchestration role in the engine. This module is responsible for managing and monitoring the whole execution procedure of a context query. We will describe the details and workflow of these components in Section 4.3.

In the next step, context requests will be pushed into the Context Service Discovery (CSD) module. This module is in charge of finding the most appropriate

context service for an incoming request. The workflow of this component consists of two parts. First, it finds context services that match the requirements of a context request. Then, based on the discovered services, it returns a sorted set of the best available context services that can satisfy the requirements of a request, considering different metrics such as Cost of Service, and Quality of Service. The underlying concepts of CSD are discussed in Section 4.4.

After selecting the best eligible context provider (i.e. context service) for each context request, the request will be passed to the Context Service Invoker (CSI). This component is responsible for fetching context from the corresponding context provider to retrieve the required contextual information and pass the retrieved information to the Context Query Aggregator (CQA). Finally, the CQA combines the results of all the context requests and generates the final result of the query. The retrieved context may also be processed by the Context Reasoning Engine (CRE) to produce high-level context.

## **4.2 CONTEXT QUERY PARSER AND EXECUTION PLAN S GENERATION**

As stated before, CDQL supports complex context queries concerning various entities where the information about each entity might be provided by a different context service. In other words, CDQL queries are capable of expressing request for contextual information related to one or several entities. Furthermore, entities used in a query can be dependent, which means the information retrieved from one entity might be used in the query definition of another entity.

For example, consider the CDQL query shown in Code block 4.1. This query consists of three context entities, namely *vehicleA*, *trafficElements*, and *targetCarparks*, and presents a request to find all the traffic incidents that might affect *vehicleA* and also available parking options near its destination.

```

prefix datex:http://vocab.datext.org,
mv:http://schema.mobivoc.org
select (trafficElements.*, targetCarpark.*)
define
entity vehicleA is from datex:vehicle where
vehicleA.vehicleRegistrationPlateIdentifier = "1hm3ea",
entity trafficElements is from datex:TrafficElement
where spatioTemporalIntersect(trafficElements.geo,
vehicleA.itinerary, 200) = true,
entity targetCarparks is from mv:ParkingGarage where
distance(targetCarparks, vehicleA.destination.geo ,
"walking") < {"@type":"shema:QuantitativeValue",
"value": 500, "unitCode":"m"}

```

#### Code block 4.1 - CDQL for finding traffic incident near a specific vehicle

As this query shows, the definition of both *trafficElements* and *targetCarparks* are dependent on *vehicleA*, as their *WHERE* clauses have a reference to one of *vehicleA*'s attributes, i.e. *vehicleA.itinerary* and *vehicleA.destination* respectively. Consequently, before querying the registered context providers about traffic incidents and parking facilities, it is necessary to send a request to *vehicleA* for fetching its planned route (i.e. *itinerary*) and destination.

On the other hand, each CDQL query might have some entities that can be queried simultaneously, which leads to reducing the overall query execution time. For example, in the query above, after retrieving the required context about *vehicleA*, both traffic incidents and parking facilities can be queried at the same time.

Based on the concepts discussed above, we have designed and developed an algorithm to generate execution plans for CDQL queries. The execution plan generation can be remodelled as a graph traversal problem, by converting CDQL queries to a directed graph, where each node represents one entity, and each edge between two nodes represents the relationship (dependency) among those entities. As a result, the execution plan can be generated by finding a path that visits all the nodes in the graph starting from a node with no dependencies (zero inbound degree).

The algorithm for the proposed execution plan generator is presented in Figure 4.2. This algorithm accepts a CDQL query in String format and generates an execution plan that specifies the order of retrieving contextual information about the entities defined in the query.

As the first step towards producing the execution plan, the incoming CDQL query will be parsed into an object model containing several attributes, namely *queryType*, *nameSpaces*, *select*, and *define*. The *queryType* identifies the type of the incoming query, which can be either pull-based or push-based. The *nameSpaces* element contains all the semantic vocabularies defined in the *PREFIX* clause. The *select* denotes the structure of the query's output and includes the entities, attributes, and functions that are defined by the *SELECT* clause of the incoming query. Lastly, the *define* element is an array of context entities described in the *DEFINE* clause of the incoming query.

---

**Algorithm 1** CDQL Execution Plan Generator

---

```

1: procedure GENERATEEXECUTIONPLAN(query)
2:   parsedQuery  $\leftarrow$  parse(query)
3:   executionOrder  $\leftarrow$  0
4:   entities  $\leftarrow$  parsedQuery.entities
5:   Initialize executionPlan, visitedEntites
6:   for each entity  $\in$  entities do
7:     if entity.getDependency().isEmpty() then
8:       entities.remove(entity)
9:       executionPlan.get(executionOrder).add(entity)
10:      visitedEntites.add(entity)
11:   flag  $\leftarrow$  true
12:   while flag and !entities.isEmpty() do
13:     executionOrder  $\leftarrow$  executionOrder + 1
14:     Initialize tempVisitedEntites
15:     flag  $\leftarrow$  false
16:     for each entity  $\in$  entities do
17:       if entity.getDependency()  $\subset$  visitedEntites then
18:         flag  $\leftarrow$  true
19:         entities.remove(entity)
20:         executionPlan.get(executionOrder).add(entity)
21:         tempVisitedEntites.add(entity)
22:         visitedEntites.addAll(tempVisitedEntites)
23:   if parsedQuery.entities.length()  $\neq$  visitedEntites.length() then
24:     throw error
25:   return executionPlan

```

---

**Figure 4.2** - CDQL Execution Plan Generator

Each context entity itself is represented by five elements:

- **entityID** denotes the unique name assigned to the entity.
- **type** represents the semantic category/ontology classes the entity belongs to.
- **dependency** captures the dependency with the other context entities that are referenced in the definition of this entity
- **RPNCondition** is the Reverse Polish Notation (RPN) representation of the *WHERE* clause. RPN is a well-known method for the expression notification in a postfix manner, instead of using the usual infix notation.
- **contextAttributes** consists of an array of context attributes that are used in the CDQL query in the *SELECT*, *WHEN*, or *WHERE* clauses.

Code block 4.2 shows the JSON representation of the parsed CDQL object for the query presented in Code block 4.1.

After generation of the parsed CDQL object, the initialization step of Algorithm 4.1 (Figure 4.2) creates an empty hashmap for storing the execution plan (*executionPlan*), and an empty set to keep track of visited context entities (i.e. *visitedNodes*). Then, the algorithm iterates over all the context entities in the *define* element to find those context entities that have no dependency (0 inbound degree). The retrieved entities in this step will be marked as visited, removed from the *define* element and will be added to the *executionPlan*, where the execution order is 1.

As the next step, the algorithm iterates through the remaining entities in the *define* element and tries to find those entities that their *dependency* is a subset of the *visitedNodes*. Then, similar to the previous step, the found entities will be removed from the *define* element, labelled as visited, and will be added in the next execution order of the *executionPlan*. This step will be repeated several times until either all the nodes in the *define* elements are visited (until the define element becomes empty) or cannot visit a new entity in an iteration. Finally, the algorithm checks if all the entities in the *define* element are visited. If not, it means the execution plan for the incoming query cannot be generated due to a cycle in the dependency graph. Otherwise, the algorithm returns the generated execution plan.

```

{
  "warnings": {},
  "errors": {},
  "queryType": "PULL_BASED",
  "nameSpaces": {
    "mv": "http://schema.mobivoc.org",
    "datext": "http://vocab.datext.org"
  },
  "select": {
    "selectAttrs": {
      "trafficElements": [...],
      "targetCarparks": [...]
    },
  },
  "define": [{
    "entityID": "vehicleA",
    "type": {
      "type": "Vehicle",
      "vocabURI": "http://vocab.datext.org/Vehicle"
    },
    "dependency": {},
    "RPNCondition": [...],
    "contextAttributes": ["*", "geo", "itinerary"]
  },
  {
    "entityID": "targetCarparks",
    "type": {
      "type": "ParkingGarage",
      "vocabURI": "http://schema.mobivoc.org"
    },
    "dependency": {
      "vehicleA": ["geo"]
    },
    "RPNCondition": [...],
    "contextAttributes": ["*"]
  },
  {
    "entityID": "trafficElements",
    "type": {
      "type": "TrafficElements ",
      "vocabURI": "http://vocab.datext.org/TrafficElements "
    },
    "dependency": {
      "vehicleA": ["itinerary"]
    },
    "RPNCondition": [...],
    "contextAttributes": ["*", "geo"]
  }
] }

```

**Code block 4.2** - An Example of Parsed CDQL Query



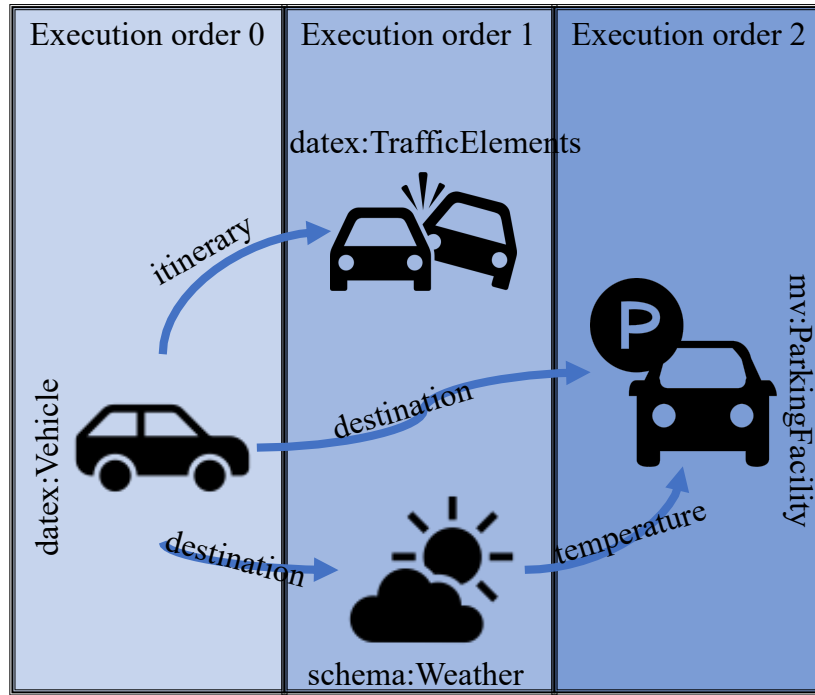
```

prefix datex:http://vocab.datext.org,
mv:http://schema.mobivoc.org, schema:http://schema.org
select (trafficElements.*, targetCarpark.*,
weatherCondition.*)
define
entity vehicleA is from datex:Vehicle where
vehicleA.vehicleRegistrationPlateIdentifier = "1hm3ea",
entity weatherCondition is from schema:Weather where
weatherCondition.location = vehicleA.destination,
entity trafficElements is from datex:TrafficElement
where
spatioTemporalIntersect(trafficElements.geo,
vehicleA.itinerary, 200) = true,
entity targetCarparks is from mv:ParkingFacility where
(goodForWalking(weatherCondition) > 0.7 and
distance(targetCarparks, vehicleA.destination,
"walking") < {"@type":"shema:QuantitativeValue",
"value": 500, "unitCode":"m"}) or
(goodForWalking(weatherCondition) <= 0.7 and
distance(targetCarparks, vehicleA.destination,
"walking") < {"@type":"shema:QuantitativeValue",
"value": 1000, "unitCode":"m"})

```

#### Code block 4.3 - Extended parking and traffic elements query

To illustrate the procedure of generating an execution plan, consider the context query shown in Code block 4.3, which is an extended version of the query discussed earlier in this section in Code block 4.1. This query consists of four entities: *vehicleA*, *weatherCondition*, *trafficElements*, and *targetCarparks*.



**Figure 4.3 - Query Execution Plan Graph**

Figure 4.3 shows the directed graph generated based on this query. As depicted in this graph, the inbound degree of entity *vehicleA* is 0. Therefore, this entity should be retrieved in the first step. In the next step, when the required information (i.e. destination and itinerary) regarding *vehicleA* is fetched, the context request related to *weatherCondition* can be issued. In the same manner, in parallel with the previous step, the request for *trafficElements* can be executed. Lastly, when the required contextual information related to *weatherCondition* is fetched, a context request will be generated to find the best available car parks. Therefore, the order of context requests execution (execution plan) for this query can be written as shown below:

Execution order 1: *vehicleA*

Execution order 2: *weatherCondition*  
*trafficElements*

Execution order 3: *targetCarparks*

### 4.3 CONTEXT QUERY COORDINATOR

In the previous section, we presented our proposed algorithm for generating execution plan for CDQL queries. Furthermore, we showed the structure of the parsed query object and described its main elements. As the next step towards executing CDQL queries, in this section, we will describe the workflow of Context Query Coordinator (CQC) module. As discussed in Section 4.1, CQC is responsible for managing the whole execution lifecycle of CDQL queries, including both pull-based and push-based queries.

As mentioned in Chapter 3, CDQL supports querying contextual information using two approaches: the pull-based approach and the push-based approach. In the remainder of this section, we will discuss how CQC handles pull-based queries in Subsection 4.3.1. Then, in Subsection 4.3.2, the workflow of managing push-based queries will be described in detail.

#### 4.3.1 PULL-BASED CDQL QUERY

In this section, we will present the workflow of executing pull-based queries, which are executed synchronously. A synchronous query is a query that maintains control over the process of the application that issues the query for the query's lifetime. In other words, when a context consumer issues a pull-based query, it has to wait for the entire round trip, from when the query is first sent to the CoaaS until the results are retrieved and returned to the context consumer.

The complete workflow of executing pull-based queries is illustrated as flow of events in a sequence diagram in Figure 4.4. When CQE receives a CDQL query, the query will be sent to CQP, which parses the raw query and generates the execution plan. Then, the CQP passes the parsed query object plus the execution plan to CQC. As described in Section 4.2, each execution plan consists of several execution orders that specify the correct sequence of retrieving the context entities defined in a CDQL query. Moreover, each execution order itself has one or several independent entities, which means they can be queried simultaneously.

Therefore, to execute an incoming context query, CQC iterates over the generated execution plan in ascending order, from the execution order 1 to the last execution

order. Following this, for each entity in the current execution order, CQC starts a new thread that forms and issues a context request to fetch the required context of the entity. As defined in Definition 3.6, context requests are represented as a triple:  $\langle E, CA, P \rangle$  where  $E$  denotes the type of entity of interest (i.e. *entityType* in the parsed query object),  $CA$  is a set of requested context attributes (i.e. *contextAttributes* in the parsed query object), and  $P$  is a set of predicates, which are defined over  $CA$  using logical expressions (i.e. *RPNCondition* in the parsed query object). Execution of context requests has four main steps, as outlined below:

**Figure 4.4 - Push-based CDQL execution workflow**

converting the incoming context requests to the underlying data storage language. Subsequently, a list of matching context entities (i.e. context responses) will be sent back to the CQC, which can have zero or more entities, depending on the ability of CSMS to find compliant entities.

**Step 2:** If the returned list is non-empty, the CQC checks the validity of the context responses by inspecting the expiry timestamps of their context attributes. If any of the attributes were expired, CQC issues a request to Context Service Invoker (CSI) to re-fetch the value of the expired context attribute from the corresponding context provider.

**Step 3:** On the other hand, if CSMS cannot find any context entity that matches the characteristics of the requested entity, CQC issues a context discovery request to CSD. Then, CSD tries to find and select the most eligible context services that match the requirements of the incoming context request. Details of how CSD discovers and selects matching context services is provided in Section 4.4. Then, CQC fetches the context of the entities of interest through the CSI module.

**Step 4:** In the final step of handling context requests, CQC re-evaluates the RPNCondition of those retrieved entities that their context attributes have been updated in Step 2. Moreover, if the RPNCondition contains any situation or aggregation function that cannot be evaluated in the previous steps, CQC re-evaluates them.

After successfully obtaining the needed context for each request in the first execution order, CQC stores the result and starts the next iteration, by incrementing the execution order by one. However, before starting the next iteration, it is required to update the RPNCondition of those entities that are dependent on at least one of the entities that are retrieved in the current execution order. Consequently, CQC traverses the context entities in the next execution orders and updates their RPNCondition by replacing the dependant context attributes according to their actual values that are fetched in the current iteration. During the process of updating the RPNConditions, there might be a case that more than one context entity is retrieved for a given context request, which is referred to in the *WHERE* clause of another entity. In this situation, if it is required, CQC reformulates the RPNCondition. Based on how the dependent attribute is used in the *WHERE* clause, five different reformulation strategies might be considered by CQC. Table 4.1 shows the reformulation strategies.

**Table 4.1** - RPNCondition reformulation strategies

Usage Type	Strategy	Example	
		Original condition	Reformulated condition
<b>In a condition using set operators (e.g. containsAny, containsAll)</b>	No changes required.	$e1.a1 \text{ containsAll } e2.a1$	$e1.a1 \text{ containsAll } [1,2,3,4]$
<b>In an equality condition</b>	The equality operator will be replaced by containsAny.	$e1.a1 = e2.a1$	$e1.a1 \text{ containsAny } [1,2,3,4]$
<b>In an inequality condition</b>	The inequality will be broken down into several inequality conditions (one for each instance of dependent entity) that are connected with <i>OR</i> operator.	$e1.a1 < e2.a1$	$(e1.a1 < 1 \text{ or } e1.a1 < 2 \text{ or } e1.a1 < 3 \text{ or } e1.a1 < 4)$
<b>Inside a function call</b>	The function call will be broken down into several function calls (one for each instance of dependent entity) that are connected with <i>OR</i> operator.	$F1(e1.a1, e2.a1) < 12$	$(F1(e1.a1, 1) = \text{true} \text{ or } F1(e1.a1, 2) = \text{true} \text{ or } F1(e1.a1, 3) = \text{true} \text{ or } F1(e1.a1, 4) = \text{true})$
<b>Inside an entityMatch operator</b>	For each instance of dependent entity, one entityMatch statement will be generated. The <i>OR</i> operator will be used to connect these statements.	$\text{entityMatch}(e1.a1 = e2.a1 \text{ and } e1.a2 < e2.a2)$	$((e1.a1 = 1 \text{ and } e1.a2 < 10) \text{ or } (e1.a1 = 2 \text{ and } e1.a2 < 8) \text{ or } (e1.a1 = 3 \text{ and } e1.a2 < 4) \text{ or } (e1.a1 = 4 \text{ and } e1.a2 < 6))$

\*assume the context response for e2 contains the following entities:  $[\{a1:1, a2:10\}, \{a1:2, a2:8\}, \{a1:3, a2:4\}, \{a1:4, a2:6\}]$

Finally, when all the context entities presented in the execution plan are retrieved, the fetched context will be passed to the Context Query Aggregator (CQA). CQA generates the final output of the incoming CDQL query based on its SELECT clause.

To further clarify the execution procedure of pull-based queries, consider the example query presented in Code block 4.4. This query is designed to find the vehicles that are driving faster than 60 km/h at a distance less than 500 meters from a school in one of Melbourne's suburbs.

```
Prefix schema:http://schema.org
select (vehicles.VIN)
define entity schools is from schema:School where
schools.address= {
  "@type": "PostalAddress", "addressCountry":
"Australia",
  "addressLocality": "Melbourne", "addressRegion":
"VIC",
  "postalCode": "3145"},
entity vehicles is from schema:Vehicle where
vehicles.speed > {"@type":"shema:QuantitativeValue",
"value": 40, "unitCode":"kmh"} and
distance(vehicles.geo, schools.geo, "driving")<
{"@type":"shema:QuantitativeValue", "value": 500,
"unitCode":"m"}
```

**Code block 4.4** - CDQL query for finding vehicles driving faster than 60 km/h near a school in Melbourne

The code block shows that this query has two entities, *schools* and *vehicles*, where the *vehicles* entity has a dependency on entity *schools*. Therefore, the execution plan of the query has two execution orders:

Execution order 1: *schools*

Execution order 2: *vehicles*

Based on the above execution plan, CQC issues a context request to CSMS to find all the schools within the specified area.

$cr_{schools}: \langle schema: School, \{address, geo\}, \{schools.address = \{\dots\}\} \rangle$

Then, CSMS queries the repository of the registered entities to find the matching schools. For this query, assume 3 schools are registered inside the identified region. Therefore, CSMS sends a context response back to CQC, which contains the address and geocoordinate of 3 schools that matches the aforementioned condition. Then, for each of these entities, CQC validates the expiry timestamp of the corresponding context attributes. However, as both address and geocoordinate for an entity like a school are considered as static values, we assume all the retrieved context attributes are valid.

Since the entity *schools* is the only entity in the first execution order, CQC starts the next execution order. However, as mentioned earlier, it is required to update the RPNCondition of the entity *vehicles* by replacing the *schools.geo* by its actual value. For the given example, as more than one entity has been found for *schools* entity, CQC reformulates the WHERE clause of entity *vehicles*. The reformulated query can be seen in Code block 4.5.

```
entity vehicles is from schema:Vehicle where
vehicles.speed > {"@type":"shema:QuantitativeValue",
"value": 40, "unitCode":"kmh"} and (
    distance(vehicles.geo, [-37.876584, 145.053531],
    "driving")< {"@type":"shema:QuantitativeValue",
"value": 500, "unitCode":"m"}) or
    distance(vehicles.geo, -37.873959, 145.057103],
    "driving")< {"@type":"shema:QuantitativeValue",
"value": 500, "unitCode":"m"}) or
    distance(vehicles.geo, [-37.877200, 145.047115],
    "driving")< {"@type":"shema:QuantitativeValue",
"value": 500, "unitCode":"m"})
)
```

**Code block 4.5 - Reformulated WHERE clause**



In the next step, CQC forms a context request based on the updated RPNCondition in order to find the vehicles that are over-speeding near one of the three schools found in the previous iteration.

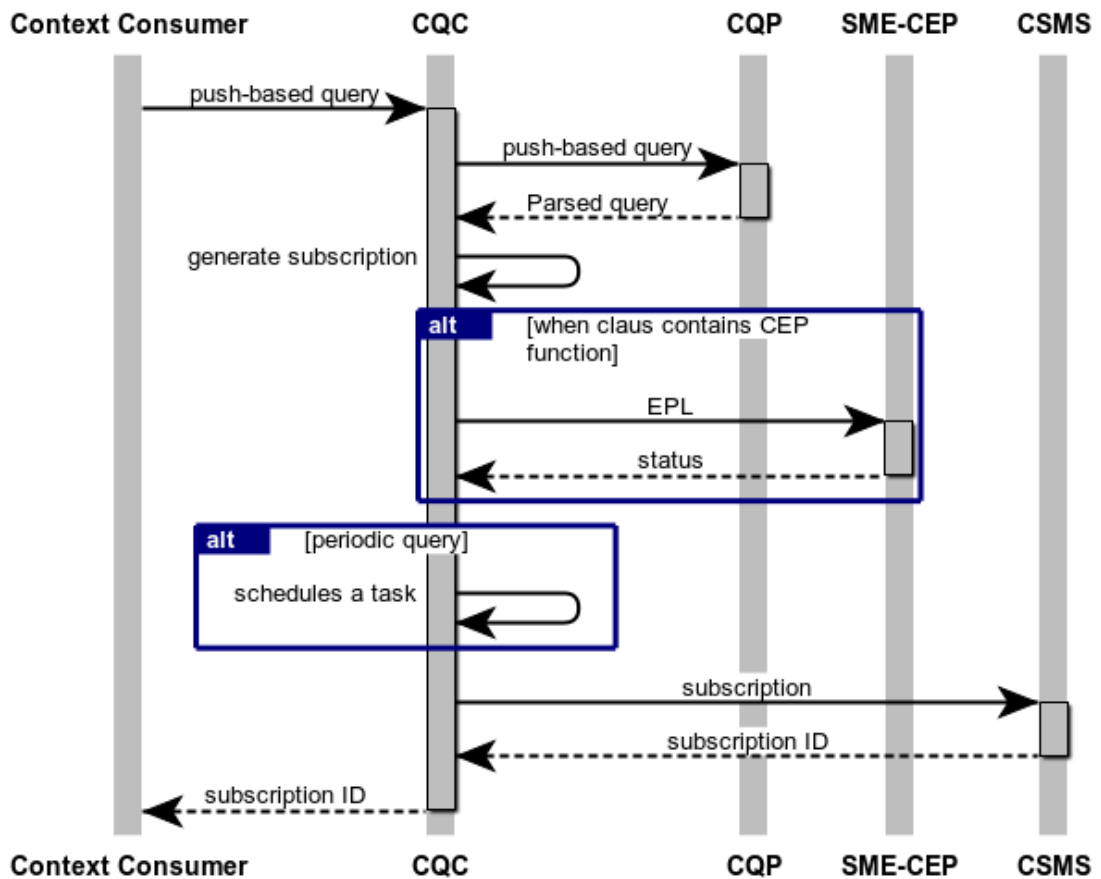
$$cr_{\text{vehicles}} : \langle \text{schema: Vehicle}, \{VIN, geo, speed\}, \{vehicles.distance < \dots\} \rangle$$

This time we assume CSMS returns 10 vehicles that each of them meets the above conditions (i.e. near the school and over-speeding). Then, for each vehicle, CQC checks the expiry date of their required attributes, namely Vehicle Identification Number (VIN), geocoordinate, and speed. As both speed and geocoordinate for a mobile entity like a vehicle have high update frequency, there is a considerable chance of having outdated values. As a result, CSI sends a request to corresponding vehicles to fetch the real-time values of the expired context attributes. Then, finally, CQC re-evaluates the RPNCondition based on the updated context attributes and returns the VIN of over-speeding cars back to the corresponding context consumer.

#### 4.3.2 PUSH-BASED CDQL QUERY

In this section, we will explain the workflow of Context Query Coordinator (CQC) for processing the second type of CDQL queries, which are referred to as push-based queries. In the case of push-based queries, the CQC is responsible for creating and registering new subscriptions based on the incoming CDQL queries.

The workflow of the execution of push-based CDQL queries is presented in Figure 4.5 as event sequences in a sequence diagram. As shown in the figure, the subscription procedure starts when the CQC receives a new push-based CDQL query (i.e. a CDQL query with *WHEN* clause) from a context consumer. In the first step, the query will be sent to Context Query Parser (CQP), which parses the query and sends the outcome to the CQC.



**Figure 4.5** - Push-based CDQL execution workflow

The CQC will then generate a subscription. Here, it will convert the parsed query to an internal representation, which is called a subscription data model. This model has four main parts:

- **Callback** stores the required information about the context consumer's endpoint and will be used for sending the result of the query.
- **Parsed Query** stores an executable version of the issued query. The reason for storing the executable version of the query is to speed up the pre-processing procedure by avoiding the need to parse a full query each time a context update is received by the platform.
- **Related Entities** contains the information about all the entities and their attributes that needs to be monitored. This information is organised in an indexed structure and is used for pre-filtering the subscriptions.

- **Situation** is the Reverse Polish Notation (RPN) representation of the *WHEN* clause.

Besides these four elements, each subscription document has a unique ID. An example of the subscription data model is shown in Figure 4.6.

```

id : 5b402f7802a21e0fa1996c5c
▼ callback {3}
  callbackMethod : HTTPPOST
  httpURL : http://138.194.106.20:8282/
  ▼ headers {1}
    Content-Type : text/xml
▶ parsedQuery {11}
▼ relatedEntities [5]
  ▼ 0 {4}
    entityID : temp
    ▶ entityType {2}
    ▶ attributes [3]
    ▶ condition {2}
  ▼ 1 {4}
    entityID : driver
    ▶ entityType {2}
    ▶ attributes [2]
    ▶ condition {2}
  ▼ 2 {4}
    entityID : car
    ▶ entityType {2}
    ▶ attributes [4]
    ▶ condition {2}
  ▼ 3 {4}
    entityID : eventLocation
    ▶ entityType {2}
    ▶ attributes [4]
    ▶ condition {2}
  ▼ 4 {4}
    entityID : events
    ▶ entityType {2}
    ▶ attributes [4]
    ▶ condition {2}
▶ situation [23]
  ▼ 0 {3}
    stringValue : timeDifference(events.startDate,currentTime(\"Australia/Melbourne\"))
    type : Function
    ▶ functionCall {3}
  ▼ 1 {3}
    stringValue : 50
    type : Constant
    constantType : Numeric
  ▼ 2 {2}
    stringValue : <
    type : Operator
  ...

```

**Figure 4.6** - An example of subscription data model

In the next step, the Query Coordinator checks if the *WHEN* clause contains any of the following processing functions: windowing (e.g., during the last five minutes), temporal relation (e.g., after, before, etc.), or trend detection (i.e., increase, decrease, stable). This class of functionality is commonly realised in the Complex Event Processing (CEP) software. Consequently, to support these types of functions, we have adopted an existing CEP engine.

If such tasks exist, Query Coordinator generates a query in the corresponding Event Processing Language (EPL) and issues it to the CEP engine. For example, consider a *WHEN* clause, which contains the following trend function to monitor the decrease in the number of parking spots :

```
decrease (parking.availableSpots, {"value":10,"unit":"minutes"})
```

In the current implementation of CoaaS platform, we are using Siddhi CEP framework (Suhothayan et al., 2011). An example of a generated EPL query (i.e. Siddhi application) for trend detection (decreasing the number of parking spots during the last 10 minutes) is presented in Code block 4.6.

```
1  @app:name('sub_123')
2  define stream eventStream(name string, amount double, timestamp long);
3  define table eventResultTable (functionSignature string, initialAmount double,
4  finalAmount double,timestamp long);
5  partition with (name of eventStream)
6  begin
7  from every e1=eventStream,
8  e2=eventStream[e1.amount > amount and (timestamp - e1.timestamp) < 10 * 60000]*,
9  e3=eventStream[timestamp - e1.timestamp > 10 * 60000 and e1.amount > amount]
10 select 'functionSignature' as functionSignature, e1.amount as initialAmount,
11 e3.amount as finalAmount, e3.timestamp insert into eventResultTable;
12 end;
```

**Code block 4.6** - An example of Siddhi application generated by the CQC

Another type of PUSH-based queries is the periodic query. This type of query is used by context consumers to receive regular updates about the situation of an entity. In such a case, the Query Coordinator schedules a task with a fixed interval to update the context consumer of situation changes during a specified period.

In the final step of the execution of push-based queries in Figure 4.5, the generated subscription will be passed to the Context Storage Management System (CSMS), which will store it for persistence. Moreover, the ID of the registered subscription will be sent back to the corresponding context consumer. Context consumers can monitor the detected situations of a registered subscription by providing this ID. On top of that, using the subscription ID, context consumers are able to deactivate or delete their registered subscription.

#### 4.4 CONTEXT SERVICE DISCOVERY

As mentioned earlier, Context Service Discovery (CSD) is responsible for discovering the context services that can provide the requested information and hence satisfy the incoming CDQL queries. In the remainder of this section, we will describe the context service discovery problem followed by our proposed solution.

To formulate the context service discovery problem, consider a platform with 'n' registered context services and a given context request. The set of context services is denoted by CSR (context service repository) and the given context request is denoted by  $cr$ . As shown in Definition 3.6, a context request is represented as a triple:  $\langle E, CA, P \rangle$  where  $E$  denotes the entity of interest,  $CA$  is a set of requested context attributes, and  $P$  is a set of predicates, which are defined over  $CA$  using logical expressions.

The goal of the context service discovery is to find all the services that best match  $cr$ . Therefore, the problem of context service discovery can be modelled with the following function:

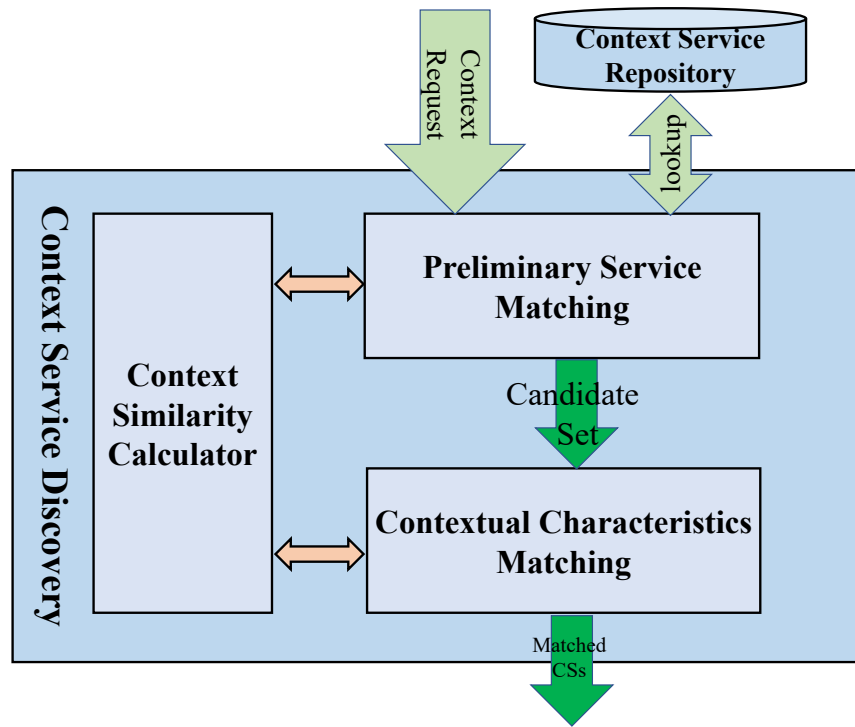
$$(Eq. 4.1) \quad cr \rightarrow \{cs_1, \dots, cs_l\}$$

such that:

$$\{cs_1, \dots, cs_l\} \subseteq CSR$$

$$\forall cs_j \in \{cs_1, \dots, cs_l\} : E_{cr} \subseteq E_{cs_j} \wedge CA_{cr} \subseteq CA_{cs_j} \wedge P_{cr} \subseteq P_{cs_j}$$

In order to solve the stated problem, we have designed a two-level approach, which is capable of discovering the most eligible services for a given context request. Figure 4.7 illustrates the overview architecture of the proposed solution. The figure shows that the CSD consists of three modules, (i) Preliminary Service Matching (PSM), (ii) Contextual Characteristics Matching (CCM), and (iii) Context Similarity Calculator (CSC).



**Figure 4.7** - Context Service Discovery architecture

When CSD receives a context request, the incoming request first goes to Preliminary Service Matching (PSM). PSM is responsible for searching through the context service repository to find those context services that their offered entity matches the incoming request. Then, the outcome of PSM, which is referred to as candidate set, will be sent to Contextual Characteristics Matching (CCM). CCM checks each context service inside the candidate state and verifies if it satisfies the characteristics of the incoming context request. Furthermore, CCM assigns a satisfiability level to each context service, which will be used to sort the matched services and choose the service that have the highest probability to serve the incoming request best.

Lastly, Context Similarity Calculator (CSC) is a tool that is designed to calculate the similarity between two context attributes. This tool is being used by both PSM and CCM. PSM uses it to compare the type of the requested entity (i.e.  $E_{cr}$ ) to offered entity's type (i.e.  $E_{cs}$ ). Moreover, CSC is used by CCM in order to compare how well two predicates, i.e. the one belonging to the context request and the other belonging to the context service, match. A detailed description of these modules is provided in the remainder of this section.

#### 4.4.1 CONTEXT SIMILARITY CALCULATOR (CSC)

In order to compute the similarity of a common context attribute ( $ca$ ) between a given context request ( $cr$ ) and a context service ( $cs$ ) (e.g.  $ca_{cr}.type = ca_{cs}.type = Temperature$ ), we define five different similarity functions. These functions accept two expressions defined on a context attribute with the same type as arguments and compute how closely a context service expression matches the corresponding attribute in the context request.

*Boolean Similarity:* Boolean Similarity is the most basic similarity function and is used to compare equality expressions that are defined on top of string-based context attributes (e.g.  $ca_{cr} = "AUD"$ ). It compares two context values and returns a Boolean value (either 0 or 1). The similarity is computed as:

$$(Eq. 4.2) \text{ Similarity}(ca_{cr}, ca_{cs}) = \begin{cases} 1 & \text{if } v(ca_{cr}) = v(ca_{cs}) \\ 0 & \text{if } v(ca_{cr}) \neq v(ca_{cs}) \end{cases}$$

where  $v(ca)$  represents the value of the context attribute  $ca$ .

*Continues Similarity function:* If the type of the context attribute is numeric or ordinal, the similarity between  $ca_{cr}$  and  $ca_{cs}$  is computed as below:

$$(Eq. 4.3)$$

$$\text{Similarity}(ca_{cr}, ca_{cs}) = \begin{cases} \frac{\text{Min}(\text{end}_{ca_{cr}}, \text{end}_{ca_{cs}}) - \text{Max}(\text{start}_{ca_{cr}}, \text{start}_{ca_{cs}})}{\text{Max}(\text{end}_{ca_{cr}}, \text{end}_{ca_{cs}}) - \text{Min}(\text{start}_{ca_{cr}}, \text{start}_{ca_{cs}})} & \text{if } ca_{cr} \cap ca_{cs} \neq \emptyset \\ 0 & \text{if } ca_{cr} \cap ca_{cs} = \emptyset \end{cases}$$

where  $\text{end}$  represents the higher bound value of  $ca$ , and  $\text{start}$  represents the lower bound value of  $ca$ .

To clarify, consider a given query that needs to find a carpark between 8:00 to 18:00 in a particular location. At the same time, assume there is a carpark service that provides information about a garage located in the requested area where its working hours are from 7:00 to 17:30. For this example, the similarity between the queried request time and carpark's working hours can be computed as below:

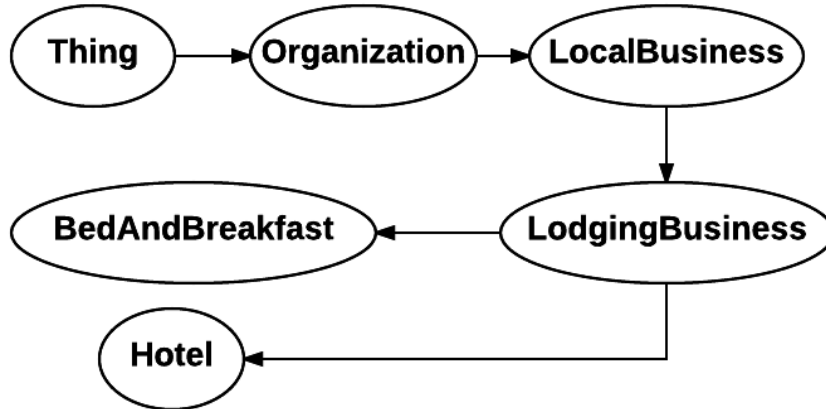
$$\frac{\text{Min}(18, 17.5) - \text{Max}(7, 8)}{18 - 7} = \frac{9.5}{11} = 0.86$$

*Semantic Similarity function:* If a context attribute refers to a semantic concept based on a hierarchical ontology; we introduce a semantic similarity function that uses a depth variable. The depth function returns the number of edges on the path from the given node to the root node (Zhang, Tang, Hong, Li, & Wei, 2006).  $LC(ca_{cr}, ca_{cs})$  denotes the lowest common concept node of both  $ca_{cs}$  and  $ca_{cr}$ . The similarity is calculated as follows:

$$(Eq. 4.4) \text{ Similarity}(ca_{cr}, ca_{cs}) = \frac{\text{depth}(LC(ca_{cr}, ca_{cs}))}{\text{depth}(ca_{cr})}$$

One of the main usages of the semantic similarity function is in entity type matching (explained in Section 4.4.2) when it is needed to check whether the type of the requested entity matches the type of the entity offered by the service or not. For example, consider a query designed to find a cheap ‘hotel’ in the city of Melbourne. However, while there is no context service related to a hotel that can satisfy the conditions of this query, i.e. cost and location, there is a service that offers ‘bed and breakfast’, which has a significant similarity with the query criteria. By considering the semantic hierarchy presented in Figure 4.8 (generated based on schema.org hierarchy), the similarity between ‘Hotel’ and ‘BedAndBreakfast’ can be computed as:

$$\frac{\text{depth}(\text{LodgingBusiness})}{\text{depth}(\text{Hotel})} = \frac{3}{4} = 0.75$$



**Figure 4.8** - Semantic hierarchy example based on schema.org



*Set Similarity* function: If the context attribute's value is a set/vector, the similarity is computed as follows:

(Eq. 4.5)

$$Similarity(ca_{cr}, ca_{cs}, op_{cr}) = \begin{cases} \frac{n(ca_{cr} \cap ca_{cs})}{n(ca_{cr})} & op_{cr} = containsAll \\ \begin{cases} 0 & ca_{cr} \cap ca_{cs} = \emptyset \\ 1 & ca_{cr} \cap ca_{cs} \neq \emptyset \end{cases} & op_{cr} = containsAny \end{cases}$$

where  $n(ca)$  is the number of elements in the set  $ca$  and  $op_{cr}$  denotes the type of operation used in the definition of the corresponding context request's expression.

*Geo-based Similarity* function: If the context is a geocoordinate or a geo-shape, the similarity is computed as:

(Eq. 4.6)

$$Similarity(ca_{cr}, ca_{cs}) = \begin{cases} 1 & ca_{cr} \cap ca_{cs} \neq \emptyset \\ \frac{distance(centre(ca_{cr}), centre(ca_{cs}))}{radius(ca_{cs}) + radius(ca_{cr})} & ca_{cr} \cap ca_{cs} = \emptyset \end{cases}$$

where  $center(ca)$  denotes the centre of the smallest bounding circle that contains  $ca$ ,  $radius(ca)$  denotes the radius of the smallest bounding circle, and  $distance$  function calculates the euclidean distance between two coordinates.

Following this section we will explain how these similarity functions will be employed in the procedure of context service discovery.

#### 4.4.2 PRELIMINARY SERVICE MATCHING (PSM)

The first phase of the proposed solution for addressing the services discovery problem is Preliminary Service Matching. In this phase, context services and context requests are coarsely checked. To pass this phase, the following conditions must hold:

1. **Entity matching:** The requested entity type is a) identical to the entity type offered by a context service or b) is a generalization of the offered entity type ( $E_{cr} \subseteq E_{cs_j}$ ).

2. **Context attribute matching:** The requested context attributes are a) the same as the attributes offered by a context service, or b) a generalization of the offered attributes. ( $CA_{cr} \subseteq CA_{cs_j}$ ).

Applying these two constraints limits the solution space of the service discovery problem by restricting the number of context services that are eligible for serving a given request. We call the set of eligible services *Candidate Set*; which will be passed to the Contextual Characteristics Matching phase for further checking.

#### 4.4.3 CONTEXTUAL CHARACTERISTICS MATCHING (CCM)

The second and last phase of the service discovery process is to go through the services' *Candidate Set* and check whether the characteristics of a given service request( $cr$ ) matches any of the services in this Set ( $cs_j \in \text{Candidate Set}$ ) ( $P_{cr} \subseteq P_{cs_j}$ ).

The  $P_{cs_j}$  of a service  $cs_j$  and the  $P_{cr}$  of a context request  $cr$  match if and only if the conjunction of both constraints is satisfiable.

$$(Eq. 4.7) P_{cr} \wedge P_{cs_j} \vdash \text{satisfiable}$$

$P$  is a set of predicates combined with logic operators to define the contextual characteristics of a context entity. Therefore, Eq. 4.7 can be rewritten as equation Eq. 4.74.8.

$$(Eq. 4.8) \text{predicate}_{cr_1} \bigwedge_{\vee} \text{predicate}_{cr_2} \bigwedge_{\vee} \dots \bigwedge_{\vee} \text{predicate}_{cr_n} \\ \wedge \\ \text{predicate}_{cs_{j_1}} \bigwedge_{\vee} \text{predicate}_{cs_{j_2}} \bigwedge_{\vee} \dots \bigwedge_{\vee} \text{predicate}_{cs_{j_m}}$$

where  $m, n \in \mathbb{N}$  and each predicate can be represented as below:

$$ca_i \text{ op } cv$$

Where:

- $op \in \{=, <, >, \leq, \geq, \neq, \in, \exists\}$

- $cv \in \{ \text{Number}, \text{String}, \text{Set}, ca_j, \text{true/false} \}$

Further, by applying logical equivalence laws, such as the double negative elimination, De Morgan's laws, and the distributive law, we can transform Eq. 4.8 to disjunctive normal form (DNF). A logical formula is considered to be in DNF if and only if it is a disjunction (sequence of ORs) consisting of one or more disjuncts, each of which is a conjunction (AND) of one or more predicates.

$$\text{(Eq. 4.9)} \quad \text{conjunction}_1 \vee \text{conjunction}_2 \vee \dots \vee \text{conjunction}_z$$

where each conjunction is a set of predicates combined with logical *AND* as illustrated below:

$$\text{conjunction} = \text{predicate}_1 \wedge \dots \wedge \text{predicate}_w$$

*w.r.t*

$$\forall \text{predicate}_y \in \{\text{predicate}_1, \dots, \text{predicate}_z\},$$

$$\text{predicate}_y \in \{\text{predicate}_{cr_1}, \dots, \text{predicate}_{cr_n}\} \quad \vee$$

$$\text{predicate}_y \in \{\text{predicate}_{cs_{j_1}}, \dots, \text{predicate}_{cs_{j_m}}\}$$

We can state that Eq. 4.7 is satisfiable if at least one of the *conjunctions* in Eq. 4.9 is satisfiable.

Based on the discussed concepts, we developed an algorithm that gets  $P_{cr} \wedge P_{cs_j}$  as its input and computes the level of satisfiability. This algorithm first converts  $P_{cr} \wedge P_{cs_j}$  to its DNF form. Then, for each conjunction, it returns two floating point values between 0 and 1. The first value denotes the satisfiability level of a context service for the given context request and the second value shows the confidence of the calculated satisfiability value. The characteristics checking algorithm is presented in the Figure 4.9.

---

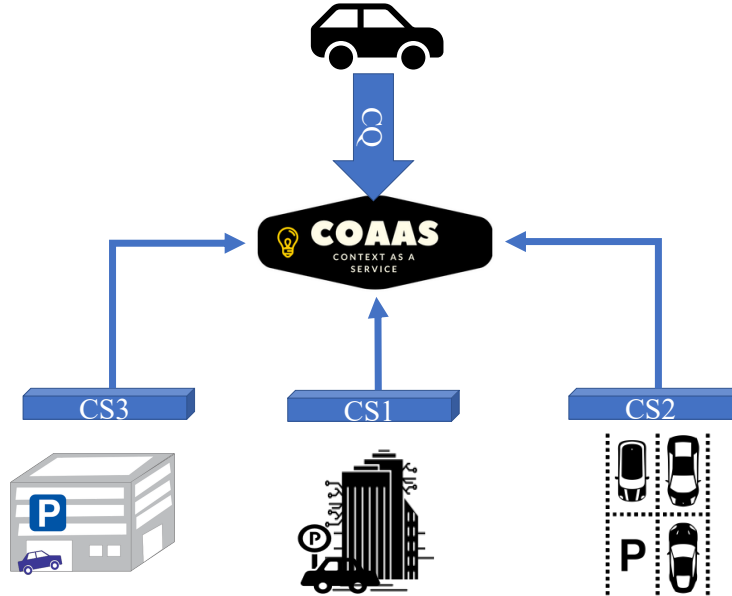
**Algorithm 2** Characteristics checking algorithm

---

```
1: function MATCH( $Ps_{cr_i} \wedge Ps_{cs_j}$ )
2:    $normalForm = toDNF(Ps_{cr_i} \wedge Ps_{cs_j})$ 
3:   for each conjunction  $c$  in  $normalForm$  do
4:     initialization;
5:     for each predicate  $p$  in  $c$  do
6:       initialization;
7:       if  $p \in visitedList$  then
8:         go to 5
9:       while  $p.hasNext \neq false$  do
10:        if  $p \in Ps_{cr_i}$  then
11:           $requestExpression \cap p$ 
12:        else
13:           $serviceExpression \cap p$ 
14:         $visitedList \leftarrow p$ 
15:        if ( $requestExpression$  and  $ServiceExpression$ ) not null then
16:           $weight += requestExpression.weight$ 
17:           $similarity += sim(requestExpression, serviceExpression) * requestExpression.weight$ 
18:         $confidence = \frac{weight}{Ps_{cr_i}.totalWeight}$ 
19:         $similarity = \frac{similarity}{weight}$ 
20:      Return  $confidence, similarity$ 
```

---

**Figure 4.9** - Contextual Characteristics Matchmaking Algorithm



**Figure 4.10** - visualisation of example for context service discovery process

#### 4.4.4 EXAMPLE

This section illustrates the CSD process by an example, which is visualised in Figure 4.10. In this example, a smart vehicle issues a context query to CoaaS in order to find available parking options near a specific location. This query is shown in Code block 4.7.

```
prefix mv:http://schema.mobivoc.org
select (targetCarpark.*)
define
entity targetCarparks is from mv:ParkingFacility where
(
(distance(targetCarparks.geo, [-
37.9133542,145.1336933], "walking")<
{"@type":"shema:QuantitativeValue", "value": 500,
"unitCode":"m"}) and targetCarparks.price <
{"@type":"shema:QuantitativeValue", "value": 5,
"unitCode":"aud"}) or
(distance(targetCarparks.geo, [-
37.9133542,145.1336933], "walking")<
{"@type":"shema:QuantitativeValue", "value": 1500,
"unitCode":"m"}) and targetCarparks.price <
{"@type":"shema:QuantitativeValue", "value": 2,
"unitCode":"aud"})
)
and targetCarparks.facilities containsAll
["ChargingPoint", "PayStation"] and
(targetCarparks.hasCapacity > 10 and
targetCarparks.hasCapacity.freshness < 200)
```

**Code block 4.7** - CDQL query for finding available parking options

Moreover, as the figure shows, we assume three context services that can serve the aforementioned query are registered in the CoaaS platform. The specifications of these context services are provided in Table 4.2.

**Table 4.2** - Registered parking facilities' context services

Name	Type	CA	P
$cs_1$	ParkingFacility	[ geo, price, totalCapacity, hasCapacity, features]	geo = [-37.907677, 145.130736] and price = 4 aud/h and features = [PayStation, ChargingPoint] and hasCapacity.freshness < 120s
$cs_2$	ParkingGrage	[ geo, totalCapacity, hasCapacity, features]	geo = [-37.908000, 145.129821] and features = [PayStation] and hasCapacity.freshness < 240s
$cs_3$	ParkingLot	[ geo, totalCapacity]	geo = [-37.914600, 145.137009] and totalCapa\city = 124

As described earlier, in the first step towards discovering the context services for an incoming context request, the Preliminary Service Matching (PSM) compares the type of the requested context entity with the type of the entities offered by the registered context services to check either they are matching or not. To do so, PSM uses the semantic context similarity function (Eq. 4.4) to compute the similarity of the offered and the requested entity's type (mv:ParkingFacility).

Then, in the next step, PSM verifies if the available context services can provide the information about the context attributes requested by the incoming  $cr$ . In order to perform this task, PSM computes the similarity of the  $CA_{cr}$  with  $CA_{cs}$  using the set similarity function (Eq. 4.5). The result of this phase of CSD process is shown in Table 4.3.

In this example, we assume the minimum similarity threshold for satisfying the preliminary service matching phase is set to 0.90. As a result, only  $cs_1$  and  $cs_2$  can satisfy the first phase of CSD. Therefore, these two context services will be passed to Contextual Characterises Matching (CCM) as the candidate set.

$$\text{candidate set} = \{cs_1, cs_2\}$$

**Table 4.3** - Result of PSM

CS	Step 1 of PSM: Entity matching	Step 2 of PSM: Context attribute matching	Average similarity
$cs_1$	$\frac{depth(parkingFacility)}{depth(ParkingFacility)}$ $= \frac{9}{9}$	$\frac{n([ geo, price, hasCapacity, facilities])}{n([ geo, price, hasCapacity, facilities])}$ $= 1$	$\frac{1 + 1}{2}$ $= 1$
$cs_2$	$\frac{depth(parkingFacility)}{depth(ParkingGrage)}$ $= \frac{9}{10}$	$\frac{n([ geo, hasCapacity, facilities])}{n([ geo, price, hasCapacity, facilities])}$ $= 1$	$\frac{1 + 0.9}{2}$ $= 0.95$
$cs_3$	$\frac{depth(parkingFacility)}{depth(ParkingLot)}$ $= \frac{9}{10}$	$\frac{n([ geo, totalCapacity])}{n([ geo, price, hasCapacity, facilities])}$ $= \frac{2}{4}$	$\frac{0.9 + 0.5}{2}$ $= 0.7$

In the next step, CCM iterates over the candidate set, and for each context service it performs the Contextual Characteristics Matchmaking algorithm to verify if the predicates of the incoming request (i.e.  $P_{cr}$ ) matches the predicates of any of the context services ( $P_{cs}$ ) in the candidate set. The following expressions show the predicates of the incoming context request and two context services in the candidate set. In order to simplify the representation of these predicates, we have assigned an ID to each predicate in these expressions, which can be seen in Table 4.4.

$$P_{cr} = ((p_{cr_1}^{cr} \wedge p_{cr_2}^{cr}) \vee (p_{cr_3}^{cr} \wedge p_{cr_4}^{cr})) \wedge p_{cr_5}^{cr} \wedge (p_{cr_6}^{cr} \wedge p_{cr_7}^{cr})$$

$$P_{cs_1} = p_{cs_1_1}^{cs_1} \wedge p_{cs_1_2}^{cs_1} \wedge p_{cs_1_3}^{cs_1} \wedge p_{cs_1_4}^{cs_1}$$

$$P_{cs_2} = p_{cs_2_1}^{cs_2} \wedge p_{cs_2_2}^{cs_2} \wedge p_{cs_2_3}^{cs_2} \wedge p_{cs_2_4}^{cs_2}$$

As described in Section 4.4.3, in the initial step, CCM combines the predicates of the incoming context request with the predicates of the context services in the candidate set using an *and* operator. Furthermore, it converts the generated expression to its disjunctive normal form. In this example, for the first context service in the candidate set, the outcome of this step is:

$$DNF(P_{cr} \wedge P_{cs_1}) = (p^{cr}_1 \wedge p^{cr}_2 \wedge p^{cr}_5 \wedge p^{cr}_6 \wedge p^{cr}_7 \wedge p^{cs_1}_1 \wedge p^{cs_1}_2 \wedge p^{cs_1}_3 \wedge p^{cs_1}_4) \vee (p^{cr}_3 \wedge p^{cr}_4 \wedge p^{cr}_5 \wedge p^{cr}_6 \wedge p^{cr}_7 \wedge p^{cs_1}_1 \wedge p^{cs_1}_2 \wedge p^{cs_1}_3 \wedge p^{cs_1}_4)$$

**Table 4.4** - Assigned ids for each predicate

Predicate ID	Predicate
$p^{cr}_1(\text{geo})$	distance(targetCarparks.geo, [-37.9133542,145.1336933], "walking")< {"@type":"shema:QuantitativeValue", "value": 500, "unitCode":"m"})
$p^{cr}_2(\text{price})$	targetCarparks.price < {"@type":"shema:QuantitativeValue", "value": 5, "unitCode":"aud"})
$p^{cr}_3(\text{geo})$	distance(targetCarparks.geo, [-37.9133542,145.1336933], "walking")< {"@type":"shema:QuantitativeValue", "value": 1500, "unitCode":"m"}
$p^{cr}_4(\text{price})$	targetCarparks.price < {"@type":"shema:QuantitativeValue", "value": 2, "unitCode":"aud"})
$p^{cr}_5(\text{facilities})$	targetCarparks.facilities containsAll ["ChargingPoint", "PayStation"]
$p^{cr}_6(\text{hasCapacity})$	targetCarparks.hasCapacity > 10
$p^{cr}_7(\text{hasCapacity.freshness})$	targetCarparks.hasCapacity.freshness < 200s
$p^{cs_1}_1(\text{geo})$	geo = [-37.907677, 145.130736]
$p^{cs_1}_2(\text{price})$	price = 4 aud/h



$p^{cs_1}_3(\text{features})$	features = [PayStation, ChargingPoint]
$p^{cs_1}_4(\text{hasCapacity.freshness})$	hasCapacity.freshness < 200s
$p^{cs_2}_1(\text{geo})$	geo = [-37.908000, 145.129821]
$p^{cs_2}_2(\text{price})$	price = 6 aud/h
$p^{cs_2}_3(\text{features})$	features = [PayStation]
$p^{cs_2}_4(\text{hasCapacity.freshness})$	hasCapacity.freshness < 240s

Then, CCM computes the satisfiability level of each conjunction in the DNF expression separately. In order to achieve this goal, for each conjunction, CCM compares the predicates that are defined on the same context attribute by using the CSC module. Table 4.5 and Table 4.6 show the outcome of CCM for conjunction 1 and 2 respectively.

Finally, for each conjunction, CCM computes two numbers between 0 and 1. The first value, which is an arithmetic average of the computed similarities, shows the overall satisfiability of each conjunction. In this example, the satisfiability level for conjunction one and two are 0.93 and 0.83 respectably.

The second value shows the confidence of the calculated satisfiability level and is calculated by dividing the number of predicates in the context request that has a matching predicate in the context service. For both conjunctions of this example, the confidence value is equal to 0.8, as 4 out of 5 predicates in the context request has a matching predicate in the context service.

**Table 4.5** - outcome of CCM for the first conjunction in  $DNF(Ps_{cr} \wedge Ps_{cs_1})$

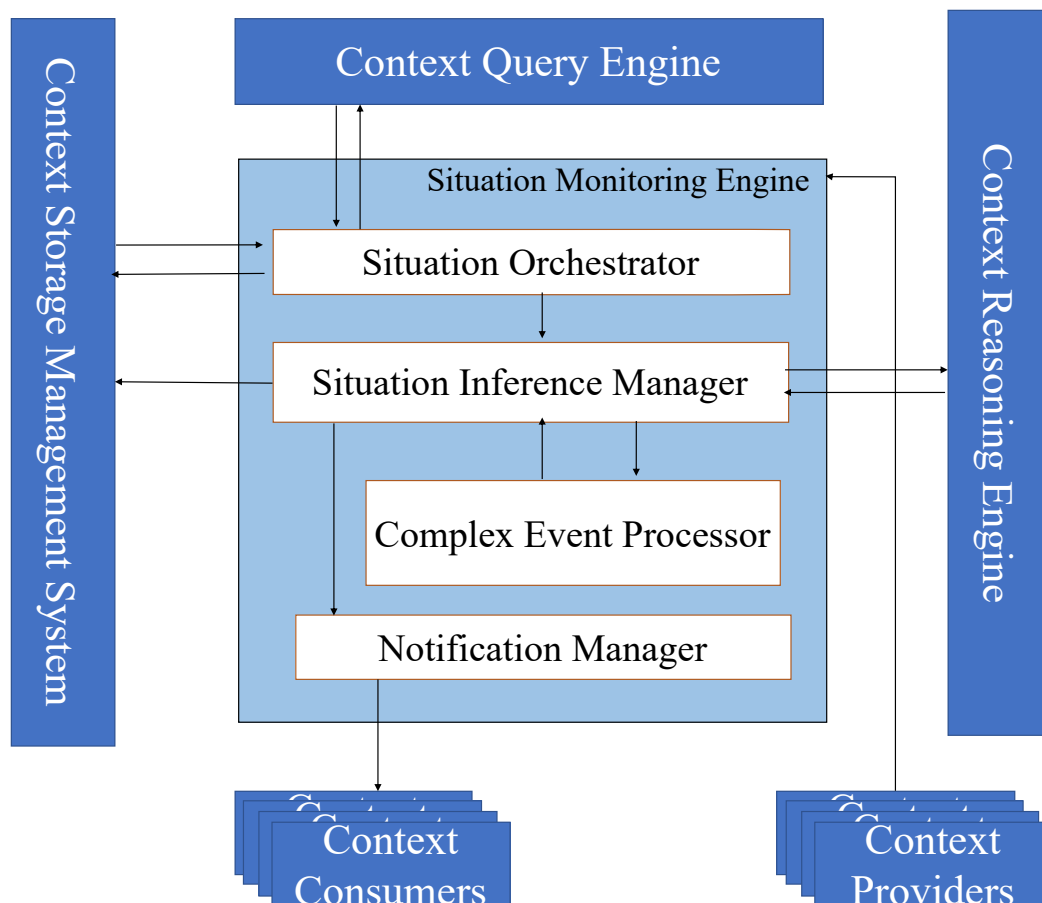
Context Request Predicate	Context Service Predicate	Similarly function	Computed Similarity
$p^{cr}_1(\text{geo})$	$p^{cs_1}_1(\text{geo})$	Geo-based Similarity	$\frac{500}{680} \approx 0.73$
$p^{cr}_2(\text{price})$	$p^{cs_1}_2(\text{price})$	Continues Similarity	$\frac{5-0}{5-0} = 1$
$p^{cr}_5(\text{facilities})$	$p^{cs_1}_3(\text{features})$	Set Similarity	$\frac{2}{2} = 1$
$p^{cr}_6(\text{hasCapacity})$	$\emptyset$	NA	$\emptyset$
$p^{cr}_7(\text{hasCapacity.freshness})$	$p^{cs_1}_4(\text{hasCapacity.freshness})$	Continues Similarity	$\frac{200-0}{200-0} = 1$

**Table 4.6** - outcome of CCM for the second conjunction in  $DNF(Ps_{cr} \wedge Ps_{cs_1})$

Context Request Predicate	Context Service Predicate	Similarly function	Computed Similarity
$p^{cr}_3(\text{geo})$	$p^{cs_1}_1(\text{geo})$	Geo-based Similarity	1
$p^{cr}_4(\text{price})$	$p^{cs_1}_2(\text{price})$	Continues Similarity	$\frac{2-0}{6-0} \approx 0.33$
$p^{cr}_5(\text{facilities})$	$p^{cs_1}_3(\text{features})$	Set Similarity	$\frac{2}{2} = 1$
$p^{cr}_6(\text{hasCapacity})$	$\emptyset$	NA	$\emptyset$
$p^{cr}_7(\text{hasCapacity.freshness})$	$p^{cs_1}_4(\text{hasCapacity.freshness})$	Continues Similarity	$\frac{200-0}{200-0} = 1$

## 4.5 SITUATION MONITORING ENGINE (SME)

The Situation Monitoring Engine (SME) is responsible for monitoring incoming contexts, detecting situations, and notifying context consumers about situations of their interest or performing the actuation process. In this section, we describe the architecture and workflow of this module.



**Figure 4.11** - Situation Monitoring Engine

Figure 4.11 shows the architecture of the SME. This engine monitors the real-time context of the IoT entities, which are used in at least one registered subscription for situation query. Moreover, SME initiates the actuation procedure by notifying context consumers when their situation of interest is detected.

SME has three main components: Situation Orchestrator (SO), Situation Inference Manager (SIM), and Notification Manager (NM). Situation Orchestrator (SO) is mainly responsible for retrieving all the related subscriptions to an incoming context update. It

also fetches the related contextual attributes, which are needed to process the retrieved subscriptions. Situation Inference Manager (SIM) is responsible for processing the retrieved subscriptions and making the decision to inform the corresponding context consumers or not. Lastly, Notification Manager (NM) is responsible for pushing notifications to subscribed Context Consumers (CC).

**Figure 4.12 - SME workflow**

The workflow of SME is illustrated as a flow of events in a sequence diagram in Figure 4.12. SME receives updates about the state of IoT entities in the form of messages from Context Providers (CP). Each message is related to one specific entity and contains real-time values of context that describes the current state of the entity. All the incoming messages are placed in a queue. The SO reads from this queue and processes the incoming messages accordingly. In the first step, SO sends a request, containing the received message to Context Storage Management System (CSMS) in order to retrieve the subscriptions that potentially can be triggered by the incoming message. To achieve

that, CSMS issues a query to the underlying data storage system, which checks the *Related Entities* part of the registered subscriptions. The matching subscriptions will be sent back to the SO for further processing. CSMS also updates the state of the related entity based on the incoming message. Then, SO starts processing each of the retrieved subscriptions in parallel.

SO is also responsible for fetching all the required contextual information for evaluating subscriptions. As mentioned earlier, CDQL allows situations to be defined based on the context of several IoT entities. However, the incoming context update only includes partial information about the situation of one entity. Therefore, to process subscriptions, it might be required to fetch the additional contextual information. Consequently, the SO executes the parsed query through Context Query Engine (CQE), which is a part of the subscription, to get all the required information for processing the subscribed situation query. After acquiring all the required contextual information, SO will pass the incoming context update, the result of the executed query, and the retrieved subscription to the Situation Inference Manager (SIM).

The SIM processes the received information and produces a Boolean output, which states whether the situation is detected or not. To do so, SIM evaluates the satisfiability of the subscription's WHEN clause. The WHEN clause contains one or several conditions connected with logical operators. Therefore, to assess the occurrence of the situation, it is needed to validate all the conditions one by one. These conditions can be classified into three types, basic conditions, CEP-based conditions, and CST-based conditions. Basic conditions only contain an individual context (e.g.  $\text{room.temp} < 10$ ) or a built-in function (e.g. "distance"). In this case, SIM retrieves the value of the context or executes the built-in function and evaluates the condition.

The CEP-based conditions use one of the CEP functions (i.e., trend, windowing, temporal relation). In this case, SIM prepares the arguments of the CEP function and passes them to the CEP engine. For example, consider the following CEP function: "decrease(distance(driver,car), 5mins)". In this case, SIM calculates the distance between the driver and the car and passes the value to the CEP engine. Then, the CEP engine will notify the SIM if the value has decreased during the last five minutes or not.

The last type of condition is CST-based. This type is used to handle uncertainty by performing probabilistic reasoning (Padovitz et al., 2004), (Medvedev et al., 2018). This type of reasoning is encapsulated as functions in CDQL. We refer to this type of functions as s-Function. When a condition contains an s-Function, in the first step, SIM prepares the arguments. Then, it will send a request to Context Reasoning Engine (CRE) by passing the parameters and the function definition.

For example, consider a condition that contains the aforementioned `goodForWalking` s-Function. In order to validate this s-Function, SIM will retrieve the context about the related entities (i.e. Weather and Location) and send it to the CRE together with the `goodForWalking` definition, which was introduced in Code block 3.11.

---

**Algorithm 3** Expression Evaluator's algorithm

---

```

1: procedure EVALUATE(event, res, sub)
2:   queue  $\leftarrow$  sub.situation
3:   Initialize stack
4:   while queue.hasNext  $\neq$  false do
5:     token  $\leftarrow$  queue.next
6:     switch token.type do
7:       case attribute
8:         value  $\leftarrow$  res.get(token)
9:         stack.push(value)
10:      case function
11:        arguments  $\leftarrow$  res.get(token.arguments)
12:        value  $\leftarrow$  execute(token, arguments)
13:        stack.push(value)
14:      case operator
15:        value  $\leftarrow$  applyOperator(token, stack)
16:        stack.push(value)
17:   return stack.pop()

```

---

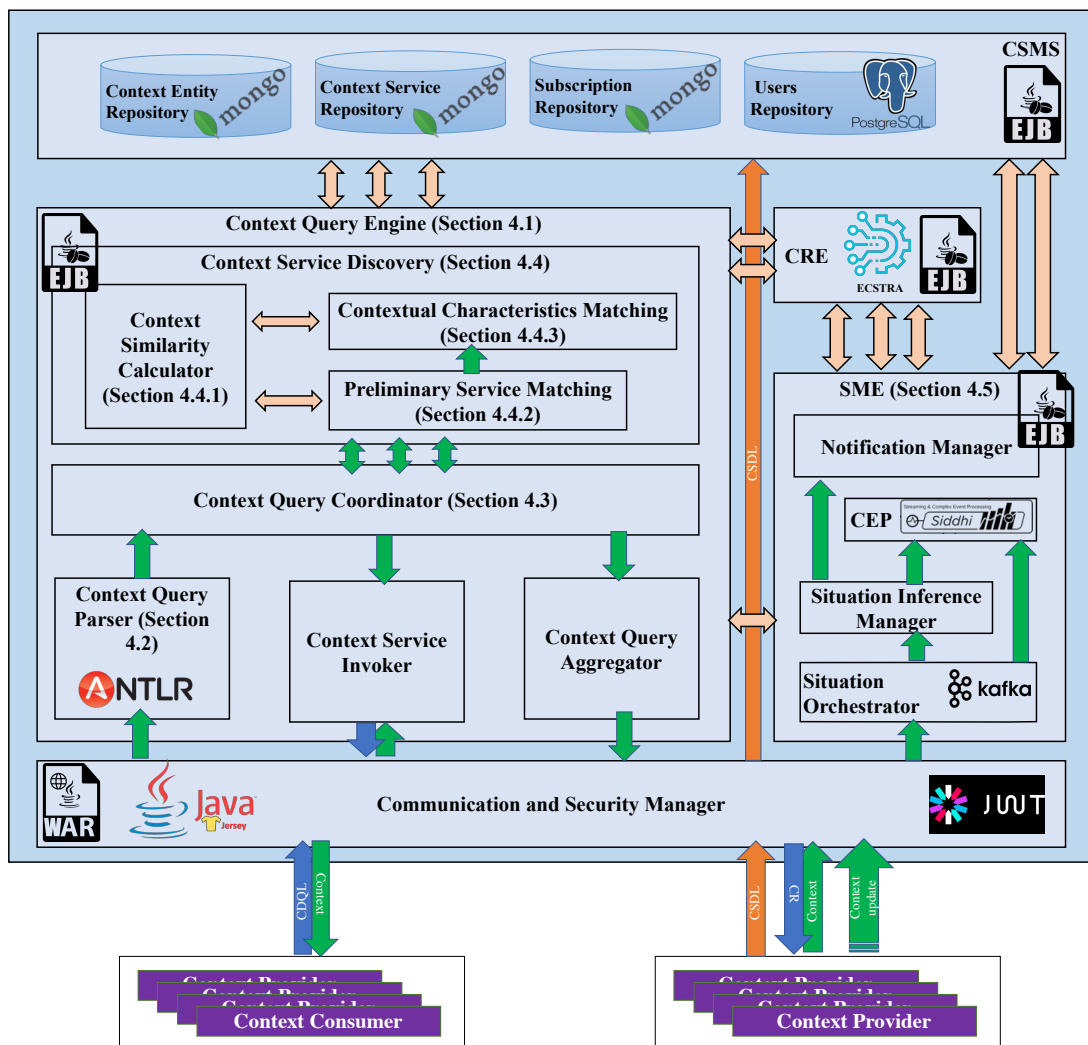
**Figure 4.13** - Situation Inference Algorithm

After evaluating all the conditions, SIM will assess the validity of the whole WHEN clause. The algorithm of situation inference process is depicted in Figure 4.13. Finally, if the situation is detected, a notification will be sent to the context consumer by the Notification Manager(NM) module using the provided callback method. If the callback method is not provided, the result of the query will be saved in CSMS, and the consumer can pull the result later using the subscription identifier. Moreover, the detected situation will be passed to the SME as a new context update. The reason behind this is that every

detected situation can be viewed as an incoming high-level context information, which can potentially trigger another subscription.

## 4.6 IMPLEMENTATION

Based on the presented reference architecture of Context-as-a-Service platform in Section 3.1.2, and the concepts presented in the previous sections of the current chapter, we have implemented a prototype of CoaaS platform. Figure 4.14 presents the architecture of the implemented context management platform.



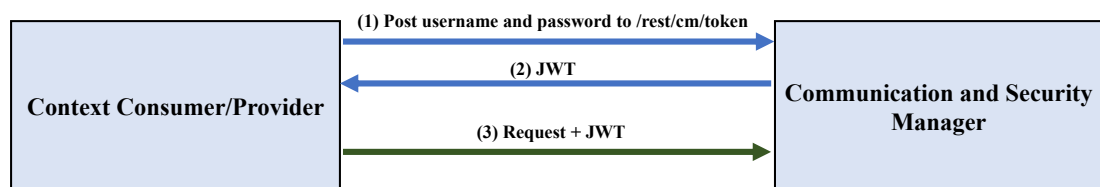
**Figure 4.14** - Architecture of prototype implementation of CoaaS platform

As described in Section 3.1.2, CoaaS platform consists of five main components: (i) Communication and Security Manager (CASM), (ii) Context Query Engine (CQE), (iii) Situation Monitoring Engine (SME), (iv) Context Storage Management System

(CSMS), and (v) Context Reasoning Engine (CRE). In the current implementation, which has around 1.3 million lines of code, we have developed CoaaS as an Enterprise application using Java Enterprise Edition 7 (Java EE 7) framework. In this regard, each of the abovementioned components is implemented as a separate Java EE component. Therefore, the implemented prototype of CoaaS platform is extensible by simply replacing its components with either newly developed parts or by integrating already existing ones. The rest of this section briefly presents the description of the implementation of each of these components.

The **Communication and Security Manager** (CASM) is implemented as a RESTful web service using Jersey 2.8 framework<sup>3</sup>. CASM provides an interface that supports the proposed languages presented in Chapter 3, namely Context Service Description Language (CSDL) and Context Definition and Query language (CDQL). Using this interface, clients can perform several operations, such as querying contextual information, registering context services, updating context information, and subscribing to certain situations about their entities of interest.

Moreover, this component is enhanced with a token-based authentication and authorisation mechanism, which is implemented through JSON Web Token (JWT)<sup>4</sup>. JWT is an open standard that defines a compact and self-contained way for securely transmitting information (Jones, Bradley, & Sakimura, 2015). The diagram depicted in Figure 4.15 shows how the implemented authentication and authorisation mechanism works.



**Figure 4.15** - Authentication and authorisation mechanism

<sup>3</sup> <https://jersey.github.io/>

<sup>4</sup> <https://jwt.io/>



In the first step, clients acquire an authorisation token by sending an authentication request that contains the client's username and password to the CASM via the URL “/rest/cm/token”. Then, based on the provided credentials, CASM authenticates the user by using Java Authentication and Authorization Service<sup>5</sup> (JAAS). If the client is successfully authenticated, a JSON Web Token (JWT) will be returned. Code block 4.8 shows how JWT can be acquired.

```
curl -X POST \
  https://localhost:8080/CoaaSMono-web/rest/cm/token \
  -H 'Content-Type: application/x-www-form-urlencoded' \
  \
```

**Code block 4.8** - Example of authentication request

Using the acquired token, clients can securely invoke CoaaS APIs. In this regard, they should provide the JWT in the ‘*Authorisation*’ header of the HTTP request using the ‘*Bearer*’ schema. Then, CASM checks for a valid JWT in the ‘*Authorisation*’ header, and if it is present, the client will be allowed to access protected resources.

The CoaaS platform has four main Restful APIs that are presented in Table 4.7. To enable secure communication between clients and CoaaS, all these APIs are only accessible via HTTPS protocol.

**Table 4.7** - CoaaS interface endpoints

Address/method	Short description	Accepts
/rest/cm/token (POST)	Authentication API	Username and Password
/rest/cm/query (POST)	CDQL query API	<ul style="list-style-type: none"> <li>• <b>CDQL query</b> <ul style="list-style-type: none"> <li>○ <b>CQL</b> <ul style="list-style-type: none"> <li>▪ <b>Pull-based query</b></li> </ul> </li> </ul> </li> </ul>

<sup>5</sup> <https://docs.oracle.com/javase/8/docs/technotes/guides/security/jaas/JAASRefGuide.html>

		<ul style="list-style-type: none"> <li>▪ <b>Push-based Query</b> <ul style="list-style-type: none"> <li>○ <b>CDL</b></li> </ul> </li> </ul>
<b>/rest/cm/register/ (POST)</b>	Context Service registration API	CSDL Service description
<b>/rest/cm/event (POST)</b>	Context update API	Context update

The main API for context consumers is the CDQL query interface, which is accessible via the URL ‘/rest/cm/token’. This interface accepts a CDQL query as input and based on the type of the provided query, it returns either contextual information (in the case of pull-based queries), a subscription ID (in the case of push-based queries), or status of the executed query (in the case of push-based query CDL queries). The code snippet provided in Code block 4.9 shows how this interface can be invoked.

```
curl -X POST \
  http://localhost:8080/CommunicationManager/rest/api/cm
\
  -H 'authorization: Bearer {auth_token}' \
  -d '{CDQL_QUERY}'
```

**Code block 4.9** - Example of issuing CDQL query

As explained in Section 3.1.1, CoaaS can interact with context providers (CP) in two ways, either by fetching context on-demand or through receiving context/data streams. In the first case, the CPs must have registered the description of their services by sending a context service registration request. In order to do this, they need to describe their context service using CSDL language and send the service description as a body of an HTTP POST request to the CoaaS service registration API (i.e. /rest/cm/register/). After successfully registering a context service, CoaaS can retrieve data about the registered service’s IoT entities by sending requests to the corresponding provider on-demand.

```

curl -X POST \
http://localhost:8080/CommunicationManager/rest/api/cm/
event \
  -H 'authorization: Bearer {token}' \
  -H 'content-type: application/json' \
  -d '{"@id":"parking.mpnash.edu/entities/p1",
    "timestamp":1520575780,
    "exitRate":"high",
    "capacity" : {
      "@Type" : "RealTimeCapacity",
      "Monash:Blue" : {
        "date" : 1520575780,
        "maximumValue" : 400,
        "currentValue" : 229
      },
      "Monash:Red" : {
        "date" : 1520575780,
        "maximumValue" : 3400,
        "currentValue" : 342
      }
    }
  }'

```

#### **Code block 4.10** - Example of sending context update

As mentioned above, CoaaS can also process streams of context updates, which CPs are sending to the platform. Context updates contain updates of the entities' states and are processed by CoaaS to monitor situations. Therefore, CoaaS has an API that allows CPs to send context updates to the CoaaS platform. As explained in Section 4.5, these updates are percolated through the registered PUSH-based queries, enabling the situation awareness. Moreover, these updates are cached in the CoaaS storage (i.e. CSMS), mainly for the purpose of using these data to serve pull-based queries. The code snippet provided in Code block 4.10 shows how the context update API can be invoked.

The **Context Query Engine (CQE)** has been implemented as an Enterprise Java Bean (EJB), based on the provided architecture in Section 4.1 and the concepts and the algorithms presented in preceding sections. To parse the incoming queries, a query parser is developed by using Antlr 4.6<sup>6</sup>. ANTLR (ANother Tool for Language Recognition) is a parser generator for reading and processing structured text. This framework accepts a formal grammar (written in an EBNF like format) as input and generates a parser for that language. The generated parser can automatically build parse trees, which are data structures representing how a grammar matches the input. ANTLR also automatically generates tree walkers that can be used to visit the nodes of those trees to execute application-specific code. The CDQL grammar for the generation of ANTLR parser is provided in Appendix B.

The **Situation Monitoring Engine (SME)** has also been developed as an EJB. As SME should be able to process millions of messages, we have implemented a distributed message queue using Apache Kafka<sup>7</sup> framework. When a context provider sends a context update message, that message would be entered in the message queue. Then, the Situation Orchestrator (SO), which has been implemented as a stateless Message Driven Bean (MDB)<sup>8</sup>, reads and processes the incoming messages. To improve the performance of SME, we have configured SO in such a way that it can process several context updates in parallel. As mentioned in Section 4.5, in order to process incoming context updates, we have integrated an existing Complex Event Processing Engine (CEP) in SME, called Siddhi. Siddhi CEP is a lightweight CEP engine that can run as an embedded Java library and process incoming context update to detect patterns and sequences.

To implement the **Context Reasoning Engine (CRE)**, we have adopted an existing context-awareness and situation-awareness framework called ECSTRA (Boytssov & Zaslavsky, 2011). ECSTRA builds on the basis of context spaces theory (Padovitz et al., 2004). This framework provides a comprehensive solution to reason about the context from the level of sensor data to the high level situation. In order to integrate ECSTRA in CoaaS, we have implemented another EJB that uses a java implementation of

---

<sup>6</sup> <https://www.antlr.org/>

<sup>7</sup> <https://kafka.apache.org/>

<sup>8</sup> <https://docs.oracle.com/javase/6/tutorial/doc/gipko.html>

ECSTRA framework. Using this EJB, other components can use the ECSTRA framework to reason about context data.

The **Context Storage and Management System (CSMS)** has been implemented based on the architecture presented by Medvedev et al. (2017). In the current implementation, CSMS has four repositories, namely context entity repository, context service repository, subscription repository, and user repository. The first three repositories, which are used to store context data, have been implemented using MongoDB<sup>9</sup>. On the other hand, the user repository that contains clients' profile, including their credentials, is implemented as a relational database using PostgreSQL<sup>10</sup>. Moreover, CSMS has an interface, which is also implemented as an EJB, that allows other components to access and store data in the aforementioned repositories.

Furthermore, to ease the development of context queries and service definitions, a specialised web-based IDE has been developed. The main features of the IDE are: (i) CDQL syntax highlighting, (ii) auto-completion of CDQL keywords and terms coming from integrated semantic vocabularies and standards, (iii) visualising the execution plan of parsed query, (iv) showing errors, warnings, and recommendations to CDQL developers, and (v) managing authorization tokens. A screen dump of the CoaaS IDE is presented in Figure 4.16.

On top of that, as it can be seen in Figure 4.17, we have implemented a web-based user interface that allows context consumers to view and manage their subscribed push-based queries.

---

<sup>9</sup> <https://www.mongodb.com/>

<sup>10</sup> <https://www.postgresql.org/>

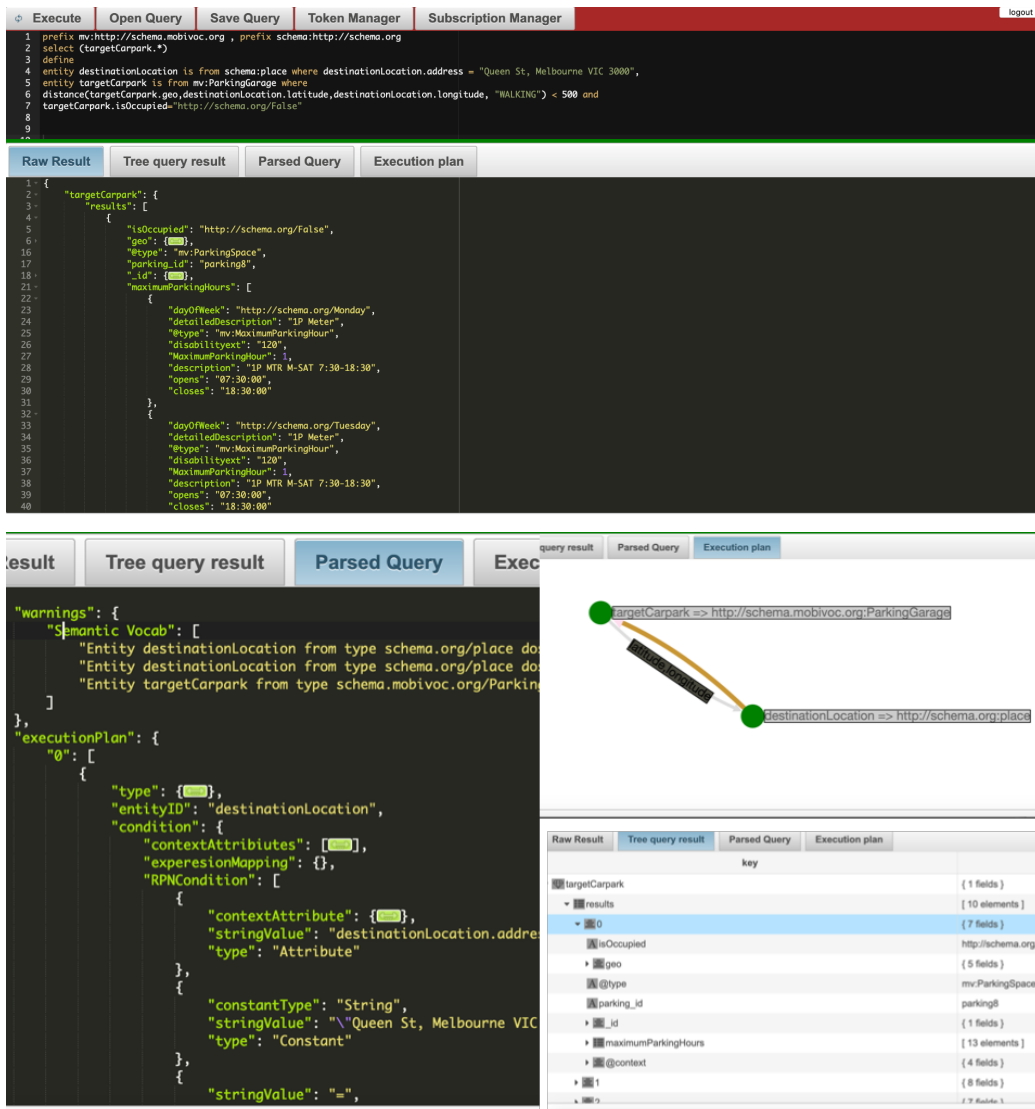


Figure 4.16 - CoaS IDE

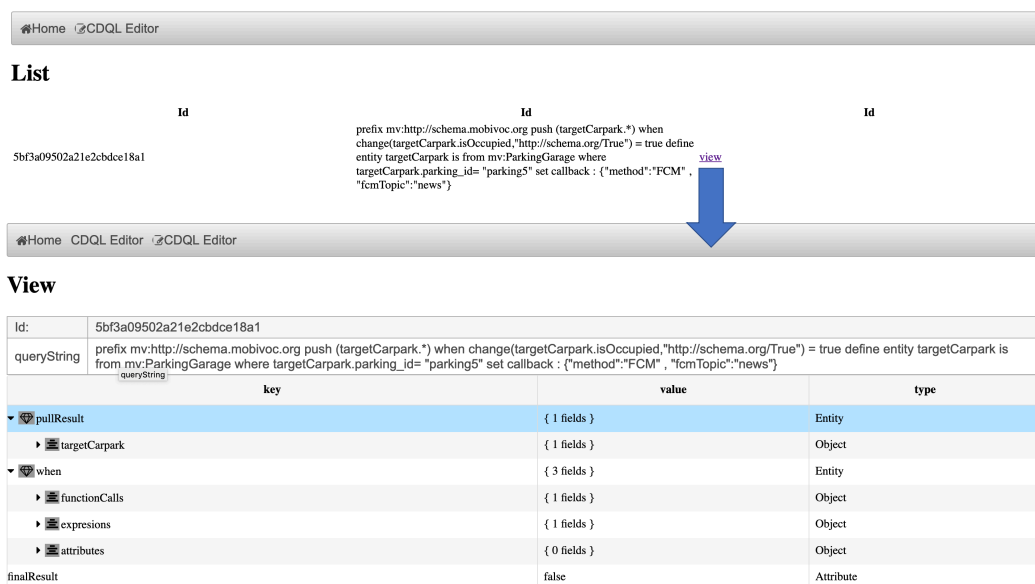
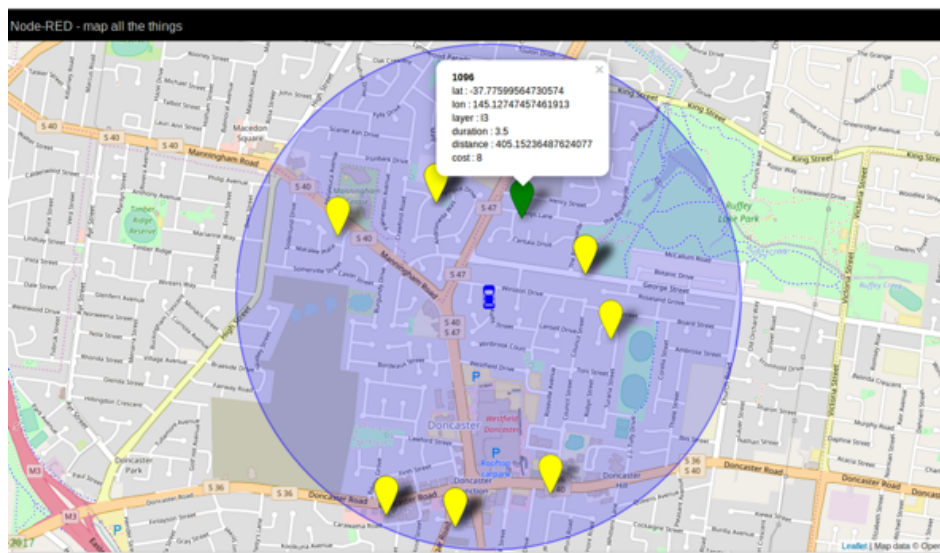
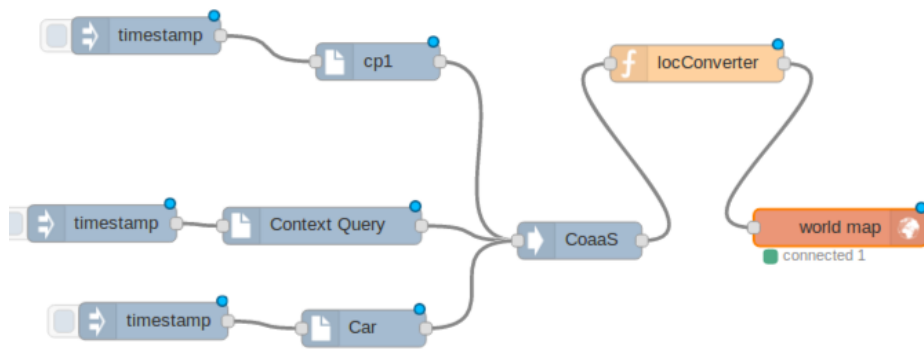


Figure 4.17 - Situation Monitoring Interface

We have also integrated the current prototype of CoaaS platform in Node-red<sup>11</sup> for visualisation purposes. Node-red is a well-known flow-based programming tool for the IoT. It allows wiring together hardware devices, APIs and online services by providing a browser-based editor.

We developed four new custom nodes and added them to node-red. These nodes are ‘CoaaS’, ‘context service’, ‘context query’, and ‘car’ entity, which is a specific type of context consumer. By using these nodes, it is possible to generate a flow to register context service and execute context queries. Figure 4.18 shows a sample workflow developed to demonstrate the car-park use-case as well as the visualisation of the query outcome.



<sup>11</sup> <https://nodered.org/>

### Figure 4.18 - Node-red based example

The current implementation of CoaaS platform is available as a Docker<sup>12</sup> image and can be downloaded from the following link: <https://hub.docker.com/r/ahas36/coaas>.

Moreover, to further ease the deployment of CoaaS platform, we have created a docker-compose file that sets up all the required development environment for CoaaS platform and automated its installation and configuration. The docker-file is available online at <https://github.com/ahas36/Context-as-a-Service>.

## 4.7 SUMMARY

In this chapter, we have proposed, designed, and implemented a mechanism for the execution of complex context query. This mechanism, which is an integral part of Context-as-a-Service platform, consists of two engines, namely Context Query Engine (CQE) and Situation Monitoring Engine (SME).

The Context Query Engine (CQE) is mainly responsible for parsing the incoming queries, generating and orchestrating the query execution plan, and producing the final query result. Furthermore, CQE is also in charge of finding the most appropriate context service for an incoming request. To achieve this goal, we have designed a Context Service Discovery (CSD) method. CSD's workflow consists of two parts. First, it finds context services that match the requirements of a context request. Then, based on the discovered services, it returns a sorted set of the best available context services that can satisfy the requirements of a request.

The Situation Monitoring Engine (SME) is designed to support continuous monitoring of incoming context, to infer situations from available context, to detect changes in situations and to provide notification of detected changes. This component monitors the real-time context of the IoT entities and reasons about their situations. It also initiates the actuation procedure by notifying context consumers when their situation of interest is detected.

Moreover, as a proof of concept, a prototype of CoaaS platform has been implemented. This prototype has a scalable, fault-tolerant microservices-based design,

---

<sup>12</sup> <https://www.docker.com/>



making it ideal for cloud deployment. To this regard, each of the CoaaS components are implemented as a Java EE component.

# Chapter 5: CDQL, CQE and SME: Evaluations

---

In Chapter 3 we have proposed and discussed two novel languages that enable IoT devices and services to publish and query context seamlessly. Moreover, in Chapter 4 we have proposed, designed, and implemented two mechanisms to execute complex context query and monitor the situation of context entities. This chapter evaluates our proposed approaches and related algorithms. We use the prototype of CoaaS platform developed in Chapter 4 to conduct our experiments.

In this chapter, we first demonstrate the feasibility and applicability of Context Definition and Query Language (CDQL) by presenting exemplary queries for each of the use cases discussed in Section 1.2. Furthermore, for two of the use cases (i.e. smart parking recommender and vehicle preconditioning), we have implemented a proof of concept application to show how CDQL queries can be utilised to develop context-aware IoT applications.

We have also conducted multiple experiments based on real-world and synthetic datasets to evaluate the ability of the proposed solution, namely Context Query Engine (CQE) and Situation Monitoring Engine (SME), to handle the load in large-scale IoT environments.

## 5.1 CDQL QUERY DEMONSTRATION

In this section, we demonstrate how CDQL facilitates querying context for the use cases described in Section 1.2, which are school safety, smart parking recommender, and vehicle preconditioning. We illustrate how CDQL can be used to represent and describe the context entities, their relationships, and context queries to fulfil the requirements we identified in Section 2.6.5.

### 5.1.1 USE CASE 1: SCHOOL SAFETY

As described in Chapter 3, CDQL is capable of representing complex context queries concerning several entities. Furthermore, it also supports definition and querying of high-level context. To illustrate these functionalities, we will use the school safety use case, where John is late and looking for a trusted parent to pick up her daughter, Hannah, from school. We start this query by defining the involved entities. As this query is designed to be executed by John's device, we first define John by using his unique user ID. Then, by applying the parenthood relationship, the entity that represents Hannah in this query can be identified. In the same manner, by using a membership relationship, we can use entity Hannah to define Hannah's school. As we define the entity that represents Hannah's school, other school students can also be defined as well.

The two remaining entities for this query are car and parent. For representing cars, we need to add two constraints: one on the available number of seats in the selected car, and whether the car is close enough to the school or not. Then, as a final step, the parent entity can be defined by using the ownership relationship that indicates the selected persons who have a car with an empty seat near Hannah's school, the parenthood relationship to show that the selected person is the parent of one of Hannah's fellow students, and the friendship relationship to indicate that selected parent is trusted by John. Code block 5.1 shows the complete CDQL query for this use case.

This example clearly illustrates the power of CDQL to express very complex queries that need to acquire contextual information from several heterogeneous entities.

Two other important aspects for such a context query that access sensitive personal information are privacy and security. In Section 4.6, we briefly explained how the current implementation of CoaaS handles authentication and authorization. As these aspects are mostly handled by the underlying platform, not the language itself, they are hence not described in detail here.

```

prefix schema:http://schema.org
select (parents.*)
define
entity john is from schema:person where john.id=
"john.id"
entity hannah is from schema:person where hannah.parent
contains john,
entity school is from schema:ElementarySchool where
school.member contains hannah,
entity otherStudents is from schema:person where
otherStudents.memberOf contains school and
distance(otherStudents.location, school.location) <
{"value":100,"unit":"m"},
entity car is from schema:Car where
distance(car.location, school.location, "driving") <
{"value":500,"unit":"m"} and car.vehicleSeatingCapacity
> 0 ,
entity parents is from schema:person where
parents.children containsAny otherStudents and
parents.knows contains john and isDriving(parents) >
0.90 and parents.owns containsAny car

```

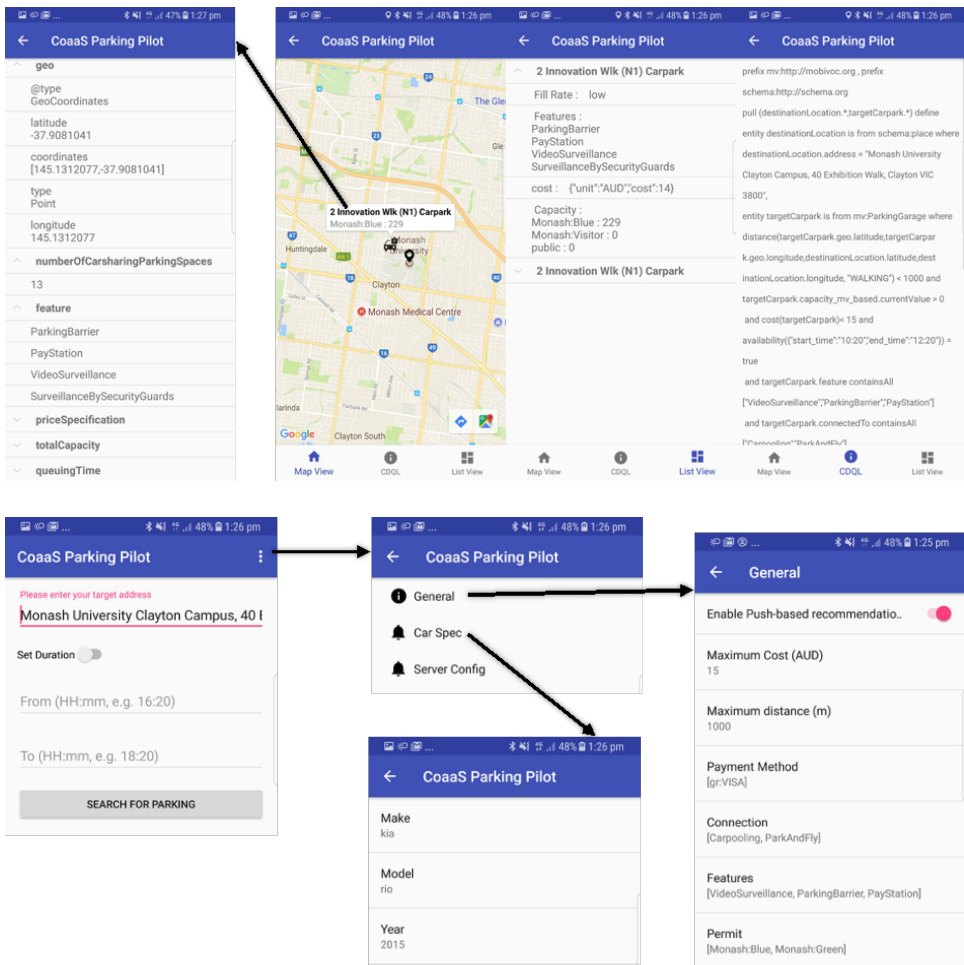
**Code block 5.1** - CDQL query for school safety use-case.

### 5.1.2 USE CASE 2: SMART PARKING RECOMMENDER

The second use case that we present in this chapter focuses on the development of a smart parking recommender application that utilises context to suggest the best available parking. To implement such an application, several challenges need to be addressed. First of all, it is essential to have access to live data regarding the availability of different parking facilities. The fact that these facilities are owned by different providers (e.g., city administrators, building owners, and organisations) makes the process of data retrieval even more complicated. Further, to be able to provide personalised suggestions to users, we need to consider additional factors, such as user

preferences, car specifications, and weather conditions. In addition, some of the data need to be inferred before being used. Addressing all these challenges needs a considerable amount of effort, even for an expert team of software developers.

However, with the help of the CDQL language, all the above-mentioned context can be retrieved by issuing a CDQL query. To prove our claim, we developed an Android mobile application, which automatically provides suggestions about available parking spaces to drivers using real data. To achieve this goal, we composed a parameterised push-based CDQL query that will be triggered when the consumer's car gets close to the user's destination. This query takes into account different contextual attributes such as weather conditions, walking distance, required parking facilities, and cost. As depicted in Figure 5.1, this application also provides an interface for users to enter their parking-related preferences in the application.



**Figure 5.1** - Smart parking suggestion application screenshot

Code block 5.2 depicts an example of this query.

```
prefix mv:http://schema.mobivoc.org ,
schema:http://schema.org
select (targetCarparks.*) when
distance(consumerCar.location,targetLocation)<{"value":
1,"unit":"km"}
define entity targetLocation is from schema:place where
targetLocation.address = "Wellington Rd, Clayton VIC
3800"
entity consumerCar is from schema:car where
consumerCar.vin="KNADN512MG6649868"
entity targetWeather is from schema:Weather where
targetWeather.location=targetLocation ,
entity targetCarparks is from mv:carpark where ( (
distance(targetCarparks.location,targetLocation,"walkin
g") < {"value":1500,"unit":"m"} and targetCarparks.cost
< 5 and goodForWalking(targetWeather)>= 0.7) or
(distance(targetCarparks.location,targetLocation,"walki
ng") < {"value":500,"unit":"m"} and
targetCarparks.cost < 10 and
goodForWalking(targetWeather) < 0.7) ) and
targetCarparks.facilities contains "charging point"
and targetCarparks.minHeight > consumerCar.height and
targetCarparks.minWidth > consumerCar.width and
isAvailable(targetCarparks.availability,{"start_time":"
11:30", "end_time":"16:50"})
```

**Code block 5.2** - CDQL query for smart parking recommender use-case.

This query is filled with preferences of a sample user as shown below:

- Required facility : Charging Point

- Parking time-frame : 11:30 to 16:50
- In case of good weather condition
  - Maximum walking distance : 1.5 km
  - Maximum cost: \$5
- Otherwise
  - Maximum walking distance : 0.5 km
  - Maximum cost: \$10

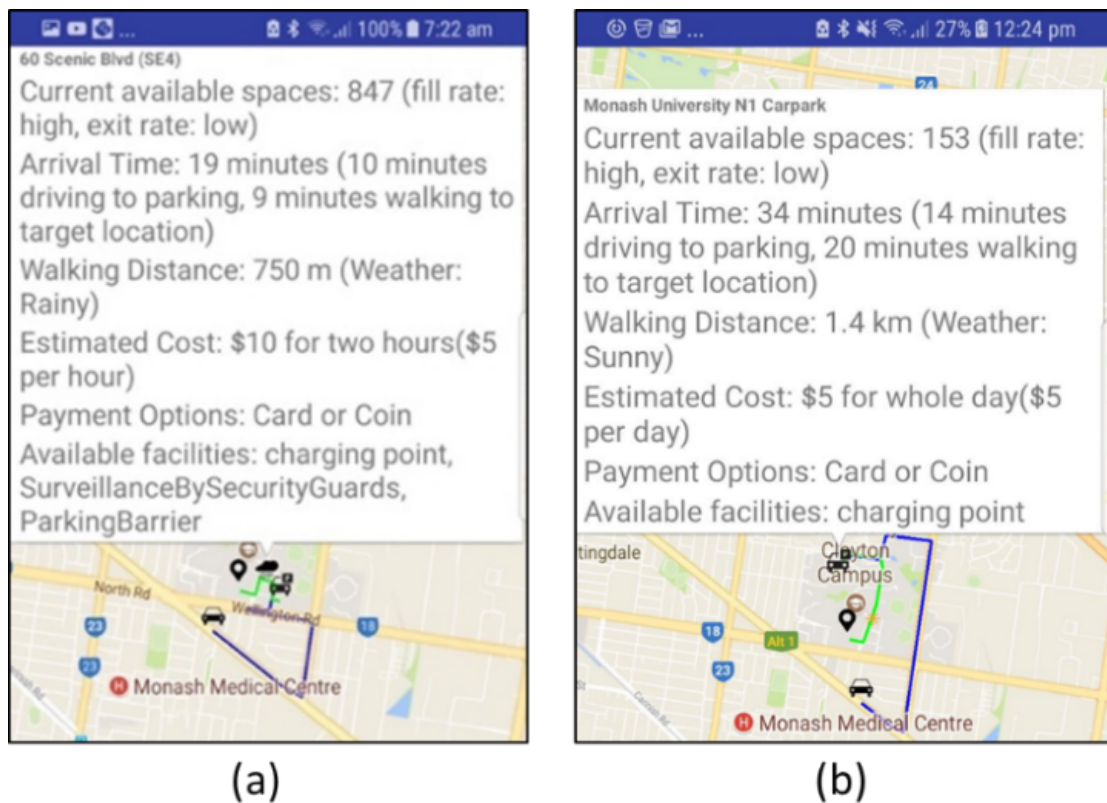
We registered four different context services in the Context-as-a-Service (CoaaS) platform based on the requirements of the scenario under consideration. The context services were:

- **Monash Parking API:** Monash has 10 different parking facilities in Clayton campus which are equipped with occupancy sensors. Further, Monash University has a web API that offers real-time vacancy information of its parking facilities. We registered this API in CoaaS as the main parking context provider.
- **VIN checker API:** In order to retrieve the specifications of the consumer car, we registered a context service that accepts Vehicle Identity Number (VIN) as input and provides the make and model of the car as output. It also provides car specifications such as height and width.
- **Google Location API:** Another context provider that is used in this scenario is Google Location API. This API has been used for reverse geocoding purposes to convert address to coordinates.
- **Weather API:** In order to fetch information about the weather conditions, we also registered a weather API that accepts location coordinates as its inputs and provides the weather conditions as output.

The application is also connected via Bluetooth to an OBD II device that reads the sensory data (e.g. VIN, speed, and fuel level) coming from the car's Controller Area

Network (CAN) bus. Then, the application takes this information, puts it in the query, and posts the CDQL query to CoaaS.

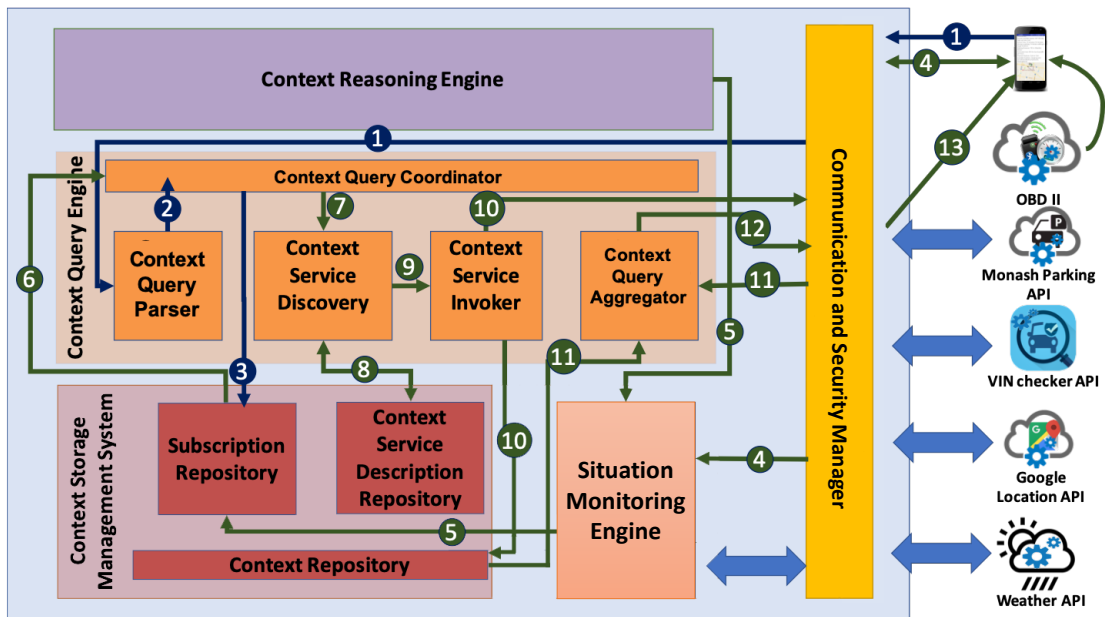
Figure 5.2 shows the screenshots of the developed application on two different days with different weather conditions. In Figure 5.2(a), the application has suggested a more expensive parking with a shorter walking distance because of the bad weather conditions. On the other hand, in Figure 5.2(b), the application has suggested a parking space that was cheaper but further away because it was a sunny day.



**Figure 5.2** - PoC parking application screenshot

Figure 5.3 illustrates the data flow of how CoaaS handles such a scenario. There are a number of different execution processes that can occur in CoaaS. However, here, we focus only on explaining the scenario where a user submits a push-based query and CoaaS provides the relevant data to the user.





**Figure 5.3** - Execution and interaction process of smart parking suggestion use-case

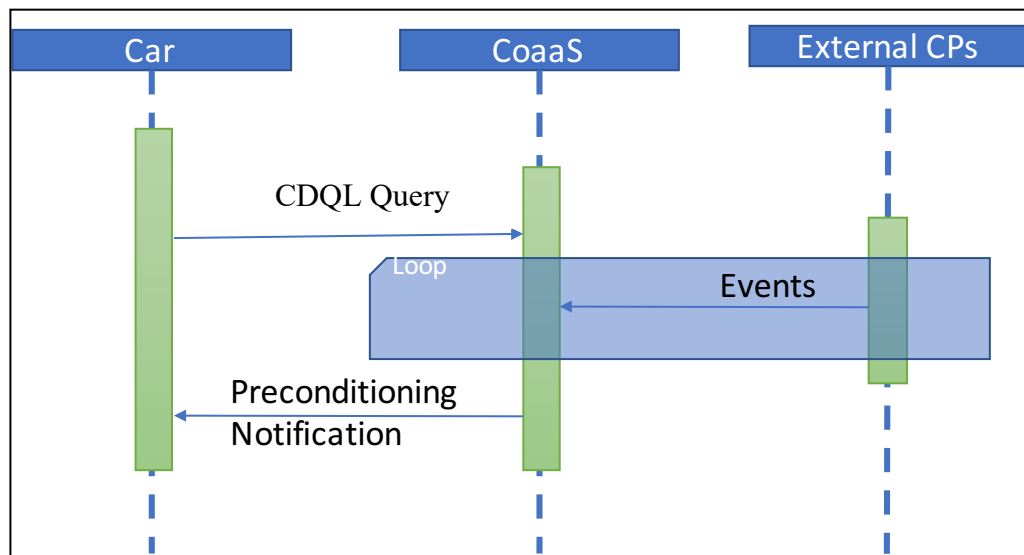
- **Step 1:** A mobile device sends the aforementioned push-based query to the CoaaS. Communication and Security Manager (CASM) receives the query and redirects it to the Context Query Engine (CQE) after the security check.
- **Step 2:** CQE parses the query, breaks the query into four context-requests, generates the execution plan, and sends the results to the Context Query Coordinator (CQC).
- **Step 3:** Since the incoming CDQL is a push-based query, the CQC registers the query and its triggering event (i.e. distance (consumerCar.location, target-Location) < 1km) in the subscription repository.
- **Step 4:** The mobile device keeps sending its location update to the CoaaS, which will be redirected into the Situation Monitoring Engine (SME).
- **Step 5:** SME analyses the incoming context updates with the help of Context Reasoning Engine (CRE), and checks whether any event/situation that triggers a query is detected or not.

- **Step 6:** When the car is getting close to Monash University, SME detects that the distance to the target is less than the predefined threshold (i.e. 1 km) and sends the corresponding query to the Context Query Coordinator (CQC).
- **Step 7:** CQC sends 4 context requests based on the query execution plan to the Context Service Discovery (CSD) module.
- **Step 8:** CSD receives the context-requests, finds the context services (providers) that can answer each context-request by looking up the Context Services Description Repository (CSDR). If more than one service is found for each request, it selects the best service based on Quality of Context (QoC) and Cost of Context (CoC).
- **Step 9:** CSD forwards the selected context services to the Context Service Invoker (CSI).
- **Step 10:** CSI sends the requests for contextual information to the selected providers. It is worth mentioning that if the required contextual information is already cached by the Context Storage Management System (CSMS), the service invoker will ask the CSMS for the required information instead of the actual context provider.
- **Step 11:** Context providers send the requested context to CoaaS, which will be forwarded to the Context Query Aggregator (CQA).
- **Step 12:** CQA produces the final query result by accumulating the incoming context from the providers. Further, it will use the Context Reasoning Engine (CRE) when it is needed to infer high-level context. For example, in this scenario, the reasoning engine will be called to infer if the weather condition is suitable for walking or not.
- **Step 13:** The result of the query will be pushed to the context-consumer.

### 5.1.3 USE CASE 3: VEHICLE PRECONDITIONING

The last use case in this section is vehicle preconditioning and shows how CDQL can be used to issue an actuation signal to turn on the car's air conditioning system. This use case is conducted in a real environment using a BMW i3 car.

Figure 5.4 shows the workflow of the experiment. The life cycle of this test is started by the car, which issues a PUSH-based CDQL query to CoaaS. This query, which is shown in Code block 5.3, represents a complex situation that contains several entities such as the driver, car, parking location and weather. Other factors are also covered and expressed in the query by considering the following questions:



**Figure 5.4** - Pre-conditioning scenario workflow

- Is there an upcoming meeting where the driver is likely to use the vehicle?
- Is the driver within walking distance from the car? Is the driver walking towards the car?
- Is the distance between the driver and the car less than the distance between the driver and the meeting location?
- Is the distance between the driver and the meeting location not within walking distance?

- Is the temperature lower or higher than a certain threshold, hence is pre-conditioning necessary?
- Is the vehicle connected to a charging point? Is the battery level sufficient for both pre-conditioning and driving to the next destination?

When CoaaS receives this query, it starts to monitor all the incoming events from external context providers that contain relevant contextual information about any of the entities mentioned above. Further, it evaluates the occurrence of the situation defined in the query and notifies the car when the situation is detected.

To test the use case, we created a meeting event in the driver's Google calendar, where the meeting location satisfied the mentioned criteria. We also developed a smartphone application that sends context updates containing the driver's current location to CoaaS. CoaaS was able to detect these changes in real time and send the corresponding situation notification (actuation) to the context consumer (BMW backend server) to start the car's climate control system as the driver started to walk towards the car. When CoaaS activated the climate control system during our test, the moment was captured by a camera and is shown in Figure 5.5



**Figure 5.5** - Activation of BMW i3 climate control system

```

prefix schema:http://schema.org
select (events.*,eventLocation.*,driver.*,car.*,temp.*)
when
timeDifference(events.startDate,currentTime("Australia/
Melbourne")) -
distance(car.geo,events.geo,"DRIVING").duration <
{"value":"30","unit":"minutes"}
and distance(car.geo,driver.geo,"WALKING").distance <
{"value":"500","unit":"meter"}
and distance(eventLocation,driver.geo,"WALKING").
distance > distance(car.geo,driver.geo,"WALKING")
and decrease(distance(driver.geo,car.geo).distance,
{"value":"2","unit":"minutes"}) and
(temp.airTemperature < 20 or temp.airTemperature > 25 )
define entity events is from schema:event where
events.attendee.email="biotope2018.au@gmail.com",
entity eventLocation is from schema:Place where
eventLocation.address = events.location.address,
entity driver is from schema:Person where
driver.driverID = "biotope",
entity car is from schema:Vehicle where
car.vehicleIdentificationNumber = "9d791e4d-8181",
entity temp is from schema:weather where
temp.location.latitude = car.geo.latitude
and temp.location.longitude = car.geo.longitude
set callback : {"method":"post" , "body":"<omienvelope
xmlns=\"http://www.opengroup.org/xsd/omi/1.0/\"
version=\"1.0\" ttl=\"0\"><write msgformat=\"odf\">
<msg><objects....\",\"url\":\"http://138.194.106.20/\",\"head
ers\":{\"ContentType\":\"text/xml\" } }

```

**Code block 5.3** -CDQL query for vehicle preconditioning use-case.

In the three use cases described in this section, we have demonstrated how complex use cases from different smart city applications can be implemented by issuing only one CDQL query. Moreover, we have shown how heterogeneous context services can be easily integrated into the CoaaS platform. While implementing each of the abovementioned use cases from scratch requires a considerable amount of time and effort from the developers, using the proposed query language significantly reduces the complexity of the task. Developers only need to issue one CDQL query for querying and monitoring context of several IoT entities and detecting complex situations.

## 5.2 COMPARISON OF CDQL WITH NGSI

As mentioned in Chapter 2, the most sophisticated existing context query language is NGSI (Open Mobile Alliance, 2012). Therefore, to illustrate the advantages of the proposed context query language, we compare CDQL with NGSI. To do so, we will first discuss how smart parking recommender use case and vehicle preconditioning use case can be implemented using NGSI language. We will then compare the implementation of these use cases in NGSI with CDQL and will discuss the outcome.

In the previous section, we showed how the required contextual information for parking recommender use case can be expressed with a single CDQL query. However, it is not possible to implement this use case with one NGSI query as NGSI is not expressive enough for such a complex scenario.

Code block 5.4 presents the pseudo-code for the of Use Case 2 using NGSI. As mentioned previously, NGSI only supports querying one entity type per request. As a result, in order to implement this use case that involves several entities, four context queries are required to be implemented and issued.

Moreover, NGSI does not support context reasoning and custom aggregation functions. Therefore, it is not possible to integrate such functions (i.e., `goodForWalking` and `isAvailable`) in NGSI queries and it is the responsibility of the developer of such an application to implement these functions. In addition, NGSI does not support ‘OR’ operator. Hence, if such an operation is needed, developers should implement several versions of a query and use ‘if’ statement to decide which one should be issued.

```

//Q1: Get information about the car
localhost:1026/v2/entities?type=Car&q=vin==KNADN512MG66
49868&attrs=width,height,length

//Q2: Get geo-coordinates by address
localhost:1026/v2/entities?type=place&q=address==Rio&q=
" Wellington Rd, Clayton VIC 3800"&attrs=latitude,
longitude

//Parse the Q2 result ...

//Q3: Get the weather information
localhost:1026/v2/entities?type=weather&georel=near;max
Distance:100&geometry=point&coords=-37.81, 144.95
&limit=1&orderBy=geo:distance&attrs=airTemperature

//Parse result of Q3 and Reason about good for walking situation ....

//Parse the result of Q1 ...

//Q4: Get the parking information
if (goodForWalking >=0.7) {
    localhost:1026/v2/entities?type=ParkingGarage&
    q=vehicleWidthLimitInM >=1500&q=cost<5&q=
    facilities == charging
    point&q=vehicleHeightLimitInM>=
    4600&q=vehicleLengthLimitInM>=1300georel=near;
    maxDistance:2000&geometry=point&coords=-37.81,
    144.95
} else {
    localhost:1026/v2/entities?type=ParkingGarage&q=ve
    hicleWidthLimitInM
    >=500&q=&q=vehicleHeightLimitInM>=
    4600&q=vehicleLengthLimitInM>=1300georel=near;
    maxDistance:500&geometry=point&coords=-37.81,
    144.95}

// iterate over the list of retrieved parking facilities and compute if they are
available or not ...

```

**Code block 5.4** - NGSi queries for smart parking recommender use-case.

Another use case that we discussed in the previous section is vehicle preconditioning (Use Case 3). Implementing this use case requires monitoring the context of several entities (e.g., driver, car, and weather) and reason about if the precondition should be initiated or not. While NGSI allows monitoring changes in context information, its subscription model is not sophisticated enough for this use case. Firstly, NGSI subscription model only supports monitoring context of a single entity type per subscription. Secondly, NGSI does not support situation inference and window functions.

Therefore, it is not possible to fully implement Use Case 3 using only NGSI without having these capabilities on the consumer's side. Code block 5.5 is an example of NGSI query for subscribing to receive notification when the distance between driver and a specific location is less than 500 meters.

```
{
  "description": "Notify when driver near specific
location",
  "subject": {
    "entities": [{"id": ".*", "type": "person"}],
    "condition": {
      "expression": {
        "q": " driverID== biotope"
        "georel": "near;maxDistance:500",
        "geometry": "point", "coords": "-37.81,144.95"}
      },
    },
  "notification": {
    "http": {
      "url":
"http://fiware:3000/subscription/preconditioning "
    },
    "attrsFormat" : "keyValues"
  }
}
```

**Code block 5.5** - NGSI subscription.



As illustrated in these use cases, the main advantage of CDQL over NGSI is the support for expressing multiple entities in one query. Due to this fact, several NGSI queries might be required to implement a use case that can be expressed with only one CDQL query. For example, in Use Case 2, the consumer will perform four NGSI queries, requiring extra time and network bandwidth for data transfer and processing. Moreover, having more queries makes the implementation and maintenance of context-aware IoT applications more complex.

Furthermore, the lack of support in NGSI to query more than one entity type in a request may lead to increase in several other unavoidable drawbacks, namely (i) difficulty to avoid retrieving data which is intermediate and may not be really needed in the final result, (ii) difficulty to protect intermediate data from unwarranted access, and (iii) difficulty to avoid network delays.

CDQL not only addresses the above shortcomings, it also has several other benefits compared to NGSI. For example, it is possible to integrate query optimisation to improve the overall performance of the system.

Apart from the number of supported entities, CDQL provides several other functionalities that are essential for a CMP and not supported in NGSI, such as the support of aggregation functions, window functions, situation inference functions, and temporal relations, just to name a few.

Based on the discussion above, we can make a claim that CDQL can provide significant benefit for CMP platforms compared to NGSI.

### 5.3 PERFORMANCE EVALUATION

This section describes the performance evaluation of the proposed Context Query Engine (CQE) and Situation Monitoring Engine (SME) that provide support for publishing and querying context through the use of CSDL and CDQL.

We will first describe the metrics and experimental environment of our evaluation in Section 5.3.1. Then, based on the provided metrics, we will present two sets of experiments to evaluate the performance of the proposed CQE and SME during the execution of CDQL queries. Section 5.3.2 presents the first set of experiments, which focuses on the execution of pull-based CDQL queries to demonstrate the performance of the CQE. The second set of experiments, which is provided in Section 5.3.3, focuses on evaluating the SME through the execution of push-based queries. Lastly, Section 5.3.4 presents another set of experiments for the evaluation of the proposed Context Service Discovery (CSD) approach.

#### 5.3.1 EXPERIMENT ENVIRONMENT AND METRICS

In order to evaluate the proposed solution, we used the current implementation of the CoaaS platform, which was described in detail in Section 4.6. During all the conducted experiments, the CoaaS platform was running as a web application on Payara Server 5.182 where the maximum JVM heap size and maximum thread pool size are 16 GB and 500 threads respectively. The Payara Server is hosted on a virtual machine located in the CSIRO Melbourne Cloud and running Debian GNU/Linux 8 (Jessie). The VM is running on an eight-core Intel(R) Xeon(R) CPU E5- 4640 0 @ 2.40 GHz instance with 64 GB RAM.

For our experiments, we used real parking data provided by the Melbourne city portal (“On-street parking data - City of Melbourne,” n.d.). This dataset contains information from in-ground car parking bay sensors deployed in the Melbourne Central Business district. Update frequency of the dataset is two minutes, and the number of parking spaces is 2767. Since we also wanted to test the scalability of CoaaS, we developed a script, which simulated more parking spots based on the aforementioned dataset.

We also developed a context provider simulator, which imitates the behaviour of IoT entities (i.e., driver location, car park status). Context updates are randomly generated in a way that each update has a 30% chance of triggering a subscription.

For all the experiments in this section, we used JMeter 4 to simulate and issue CDQL queries. We deployed the JMeter 4 in the same network where the CoaaS instance was running to minimise the network delay since we are only interested in measuring the performance of the CoaaS.

To measure the performance of the proposed solution, an evaluation framework is required. In recent years, several academic papers in the area of IoT platforms evaluation were published (da Cruz, Rodrigues, Sangaiah, Al-Muhtadi, & Korotaev, 2018; Medvedev, Hassani, et al., 2017; Pereira, Cardoso, Aguiar, & Morla, 2018; Salhofer & Joanneum, 2018; Williams, Aggour, Interrante, McHugh, & Pool, 2015). The TPC group also proposed a benchmarking framework for the evaluation of IoT Gateway Systems (“TPCx-IoT,” n.d.). These works are mainly focused on measuring the performance of IoT platforms in terms of ingestion and not paying enough attention to data retrieval performance. There are only a few works (“TPCx-IoT,” n.d.; Williams et al., 2015) that took data retrieval performance into account. However, in our opinion, the metrics used in these works are too basic and could not feature the actual performance of IoT platforms. Consequently, in the rest of this section we will discuss the main metrics that need to be considered in order to measure the data retrieval related aspects of CMPs.

During the development of the CoaaS platform our team was closely collaborating with academic and industrial partners involved in the bIoTope project (“bIoTope Project,” n.d.). As a result, we have determined the typical workflows and main requirements for the integration of a CMP with real-world smart city use cases. Consequently, we identified three main factors that can affect query execution performance. These factors are: (i) the number of registered entities in the platform, (ii) the number of parallel queries, and (iii) the complexity of incoming queries.

The first factor we take into account is the number of registered entities in the platform. This factor affects the query performance in two main ways: (i) the amount

of data returned as a result potentially increases, and (ii) scanning and processing large volumes of data is time consuming.

The second important factor is the number of parallel queries getting executed. As this number increases, race condition can occur due to limited available resources, which may lead to performance degradation.

The last factor we consider is the complexity of a query. Measuring this factor is more complicated compared to other factors, as it is a multidimensional metric and depends on a number of features.

The first feature that influences the complexity of a query is the number of defined entities. As mentioned earlier, each context query can contain several entities, where each defined entity represents a group of real-world entities (i.e. one or several) with specific characteristics. Further, the query execution plan generator will construct a context request for each entity defined in a query. Consequently, the number of defined entities in a query has a high impact on the query execution time as it indicates the number of nodes that needs to be traversed in the query graph. Therefore, this feature plays a vital role in determining the complexity of a query.

As mentioned in Chapter 3, each defined entity in CDQL is represented by several constraints connected with logical operators (i.e. and/or); where each constraint itself consists of two operands connected with an operator (e.g. '=', '<', '>', 'containsAny', etc). In CDQL, operands can be a literal (e.g. string or number), context attribute (e.g. car.speed), or a function. Functions can be categorised into four main groups, namely (i) geospatial functions, (ii) situation functions, (iii) windowing functions, (iv) aggregation and computational functions.

By considering the above description, the other influencing features in defining complexity of context queries are (i) the number of constraints, (ii) the ratio of 'ANDs' to 'ORs', (iii) the number of functions, and (iv) the type of functions.

In the rest of this section, we will design several experiments based on the factors discussed in this section to measure the performance of CoaaS data retrieval.

### 5.3.2 EXPERIMENT 1: CONTEXT QUERY ENGINE - PULL-BASED QUERIES

This experiment focuses on the performance evaluation of PULL-based queries that illustrate how well the CQE works. As results of the experiment are dependent on the infrastructure, especially on the application server, we conducted an initial test to find the maximum number of requests that our application server can handle. We found that in the current setup it is possible to serve a maximum of 570 HTTP POST requests per second.

At first, we studied the impact of query load to show how CQE performed when the number of concurrent queries increased. To this end, we gradually increased the query load (query per second) from 40 to 550 and measured the query response time. It is worth mentioning that during this experiment the number of registered parking spaces was equal to 2,767. Moreover, to take the impact of the complexity of queries into account, we repeated this experiment using four queries with an increasing level of complexity. These queries are shown in Table 5.1.

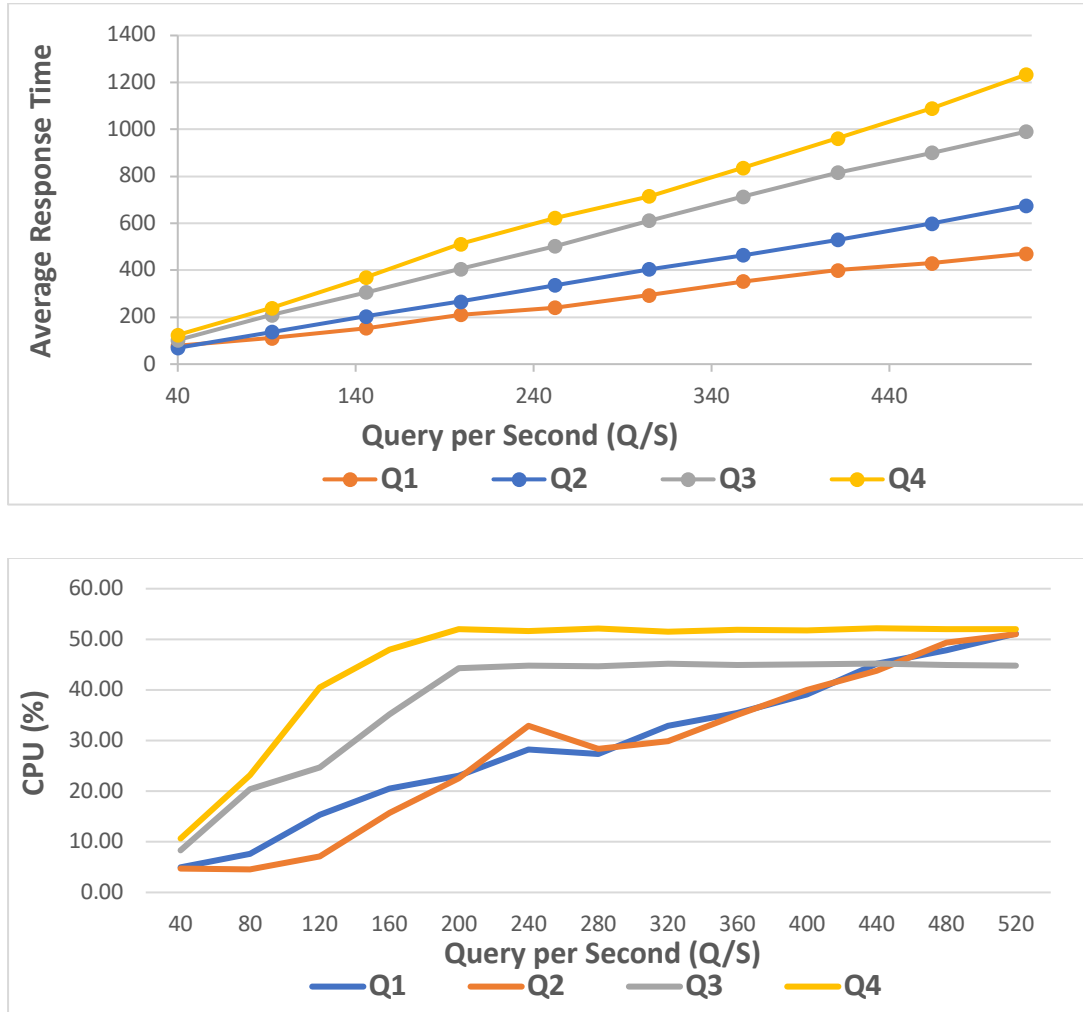
The first query (Q1) represented a search for a car park by providing its ID. The second query (Q2) was a location-based query, which searched for available parking spots near a specific coordinate. In the third query (Q3), we extended the previous query by taking the car specification (i.e. width, length, and height) into account, which required adding an entity representing the car in the query. In the last query (Q4), we added a situation reasoning function to the previous query. This function added the walking conditions between the destination and the car park into the scope.

**Table 5.1** - CDQL queries for performance evaluation

ID	Description	Query String
Q1	Query by ID	prefix mv:http://schema.mobivoc.org select (targetCarpark.*) define entity targetCarpark is from mv:ParkingGarage where targetCarpark.parking_id="parking1"
Q2	Location and isOccupied	prefix mv:http://schema.mobivoc.org , prefix schema:http://schema.org select (targetCarpark.*) define entity targetCarpark is from mv:ParkingGarage where distance(targetCarpark.geo,"-37.80303441012997","144.96765439598772","WALKING") < 50 and targetCarpark.isOccupied="http://schema.org/False"
Q3	3 entities with join (Car spec, Location, and Parking) + isOccupied	prefix mv:http://schema.mobivoc.org , prefix schema:http://schema.org select (targetCarpark.*) define entity targetLocation is from schema:Place where targetLocation.address="55 Pelham St, Carlton VIC 3053, Australia",entity targetCarpark is from mv:ParkingGarage where distance(targetCarpark.geo,targetLocation.latitude,targetLocation.longitude,"WALKING") < 20 and targetCarpark.vehicleLengthLimitInM > car.length,entity car is from schema:car where car.manufacturer="kia" and car.model="rio" and car.vehicleModelDate=2015 "

<b>Q4</b>	4 entities with join (Car spec, Location, and Parking , weather) + situation inference (goodForWalking)	<pre> prefix mv:http://schema.mobivoc.org , prefix schema:http://schema.org select (targetCarpark.*,situWeather(targetWeather).goodForWalking) define entity targetLocation is from schema:Place where targetLocation.address="55 Pelham St, Carlton VIC 3053, Australia",entity car is from schema:car where car.manufacturer="kia" and car.model="rio" and car.vehicleModelDate=2015,entity targetWeather is from schema:weather where targetWeather.location.latitude = targetLocation.latitude and targetWeather.location.longitude = targetLocation.longitude , entity targetCarpark is from mv:ParkingGarage where targetCarpark.vehicleLengthLimitInM &gt; car.length and (distance(targetCarpark.geo,targetLocation.latitude,targetLocation.longitude,"W ALKING") &lt; 100 or (distance(targetCarpark.geo,targetLocation.latitude,targetLocation.longitude, "WALKING") &lt; 3000 ) and situWeather(targetWeather).goodForWalking &gt; 70) </pre>
-----------	---	--

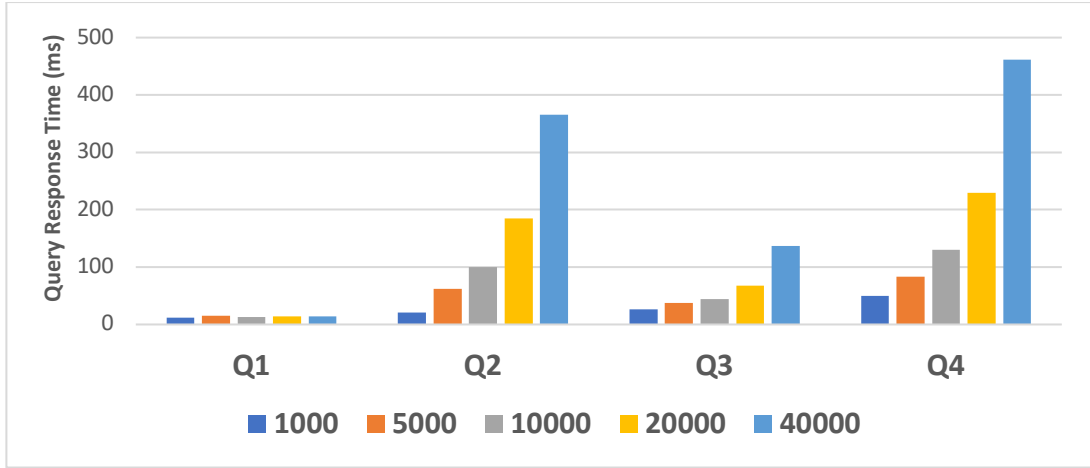
Results of the experiment are presented in Figure 5.6. The result shows the processing time of a query grows linearly with the increase in query load. The increase in query complexity increases the steepness of the graph. However, it can be seen that even for the most complex query (Q4) while the incoming query load was 550 query/sec, the response time is close to one second. This response time is within the acceptable range for most of IoT applications.



**Figure 5.6 - Query response time vs input rate**

Next, we designed another experiment to study the impact of the number of registered entities (i.e. parking spaces) on query execution time. In this experiment, we varied the number of registered parking spaces from 1000 to 40,000. The query load was equal to 100 query/sec. Similar to the previous experiment, we ran this experiment four times according to the queries above. Results of the experiment are presented in Figure 5.7.





**Figure 5.7** - Query response time vs number of registered entities

As shown in the bar chart, the response time of Q1 remained unchanged while the query load increased. The reason is this query searches for only one unique indexed attribute. However, in the case of other queries, we observed a linear growth of processing time. These queries are geolocation-based, and the number of entities can directly affect the search space. Interestingly, we observed Q3, which had more attributes than Q2, and had a lower response time. The reason for this effect is short-circuiting. Short-circuiting means the second argument of a logical expression is evaluated only if the result of the first argument is insufficient to determine the value of the expression.

### 5.3.3 EXPERIMENT 2: SITUATION MONITORING ENGINE - PUSH-BASED QUERIES

In this experiment, we focus on the evaluation of the push-based queries by conducting two sub-experiments to show how the CoaaS platform, in particular the proposed Situation Monitoring Engine (SME), deals with the increase in the number of context updates and the number of subscriptions. In both experiments, we used the preconditioning push-based query, which was presented in Code block 5.3.

The first experiment shows the impact of the number of incoming context updates on the execution time of push-based queries. To reveal the results, we used the following metrics: *input rate*, *throughput*, *processing time*, *CPU usage* and *memory consumption*.

The *input rate* denotes the number of incoming context updates per second.

$$\text{Input Rate} = \frac{\text{Number of incoming Context Updates}}{T_{end} - T_{start}}$$

The *throughput* depicts the number of context updates which were fully processed by the platform.

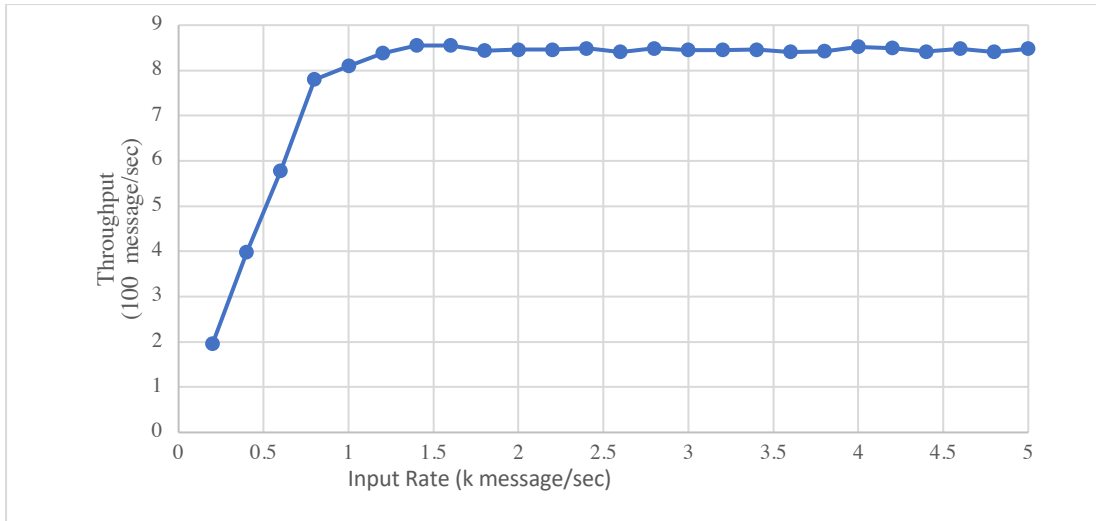
$$\text{Throughput} = \frac{\text{Number of Processed Context Updates}}{T_{end} - T_{start}}$$

*Processing time* represents the time, which is needed to process a context update from the time it reached the situation framework ( $T_{in}$ ) until the moment it gets fully processed ( $T_{out}$ ).

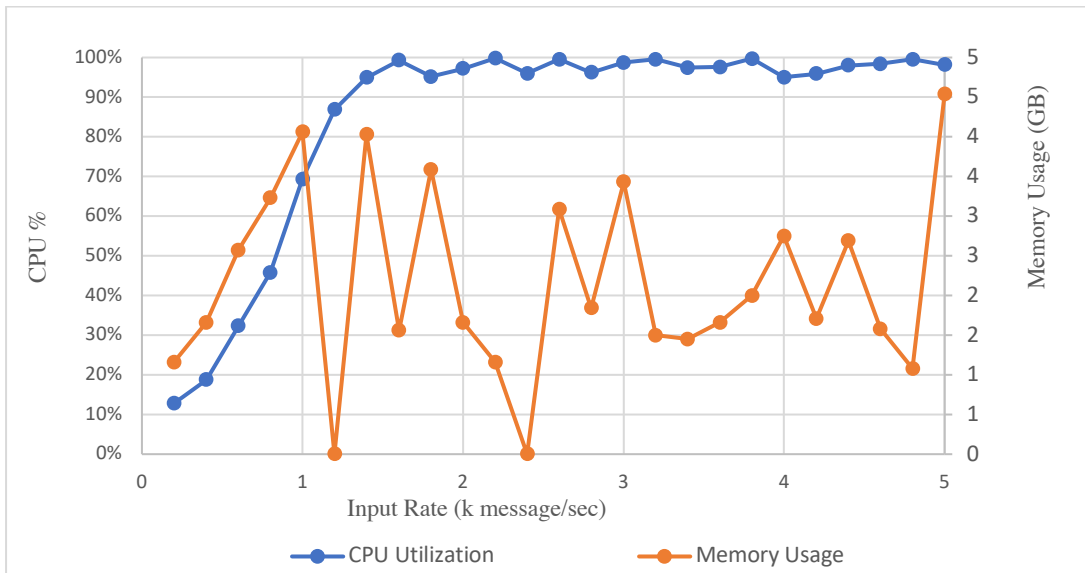
$$T_{\text{Processing}} = T_{out} - T_{in}$$

During the experiment, we were increasing the input rate from 200 updates per second to 5000 updates per second, while keeping the number of subscriptions equal to 10. The result of this experiment is depicted in figures 5.8–5.12.

Figure 5.8 shows the impact of increasing the input rate on throughput. As the graph shows, while the input rate increases from 200 to 1000, the throughput grows linearly with almost direct ratio from 196 updates/sec to 878 updates/s. From that moment until the end of the experiment, the throughput remains on the same level as the CPU utilisation (Figure 5.9) reaches its maximum. During this period, as the input rate becomes higher than the throughput, the messages are queued.

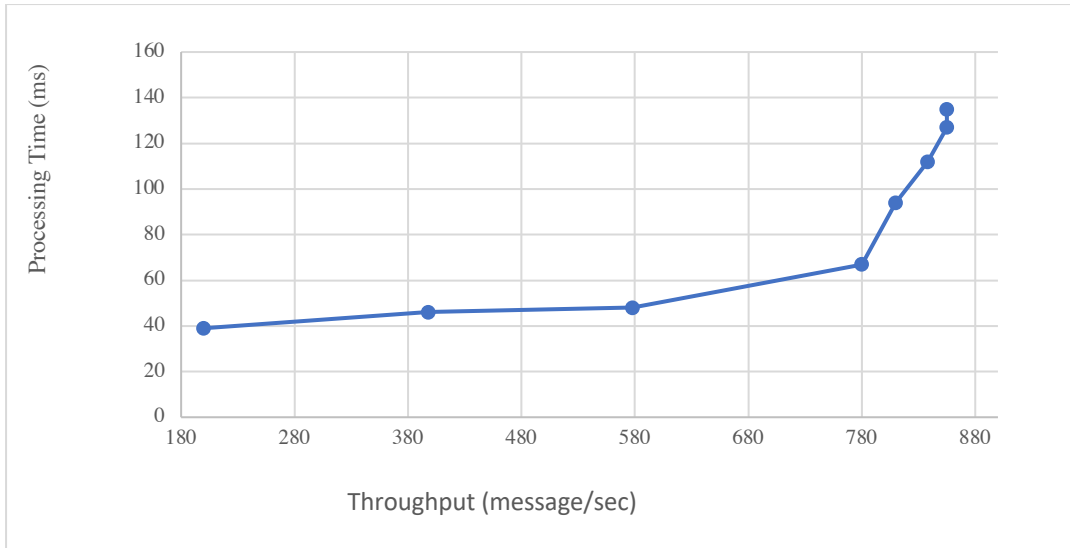


**Figure 5.8 - Throughput vs Input rate**



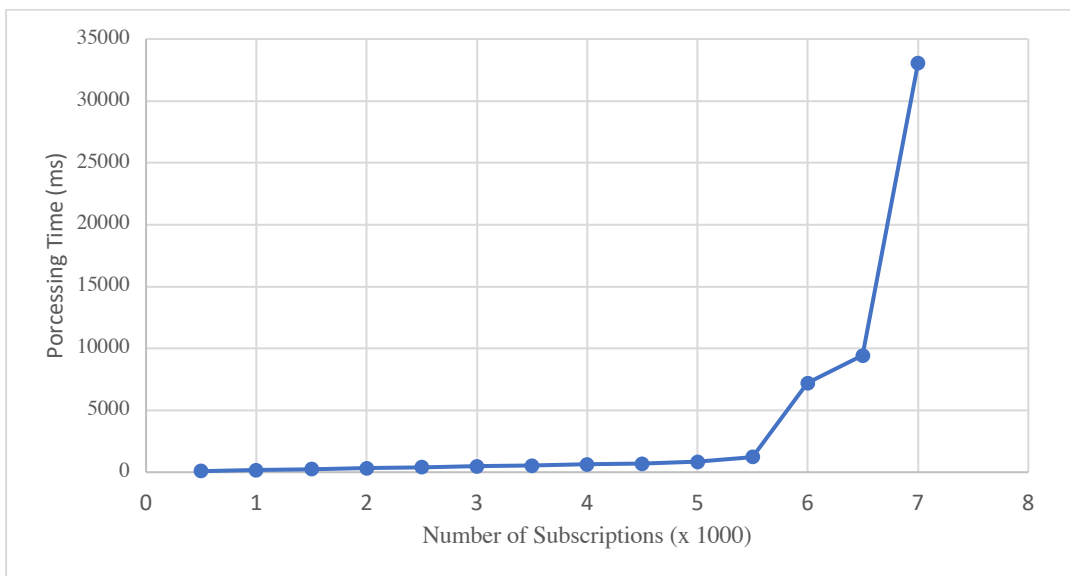
**Figure 5.9 - Resource utilization**

To demonstrate the effect of throughput on the processing time of an update, we plotted Figure 5.10. This graph shows a gradual linear growth of the processing time from 40ms to slightly more than 67ms until the throughput reaches 780 updates per second, which is the CPU saturation point. Then, updates start queuing and the processing time dramatically increases.

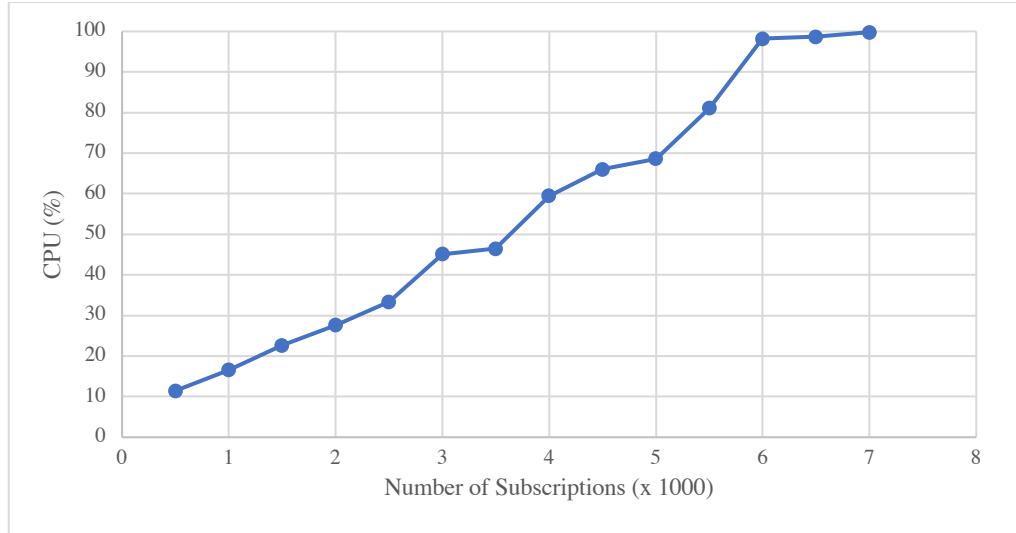


**Figure 5.10 - Processing time vs throughput**

The second experiment analyses how the number of subscriptions affects the context update processing time. We varied the number of subscriptions from 500 to 7000 while the input rate was equal to 100 updates per second. The result of this experiment is presented in Figure 5.11 and Figure 5.12. As it can be seen, the processing time increases gradually from 84ms to 1239ms, while the number of subscriptions increases from 500 to 5500. After that point, as the CPU utilisation reaches its maximum, the processing time increases dramatically.



**Figure 5.11 - Processing time vs number of subscriptions**



**Figure 5.12** - CPU utilisation vs number of subscriptions

In both sets of experiments, we demonstrated how the CoaaS platform could handle increasing load with high performance. We observed a drop in performance when the incoming load became too high. The result of our analysis shows and demonstrates that the drop in performance was caused by the resource limitation as we conducted our study with one server instance. However, all the components used in the system design were stateless and could be easily scaled out to several instances to provide near real-time performance for IoT scale applications.

### 5.3.4 EVALUATION OF CONTEXT SERVICE DISCOVERY ALGORITHM

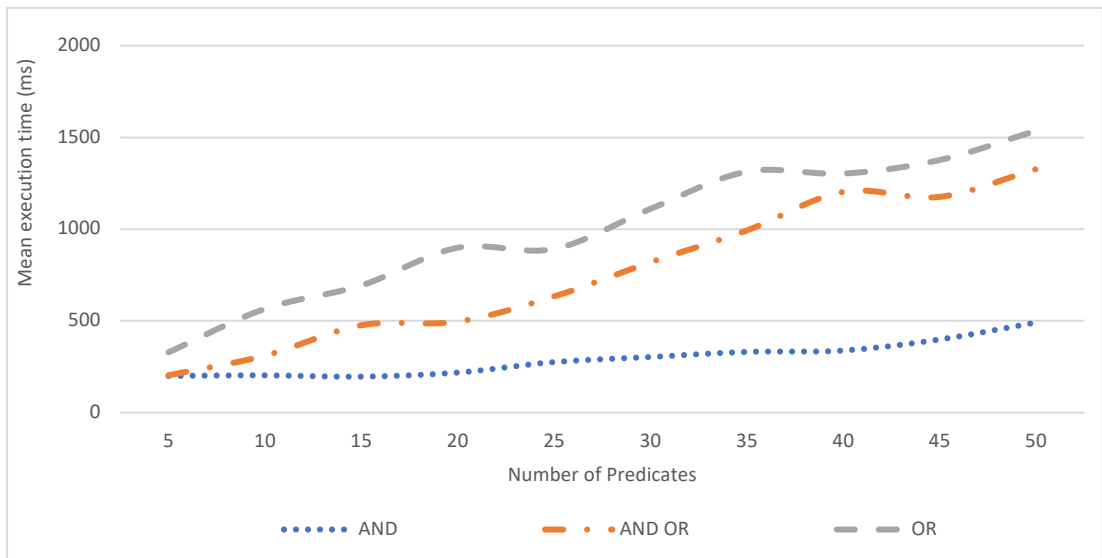
In order to evaluate the proposed Context Service Discovery (CSD) approach presented in Section 4.4, we have considered two aspects of performance and correctness. The performance metric that we consider here is how fast the CSD can respond to incoming context requests. On the other hand, the correctness metric measures the accuracy and the validity of car park recommendations.

To evaluate the performance of the proposed approach, 1000 context services were generated randomly. As we stated, each context service consists of three main components, namely Entity (E), Context Attributes (CA), and Predicates (P). In this regard, we extracted 50 different entities from schema.org semantic vocabulary and allocated them to context services randomly to generate the entity type and context attributes of each service. Further, in order to generate the predicates, 10 attributes from each entity were randomly selected. For each attribute, based on its type, we generated

a predicate (i.e. equality or inequality) and later connected these predicates by logic operators (i.e. ‘and’ / ‘or’), where the ratio of conjunctions to disjunctions equals 1:3.

For the evaluation, we distinguished three different scenarios: best, average and worst-case scenarios. Every test was performed with 5 to 50 predicates with a step size of 5. Furthermore, every test was repeated 20 times to provide a certain statistical persistence. The three different scenarios differ in the way the predicates are composed. According to our context service contextual characteristics matchmaking algorithm, a best-case scenario happens when only ‘AND’ is used to connect predicates in the logical expression. In contrast, the worst-case scenario occurs when only ‘OR’ is used to connect predicates. To create an average case scenario, a trade-off between best and worst-case scenario has to be found. An average scenario can be simulated by choosing a combination of ‘AND’ and ‘OR’ to connect the predicates. The chosen trade-off between ‘AND’ and ‘OR’ is a 1:3 ratio.

The chart presented in Figure 5.13 shows the result of the conducted experiments. As it was expected, increasing the number of predicates leads to higher execution time in all scenarios. However, with a reasonable number of predicates (25) the execution time of a query is kept under 1 second.

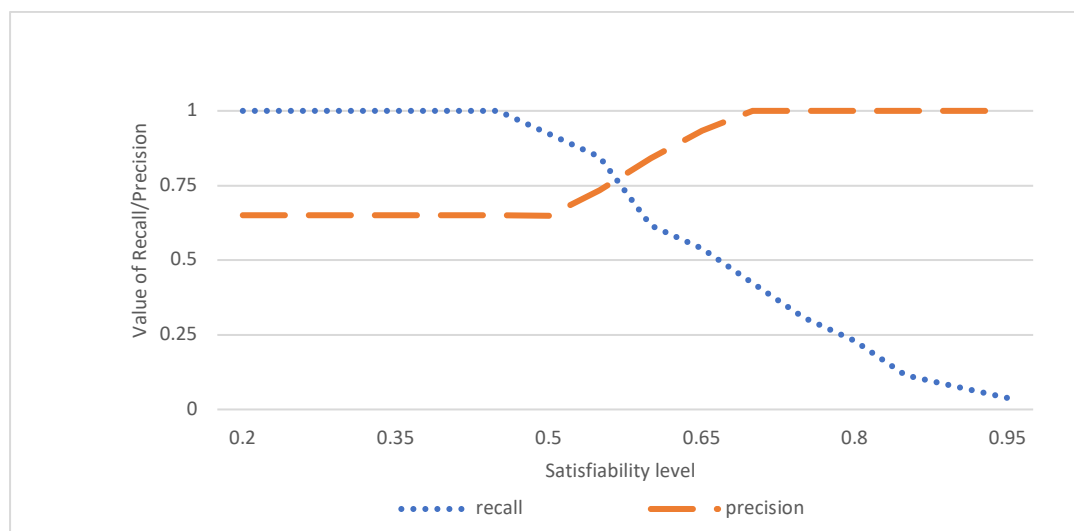


**Figure 5.13 - Performance of CSD**

In the case of correctness evaluation, we use standard metrics applied to information retrieval systems. More precisely, we used *precision/recall* metrics, where *precision* indicates how accurate the retrieved context services match a given context request, and *recall* is the proportion of a correctly matched context services to all the available services that can actually serve a given context request. In this experiment, we generated a test set of different context service descriptions and context requests. Further, to evaluate the precision recall of the matchmaking technique we generated a graded relevance set, which uses a 3-graded scale: full match, partial match, and no match.

As explained in Section 4.5, the output of the service discovery algorithm is a floating number that indicates how well a service satisfies a given context request. Finally, we executed the queries with various satisfiability level thresholds and compared the results with graded relevance set to calculate the precision and recall.

Figure 5.14 represents the dependence of the recall and the precision on the satisfiability level. As illustrated, increasing the threshold of the satisfiability level leads to a decrease of the recall level and to an increase of the precision of matchmaking. We set the default threshold to 0.7 level; however, this level can be overridden as part of a context query.



**Figure 5.14 - Precision vs Recall of CSD**

## 5.4 SUMMARY

In this chapter, in order to demonstrate the fulfilment of the different requirements and the feasibility of the proposed context publishing and querying approach, we have conducted a comprehensive evaluation, which consists of three parts.

In the first part of the evaluation, we validated the main functionalities of the proposed approach based on real-world scenarios. More precisely, we have integrated several context services (based on existing works) in the CoaaS platform and then designed and executed various context queries to demonstrate the capability of the proposed solution to represent a wide range of complex context queries, such as pull-based queries, push-based queries, and queries concerning various context entities and constraints.

The demonstration clearly shows how the proposed solution can be utilised to ease the development of a wide range of context-aware IoT applications. Moreover, it shows that CDQL can satisfy all the six requirements we have identified for a context query language in Chapter 2.

Moreover, to demonstrate the advantages of the proposed context query language compared to existing CQLs in the literature, we conducted a comparative evaluation. In this evaluation, we compared CDQL with NGSI, which is the most sophisticated existing context query language in existence.

Besides highlighting the practical usability of the proposed concepts, we also have conducted a set of experiments to illustrate the performance of the CoaaS platform, in particular Context Query Engine and Situation Monitoring Engine, by executing the context queries under different circumstances. We have designed a set of context queries as benchmarks and executed them by varying the experimental setting (e.g. number of context providers, and number of concurrent queries) and compare the results based on different evaluation metrics (e.g. execution time, CPU usage, and memory usage). These experiments showed the proposed solution can be utilised for large-scale IoT applications with decent performance.

Finally, we performed a systematic evaluation of the proposed Context Service Discovery (CSD) approach to prove its correctness and performance.



## Chapter 6: Conclusion

---

This chapter concludes the dissertation by summarising the research contributions and providing a discussion on future work.

### 6.1 SUMMARY OF CONTRIBUTIONS

The Internet of Things (IoT) envisions an ecosystem in which everyday objects (e.g., refrigerator, air conditioner, smartphones, and cars) are enhanced with sensing, computation, and communication capabilities. These ‘smart’ devices (i.e. IoT devices) can sense and collect an enormous amount of data about their surroundings, which is known as context, and share it with each other via the Internet. Utilising the context data produced by IoT devices, it is possible to enhance a wide range of applications in a way that they adapt their behaviour according to the context of their related entities, including themselves. Such applications are known as context-aware applications.

While context-driven intelligence is a fundamental factor for IoT sustainability, growth, interoperability and acceptance, IoT’s characteristics, such as scalability, big data, heterogeneity and dynamism, will make the development of context-aware applications and services a very challenging task. To address these challenges and facilitates the development of context-aware IoT applications, in this dissertation, we proposed and evaluated a comprehensive solution for publishing, querying, monitoring, and sharing context.

The proposed solution consists of four main components, which are (i) Context Service Description Language (CSDL), (ii) Context Definition and Query Language (CDQL), (iii) Context Query Engine (CQE), and (iv) Situation Monitoring Engine (SME). The first two components, namely CSDL and CDQL, are two specially designed high-level languages that provide a novel mechanism for publishing and querying context. The other two components, which are CQE and SME, are two novel mechanisms that enable the execution of complex context queries in IoT environment and continuous monitoring of changes in context of IoT entities respectively.

Context Service Description Language (CSDL) is a JSON-LD-based language. This language enables developers of context services to describe their services in terms of semantic signature and contextual behavioural specification; where the semantic signature defines the service name, number and types of its parameters, and the type of its output, and the contextual behavioural presents the context of IoT entities provided by the service. CSDL enables IoT applications to discover and consume context services owned and operated by different individuals and organisations.

Context Definition and Query Language (CDQL) provides a generic and flexible approach to defining, representing, inferring, monitoring, and querying context. CDQL consists of two main parts, namely: Context Definition Language (CDL), which is designed to describe situations and high-level context; and Context Query Language (CQL), which is a powerful and flexible query language, to express contextual information requirements without considering the details of the underlying data structures. An important feature of the proposed query language is its ability to query entities in IoT environments based on their situation in a fully dynamic manner where users can define situations and context entities as part of the query.

Context Query Engine (CQE) is mainly responsible for parsing incoming context queries, generating and orchestrating the query execution plan, and producing the final context query result. Furthermore, CQE is also in charge of finding the most appropriate context services for incoming requests. To achieve this goal, we have designed a Context Service Discovery (CSD) approach. CSD's workflow consists of two parts. First, it finds context services that match the requirements of a context request. Then, based on the discovered services, it returns a sorted set of the best available context services that can satisfy the requirements of a request considering different metrics such as Cost of Service and Quality of Service.

The Situation Monitoring Engine (SME) is designed to support continuous monitoring of incoming context, inferring situations from available context, detecting changes in such inferred situations and providing notification of detected changes. This component monitors in real-time contextual changes of IoT entities. It also initiates the actuation procedure by notifying context consumers when their situation of interest is detected.

The proposed solution in this dissertation has been integrated in a context management platform called Context-as-a-Service (CoaaS), which is part of EU Horizon-2020 project (“Software - bIoTpe Project,” n.d.). CoaaS enables IoT devices to provide context and IoT application to consume context seamlessly. We present a blueprint architecture of CoaaS platform and demonstrate its functionalities through a proof of concept implementation. The blueprint architecture follows a scalable and fault-tolerant design.

To demonstrate the feasibility of the proposed approaches in developing context-aware IoT applications that can seamless share context, we developed 3 application demonstrators based on real-world scenarios. These include school safety scenario, context-aware parking suggestions application, and a connected car pre-conditioning application. These application demonstrators validate the proposed approaches’ ability to represent a wide range of complex context queries, execute pull-based and push-based queries (supporting various needs to IoT applications), query high-level context and situation, and complex queries that include several context entities and constraints.

To demonstrate the advantages of the proposed context query language compared to the existing CQLs, we have conducted a qualitative evaluation. In this evaluation, we compared CDQL with NGSI. We demonstrated that in order to implement the use case of smart parking five NGSI queries plus some additional coding on the application side were required. Whereas, the same scenario was implemented using a single CDQL query. Moreover, we demonstrated that NGSI is not capable of implementing use cases that need to monitor the change in the context of multiple IoT entities.

To assess the performance and scalability of the proposed solution in execution of context queries under different scenarios, we have conducted three sets of experiments. In the first set of experiments, we have evaluated the performance of Context Query Engine (CQE) in the execution of pull-based CDQL queries. In this regard, we have studied the impact of the query load on query execution performance by increasing the load from 40 to 550 query per second. The result showed the average query execution time grows linearly, from 91ms to 842ms, with the increase in the query load.

In the second set of experiments, we have evaluated the execution of push-based queries. This set of experiments demonstrated how the CoaaS platform, in particular the proposed Situation Monitoring Engine (SME), deals with the increase in the number of context updates and the number of subscriptions. The outcome of this experiment showed a single instance of CoaaS platform can handle 780 context updates per second without any degradation in the performance of the system where the average processing time for each context update is around 67ms. We have also varied the number of subscriptions from 500 to 7000, while the input rate was equal to 100 updates per second. The result of this experiment showed that the processing time increases gradually (i.e. from 84ms to 1239ms) with the increase in the number of subscription (i.e. 500 to 5500) until the point CPU utilisation reaches its maximum.

In the third set of experiments, we have evaluated the proposed Context Service Discovery (CSD) approach. This set of experiments showed that for 1000 registered context services, with a reasonable number of predicates (25), the execution time of the CSD is under 1 second. These experiments demonstrate and validate the ability of the proposed solution to be utilised for large-scale IoT application development.

## 6.2 LIMITATIONS AND FURTHER RESEARCH

Despite the significant contributions of this work towards operationalising context-awareness in IoT ecosystem especially with the focus on context diffusion and distribution, there are still several open issues in this domain that require further investigation. Some of the most interesting problems that deserve attention in further studies are listed below.

- **Advanced context storage and management system:** In order to achieve a higher throughput of the system, it is essential to have an advanced context storage system that can store and retrieve context data efficiently. An important aspect related to context storage management system that needs to be addressed is developing a proactive caching mechanism that is able to cache contextual data dynamically.
- **Automatic Annotation of Context Provider/Service:** An essential aspect of IoT is utilising the data produced by IoT devices. In this regard a CMP platform should be capable of understanding and contextualising such data in order to use it effectively. However, the metadata required to make sense of this data is mostly

inaccurate, incomplete, and in many situations, only human interpretable. To address this challenge, it is required to investigate and design a mechanism that can classify, annotate and semantically enrich IoT data-streams/services.

- **Privacy, Security, and Access control:** In this dissertation, we have briefly discussed the topics of security, privacy, and access control. However, considering the importance of these topics, a CMP should be enhanced with an advanced authentication and authorisation mechanism that ensures the privacy and the security of the users' data. Moreover, there is a need for an access control mechanism that is capable of sharing context data only to selected context consumers.
- **Context Prediction:** Another important aspect of context processing that has not been discussed in this dissertation is context prediction. Context predication is referred to the process of exploiting expected future context of IoT entities based on the historical context. A CMP that supports context predication offers distinct advantages to the context consumers, which enables a range of new use-cases. Hence, it is important to investigate, design, and implement a generic mechanism that allows context consumers predict the future context of IoT entities.
- **Auto-Scaling Strategy:** In this dissertation, we have conducted all the experiments on a single instance of CoaaS platform. However, in production environments, context management platforms are needed to be scaled-out to deal with the massive number of requests generated by the billions of IoT devices. To address this challenge, it is required to investigate and design an auto-scaling strategy for CMPs that automatic scale-out or scale-in based on the scale of incoming requests.

## Glossary

**Context** is any information that can be used to characterise the state of an entity. Entities are persons, locations, or objects that affects the behaviour of an application.

**Context-Awareness** refers to a system that adapts its behaviour to the context of itself, its users, or its surrounding environment.

**Context as a Service** is a context management framework which is responsible for providing a comprehensive method to allow a smart entity, namely context service consumers, to consume a context service provided by another entity that is a context service provider.

**Context Entity/Attribute** In context-aware systems, an entity (denoted by  $E_k$ ) accounts for a physical or virtual object (such as a person, car, electronic device, event) that can be associated with one or more context attributes (denoted by  $ca_i$ ), which can be any type of data that characterises this entity.

**Context Provider** Context Provider (CP) refers to any device or system that can provide context or data that can be used to infer context about one or several entities.

**Context Consumer** (CC) refers to any device or system that query and receive context about one or several entities.

**Context Service** provides contextual information about a particular entity. Context service can be represented as a triple:  $\langle E, CAs, Ps \rangle$  where  $E$  denotes the related entity,  $CAs$  is a set of provided context attributes, and Predicates (denoted by  $Ps$ ) form a set of logical expressions defined over  $CAs$ .

**Context Query** is a request for contextual information (either context attributes or high-level context inferred from context attributes) extracted from one or many entities.

**Context Request** represents a request for contextual information about a particular entity. Context request can be represented as a triple:  $\langle E, CAs, Ps \rangle$  where  $E$  denotes the entity of interest,  $CAs$  is a set of requested context attributes, and  $Ps$  is a set of predicates, which are defined over  $CAs$  using logical expressions.

## References

- Abowd, G. D., Dey, A. K., Brown, P. J., Davies, N., Smith, M., & Steggles, P. (1999). Towards a better understanding of context and context-awareness. In *Proceedings of the First International Symposium on Handheld and Ubiquitous Computing (HUC '99)* (pp. 304–307). [https://doi.org/10.1007/3-540-48157-5\\_29](https://doi.org/10.1007/3-540-48157-5_29)
- Abowd, G. D., & Mynatt, E. D. (2000). Charting past, present, and future research in ubiquitous computing. *ACM Trans. Comput.-Hum. Interact.*, 7(1), 29–58. <https://doi.org/10.1145/344949.344988>
- Allen, J. F. (2013). Maintaining Knowledge about Temporal Intervals. In *Readings in Qualitative Reasoning About Physical Systems*. <https://doi.org/10.1016/B978-1-4832-1447-4.50033-X>
- Atzori, L., Iera, A., & Morabito, G. (2010). The internet of things: A survey. *Computer Networks*, 54(15), 2787–2805.
- Baldauf, M., Dustdar, S., & Rosenberg, F. (2007). A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2(4), 263. <https://doi.org/10.1504/IJAHUC.2007.014070>
- Bassi, A., & Horn, G. (2008). Internet of Things in 2020: A Roadmap for the Future. *European Commission: Information Society and Media*, 22, 97–114.
- Bauer, M., Becker, C., & Rothermel, K. (2002). Location Models from the Perspective of Context-Aware Applications and Mobile Ad Hoc Networks. *Personal and Ubiquitous Computing*, 6(5), 322–328. <https://doi.org/10.1007/s007790200036>
- Becker, C., & Nicklas, D. (2004). Where do spatial context-models end and where do ontologies start? A proposal of a combined approach. *Proceedings of the First International Workshop on Advanced Context Modelling, Reasoning and Management in Conjunction with UbiComp 2004*, 48–53. Retrieved from [http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL\\_view.pl?id=INPROC-2004-38&engl=](http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2004-38&engl=)

- Bettini, C., Brdiczka, O., Henriksen, K., Indulska, J., Nicklas, D., Ranganathan, A., & Riboni, D. (2010). A survey of context modelling and reasoning techniques. *Pervasive and Mobile Computing*, 6(2), 161–180.  
<https://doi.org/10.1016/j.pmcj.2009.06.002>
- bioTope Project. (n.d.). Retrieved May 27, 2019, from <https://biotope-project.eu/>
- Boytssov, A., & Zaslavsky, A. (2011). ECSTRA - Distributed context reasoning framework for pervasive computing systems. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. [https://doi.org/10.1007/978-3-642-22875-9\\_1](https://doi.org/10.1007/978-3-642-22875-9_1)
- Brickley, D., & Guha, R. V. (2004). RDF Vocabulary Description Language 1.0: RDF Schema. *W3C*, (February), 1–15. <https://doi.org/10.1002/9780470773581>
- Brock, D. L. (2001). *The Electronic Product Code (EPC). A Naming Scheme for Physical Objects (white paper)*. Auto-ID Center.
- Brown, P. J., Bovey, J. D., & Chen, X. (1997). Context-aware applications: From the laboratory to the marketplace. *IEEE Personal Communications*, 4(5), 58–64.  
<https://doi.org/10.1109/98.626984>
- Brümmer, M., & Weilandt, A. (2018). MobiVoc: Open Mobility Vocabulary. Retrieved April 8, 2019, from <http://schema.mobivoc.org/>
- Buchholz, T., Küpper, A., & Schiffers, M. (2003). Quality of Context: What It Is And Why We Need It. *Proceedings of the Workshop of the HP OpenView University Association*, 1–14. <https://doi.org/10.1.1.147.565>
- Chen, G., & Kotz, D. (2000). *A Survey of Context-Aware Mobile Computing Research. Technical Report TR2000-381* (Vol. 3755).  
<https://doi.org/10.1.1.140.3131>



- Chen, H., Finin, T., & Joshi, A. (2004). A context broker for building smart meeting rooms. *Proceedings of the AAAI Symposium on Knowledge Representation and Ontology for Autonomous Systems Symposium, 2004 AAAI Spring Symposium*, 53–60. Retrieved from <http://www.aaai.org/Papers/Symposia/Spring/2004/SS-04-04/SS04-04-008.pdf>
- Chen, H. L. (2004). COBRA : An Intelligent Broker Architecture for Pervasive Context-Aware Systems. *Interfaces*, 54(November), 129.  
<https://doi.org/10.1.1.4.4259>
- Chen, P., Sen, S., Pung, H. K., & Wong, W. C. (2014). A SQL-based Context Query Language for Context-aware Systems. In *IMMM 2014 : The Fourth International Conference on Advances in Information Mining and Management* (pp. 96–102).
- da Cruz, M. A. A., Rodrigues, J. J. P. C., Sangaiah, A. K., Al-Muhtadi, J., & Korotaev, V. (2018). Performance evaluation of IoT middleware. *Journal of Network and Computer Applications*. <https://doi.org/10.1016/j.jnca.2018.02.013>
- Davies, R. (2015). *Industry 4.0. Digitalisation for productivity and growth. European Parliamentary Research Service* (Vol. 10).
- Deborah L. McGuinness, F. van H. (2004). Owl web ontology language overview. *W3C Recommendation 10.2004-03, 2004*(February), 1–12.  
<https://doi.org/10.1145/1295289.1295290>
- Delir Haghighi, P., Krishnaswamy, S., Zaslavsky, A., & Gaber, M. M. (2008). Reasoning about context in uncertain pervasive computing environments. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5279 LNCS, 112–125. <https://doi.org/10.1007/978-3-540-88793-5-9>
- Dey, A. K. (2001). Understanding and using context. *Personal and Ubiquitous Computing*, 5(1), 4–7. <https://doi.org/10.1007/s007790170019>

- Dohr, A., Modre-Osprian, R., Drobits, M., Hayn, D., & Schreier, G. (2010). The internet of things for ambient assisted living. In *ITNG2010 - 7th International Conference on Information Technology: New Generations*.  
<https://doi.org/10.1109/ITNG.2010.104>
- Domingue, J., Roman, D., & Stollberg, M. (2005). Web Service Modeling Ontology (WSMO): an ontology for Semantic Web Services. *Architecture*, 776–784.  
 Retrieved from  
[http://www.w3.org/2005/04/FSWS/Submissions/1/wsmo\\_position\\_paper.html](http://www.w3.org/2005/04/FSWS/Submissions/1/wsmo_position_paper.html)
- Esper. (n.d.).
- Espressif Systems - Wi-Fi and Bluetooth chipsets and solutions. (n.d.). Retrieved March 7, 2019, from <https://www.espressif.com/>
- ETSI - ETSI ISG CIM group releases first specification for context exchange in smart cities. (n.d.). Retrieved February 18, 2019, from  
<https://www.etsi.org/newsroom/news/1300-2018-04-news-etsi-isg-cim-group-releases-first-specification-for-context-exchange-in-smart-cities>
- Feng, L. (2010). Supporting context-aware database querying in an Ambient Intelligent environment. *2010 3rd IEEE International Conference on Ubi-Media Computing, U-Media 2010*, 161–166.  
<https://doi.org/10.1109/UMEDIA.2010.5544479>
- FIWARE. (n.d.).
- Floreen, P., Przybilski, M., Nurmi, P., Koolwaaij, J., Tarlano, a, Wagner, M., ... Lau, S. (2005). Towards a Context Management Framework for MobiLife. *Management*, 120–131. Retrieved from  
<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.123.1335&rep=rep1&type=pdf>
- Franklin, D., & Flachsbart, J. (1998). All Gadget and No Representation Makes Jack a Dull Environment Sensing. In *Proceedings of AAAI 1998 Spring Symposium on Intelligent Environments (SprSym '98)* (pp. 155–160).

- Fujii, K., & Suda, T. (2009). Semantics-based context-aware dynamic service composition. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2), 1–31. <https://doi.org/10.1145/1516533.1516536>
- Grossmann, M., Bauer, M., Hönle, N., Käppeler, U. P., Nicklas, D., & Schwarz, T. (2005). Efficiently managing context information for large-scale scenarios. In *Proceedings - Third IEEE International Conference on Pervasive Computing and Communications, PerCom 2005* (Vol. 2005, pp. 331–340). <https://doi.org/10.1109/PERCOM.2005.17>
- Gu, T., Pung, H. K., & Zhang, D. Q. (2005). A service-oriented middleware for building context-aware services. *Journal of Network and Computer Applications*, 28(1), 1–18. <https://doi.org/10.1016/j.jnca.2004.06.002>
- Guinard, D., Trifa, V., Karnouskos, S., Spiess, P., & Savio, D. (2010). Interacting with the SOA-based internet of things: Discovery, query, selection, and on-demand provisioning of web services. *IEEE Transactions on Services Computing*, 3(3), 223–235. <https://doi.org/10.1109/TSC.2010.3>
- Haghighi, P. D. D., Zaslavsky, a., & Krishnaswamy, S. (2006). An Evaluation of Query Languages for Context-Aware Computing. *Database and Expert Systems Applications, 2006. DEXA '06. 17th International Workshop On*, 455–462. <https://doi.org/10.1109/DEXA.2006.25>
- Haghighi, P. D., Zaslavsky, A., & Krishnaswamy, S. (2006). An Evaluation of Query Languages for Context-Aware Computing. *Database and Expert Systems Applications, 2006. DEXA '06. 17th International Workshop On*, 455–462. <https://doi.org/10.1109/DEXA.2006.25>
- Hassani, A., Haghighi, P. D., Jayaraman, P. P., Zaslavsky, A., & Ling, S. (2018). Querying IoT Services: A Smart Carpark Recommender Use Case. In *4th IEEE World Forum on Internet of Things WF-IoT 2018*.

- Henricksen, K., & Indulska, J. (2004). A software engineering framework for context-aware pervasive computing. *2nd IEEE International Conference on Pervasive Computing and Communications (PerCom 2004)*, 77–86.  
<https://doi.org/10.1109/PERCOM.2004.1276847>
- Henricksen, K., Indulska, J., & Rakotonirainy, A. (2002). Modeling Context Information in Pervasive Computing Systems. *Pervasive*, 2414, 167–180.
- Hofer, T., Schwinger, W., Pichler, M., Leonhartsberger, G., Altmann, J., Hagenberg, A.-, ... Kepler, J. (2002). Context-Awareness on Mobile Devices - the Hydrogen Approach, 43(7236).
- Hong, J. yi, Suh, E. ho, & Kim, S. J. (2009). Context-aware systems: A literature review and classification. *Expert Systems with Applications*, 36(4), 8509–8522.  
<https://doi.org/10.1016/j.eswa.2008.10.071>
- Hönle, N., Käppeler, U.-P., Nicklas, D., Schwarz, T., & Grossmann, M. (2005). Benefits of Integrating Meta Data into a Context Model. *Third IEEE International Conference on Pervasive Computing and Communications Workshops*, 25–29. <https://doi.org/10.1109/PERCOMW.2005.20>
- Hossain, M. A., Parra, J., Atrey, P. K., & El Saddik, A. (2009). A framework for human-centered provisioning of ambient media services. *Multimedia Tools and Applications*, 44(3), 407–431. <https://doi.org/10.1007/s11042-009-0285-9>
- Huebscher, M. C., & McCann, J. A. (2004). Adaptive middleware for context-aware applications in smart-homes. *Proceedings of the 2nd Workshop on Middleware for Pervasive and Ad-Hoc Computing -*, 111–116.  
<https://doi.org/10.1145/1028509.1028511>
- Hull, R., Neaves, P., & Bedford-Roberts, J. (1997). Towards situated computing. *Digest of Papers. First International Symposium on Wearable Computers*, 146–153. <https://doi.org/10.1109/ISWC.1997.629931>
- IEEE. (1990). *IEEE Standard Glossary of Software Engineering Terminology. report IEEE Std 610.12- 1990*. <https://doi.org/10.1109/IEEESTD.1990.101064>

- Ikram, A., Baker, N., Knappmeyer, M., Reetz, E. S., & Tonjes, R. (2011). An artificial chemistry based framework for personal and social context aware smart spaces. *2011 7th International Wireless Communications and Mobile Computing Conference*, 2009–2014. <https://doi.org/10.1109/IWCMC.2011.5982843>
- Jones, M., Bradley, J., & Sakimura, N. (2015). Rfc 7519: Json web token (jwt). *Date of Retrieval*, 5, 2017.
- Klyne, G., Reynolds, F., Woodrow, C., Ohto, H., Hjelm, J., Butler, M. H., & Tran, L. (2004). Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies 1.0. *W3C Recommendation*, (April), 1–56. <https://doi.org/http://www.w3.org/TR/CCPP-struct-vocab/>
- Knappmeyer, M., Kiani, S. L., Frà, C., Moltchanov, B., & Baker, N. (2010). ContextML: A light-weight context representation and context management schema. In *ISWPC 2010 - IEEE 5th International Symposium on Wireless Pervasive Computing 2010* (pp. 367–372). <https://doi.org/10.1109/ISWPC.2010.5483753>
- Knappmeyer, M., Kiani, S. L., Reetz, E. S., Baker, N., & Tonjes, R. (2013). Survey of context provisioning middleware. *IEEE Communications Surveys and Tutorials*, 15(3), 1492–1519. <https://doi.org/10.1109/SURV.2013.010413.00207>
- Kofod-petersen, A., & Mikalsen, M. (2005). Context : Representation and Reasoning Environment. *Communication*, 19(3), 479–498. <https://doi.org/10.3166/ria.19.479-498>
- Kopecký, J., Vitvar, T., Bournez, C., & Farrell, J. (2007). SAWSDL: Semantic annotations for WSDL and XML schema. *IEEE Internet Computing*, 11(6), 60–67. <https://doi.org/10.1109/MIC.2007.134>
- Korpipää, P., & Mäntyjärvi, J. (2003). An ontology for mobile device sensor-based context awareness. *Modeling and Using Context*, 451–458. [https://doi.org/10.1007/3-540-44958-2\\_37](https://doi.org/10.1007/3-540-44958-2_37)

- Krause, M., & Hochstatter, I. (2005). Challenges in modelling and using quality of context (QoC). *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3744 LNCS, 324–333. [https://doi.org/10.1007/11569510\\_31](https://doi.org/10.1007/11569510_31)
- Le-Phuoc, D., Polleres, A., Hauswirth, M., Tummarello, G., & Morbidoni, C. (2009). Rapid prototyping of semantic mash-ups through semantic web pipes. <https://doi.org/10.1145/1526709.1526788>
- Li, X., Eckert, M., Martinez, J. F., & Rubio, G. (2015). Context aware middleware architectures: Survey and challenges. *Sensors (Switzerland)*, 15(8), 20570–20607. <https://doi.org/10.3390/s150820570>
- M. Razzaque, S. Dobson, & P. Nixon. (2005). Categorisation and modelling of quality in context information. *Workshop on AI and Autonomic Communications*.
- Manzoor, A., Truong, H. L., & Dustdar, S. (2008). On the evaluation of quality of context. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (Vol. 5279 LNCS, pp. 140–153). <https://doi.org/10.1007/978-3-540-88793-5-11>
- Mavrommatis, A., Artikis, A., Skarlatidis, A., & Paliouras, G. (2016). A distributed event calculus for event recognition. In *CEUR Workshop Proceedings*.
- Mccarthy, J. (1986). NOTES ON FORMALIZING CONTEXT \*. In *5th National Conference on AI*.
- McFadden, T., Henricksen, K., & Indulska, J. (2004). Automating Context-aware Application Development. *Procs. of the 1st Int'l Workshop on Advanced Context Modelling, Reasoning and Management (UbiComp)*, 90–95. <https://doi.org/10.1.1.1.9810>
- McIlraith, S. A., San, T. C., & Zeng, H. (2001). Semantic Web services. *IEEE Intelligent Systems*, 16, 46–53. <https://doi.org/10.1109/5254.920599>

- Medvedev, A., Hassani, A., Zaslavsky, A., Haghighi, P. D., Ling, S., Santiago, M. I., ... Kolbe, N. (2018). Situation Modelling , Representation , and Querying in Context-as-a-Service IoT Platform. In *GIoTS 2018 - Global Internet of Things Summit, Proceedings*.
- Medvedev, A., Hassani, A., Zaslavsky, A., Jayaraman, P. P., Indrawan-Santiago, M., Haghighi, P. D., & Ling, S. (2017). Data ingestion and storage performance of IoT platforms: Study of OpenIoT. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (Vol. 10218 LNCS, pp. 141–157). [https://doi.org/10.1007/978-3-319-56877-5\\_9](https://doi.org/10.1007/978-3-319-56877-5_9)
- Medvedev, A., Indrawan-Santiago, M., Delir Haghighi, P., Hassani, A., Zaslavsky, A., & Jayaraman, P. P. (2017). Architecting IoT context storage management for context-as-a-service platform. In *GIoTS 2017 - Global Internet of Things Summit, Proceedings*. <https://doi.org/10.1109/GIOTS.2017.8016228>
- Meulen, R. van der. (2017). Gartner says 8.4 billion connected “Things” will be in use in 2017 up 31 percent from 2016. *Gartner. Letzte Aktualisierung*, 7, 2017.
- Miorandi, D., Sicari, S., De Pellegrini, F., & Chlamtac, I. (2012). Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7), 1497–1516. <https://doi.org/10.1016/j.adhoc.2012.02.016>
- Ngan, L. D., Kir, M., & Kanagasabai, R. (2010). Review of Semantic Web Service Discovery Methods. *2010 6th World Congress on Services*, 176–177. <https://doi.org/10.1109/SERVICES.2010.85>
- Noura, M., Atiquzzaman, M., & Gaedke, M. (2019). Interoperability in Internet of Things: Taxonomies and Open Challenges. *Mobile Networks and Applications*, 24(3), 796–809. <https://doi.org/10.1007/s11036-018-1089-9>
- NSGIV2 API Walkthrough - Fiware-Orion. (n.d.).
- On-street parking data - City of Melbourne. (n.d.). Retrieved December 2, 2018, from <https://www.melbourne.vic.gov.au/about-council/governance-transparency/open-data/Pages/on-street-parking-data.aspx>

- Open Mobile Alliance. (2012). NGSI Context Management. Retrieved from [http://www.openmobilealliance.org/release/NGSI/V1\\_0-20120529-A/OMA-TS-NGSI\\_Context\\_Management-V1\\_0-20120529-A.pdf](http://www.openmobilealliance.org/release/NGSI/V1_0-20120529-A/OMA-TS-NGSI_Context_Management-V1_0-20120529-A.pdf)
- Padovitz, A., Loke, S. W., & Zaslavsky, A. (2004). Towards a theory of context spaces. In *Proceedings - Second IEEE Annual Conference on Pervasive Computing and Communications, Workshops, PerCom* (pp. 38–42). <https://doi.org/10.1109/PERCOMW.2004.1276902>
- Padovitz, A., Loke, S. W., Zaslavsky, A., Burg, B., & Bartolini, C. (2005). An approach to data fusion for context awareness. In *International and Interdisciplinary Conference on Modeling and Using Context* (pp. 353–367). <https://doi.org/10.1.1.60.6380>
- Patel, K., & Patel, S. M. (2016). Internet of Things-IOT: definition, characteristics, architecture, enabling technologies, application & future challenges. *International Journal of Engineering Science and Computing*. <https://doi.org/10.4010/2016.1482>
- Patrourmpas, K., & Sellis, T. (2006). Window specification over data streams. *Current Trends in Database Technology–EDBT ...*, 445–464. <https://doi.org/10.1007/b97859>
- Pereira, C., Cardoso, J., Aguiar, A., & Morla, R. (2018). Benchmarking Pub/Sub IoT middleware platforms for smart services. *Journal of Reliable Intelligent Environments*, 4(1), 25–37. <https://doi.org/10.1007/s40860-018-0056-3>
- Perera, C., Zaslavsky, A., Christen, P., & Georgakopoulos, D. (2012). CA4IOT: Context awareness for Internet of Things. In *Proceedings - 2012 IEEE Int. Conf. on Green Computing and Communications, GreenCom 2012, Conf. on Internet of Things, iThings 2012 and Conf. on Cyber, Physical and Social Computing, CPSCoM 2012* (pp. 775–782). <https://doi.org/10.1109/GreenCom.2012.128>



- Perera, C., Zaslavsky, A., Christen, P., & Georgakopoulos, D. (2014). Context aware computing for the internet of things: A survey. *IEEE Communications Surveys and Tutorials*, 16(1), 414–454.  
<https://doi.org/10.1109/SURV.2013.042313.00197>
- Polleres, A., Ruben, L., Lausen, H., Roman, D., Bruijn, J. De, & Fensel, D. (2005). A conceptual comparison between WSMO and OWL-S. *WSMO Working Group Working Draft*. <https://doi.org/10.1007/b100919>
- Prud'hommeaux, E., & Seaborne, A. (2008). SPARQL Query Language for RDF. *W3C Recommendation, 2009*(January), 1–106. <https://doi.org/citeulike-article-id:2620569>
- Reichle, R., Wagner, M., Khan, M. U., Geihs, K., Valla, M., Fra, C., ... Papadopoulos, G. a. (2008). A Context Query Language for Pervasive Computing Environments. *2008 Sixth Annual IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 434–440.  
<https://doi.org/10.1109/PERCOM.2008.29>
- Riva, O., & Di Flora, C. (2006). Contory: A smart phone middleware supporting multiple context provisioning strategies. In *Proceedings - International Conference on Distributed Computing Systems*.  
<https://doi.org/10.1109/ICDCSW.2006.33>
- Rodden, T., Cheverst, K., Davies, N., & Dix, A. (1998). Exploiting context in HCI design for mobile systems. *Workshop on Human Computer Interaction with Mobile Devices*, 21–22. <https://doi.org/10.1.1.57.1279>
- Ryan, N., Pascoe, J., & Morse, D. (1999). Enhanced Reality Fieldwork: the Context Aware Archaeological Assistant. *Archaeology in the Age of the Internet. CAA97. Computer Applications and Quantitative Methods in Archaeology. Proceedings of the 25th Anniversary Conference, University of Birmingham, April 1997 (BAR International Series 750)*, 269–274.

- Salhofer, P., & Joanneum, F. H. (2018). Evaluating the FIWARE Platform: A Case-Study on Implementing Smart Application with FIWARE. In *Proceedings of the 51st Hawaii International Conference on System Sciences* (pp. 5797–5805).
- Schmidt, A., Aidoo, K. A., Takaluoma, A., Tuomela, U., Van Laerhoven, K., & Van De Velde, W. (1999). Advanced interaction in context. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (Vol. 1707, pp. 89–101).  
[https://doi.org/10.1007/3-540-48157-5\\_10](https://doi.org/10.1007/3-540-48157-5_10)
- Schreiber, F., & Camplani, R. (2012). Perla: A language and middleware architecture for data management and integration in pervasive information systems. *Software ...*, 38(2), 478–496. <https://doi.org/10.1109/TSE.2011.25>
- Sheikh, K., Wegdam, M., & van Sinderen, M. (2008). Quality-of-context and its use for protecting privacy in context aware systems. *Journal of Software*.  
<https://doi.org/10.4304/jsw.3.3.83-93>
- Sheikh, K., Wegdam, M., & Van Sinderen, M. (2007). Middleware support for quality of context in pervasive context-aware systems. In *Proceedings - Fifth Annual IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops 2007* (pp. 461–466).  
<https://doi.org/10.1109/PERCOMW.2007.81>
- Sheng, Q. Z., & Benatallah, B. (2005). ContextUML: A UML-based modeling language for model-driven development of context-aware Web services. In *4th Annual International Conference on Mobile Business, ICMB 2005* (pp. 206–212). <https://doi.org/10.1109/ICMB.2005.33>
- Software - bIoTpe Project. (n.d.). Retrieved June 12, 2019, from <https://biotope-project.eu/software>

- Soldatos, J., Kefalakis, N., Hauswirth, M., Serrano, M., Calbimonte, J. P., Riahi, M., ... Herzog, R. (2015). OpenIoT: Open source internet-of-things in the cloud. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (Vol. 9001, pp. 13–25). [https://doi.org/10.1007/978-3-319-16546-2\\_3](https://doi.org/10.1007/978-3-319-16546-2_3)
- Sophia Antipolis. (2017). ETSI launches new group on Context Information Management for smart city interoperability. Retrieved December 2, 2018, from <https://www.etsi.org/news-events/news/1152-2017-01-news-etsi-launches-new-group-on-context-information-management-for-smart-city-interoperability>
- Strang, T., & Linnhoff-Popien, C. (2004). A Context Modeling Survey. *Graphical Models, Workshop o*, 1–8. <https://doi.org/10.1.1.2.2060>
- Suhothayan, S., Gajasinghe, K., Loku Narangoda, I., Chaturanga, S., Perera, S., Nanayakkara, V., & Narangoda, I. (2011). Siddhi: a second look at complex event processing architectures. *Proceedings of the 2011 ACM Workshop on Gateway Computing Environments - GCE '11*. <https://doi.org/10.1145/2110486.2110493>
- Sundmaeker, H., Guillemin, P., Friess, P., & Woelfflé, S. (2010). Vision and challenges for realising the Internet of Things. *Cluster of European Research Projects on the Internet of Things, European Commission*, 3(3), 34–36.
- Tan, L., & Wang, N. (2010). Future Internet: The Internet of Things. In *ICACTE 2010 - 2010 3rd International Conference on Advanced Computer Theory and Engineering, Proceedings*. <https://doi.org/10.1109/ICACTE.2010.5579543>
- Theimer, M. M., & Schilit, B. N. (1994). Disseminating Active Map Information to Mobile Hosts. *IEEE Network*, 8(5), 22–32. <https://doi.org/10.1109/65.313011>
- TPCx-IoT. (n.d.). Retrieved December 10, 2018, from <http://www.tpc.org/tpcx-iot/>

- Truong, H.-L., & Dustdar, S. (2009). A survey on context-aware web service systems. *International Journal of Web Information Systems*, 5(1), 5–31.  
<https://doi.org/10.1108/17440080910947295>
- Union, I. T. (2005). *ITU Internet Reports 2005: The Internet of Things. Communications Engineer*. <https://doi.org/10.1049/ce:20060603>
- Vermesan, O., Friess, P., Guillemin, P., Gusmeroli, S., Sundmaecker, H., Bassi, A., ... others. (2011). Internet of things strategic research roadmap. *Internet of Things-Global Technological and Societal Trends*, 1(2011), 9–52.
- Villalonga, C., Roggen, D., Lombriser, C., Zappi, P., & Tr??ster, G. (2009). Bringing quality of context into wearable human activity recognition systems. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (Vol. 5786 LNCS, pp. 164–173). [https://doi.org/10.1007/978-3-642-04559-2\\_15](https://doi.org/10.1007/978-3-642-04559-2_15)
- W3C. (2004). OWL-S: Semantic markup for web services. *W3C Member ...*, (November), 1–29. Retrieved from <http://www.ai.sri.com/daml/services/owl-s/1.2/overview/>
- Wang, X. H., Da Qing Zhang, Tao Gu, & Pung, H. K. (2004). Ontology based context modeling and reasoning using OWL. *IEEE Annual Conference on Pervasive Computing and Communications Workshops, 2004. Proceedings of the Second*, 18–22. <https://doi.org/10.1109/PERCOMW.2004.1276898>
- Want, R., Hopper, A., Falcão, V., & Gibbons, J. (1992). The active badge location system. *ACM Transactions on Information Systems*, 10(1), 91–102.  
<https://doi.org/10.1145/128756.128759>
- Ward, A., Jones, A., & Hopper, A. (1997). A new location technique for the active office. *IEEE Personal Communications*, 4(5), 42–47.
- Wei, E. J. Y., & Chan, A. T. S. (2013). CAMPUS: A middleware for automated context-aware adaptation decision making at run time. *Pervasive and Mobile Computing*, 9(1), 35–56. <https://doi.org/10.1016/j.pmcj.2011.10.002>

- Williams, J. W., Aggour, K. S., Interrante, J., McHugh, J., & Pool, E. (2015). Bridging high velocity and high volume industrial big data through distributed in-memory storage & analytics. In *Proceedings - 2014 IEEE International Conference on Big Data, IEEE Big Data 2014*.  
<https://doi.org/10.1109/BigData.2014.7004325>
- Wirth, N. (1996). Extended Backus-Naur Form (EBNF). *ISO/IEC, 14977*, 2996.
- Youn, J. (2018). Communication Scheme to Construct Self-Organization IoT Network in Heterogeneous IoT Environments. *International Journal of Control and Automation*, 11(4), 155–164.
- Zhang, K., Tang, J., Hong, M., Li, J., & Wei, W. (2006). Weighted ontology-based search exploiting semantic similarity. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3841 LNCS, 498–510. [https://doi.org/10.1007/11610113\\_44](https://doi.org/10.1007/11610113_44)

## Appendices

### Appendix A. CQL EBNF

```
CDQL      ::= DML_STATEMENT
           | DDL_STATMENT
```

```
DML_STATEMENT ::= PREFIX SELECT WHEN? DEFINE SET?
```

```
PREFIX     ::= 'prefix' PREFIX_ID ':' URI ( ',' 'prefix'
PREFIX_ID ':' URI )*
```

```
SELECT     ::= 'select' '(' ( CONTEXT-ATTRIBUTE | CONTEXT-
ENTITY | FUNCTION-CALL ) ( 'as' IDENTIFIER )? ( ',' (
CONTEXT-ATTRIBUTE | CONTEXT-ENTITY | FUNCTION-CALL ) (
'as' IDENTIFIER )? )* ')'
```

```
CONTEXT-ATTRIBUTE
```

```
          ::= CONTEXT-ENTITY-ID ( '.' IDENTIFIER )+
```

```
FUNCTION-CALL
```

```
          ::= ( PACKAGE-TITLE '::' )? FUNCTION-NAME '(' (
CONTEXT-ATTRIBUTE | CONTEXT-ENTITY-ID | FUNCTION-CALL ) (
',' ( CONTEXT-ATTRIBUTE | CONTEXT-ENTITY-ID | FUNCTION-
CALL ) )* ')' ( '.' IDENTIFIER )*
```

```

DEFINE ::= 'define' 'entity' CONTEXT-ENTITY-ID 'is
from' Prefix_ID ':' Entity_title ( 'where' CONDITION )?
SORT-BY? ( ',' 'entity' CONTEXT-ENTITY-ID 'is from'
Prefix_ID ':' Entity_title ( 'where' CONDITION )? SORT-
BY? )*

```

CONDITION

```

::= ( CONTEXT-VALUE | CONTEXT-ATTRIBUTE |
FUNCTION-CALL ) ( Comparison-Operator | Logical-Operator
) ( CONTEXT-VALUE | CONTEXT-ATTRIBUTE | FUNCTION-CALL )?

| ( CONDITION ( 'and' | 'or' ) | 'not' )
CONDITION

| '(' CONDITION ')'

```

```

SORT-BY ::= 'sort by' (CONTEXT-ATTRIBUTE | FUNCTION-CALL
| ARITHMETIC-EXPRESSION) (',' (CONTEXT-ATTRIBUTE |
FUNCTION-CALL | ARITHMETIC-EXPRESSION))* ('asc' |
'desc')?

```

```

WHEN ::= ( 'when' HIGH-LEVEL-SITUATION | 'every'
duration ) ( 'until' (date '/' ?) ( date | duration |
number 'occurrences' ) )?

```

```

duration ::= 'P' ( digit+ 'Y' )? ( digit+ 'M' )? ( digit+
'D' )? ( 'T' ( digit+ 'H' )? ( digit+ 'M' )? ( digit+
'S' )? )?

```

```

SET      ::= 'set' ( 'callback' ':' '{' 'method' ':'
METHOD ',' 'body' ':' string | 'meta' ':' '{' META-DATA-
KEY ':' CONTEXT-VALUE ( ',' META-DATA-KEY ':' CONTEXT-
VALUE )* | 'output' ':' '{' OUTPUT-CONFIG ) '}'

```

OUTPUT-CONFIG

```

      ::= 'structure' ':' STRUCTURE ( ','
'vocabulary' ':' '{' CONTEXT-ENTITY-ID ':' PREFIX_ID ':'
Entity_title ( ',' CONTEXT-ENTITY-ID ':' PREFIX_ID ':'
Entity_title )* '}' )?

```

DDL\_STATEMENT

```

      ::= CREATE-FUNCTION

      | 'create' 'package' PACKAGE-NAME

      | 'alter' 'package' PACKAGE-NAME 'set' 'title'
PACKAGE-TITLE

      | 'drop' 'function' ( PACKAGE-TITLE '::' )?
FUNCTION-NAME

```

CREATE-FUNCTION

```

      ::= PREFIX 'create function' FUNCTION-NAME 'is
on' ( Prefix_ID ':' Entity_title | Data_Type ) 'as'
Identifier ( ',' ( Prefix_ID ':' Entity_title | Data_Type
) 'as' Identifier )* ( SITUATION-FUNCTION | AGGREGATION-
FUNCTION ) ('set package' PACKAGE-TITLE)?

```



#### AGGREGATION-FUNCTION

```
 ::= ( 'post' | 'get' ) ( 'http' | 'https' )  
'://' host ( ':' port )? ( '/' ( normal_path | path_param  
) )? ( '?' ( normal_query | query_param ) )?
```

#### SITUATION-FUNCTION

```
 ::= CST-SITUATION  
  
 | HIGH-LEVEL-SITUATION
```

#### CST-SITUATION

```
 ::= SITUATION-NAME ':' '{' CONTEXT-ATTRIBUTE ':'  
CST-ATTRIBUTE-DEFINITION ( ',' CONTEXT-ATTRIBUTE ':' CST-  
ATTRIBUTE-DEFINITION )* '}' ( ',' SITUATION-NAME ':' '{'  
CONTEXT-ATTRIBUTE ':' CST-ATTRIBUTE-DEFINITION ( ','  
CONTEXT-ATTRIBUTE ':' CST-ATTRIBUTE-DEFINITION )* '}' )*
```

#### CST-ATTRIBUTE-DEFINITION

```
 ::= '{' 'ranges' ':' '[' '{' 'value' ':' ( '[' |  
'(' ) number ';' number ( ')' | ']' ) ',' 'belief' ':'  
number '}' ( ',' '{' 'value' ':' ( '[' | '(' ) number ';'   
number ( ')' | ']' ) ',' 'belief' ':' number '}' )* '['  
' ','weight' ':' number '}' '}'
```

#### HIGH-LEVEL-SITUATION

```

      ::= ( CONTEXT-VALUE | CONTEXT-ATTRIBUTE |
FUNCTION-CALL ) ( Comparison-Operator | Logical-Operator
) ( CONTEXT-VALUE | CONTEXT-ATTRIBUTE | FUNCTION-CALL )?

      | ( HIGH-LEVEL-SITUATION ( Logical-Operator |
Allens-Algerbar-OP ) | 'not' ) HIGH-LEVEL-SITUATION

      | '(' HIGH-LEVEL-SITUATION ')'

```

## Appendix B. CQL ANTLR Grammar

```
**
* @Generated
*/
grammar Cdql;

rule_Cdql : rule_Prefixs? (rule_ddl_statement |
rule_dml_statement) ;

rule_ddl_statement : rule_create_function |
rule_create_package |
                    rule_alter_function | rule_alter_package |
                    rule_drop_function | rule_drop_package |
;

rule_dml_statement : rule_query;

rule_query : (rule_Pull | rule_Push rule_When
rule_repeat?) rule_Define rule_Set_Config?
rule_Set_Callback?;

rule_create_function : CREATE (rule_sFunction |
rule_aFunction) rule_set_package?;

rule_set_package : SET PACKAGE rule_package_title;

rule_create_package : CREATE PACKAGE rule_package_title;

rule_alter_package : ALTER PACKAGE rule_package_title SET
TITLE rule_package_title;

rule_alter_function : 'tbd';

rule_drop_package: DROP PACKAGE rule_package_title;

rule_drop_function: DROP FUNCTION rule_function_id;

rule_package_title: ID;

rule_Set_Config : SET (rule_Output_Config);

rule_Set_Callback : SET (rule_Callback_Config);

rule_Output_Config : OUTPUT COLON obj;

rule_Callback_Config : CALLBACK COLON obj;
```

```

rule_Prefixs : rule_Prefix (COMMA rule_Prefix)*;

rule_Prefix : PREFIX ID COLON rule_url;

rule_Pull : PULL rule_Select;

rule_Select : LPAREN (rule_select_Attribute |
rule_select_FunctionCall) (COMMA (rule_select_Attribute |
rule_select_FunctionCall))* RPAREN;

rule_select_Attribute : rule_Attribute | rule_EntityTitle
DOT ASTERISK;

rule_select_FunctionCall : rule_FunctionCall;

rule_Attribute : rule_EntityTitle (DOT
rule_AttributeTitle)*;

rule_EntityTitle : ID;

rule_AttributeTitle : ATTRIBUTEID;

rule_FunctionCall : rule_call_FunctionTitle LPAREN
rule_call_Operand (COMMA rule_call_Operand)* RPAREN
rule_function_call_method_chaining ;

rule_function_call_method_chaining : (DOT ID)*;

rule_call_FunctionTitle : rule_FunctionTitle;

rule_call_Operand : rule_Operand | rule_Name_Operand;

rule_Name_Operand : ID COLON rule_Operand;

rule_FunctionTitle : ID (DOT ID)?;

rule_Operand : rule_EntityTitle | rule_Attribute |
rule_FunctionCall | rule_ContextValue;

rule_ContextValue : NUMBER | STRING | json;

rule_When : WHEN rule_Start;

rule_repeat : (EVERY NUMBER UNIT_OF_TIME) (UNTIL
rule_Occurrence)? | (UNTIL rule_Occurrence);

rule_Start : rule_Condition | rule_Date_Time_When;

```

```

rule_Date_Time_When : 'time' COLON rule_Date_Time;

rule_Occurrence : NUMBER UNIT_OF_TIME | NUMBER
OCCURRENCES | rule_Date_Time | LIFETIME;

rule_Date_Time : rule_Date rule_Time?;

rule_Date : NUMBER FSLASH NUMBER FSLASH NUMBER;

rule_Time : NUMBER COLON NUMBER (COLON NUMBER)?
TIME_ZONE?;

rule_Condition : rule_Constraint | rule_Condition
rule_expr_op rule_Condition | LPAREN rule_Condition
RPAREN | NOT rule_Condition;

rule_expr_op : AND | XOR | OR | NOT;

rule_Constraint : rule_left_element
rule_relational_op_func rule_right_element |
rule_target_element rule_between_op rule_left_element AND
rule_right_element | rule_target_element
rule_is_or_is_not NULL;

rule_left_element : rule_Operand;

rule_right_element : rule_Operand;

rule_target_element : rule_Operand;

rule_relational_op_func : rule_relational_op | OP LPAREN
rule_relational_op COMMA NUMBER RPAREN;

rule_relational_op: EQ | LTH | NOT_EQ | GTH | LET | GET |
CONTAINS_ANY | CONTAINS_ALL;

rule_between_op : BETWEEN | OP LPAREN BETWEEN COMMA
NUMBER RPAREN;

rule_is_or_is_not : IS | IS NOT;

ruel_Push: PUSH rule_Select ;

rule_callback : rule_http_callback | rule_fcm_callback;

rule_http_callback : METHOD EQ HTTPPOST URL EQ
rule_callback_url;

rule_fcm_callback : METHOD EQ FCM (rule_fcm_topic |

```

```

rule_fcm_token);

rule_fcm_topic: TOPIC EQ STRING;

rule_fcm_token: TOKEN EQ STRING;

rule_callback_url : rule_url;

rule_Define : DEFINE rule_Define_Context_Entity (COMMA
rule_Define_Context_Function)?;

rule_Define_Context_Entity: rule_context_entity (COMMA
rule_context_entity)*;

rule_context_entity : ENTITY rule_entity_id IS_FROM
rule_entity_type (WHERE rule_Condition)?;

rule_entity_type : (ID COLON)? ID (DOT ID)?;

rule_Define_Context_Function : rule_context_function
(COMMA rule_context_function)*;

rule_context_function : rule_aFunction | rule_sFunction;

rule_aFunction : 'aFunction' rule_function_id rule_url;

rule_sFunction : 'sFunction' rule_function_id rule_is_on
( cst_situation_def_rule);

rule_is_on : 'is on' rule_is_on_entity (COMMA
rule_is_on_entity)* ;

rule_is_on_entity : rule_entity_type AS ID;

cst_situation_def_rule : '{' rule_single_situatuin (COMMA
rule_single_situatuin)* '}';

rule_single_situatuin : STRING COLON '{'
rule_situation_pair (COMMA rule_situation_pair)* '}';

rule_situation_pair : rule_situation_attributes ':' '{'
situation_pair_values '}';

rule_situation_attributes : rule_situation_attribute_name
| '[' rule_situation_attribute_name (COMMA
rule_situation_attribute_name)+ ']';

rule_situation_attribute_name : ID (DOT ID)*;

```

```

situation_pair_values : (situation_range_values COMMA
situation_weight) | (situation_weight COMMA
situation_range_values);

situation_weight : 'weight' COLON NUMBER;

situation_range_values: 'ranges' COLON '['
situation_pair_values_item (COMMA
situation_pair_values_item)* ']';

situation_pair_values_item : '{' ((rule_situation_belief
COMMA rule_situation_value) | (rule_situation_value COMMA
rule_situation_belief)) '}';

rule_situation_belief: 'belief' COLON NUMBER;

rule_situation_value : 'value' COLON ( rule_region_value
| rule_discrete_value | discrete_value);

rule_discrete_value : '[' discrete_value (COLON
discrete_value)* ']';

discrete_value : json;

rule_region_value : region_value_inclusive |
region_value_left_inclusive |
region_value_right_inclusive | region_value_exclusive;

region_value_inclusive: '[' region_value_value ']';
region_value_left_inclusive: '[' region_value_value ')';
region_value_right_inclusive: '(' region_value_value ']';
region_value_exclusive: '(' region_value_value ')';
region_value_value: NUMBER ';' NUMBER ;

rule_entity_id : ID;

rule_function_id : ID;

rule_url
: authority '://' host (':' port)? ('/' path)? ('?'
search)?
;

authority
: ID

```

```

;

host : hostname| hostnumber;

hostname : ID ('.' ID)*;

hostnumber : INT '.' INT '.' INT '.' INT;

search : searchparameter ('&' searchparameter)*;

searchparameter : ID ('=' (ID |INT | HEX))?;

port
    : INT
    ;

path
    : (normal_path | path_param) ('/' (normal_path |
path_param))*
    ;

normal_path : ID;

path_param : '{' ID '}';

TITLE : 'title';

PACKAGE: 'package';

FUNCTION : 'function';

CREATE : 'create';

SET : 'set';

ALTER : 'alter';

DROP : 'drop';

DEFINE : 'define';

CONTEXT_ENTITY : 'context entity';

IS_FROM : 'is from';

WHERE : 'where';

WHEN : 'when';

```



```
DATE : 'date';
LIFETIME : 'lifetime';
BETWEEN : 'between';
IS : 'is';
PULL : 'pull';
ENTITY : 'entity';
AS : 'as';
EVERY : 'every';
UNTIL : 'until';
LPAREN : '(';
COMMA : ',';
RPAREN : ')';
DOT : '.';
NOT : '~' | '!' | 'not';
AND : 'and' | '&&' ;
OR : 'or' | '||';
XOR : 'xor';
IN : 'in';
CONTAINS_ANY : 'containsAny';
CONTAINS_ALL : 'containsAll';
NULL : 'null';
EQ : '=';
LTH : '<';
GTH : '>' ;
```

```
LET : '<=';
GET : '>=';
NOT_EQ : '!=';
PUSH : 'push';
INTO : 'into';
PREFIX : 'prefix';
HTTPPOST: 'http/post';
POST : 'post';

METHOD : 'method';
URL: 'url';
FCM : 'fcm';
TOPIC : 'topic';
TOKEN : 'token';
TYPE : 'type';
COLON : ':';
ASTERISK : '*';
UNIT_OF_TIME : 'h' | 's' | 'ms' | 'd' | 'm' | 'ns';
OCCURRENCES : 'occurrences';
FSLASH: '/';
OP : '$op';
OUTPUT : 'output';
CALLBACK : 'callback';
TIME_ZONE : 'UT'
            | 'GMT'
            | 'EST'
```

```

| 'EDT'
| 'CST'
| 'CDT'
| 'MST'
| 'MDT'
| 'PST'
| 'PDT'
| (('+' | '-') NUMBER);

json
: value
;

obj
: '{' pair (',' pair)* '}'
| '{' '}'
;

pair
: STRING ':' value
;

array
: '[' value (',' value)*? ']'
| '[' ']'
;

value
: STRING
| NUMBER
| obj
| array
| 'true'
| 'false'
| 'null'
;

STRING
: '"' (ESC | ~ ["\\])* '"';

ID : ('a'..'z' | 'A'..'Z' | '_' ) ('a'..'z' | 'A'..'Z' |
 '_' | '0'..'9')* ;

ATTRIBUTEID : '@'? ('a'..'z' | 'A'..'Z' | '_' ) ('a'..'z'
 | 'A'..'Z' | '_' | '0'..'9')* ;

```

```

fragment ESC
: '\\ ' (["\\/\bfnrt] | UNICODE)
;

fragment UNICODE
: 'u' HEX HEX HEX HEX
;
fragment HEX
: [0-9a-fA-F]
;

COMMENT : ('/*' .* '*/' | '// ' ~('\r' | '\n')*) -> skip
;

WS: (' ' | '\r' | '\t' | '\u000C' | '\n') -> skip ;

NUMBER
: '-'? INT '.' [0-9] + EXP? | '-'? INT EXP | '-'? INT
;

fragment INT
: '0' | [1-9] [0-9]*
;

fragment EXP
: [Ee] [+|-]? INT
;

```