Modelling, Planning and Adaptive Implementation of Semantic Event Service

Author:
Feng Gao

Supervisor:
Dr. Edward Curry

Co-Supervisor:
Dr. Alessandra Mileo

Submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy

INSIGHT Centre for Data Analytics
National University of Ireland, Galway

May 13, 2016

Declaration of Authorship

I, Feng Gao, declare that this thesis titled, 'Modelling, Planning and Adaptive Implementation of Semantic Event Service' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ____________________________________

Date: ____________________________________

Abstract

Recent developments in sensor networks, social media, process management and data analysis envisions interlinked devices, people, processes and data, constituting an Internet-of-Everything. These networks can be used to help create systems that detect situations occuring in urban life and respond to those situations in a timely manner, so that smart decisions can be made for citizens, organisations, companies and city administrations.

Event processing is an important technique in developing Smart City applications that target detecting patterns in events captured at real-time. However, event processing faces many challenges in the context of creating Smart City applications and building the Internet-of-Everything, including integrating heterogeneous data sources and data interfaces, ensuring quality-of-service and providing an easy-to-use and easy-to-maintain platform. This thesis addresses these challenges by integrating Semantic Web, Service Oriented Architecture and Complex Event Processing techniques, realising a network of semantic complex event services. The modelling, planning and adaptive implementation of semantic event services are researched, with the objective of facilitating an efficient and effective management of the life-cycle of a semantic event service. Service composition algorithms based on pattern matchmaking are developed and evaluated. Experiments show that by leveraging an event service reusability index, the composition time for large repositories and complicated queries can be reduced significantly, making on-demand composition possible for scenarios with large solution spaces. Genetic algorithms are developed for optimising event service compositions with regard to quality-of-service metrics. The evaluation shows that the genetic algorithms are effective and scalable. Different quality-aware event service adaptation strategies are developed to recover the quality of the system at run-time. Experiments show that applying different adaptation strategies have trade-offs between adaptation efficiency and effectiveness.

The techniques are integrated in the Automatic Complex Event Implementation System, which serves as a middleware for handling activities in the life-cycle of semantic event services. A prototype of the system is developed and tested over different smart city application scenarios, based on both real data collected from the city of Aarhus as well as synthetic data. By addressing the requirements in real-world scenarios, the prototype demonstrates the validity and feasibility of the system.

Acknowledgements

The completion of my Ph.D. thesis was a result from a long academic journey at the INSIGHT Centre for Data Analytics, National University of Ireland, Galway. This journey is made of almost equal parts of joy and struggle. Now I have reached a notable milestone in my life, which they say, and I concur, that one cannot reach all by him/herself. Indeed, there is a long list of names that I feel obligated to thank for.

First and foremost, I would like to thank Dr. Edward Curry, my thesis supervisor, for his patience and excellence in helping me organise my research and this dissertation. I have lost count on how many times we sit together and discuss the issues of the thesis, some of them repeatedly, as well as the research publications relevant for the thesis. His expertise and positive attitude encouraged me throughout the process of the thesis writing. I would like to also thank my project advisor and co-supervisor, Dr. Alessandra Mileo, not only for funding my research and keeping me warm and fed during my Ph.D., but also for the various discussions we had for the CityPulse project, which cultivates many of the research ideas in my thesis. I am also thankful for the valuable suggestions provided by Dr. Sami Bhiri in my early stages of pursuing the Ph.D.

The members of my Graduate Research Committee, Prof. Manfred Hauswirth, Prof. Stefan Decker and Dr. Adegboyega Ojo have been keeping an eye on my research and making sure that I am on track since 2010. The research presented in this thesis has received many insightful feedbacks from the committee. The study at INSIGHT has been a great experience and I appreciate all the discussions, debates and collaborations I had with my colleagues and friends from the Unit of Reasoning and Query, including Dr. Muhammad Initizar Ali, Soheila Dehghanzadeh, Thu-Le Pham and Zia Ush Shamszaman. Their help in proof-reading the thesis chapters is also deeply appreciated. Special thanks to Dr. Michael Schukat and Dr. Payam Barnaghi for their time and efforts spent reviewing the thesis.

Last but not the least, I would like to extend my deepest and sincerest gratitude to my family. I would not be where I am now without the support (and genes) from my parents, or without the earnest and faithful love from my wife, YiZhuo Zhu, who has always been there for me and our lovely daughter RuoMin Gao, even when my foreign study separates us apart physically. I dedicate this thesis to you.

Contents

List of Figures

List of Tables

Abbreviations

ACP	**A**bstract **C**omposition **P**lan
ACEIS	**A**utomatic **C**omplex **E**vent **I**mplementation **S**ystem
BEMN+	extended **B**usiness **E**vent **M**odelling **N**otations
CEP	**C**omplex **E**vent **P**rocessing
CES	**C**omplex **E**vent **S**ervice
CESO	**C**omplex **E**vent **S**ervice **O**ntology
CCP	**C**oncrete **C**omposition **P**lan
DSMS	**D**ata **S**tream **M**anagement **S**ystem
DST	**D**irect **S**ub **T**ree
ESN	**E**vent **S**ervice **N**etwork
EST	**E**vent **S**yntax **T**ree
FP	**F**unctional **P**roperty
GA	**G**enetic **A**lgorithm
IoE	**I**nternet of **E**verything
IoT	**I**nternet of **T**hings
NFP	**N**on-**F**unctional **P**roperty
PES	**P**rimitive **E**vent **S**ervice
QoS	**Q**uality of **S**ervice
RSP	**R**DF **S**tream **P**rocessing
SES	**S**emantic **E**vent **S**ervice

致我的妻子与女儿

To Yizhuo Zhu and Ruomin Gao

Part I

Foundation - Motivation and Background

Chapter 1

Introduction

1.1 Motivation

During the past decade, the developments in hardware infrastructures and software techniques have resulted in a rapid growth in the amount of information collected, shared and processed by human as well as machines on a daily basis [4]. Typical sources of information include sensor networks, smart phones, social networks and business processes. Recently, there has been a wide-spread discussion on research topics like "Smart City" and "Internet-of-Everything" (IoE) [5], which envisions building new applications, platforms or methodologies based on these data sources.

A major challenge in realising the Smart City and IoE is analysing huge volumes of heterogenous data in (near) real-time, which is beyond the capability of traditional Relational Database Management Systems (RDBMS) [6] as well as other conventional information management approaches.

Complex Event Processing (CEP) [7] has been a widely discussed and used for deductive reasoning over dynamic event streams to detect complex events according to predefined event patterns [1]. CEP has been proved to be efficient for processing streams with high frequency and queries with complicated semantics, making it an important technique in solving the problem of real-time information flow analysis [8]. Typically in CEP systems, an "event" is broadly defined as "an occurrence within a particular system or domain" or a change of state in the universe [1]. A "complex event" (or "composite event") can be defined as an event consisting several different event instances [1]. A complex event can be detected by a matching of an event pattern, which is a template containing event templates, relational operators and variables[1].

[1] Event Processing Technical Society (EPTS): http://www.ep-ts.com/, last accessed: Dec. 2015.

In the context of a Smart City where applications are deployed in different domains have the potentials and needs to collaborate, conventional CEP faces new challenges, e.g., incorporating heterogeneous data sources and reusing CEP applications built on different platforms [4]. Moreover, there is a lack of automatic means to discovery and compose event streams according to users' requests or to recover the system from erroneous states [9, 10]. Scalability is also an issue here, since Smart City applications may need to integrate thousands to tens of thousands of data sources and perhaps even more users. To tackle these issues, the works in this thesis integrate CEP systems with the Service Oriented Architecture (SOA) to provide CEP capability as services, which are self-contained black boxes that offer the CEP capability to the event consumers[2]. This way CEP applications can benefit from the flexibility and reusability offered by SOA [11].

Integrating CEP with SOA is not a new idea and has been adopted at enterprise levels as parts of Business Process Management (BPM) solutions. For example, companies like TIBCO[3], IBM[4] and Oracle[5], have been providing CEP solutions connected with Enterprise Service Bus[6] (ESB) thus realising an event-driven SOA [12]. Figure 1.1 illustrates the high-level architecture of these systems. However, these solutions rely on existing Web Service standards, which do not provide adequate models for complex events. As a result, conventional service discovery and composition techniques (i.e., keyword, type or attribute matching) are used for such CEP services, if and when these services are published in the ESB [13, 14]. These techniques are insufficient for identifying the semantics of complex events defined as event patterns [15].

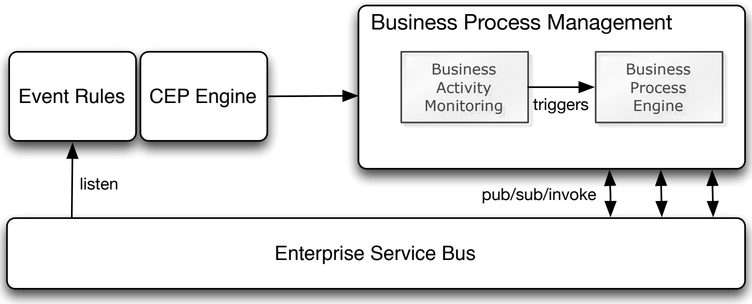


FIGURE 1.1: Existing CEP and SOA integration architecture

[2] Open Group's definition for service: https://www.opengroup.org/soa/source-book/soa/soa.htm, last accessed: May, 2015.

[3] TIBCO BusinessEvents: http://www.tibco.com/products/event-processing/complex-event-processing/businessevents/, last accessed: May, 2015.

[4] IBM WebSphere Business Events: http://www-01.ibm.com/software/integration/wbe/, last accessed: May, 2015.

[5] Oracle Event Processing: http://www.oracle.com/us/products/middleware/soa/event-processing/overview/index.html, last accessed: May, 2015.

[6] Oracle's definition on ESB: http://www.oracle.com/technetwork/articles/soa/ind-soa-esb-1967705.html, last accessed: May, 2015.

In order to fully integrate CEP with SOA, the CEP capability should be provided as services, in the following, the definition for an Event Service (Definition 1.1) as well as a Complex Event Service (Definition 1.2) are given.

Definition 1.1 (Event Service)**.** An asynchronous notification service that accepts subscriptions from event consumers and delivers events.

Definition 1.2 (Complex Event Service)**.** An event service that delivers complex events detected by an underlying event engine for its consumers during the subscription, with the event pattern(s) of the complex event(s) published as part(s) of its service description.

Event services not equipped with CEP capability or do not describe the event patterns in their service descriptions are called Primitive Event Services (PES)[7]. Together, a set of CESs, PESs and the communication among them constitutes an Event Service Network (ESN), in which CES can be reused by or composed from other event services, making CEP capability a first-class citizen in service computing. Figure 1.2 illustrates an overview of an ESN.

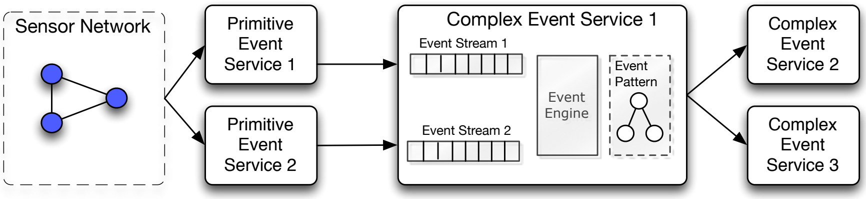


FIGURE 1.2: Overview of an event service network

Another challenge for CEP in the context of a Smart City is to provide a common ground for data semantics [16, 17]. The concept of Semantic Web (SW) and relevant techniques have been explored to enhance the semantic interoperability of data [18]. Recent research show interests in using semantics in event processing in order to bridge the semantic differences from various event sources and facilitate knowledge-aware event processing [19–24]. Semantic Web techniques have been also used in web services to describe service metadata, thus realising a Semantic Web Service (SWS) [25]. SWS facilitates knowledge-based service discovery and composition and improves the semantic interoperability of service metadata [26–28]. When event services in an ESN use semantic annotations in both the delivered event messages as well as in the service descriptions, they are referred to as Semantic Event Services (SESs). Table 1.1 summarises the concepts introduced in this section.

[7] It is worth mentioning that a PES may deliver complex events detected based on event patterns. However, since PESs do not publish the patterns in the service descriptions the event logics are not recognisable during service planning, therefore, these events are treated as primitive events, i.e., identifiable by event types or attributes but not patterns.

Concepts	**Definitions**	**Examples**
Event	"Anything that takes place or happens, especially something important." – *Collins English Dictionary* [29]. "An occurrence within a particular system or domain..." – *Event Processing in Action* [1].	Any arrival or non-arrival of new data, or information derived from those data, in an information system.
Primitive/Simple Event	"An event that is not viewed as summarizing, representing, or denoting a set of other events." – *EPTS*	A traffic sensor observation reporting the vehicle count and average speed on a street segment.
Complex Event	"An event consisting several different event instances" – *Event Processing in Action* [1]. "An event that summarises, represents, or denotes a set of other events." – *EPTS*	A traffic jam event detected from traffic sensor readings.
Event Pattern	"A template containing event templates, relational operators and variables." –*EPTS*	A set of rules specifying how the traffic jam is detected from sensor readings, e.g., 80% of the sensors have reported high vehicle count and low average vehicle speed during the past 30 minutes repeatedly.
Service	"A service is a self-contained, logical representation of a repeatable business activity that has a specified outcome", "is a 'black box' to the consumer of the service" – *The Open Group* (Footnote 2)	A data service provided via REST APIs allowing citizens to query real-time status of city infrastructures.
Event Service	"An ... service that ... delivers events." (Definition 1.1)	A service publishing city events to citizens based on their subscriptions.
Complex Event Service (CES)	"An event service that delivers complex events ..., with the event patterns ... published as ... its service description." Definition 1.2	An event service publishing traffic jam notifications.
Primitive Event Service (PES)	An event service not equipped with CEP capability or does not describe the event pattern in the service description, i.e., an event service that is not a CES.	An event service publishing directly traffic sensor readings.
Semantic Event Service (SES)	An event service processes semantically annotated events (with background knowledge) and equipped with a semantically annotated service description.	When the service descriptions and event messages above traffic jam/reading services are semantically annotated.
Event Service Network (ESN)	A network consisting a set of interconnecting event services.	The traffic jam service, the traffic reading service, and the network allowing the former to utilize the latter.

Table 1.1: Concepts and definitions relevant for event services

Consider a traffic monitoring application in a Smart City. The traffic sensors deployed on the streets are continuously generating traffic reports. These traffic sensors can be wrapped as PESs, providing primitive events about the traffic conditions. When a traffic monitoring query over several streets is needed, one can find the relevant sensors to answer the query leveraging the functional and non-functional aspects described in the service descriptions of the PESs. Meanwhile, if these PES descriptions are semantically annotated, a semantic discovery process is possible. Moreover, the stream of continuous query results can be modelled as a CES, since the results are derived from multiple traffic reports. This CES can be reused by other queries covering the streets involved in the query, or even by a query from a different application domain, such as a smart travel planning or parking application. The details of scenarios using the event services in a Smart City are given in Chapter 2.

1.2 Research Problems

This thesis investigates means for realising easy-to-use, on-demand and scalable event processing using semantic event services. To achieve this goal, different activities related to event services from their creation to termination are analysed, including *service description, request definition, planning, deployment/execution* and *adaptation.* These activities are iterated during the life cycle of SESs (by analogy to the service life cycle in [30, 31]). In the following, first the requirements for the realising and managing the above activities in the life cycle are analysed, then, the limitations of current approaches are discussed and finally the research questions are detailed.

1.2.1 Requirements Analysis

The following four basic requirements are considered for realising an efficient and effective management of the SES life cycle.

User-centric SES Modelling: During *request definition* phase, event requests should reflect each individual user's requirements or constraints on both functional and non-functional properties (NFP) of complex events. Users should be able to specify different events they are interested in by specifying Functional Properties (FP) like event type and pattern. Meanwhile, for the same complex event, users may have different focuses on the NFPs: some may ask for accurate results while others may ask for more timely notifications, etc [32]. The implemented event services should fulfil those requirements and constraints, which implies that NFPs should be provided in the service metadata during *service description.* Moreover, to integrate SES as part of business process models that consumes the complex events, the *request definition* should be user-friendly for process model designers [3, 33]. It should allow non-technical or business users to specify requests with minimal effort and learning overhead.

Automatic and Customised SES Planning: The service *planning* activity should be able to automatically discover and compose SESs according to users' functional and non-functional requirements [34]. To fully benefit from automatic implementation and enable an on-demand ESN implementation, the automatic planning should be efficient to be carried out at run-time [35].

Automatic and Adaptive SES Implementation: The *deployment* of composition plans should also be automatic to facilitate automatic evaluation of event rules over different CEP systems. Adaptability is also important in SOA [36].The *adaptation* activity should have the ability to automatically detect service failures or constraint violations according to users' requirements at run-time and make appropriate adjustments, including re-composition and re-deployment of composition plans, to adapt to changes. The adaptation process should be efficient to minimise information loss and maximise the performance of the ESN over time.

Efficient SES Execution: The *execution* and performance optimisation of services is the responsibility of service providers [31]. In the context of SESs, the efficiency of semantic event detection depends on the underpinning semantic event processing engines. In this thesis, the RDF Stream Processing (RSP) engines (e.g., CQELS [37] and CSPARQL [38]) are utilised for the execution of SESs. The implementation of a novel RSP engine or optimisation for the internal processing mechanisms of existing engines are out of the scope of this thesis. However, given a specific problem setting, a benchmarking system can help to determine which specific type of RSP engine is the most efficient choice to execute a composition plan [39, 40]. Moreover, when multiple engine instances are used in a distributed fashion, the efficiency of service composition plan execution can be improved by shedding the load from overloaded engines [41–43], without tampering with their internal implementations. A benchmark system can also be used to evaluate the performance improvement of the load balancing technique.

1.2.2 Limitations of Current Approaches

Current event based systems do not fully satisfy the requirements listed above. A summary of the limitations of current approaches with regard to the requirements is shown in Table 1.2. In the following sections these limitations are discussed in more details.

Requirements	Limitations	Evidences
User-centric Modelling	Lack of NFP customization in CEP systems.	Rapide, StreamDrill, Esper, StreamBase.
	Lack of pattern definition in services	WS-Notification, SAS, DPWS etc.
Automatic and Customised Planning	Lack of pattern-based and constraint-aware composition	SIENA[44], REBECA [45], PADRES [46], [27]
Automatic and Adaptive Implementation	Lack of automatic query creation for different CEP systems.	BEMN [3], ERDT [33], [47]
	Lack of a holistic and platform-independant QoS-aware adaptation	[48, 49], PIRATES [50], StreamHub [51, 52], AR [53], [54], [55–57]
Efficient Execution	Lack of configurable RSP benchmark with realistic datasets	LS-Bench [39], SR-Bench [40]

Table 1.2: Limitations of current approaches

1.2.2.1 User-centric SES Modelling

In event processing, requested complex events are usually defined by event patterns. An event pattern describes how a set of member events are correlated and contribute to the detection of a complex event, which is the functional specification of a complex event. Existing event processing engine (e.g., Rapide[8], EPL[9]) and CEP engines (e.g., StreamDrill[10], Esper[11] and StreamBase[12]) do not allow constraints on NFPs. However, to fully support customisation, the ability to describe constraints on NFPs for both complex and primitive events are needed. Supporting NFP has recently gained some research interest in the Data Stream Management Systems (DSMS) and Event Broker Networks (EBN) community. In [58, 59] the authors discuss how Software Defined Network (SDN) concepts can be used to improve the performance of EBNs. On the other hand, in service computing, existing eventing service models can describe NFPs for event services and event types for PESs, however, they do not support describing complex events with patterns.

Most of the existing event pattern definition languages have SQL-like syntax (e.g., Rapide, EPL). For business users, manipulating SQL-like queries are more difficult than using graphical notations. There exists only a few works (e.g., [33, 47, 60]) which use graphical notations to specify event patterns, however they are platform-specific. BEMN [3] provides graphical notations for complex events that are compatible with Business Process Model and Notation[13] (BPMN) with formal semantics. However, the users need

[8] RAPIDE: http://www.complexevents.com/rapide/, last accessed: Jan, 2015.

[9] Event Processing Language: https://docs.oracle.com/cd/E13157_01/wlevs/docs30/epl_guide/overview.html, last accessed: Jan, 2015.

[10] StreamDrill: https://streamdrill.com/, last accessed: Jan, 2015.

[11] Esper engine: http://esper.codehaus.org/, last accessed: Jan, 2015.

[12] StreamBase CEP engine: http://www.streambase.com/wp-content/, last accessed: Jan, 2015.

[13] BPMN: http://www.bpmn.org/, last accessed: May, 2015.

to know the exact data structure of the event instances in the event streams in order to create event queries.

1.2.2.2 Automatic and Customised SES Planning

Current event processing systems can detect complex events automatically, when the event sources are determined and event queries are specified [1, 7]. But they do not provide automation support for discovering or composing event streams (or data services) for event queries, which allows reusing complex events detected by other queries. There are only a few works on complex event reuse, and most of them support only syntactical-content and simple pattern based subscription reuse [44–46]. On the other hand, existing automatic service discovery and composition techniques (e.g., surveyed in [27]) do not cater for complex event services, because their matchmaking is based on Input, Output, Preconditions and Effects (IOPE), e.g., in [61]. However, the functionalities of event services are determined by the semantics of the events they deliver, which is captured by the event patterns (defined within an event algebra), not by the IOPEs [62]. In addition, a conventional web service composition plan is an imperative workflow, while an event service composition plan is a declarative query. As a result, a novel QoS aggregation schema and efficient, pattern-based composition algorithms are needed for automatic and customised SES planning.

1.2.2.3 Automatic and Adaptive SES Implementation:

Existing publish-subscribe systems do not have a holistic Quality-of-Service aware adaptation and recovery mechanism [48, 49]. Most of them [44, 50–55] focus on a single QoS metric like latency, throughput or reliability etc. On the other hand, existing adaptive CEP systems consider only the adaptation on the data level, i.e., they can dynamically rewrite query execution plans based on the data coming into the systems [56, 57], however they cannot perform adaptations on the stream level, i.e., replacing input streams because the QoS of the original streams have changed. Also, as existing adaptive event systems are platform-dependent there is a lack of cross-platform solutions.

Since adaptation may result in using different streams with different operators, i.e. the composition plan may change after adaptation, automatic query creation based on composition plans is needed to realise automatic adaptation. However, the compositions of event streams is hard-coded as engine-specific event queries and must be manually created by users, i.e., an automatic way of creating queries from composition plans for different event engines is missing.

1.2.2.4 Efficient SES Execution:

Existing benchmarks for RSP engines like SR-Bench [40] and LS-Bench [39] use pre-configured, synthetic datasets. However, in real-world scenarios, different configurations need to be tested to study the performance of RSP engines. In addition, using synthetic datasets in benchmarking systems may produce un-reliable results [63]. On the other hand, existing RSP engines have scalability issues and cannot deal with large-amount of queries concurrently [39], which hinders the application of RSP in large-scale applications. Service load balancing techniques have been studied extensively in the literature [41–43, 64, 65]. Various metics, from basic execution latency and bandwidth usage [41] to sophisticated service correlations [42, 66] and network path analysis [43, 67] have been proposed to evaluate the load and determine the shedding strategy. In this thesis, some basic metrics, e.g., number of queries deployed and average latency of engine instances, are applied to SES.

1.2.3 Research Questions

To overcome the above limitations of current approaches, the following research questions need to be answered.

RQ1: What is the suitable information model for describing event services and event patterns?

RQ1 aims at addressing the user-centric SES modelling requirement. It can be divided into three sub-questions:

(a) How to semantically annotate the description for event services and requests?

(b) How to define the formal semantics of complex event patterns?

(c) How to graphically present the event semantics so that the business users can understand and define them easily?

RQ2: How to facilitate efficient and customised event service composition?

RQ2 aims at addressing the automatic and customised SES planning requirement. It can be divided into two sub-questions:

(a) How to efficiently create event service compositions based on the functional aspects of event services.

(b) How to efficiently create event service compositions based on the non-functional aspects of event services.

RQ3: How to realise automatic and adaptive event service implementation?

RQ3 aims at addressing the automatic and adaptive SES implementation requirement. It can be divided into two sub-questions:

(a) How to automatically deploy executable services according to composition plans?

(b) How to efficiently re-deploy event service compositions when constraint violations are detected at run-time?

RQ4: How to evaluate and improve the performance of semantic event service execution?

RQ4 aims at addressing the efficient semantic event service execution requirement. It can be divided into three sub-questions:

(a) How to design a configurable benchmark for RDF stream processing engines?

(b) What are the differences of the RDF stream processing engines in terms of query performance?

(c) How to improve the performance of RDF stream processing engines with regard to handling concurrent queries?

1.3 Methodology

The research methods in this thesis consist of the following steps:

1. *Survey the state-of-the-art* to get a view of the landscape of the relevant research areas, and understand the achievements and limitations of the current process event modelling, discovery, deploy and detection approaches.
2. *Analyse the requirements* to identify concrete research problems that need to be dealt with in order to promote the efficiency and effectiveness of event service life cycle.
3. A *theoretical design* phase to include the overall design of the ACEIS system, as well as the development of formalisms and algorithms used in each functional module.

4. A *prototype implementation* phase to include the development of concrete ACEIS components and sub systems, namely the event pattern modelling tool, event service discovery/composition engine, constraint analysis module and the run-time event adaptation system.

5. Finally the *feasibility and performance evaluation* phase evaluates the proposed ACEIS with regard to applicability and domain-specific performance measures, using simulated or real-world datasets.

1.4 Overview of the Proposed Approach

The goal of this thesis is to provide means for the efficient and effective CES life cycle management, i.e., providing answers for the research questions in Section 1.2.3 and fulfil the requirements analysed in Section 1.2.1. To answer the first three questions, an Automatic Complex Event Implementation System (ACEIS) is designed and implemented in this thesis. ACEIS should offer the following four key functionalities.

Event Service Annotation and Event Pattern Definition. This functionality answers the research question *RQ1*. An event service ontology should be used as the information model to semantically annotate complex event services with event patterns and NFPs to increase the data interoperability. The event service ontology should also be used to formulate customised event service requests. The event pattern syntax and semantics in BEMN [3] can be extended and revised to define event patterns annotated in the ontology. The graphical notations in BEMN should be adopted and extended to allow user-friendly event pattern definition for non-technical/business users.

Pattern-based Event Service Discovery and Composition. This functionality answers the research question *RQ2(a)*. Using event service ontology and event pattern definitions, an event service discovery/composition engine should be able to make match-makings between event service requests and service candidates by comparing the event pattern semantics, and derive optimal service composition plans that reuse as many as possible existing event services to reduce the event traffic over the network.

Constraint-aware Event Service Discovery and Composition. This functionality answers the research question *RQ2(b)*. Users should be able to specify customised constraints and preferences for NFPs of event service compositions. Using the specifications on NFPs in both event service requests and descriptions, the discovery/composition engine should be able to derive (near-) optimal composition plans to best suit the users' needs. Meanwhile, the composition algorithm should be efficient to derive results upon large scale ($\gg 10^3$) service repositories deployed in an enterprise or a city.

Automatic Event Service Implementation and Adaptation. This functionality answers the research question *RQ3*. ACEIS should facilitate an automatic and adaptive event implementation. It should be able to transform event service composition plans into event queries and deploy them on event processing engines automatically to implement the CEP capability. ACEIS should also be able to detect NFP constraint violations and service failures event service implementations at runtime and find replacements for the malfunctioning/invalid parts of the event service compositions with little or no human intervention.

Apart from implementing the core functionalities of ACEIS, in order to answer the fourth research question, a benchmark for RSP engines is needed and load balancing techniques are desired for the **Benchmarking and Optimisation of Event Service Execution**. This functionality answers the research question *RQ4*. The performance of event queries from Smart City scenarios should be evaluated based on real-world datasets, using different benchmark configurations. In order to improve the capacity of handling concurrent queries in RSP engines, different mechanisms should be used to distribute the workload among a set of RSP engine instances. ACEIS should be augmented with a scheduler to implement the load balancing mechanisms. The impact of the optimisation should be evaluated leveraging the benchmark functionality. Figure 1.3 illustrates the relations between the research questions and proposed solutions.

1.5 Summary of Contributions

Following the research methodology, the research presented in this thesis results in the following contributions in event service modelling, planning, implementation and adaptation in order to provide an easy-to-use, on-demand event processing capability in application domains such as a Smart City:

Event Service Modelling. A semantic event service model (i.e., Complex Event Service Ontology) is developed to annotate event service descriptions and service requests. The event service ontology allows discovery and composition of event services based on both functional and non-functional service properties. The event service ontology addresses *RQ1(a)*. A graphical language for specifying event patterns is provided as an extension from BEMN [3], called BEMN$^+$. The graphical notations in BEMN$^+$ is compatible with BPMN. The abstract syntax and semantics of the event patterns are defined. BEMN$^+$ addresses *RQ1(b,c)*.

Event Service Planning. Algorithms are developed to index event patterns based on their semantics, inserting an event pattern with 10 to 70 nodes into a 1500-node

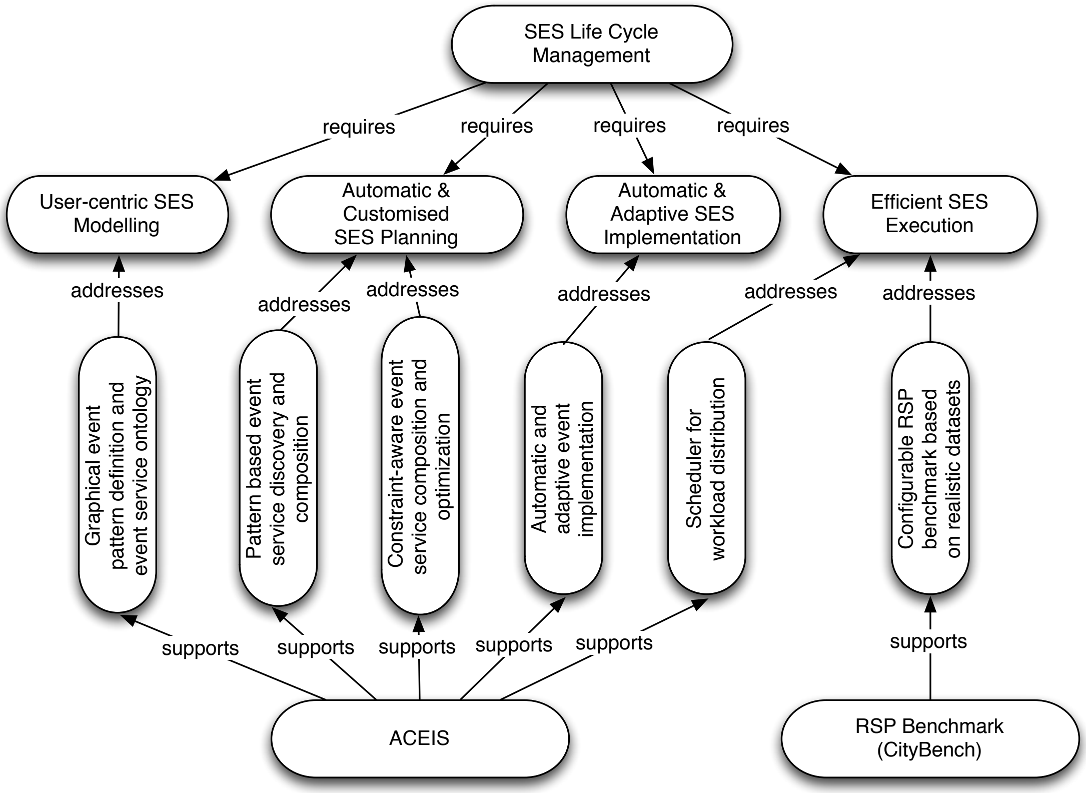

SES Life Cycle Management
requires
requires
requires
requires
User-centric SES Modelling
Automatic & Customised SES Planning
Automatic & Adaptive SES Implementation
Efficient SES Execution
addresses
addresses
addresses
addresses
addresses
addresses
Graphical event pattern definition and event service ontology
Pattern based event service discovery and composition
Constraint-aware event service composition and optimization
Automatic and adaptive event implementation
Scheduler for workload distribution
Configurable RSP benchmark based on realistic datasets
supports
supports
supports
supports
supports
supports
ACEIS
RSP Benchmark (CityBench)

Figure 1.3: Relations between research questions and proposed solutions

index may take several to ∼2000 milliseconds (Figure 6.9). Event service composition algorithms (with or without index) are developed to compose event services based on comparing semantics of event patterns. The algorithms for event service composition based on the reusability index are inspired by the query subsumption techniques in database systems [68]. These algorithms address *RQ2(a)* by providing efficient and accurate means of composing event services using event patterns. The reusable relation (and the index build upon the relation) facilitates Experiments show that the indexed composition algorithm uses ∼41% of the time taken by the un-indexed one when handling large queries (25 nodes) on service repository with 1000 candidates (Figure 6.10).

Algorithms are developed to the QoS of event services in event service compositions, in order to estimate the overall QoS of event service compositions. Based on the event service QoS aggregation and estimation, heuristic algorithms are developed to derive optimal event service compositions with regard to non-functional constraints and preferences. The QoS aggregation schema and QoS-aware event service composition algorithm address *RQ2(b)* by extending the genetic encoding schema and operators in existing genetic algorithms to cater complex event services. Experiments show that the genetic algorithm can achieve ∼89% optimal results within 2 seconds for a service repository with 9000 candidates (Figure 7.9 and 7.10).

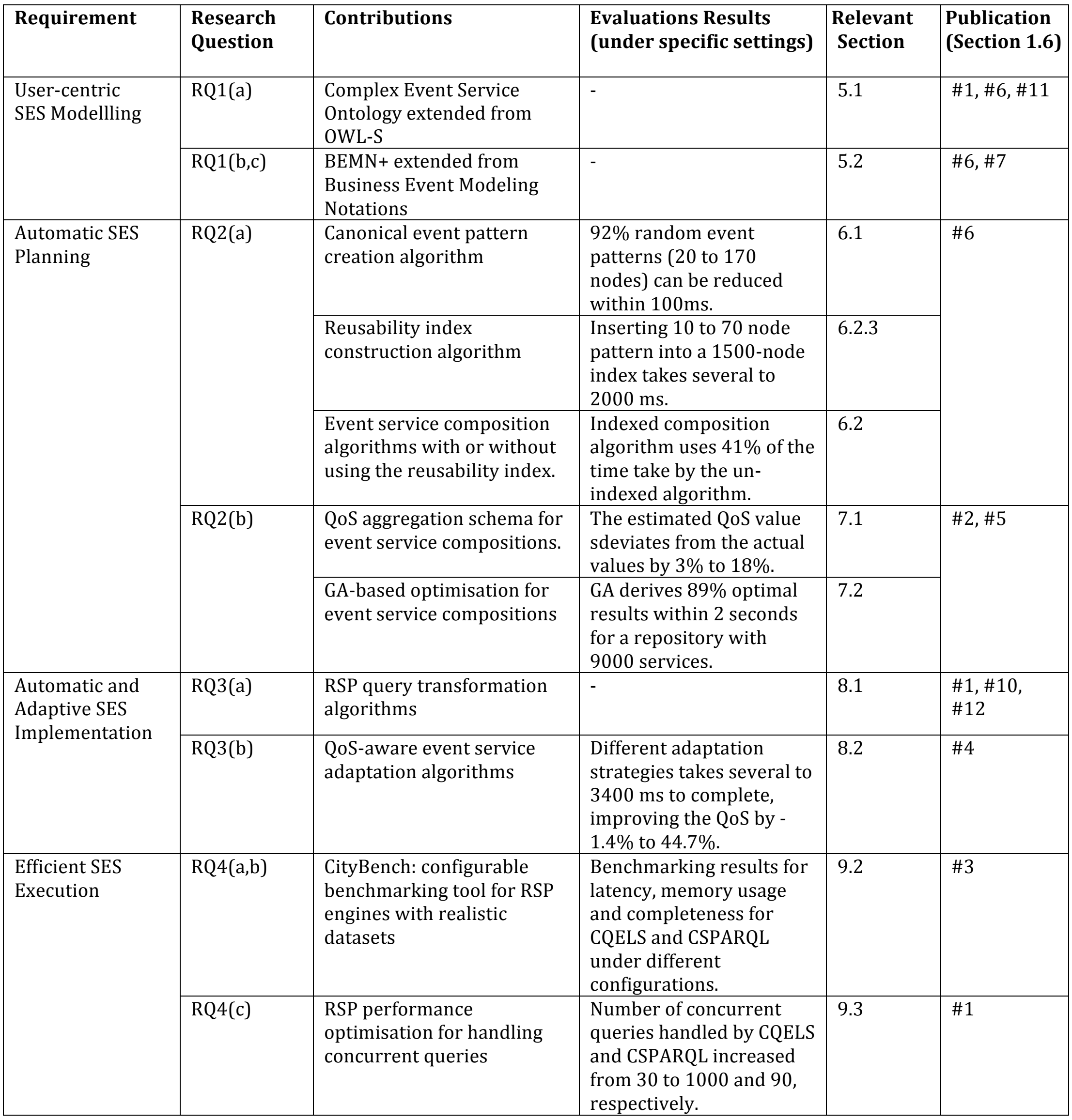
Event Service Implementation and Adaptation. Methods to transform composition plans into RSP queries are provided so that the event service compositions can be deployed and executed automatically. The query transformation relies on the semantics alignment between BEMN$^+$ event operators and the query operators in RSP engines to ensure the correctness of queries created. The semantics alignment and query transformation address *RQ3(a)*. Leveraging the automatic composition plan deployment capability, a QoS-aware event service adaptation is realised, addressing *RQ3(b)*. Unlike existing adaptation mechanisms in CEP systems which adjust the execution plans for the query to improve the query latency, ACEIS allows recomposing the event service over different event streams and can adapt to multiple QoS criteria. Different adaptation strategies are implemented and evaluated. Experiments show a trade-off between adaptation time and QoS improvement when using different adaptation strategies, i.e., they may take several to 3446 milliseconds to recover the system, and could improve the overall QoS of the event service compositions by -1.41% to 44.67% (Table 8.2).

Benchmarking and Optimisation of Event Service Execution. In addition to the above contributions, the usage of the ACEIS in Smart City applications are presented. The practical problem of the query engine performance is analysed and a benchmarking tool for RSP engines are developed. The benchmark addresses *RQ4(a,b)* and is the first attempt (to the best of my knowledge) that evaluates RSP engines with realistic datasets and scenarios in a configurable way. Optimisation techniques based on the benchmarking results are discussed and tested in order to improve the engine performance with regard to concurrent queries. The optimisation leverages multiple engine instances and applies load balancing strategies to address *RQ4(c)*. Experiments show the number of concurrent queries handled stably by CQELS and CSPARQL engines can be increased from less than 30 to 1000 and 90, respectively, when using the optimisation techniques (Figure 9.32 and 9.33). Table 1.3 summarises the contributions of the thesis.

1.6 Thesis Outline

In this chapter, the research in this thesis is introduced, the context and motivation of the research are explained, the research problems to resolve are listed, the overview of the proposed solutions are described, the research methods are presented and the major contributions of the thesis are summarised. The rest of the thesis is organised as follows:

Chapter 2 puts the research problems in the context of smart city applications and explains the motivation of the research within concrete scenarios.



Requirement	Research Question	Contributions	Evaluations Results (under specific settings)	Relevant Section	Publication (Section 1.6)
User-centric SES Modellling	RQ1(a)	Complex Event Service Ontology extended from OWL-S	-	5.1	#1, #6, #11
	RQ1(b,c)	BEMN+ extended from Business Event Modeling Notations	-	5.2	#6, #7
Automatic SES Planning	RQ2(a)	Canonical event pattern creation algorithm	92% random event patterns (20 to 170 nodes) can be reduced within 100ms.	6.1	#6
		Reusability index construction algorithm	Inserting 10 to 70 node pattern into a 1500-node index takes several to 2000 ms.	6.2.3	
		Event service composition algorithms with or without using the reusability index.	Indexed composition algorithm uses 41% of the time take by the un-indexed algorithm.	6.2	
	RQ2(b)	QoS aggregation schema for event service compositions.	The estimated QoS value sdeviates from the actual values by 3% to 18%.	7.1	#2, #5
		GA-based optimisation for event service compositions	GA derives 89% optimal results within 2 seconds for a repository with 9000 services.	7.2	
Automatic and Adaptive SES Implementation	RQ3(a)	RSP query transformation algorithms	-	8.1	#1, #10, #12
	RQ3(b)	QoS-aware event service adaptation algorithms	Different adaptation strategies takes several to 3400 ms to complete, improving the QoS by -1.4% to 44.7%.	8.2	#4
Efficient SES Execution	RQ4(a,b)	CityBench: configurable benchmarking tool for RSP engines with realistic datasets	Benchmarking results for latency, memory usage and completeness for CQELS and CSPARQL under different configurations.	9.2	#3
	RQ4(c)	RSP performance optimisation for handling concurrent queries	Number of concurrent queries handled by CQELS and CSPARQL increased from 30 to 1000 and 90, respectively.	9.3	#1

Table 1.3: Summarised contribution of the thesis.

Chapter 3 introduces the research background of this thesis, which involves Semantic Web, Service Oriented Computing and Complex Event Processing.

Chapter 4 provides an overview of the proposed solutions as an integrated system. It describes the functionalities of the components in the system as well as their interactions.

Chapter 5 presents a Complex Event Service Ontology (CESO) to semantically annotate for CESs. This chapter also presents the graphical event pattern definition language extended from BEMN, called BEMN$^+$. The syntax and formal semantics of BEMN$^+$ are elaborated.

Chapter 6 proposes a complex event service discovery and composition based on complex event patterns annotated with CESO.

Chapter 7 extends the pattern-based event service discovery and composition introduced in the previous chapter to address non-functional constraints and preferences.

Chapter 8 discusses how event composition plans created in the previous chapters are automatically transformed into stream queries and evaluated over event streams. This chapter also provides a means to enable QoS-aware event service adaptation.

Chapter 9 presents the prototype implementation of the system (i.e., ACEIS) in Smart City applications. It introduces a benchmarking tool for evaluating the performance of the RSP engines used in ACEIS. It also presents means for optimising the performance of the RSP engines using load balancing.

Chapter 10 gives the concluding remarks of this thesis and discusses possible future work.

1.7 List of Relevant Publications

In this section, publications in research journals, conferences and workshops relevant to this thesis are listed.

1.7.1 Journal

1. Automatic Discovery and Integration of Urban Data Streams: The ACEIS Middleware, Feng Gao, Muhammad Intizar Ali, Edward Curry, Alessandra Mileo, Future Generation Computer System (FGCS), 2016, (submitted Arp., 2016, under review)
2. QoS-aware Stream Federation and Optimization based on Service Composition, Feng Gao, Muhammad Intizar Ali, Edward Curry, Alessandra Mileo, International Journal on Semantic Web Information Systems (IJSWIS), 2015, (revision submitted Apr., 2016)

1.7.2 Conference

3. QoS-aware Adaptation for Complex Event Service, Feng Gao, Muhammad Intizar Ali, Edward Curry, Alessandra Mileo, Proceedings of the 31st ACM Symposium On Applied Computing, (SAC 16'), 2016.
4. CityBench: A Configurable Benchmark to Evaluate RSP Engines using Smart City Datasets, Muhammad Intizar Ali, Feng Gao and Alessandra Mileo, Proceedings of the 14th International Semantic Web Conference, (ISWC 15'), 2015.

5. QoS-aware Complex Event Service Composition and Optimization using Genetic Algorithms, Feng Gao, Edward Curry, Muhammad Intizar Ali, Sami Bhiri, Alessandra Mileo, Proceedings of the 13th International Conference on Service Oriented Computing (ICSOC 14'), France, 2014 (short paper).

6. Complex Event Service Provision and Composition based on Event Pattern Matchmaking, Feng Gao, Edward Curry, Sami Bhiri, Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS 14'), 2014.

7. User-centric Complex Event Modeling and Implementation Based on Ubiquitous Data Service, Feng Gao, Sami Bhiri, The Sixth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (Ubicomm 12'), 2012 (short paper).

8. Extending BPMN 2.0 with Sensor and Smart Device Business Functions, Feng Gao, Maciej Zaremba, Sami Bhiri, Wassim Derguech, 20th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 11'), 2011.

1.7.3 Workshop and Demo

9. RDF Stream Processing for Smart City Applications, Feng Gao, Muhammad Intizar Ali, Alessandra Mileo, RDF Stream Processing Workshops in ESWC'15, 2015.

10. Semantic Discovery and Integration of Urban Data Streams, Feng Gao, Muhammad Intizar Ali, Alessandra Mileo, Semantics for Smart Cities (SSC), Proceedings of Workshops in International Semantic Web Conference. (ISWC 14' Workshops), 2014.

11. Ubiquitous Service Capability Modeling and Similarity Based Searching, Feng Gao, Wassim Derguech, Web Information Systems Engineering (WISE 12' Workshops), 2012.

12. User centric complex event processing based on service oriented architectures, Feng Gao, Sami Bhiri, Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC 13' Demo), 2013.

1.7.4 Additional

13. CityPulse: Large Scale Data Analytics Framework for Smart Cities, Dan Puiu, Payam Barnaghi, Ralf Tnjes, Daniel Kmper, Muhammad Intizar Ali, Alessandra Mileo, Josiane Xavier Parreira, Marten Fischer, Sefki Kolozali, Nazli Farajidavar, Feng Gao, Thorben Iggena, Thu-Le Pham, Cosmin-Septimiu Nechifor, Daniel Puschmann, Joao Fernandes , in IEEE Access, vol. 4, no. , pp. 1086-1108, 2016.

14. Real-time Data Analytics and Event Detection for IoT-enabled Communication Systems, Muhammad Intizar Ali, Naomi Ono, Mahedi Kaysar, Zia Ush Shamszaman, Thu-Le Pham, Feng Gao, Keith Griffin, Alessandra Mileo, Journal of Web Semantics (JWS), 2016.

15. Observing the Pulse of a City: A Smart City Framework for Realtime Discovery, Federation, and Aggregation of Data Streams, Sefki Kolozali, Maria Bermudez-Edo, Payam Barnaghi, Feng Gao, Muhammad Intizar Ali, Alessandra Mileo, Marten Fischer, Thorben Iggena, Daniel Kuemper, Future Generation Computer System (FGCS), 2016, (submitted Apr., 2016, under review)

16. Optimizing the Performance for Concurrent RDF Stream Processing Queries, Chan Le Van, Feng Gao, Muhammad Intizar Ali, Alessandra Mileo, Proceedings of the 15th International Semantic Web Conference, (ISWC 16'), 2016, (submitted Apr., 2016, under review).

17. AgriPulse: A Semantic Framework for Internet of Things-enabled Smart Farming Applications, Andreas Kamilaris, Feng Gao, Ali Intizar, Francesc X. Prenafeta-Bold, Proceedings of the 15th International Semantic Web Conference, (ISWC 16'), 2016, (submitted Apr., 2016, under review).

Chapter 2

Motivation

In this chapter, the research problems are motivated within the context of Smart City applications. I will first explain at a high-level what is the vision of a smart city. I will then discuss different smart city scenarios and some sample queries. Furthermore, the requirements of smart city applications are analysed and the gaps in the state-of-the-art are discussed.

2.1 Context: Smart City Applications

Building a smart city requires providing both hardware to collect information on the events happening within the urban environment as well as software to utilise the collected information and help decision makings in urban life. The main goal of smart city applications is to enhance the quality of urban services, to reduce costs, and to engage the citizens in a more effective way [69]. The potential domains for smart city applications span from government services, public transport and crisis management, to individual health care, smart home and travel planning. Noticeably, many of these application domains require real-time responses to situations happening in the city, in order to react with minimal delay. Figure 2.1 illustrates some exemplifying applications in smart cities. In the following we discuss three smart city scenarios and provide some concrete usecases (functionalities as queries) in those scenarios. These scenarios are extracted from CityPulse 101 scenarios[1] and the full set of queries are listed in CityBench [70]. The datasets and queries in these scenarios are revisited in details later in Section 9.2.

[1] CityPulse 101 scenarios: http://www.ict-citypulse.eu/scenarios/, last accessed: May, 2015.

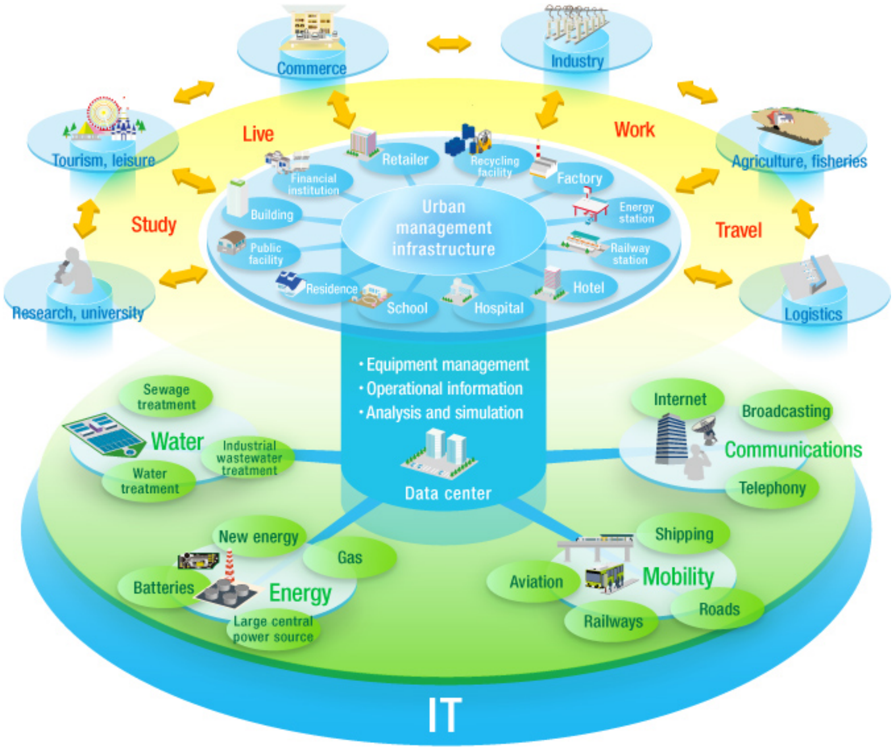


FIGURE 2.1: Overview of smart city applications (from Hitachi: https://community.hds.com/community/innovation-center)

- **Multi-modal Context-aware Travel Planner** (**$S1$**)**:** This application relies on a routing module that offers routes given the start and end travel locations. On top of this module, the application aims at optimising users' travel path based on their preferences on route type, air pollution and travel cost etc. In addition to that, the application continuously monitors factors and events that can impact this optimisation (including traffic, weather, parking availability and so on) to promptly adapt to provide the best up-to-date option. Sample queries:
  Q1: What is the traffic congestion level on each road of my planned journey?

  Q3: What are the average congestion level and estimated travel time to my destination?

- **Parking Space Finder Application** (**$S2$**)**:** This application is designed to facilitate car drivers in finding a parking spot combining parking data streams and predicted parking availability based on historical patterns. Additional sources such as timed no parking zones, congested hot spots and walking time from parking to a point of interest, the user can reduce circulation time and optimise parking management in the city. Sample query:
  Q7: Notify me whenever a parking place near to my destination is full.

- **Smart City Administration Console** (**$S3$**)**:** This application facilitates city administrators by notifying them of the occurrence of specific events of interest.

The dashboard relies on data analytics and visualisation to support early detection of any unexpected situation within the city and takes immediate actions, but it can also be used as a city observatory for analysing trends and behaviours as they happen. Sample query:
Q10: Notify me every 10 minutes, about the most polluted area in the city.

2.2 Basic Requirements of Smart City Applications

The data processing procedure in real-time smart city applications has three phases: data gathering, data analysis and result delivery [2, 71]. A smart city application could be an integration across different application domains, as well as the engagement of different city departments, city-contracted entrepreneurs and individual enterprises providing services. Thus, raw urban data gathered for smart city applications may arrive in different formats, e.g., traffic information, parking spaces, bus timetables etc, as well as from different interfaces, e.g., APIs, websites, web services etc [4]. Due to the data and interface heterogeneity, the data aggregation or abstraction from urban data sources is typically carried out manually, resulting in static or outdated information. A real-time data analysis is needed for urban data in huge volumes [72]. Moreover, the analysis results should be context-aware and knowledge-based to provide insights of the current situations [73]. The vision of the data processing pipeline in smart city applications is illustrated in Figure 2.2.

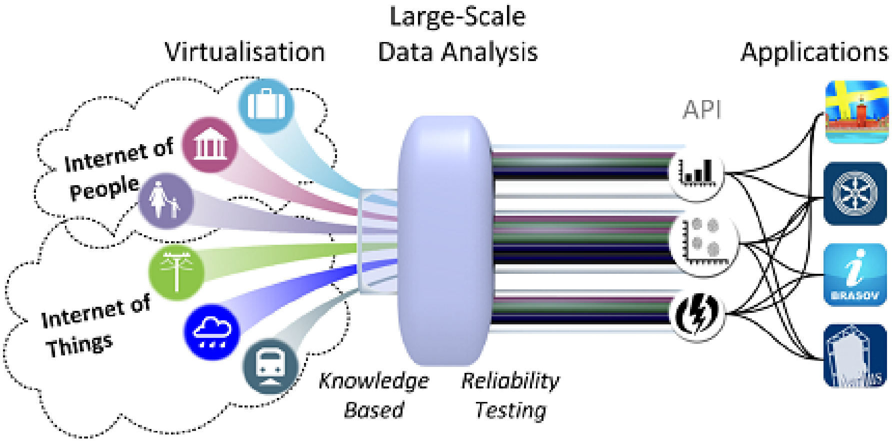


FIGURE 2.2: Data Processing in Smart City applications (from Citypulse project: http://www.ict-citypulse.eu/)

In summary, the basic requirements for smart city applications are 1) federation of heterogeneous data interfaces, 2) real-time data analysis at a large-scale and 3) federation of heterogeneous data formats and semantics. More specifically, the first requirement implies the application may need to interact with *different data interfaces*, the second requirement implies the queries used in the applications are expected to give *real-time results* over *multiple streams* and the third requirement implies that the queried data may have *different formats* and the *use of background knowledge* might be needed. In the following the techniques can be used to fulfil these requirements, namely, Service Oriented Architecture (SOA), Complex Event Processing (CEP) and Semantic Web (SW), are discussed.

Federation of Heterogeneous Data Interface. In order to integrate different data streaming interfaces, i.e., the interface heterogeneity, the SOA paradigm can be used, i.e., to provide data/event streams as services. Using SOA, the metadata of the data/event streams are structurally formatted as a service description according to a service model. The service description will provide information on the functional/non-functional service properties for service discovery/composition, as well as information on the access mechanisms of the services to allow automatic service invocation, making the events reusable across different platform. For example, the traffic data stream will be described as a traffic condition reporting service, with functionalities like observing vehicle speed and count as well as non-functional properties like sampling frequency, latency etc. The access mechanism could be a CKAN query via http request and the result format is a proprietary XML schema.

Real-time Data Analysis at a Large-scale. In order to provide real-time data analysis for multiple streams, techniques like CEP and Data Stream Management System (DSMS) can be considered. The former has a focus on providing complex events with traceable causal events and the latter emphasises on providing real-time results to data processing queries similar to conventional database queries. For example, most queries in Section 2.1 can be easily written in EPL.

Federation of Heterogeneous Data Formats and Semantics. Data heterogeneity needs to be dealt with on two levels: data and metadata. SW techniques can be used to help improve data interoperability on both levels. Data in SW are modelled as graphs with labelled nodes and edges, where nodes represent terms and edges represent relationships between terms. This simple yet powerful formalism can be used to define terms and entities as commonly accepted knowledge, or *ontology.* By annotating data in different formats with terms and relationships from ontologies, a machine can "understand" the meaning of the data and it can correlate it with other data. For example, the metadata of a traffic data stream can be semantically annotated as a Semantic Web Service (SWS) document, with the vehicle speed observation functionality annotated as a sub-class of the term "traffic observation" in an ontology, so that users querying the super-class can find the traffic report service via semantic subsumption. Also, the data transmitted by the traffic service in XML format can be annotated with sensor observation ontologies to describe what the data is about, and a CEP/DSMS engine with semantic reasoning capability can correlate the annotated observation with background knowledge.

In summary, integrating SOA, CEP and SW techniques and implementing the Semantic Event Services (SESs) to handle streaming data can be a starting point for addressing the requirements of Smart City applications. A more detailed discussion on these technologies are presented in Chapter 3.

2.3 Advanced Requirements and Limitations of Existing Approaches

Like conventional services, the life-cycle of SES consists of different phases, which can be categorised into service modelling (including service description and service request definition), service planning and service implementation (including service deployment, execution, and service adaptation), as depicted in Figure 2.3:

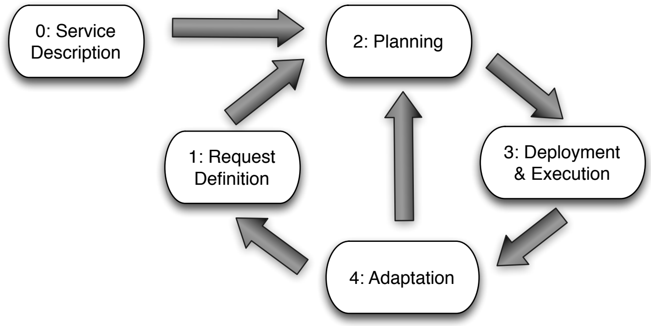


Figure 2.3: Life cycle of event services

0. **Service Description:** the static description on the service metadata is created and stored in the service repository. Describing services and storing the descriptions is a preliminary step for any service requests to be realised by the described services.

1. **Request Definition:** an event service consumer identifies the requirements on the interested complex events (as well as the services that deliver the events) and specify those requirements in an event service request.

2. **Planning:** an agent receives a consumer's request and match it against service descriptions in the service repository. If direct matches are found, the matching

service descriptions are retrieved and the matching process ends. Otherwise, existing event services are composed to fulfil the requirements and composition plans are derived.

3. **Deployment & Execution:** an agent establishes connections between the event service consumer and providers by subscribing to event services (for the consumer) based on a composition plan, then it starts the event detection (if necessary) and messaging process.

4. **Adaptation:** an agent monitors the status of the service execution to find irregular states. When irregular states are detected, the planning activity is invoked to create new composition plans and/or service subscriptions. If the irregular states occur too often, it may suggest that the service request needs to be re-designed.

In the following the requirements for an efficient and effective management of the SES life-cycle is discussed and the relevant gaps between the existing approaches are analysed using examples from the motivation scenarios.

2.3.1 User-Centric SES Modelling

Existing syntactical event service models allow describing the type and location of events for PESs. Listing 2.1 and 2.2 show snippets of traffic data streams modelled in WS-Eventing and Sensor Alert Service (SAS), respectively. Notice that the location information is provided by the SAS schema via *Feature of Interest* and *Operation Area* directly, while in WS-Eventing it is not provided directly but it can be specified using domain specific XML schema. Neither provides native support for NFPs like quality-of-service, but they can be provided via extending the schema, e.g., in [74]. The event type and location information can be annotated using the *modelReference* proposed in SA-WSDL [75].

```
<wsdl:definitions xmlns:wse=”http://schemas.xmlsoap.org/ws/2004/08/eventing”
                  xmlns:wsdl=”http://schemas.xmlsoap.org/wsdl/”
                   xmlns:xs=”http://www.w3.org/2001/XMLSchema” >
 <wsdl:import namespace=”http://schemas.xmlsoap.org/ws/2004/08/eventing”
                         location= ”http://schemas.xmlsoap.org/ws/2004/08/eventing/eventing.wsdl” />
 <wsdl:types>
  <xs:schema targetNamespace=”http://www.example.org/traffic” >
   <xs:element name=”TrafficReport” >
    <xs:complexType>
     <xs:sequence>
      <xs:element name=”VehicleSpeed” type=”xs:string” />
      <xs:element name=”VehicleCount” type=”xs:string” />
      <xs:element name=”Street” type=”xs:string” />
      <xs:element name=”City” type=”xs:string” />
```

```
            <xs:element name=”Country” type=”xs:string” />
            <xs:element name=”Lat” type=”xs:string” />
            <xs:element name=”Long” type=”xs:string” />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:schema>
  </wsdl:types>
  <wsdl:message name=”trafficMsg” >
    <wsdl:part name=”body” element=”tns:TrafficReport” />
  </wsdl:message>
  <wsdl:portType name=”TrafficReportPort” wse:EventSource=”true” >
    <wsdl:operation name=”TrafficOp” >
      <wsdl:output message=”tns:trafficMsg” />
    </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>
```

Listing 2.1: WS-Eventing encoding for traffic reporting service

```
<?xml version=”1.0” encoding=”UTF−8”?>
<Capabilities xmlns=”http://www.opengis.net/sas”
 xmlns:swe=”http://www.opengis.net/swe”
 xmlns:xsi=”http://www.w3.org/2001/XMLSchema−instance”>
 <Contents>
  <AcceptAdvertisements>true</AcceptAdvertisements>
   <SubscriptionOfferingList>
    <SubscriptionOffering>
     <SubscriptionOfferingID>1</SubscriptionOfferingID>
     <AlertMessageStructure>
      <QuantityProperty>
       <Content definition=”city:VehicleSpeed” uom=”km/h” min=”0” max=”300”/>
      </QuantityProperty>
     </AlertMessageStructure>
     <FeatureOfInterest>
      <Name>Street 1</Name>
      <Description>The vehicle speed applies to the street segment.</Description>
     </FeatureOfInterest>
     <OperationArea>
      <swe:GeoLocation>
       <swe:longitude>
        <swe:Quantity>56.104156</swe:Quantity>
       </swe:longitude>
       <swe:latitude>
        <swe:Quantity>10.233607</swe:Quantity>
       </swe:latitude>
      </swe:GeoLocation>
     </OperationArea>
     <AlertFrequency>0.5</AlertFrequency>
    </SubscriptionOffering> ...
   </SubscriptionOfferingList>
  </Contents>
</Capabilities>
```

Listing 2.2: SAS encoding for traffic reporting service

However, existing service models are not suitable for describing CESs. Consider a variant of *Q1* (i.e., *Q1'*) gives the estimated travel time of a route R_1, where $R_1 = \{seg_1, seg_2, ...\}$ consists of a set of street segments. A CES can be implemented by evaluating *Q1'* continuously and send the results to subscribers. A service request for estimated travel time on R_2 cannot find the service evaluating *Q1'* by only comparing message types or sensor locations: the semantic subsumption or equivalence relation of the provided and requested events must be examined. It is not possible to specify a complex event pattern within these service models, nor is it possible to specify patterns in existing SWS standards like SAWSDL, OWL-S or WSMO. In SemSOS [76], semantic rules can be used to assert new facts based on sensor readings. However, various ways that the rules can be specified within the semantic service description is not discussed in [76]. Also, [76] does not describe how temporal/logical rules can be specified for events. In summary, existing service models do not provide adequate conceptualisation for event patterns.

```
SELECT  AVG(congest) AS avgCong, SUM(travelTime) AS sumTime
FROM TrafficReportOnSeg1 T1, TrafficReportOnSeg2 T2, TrafficReportOnSeg3 T3
RETAIN 10 seconds
GROUP BY T1.rid, T2.rid, T3.rid
```

Listing 2.3: Q3 in EPL

Various CEP engines use pattern definition languages (e.g., in [77–81]) to describe event patterns in different syntaxes. For example, Event Processing Language (EPL) used in Oracle's Esper[2] CEP engine provides a list of operators to describe correlations of events in an SQL-like syntax. Using EPL, *Q3* can be modelled as shown in Listing 2.3. However, if a user wants to specify a non-functional requirement for the query results, e.g., the latency of the results should be less than 500 milliseconds and the accuracy should be more than 90%, he/she cannot define these requirements with EPL, or other pattern definition languages. Also, preferences on NFPs, e.g., a user is more concerned with the accuracy than the latency, cannot be described. Moreover, most pattern definition languages require some knowledge in relational database systems or logic programs, since they use an SQL-like or rule-based syntax. When complex events are used to trigger a business process or a workflow, it may increase the learning overhead for the business/non-technical users who define those process models in standardised business process modelling languages, e.g., Business Process Model Notation[3] (BPMN). Therefore, it is desirable to have graphical notations for event patterns which are compatible with process modelling languages.

[2] Esper engine: http://www.espertech.com/products/esper.php, last accessed: May, 2015.

[3] BPMN: http://www.bpmn.org/, last accessed: May, 2015.

2.3.2 Automatic and Customised SES Planning

Service planning determines which service(s) can be used to fulfil a service request and in the case of using multiple services, how to integrate them in a service *composition plan*. Composition plans generated as the outcome of service planning should fulfil both functional and non-function requirements specified in service requests. In the following the limitations of current service planning approaches for addressing functional and non-functional requirements of event service requests are discussed.

2.3.2.1 Event Service Planning based on Functional Aspects

The services used in composition plans can be composed services themselves. For example, to find the relevant services to implement *Q3*, one can look for the sensors deployed on the user-selected route by examining the location information specified in the traffic report service descriptions and comparing it with the route information specified in the service request. Eventually three services will be retrieved to create streams for *TrafficReportOnSeg1, TrafficReportOnSeg2* and *TrafficReportOnSeg3* in Listing 2.3. However, assume there are two CESs, one calculating the average congestion level and the other the sum of the travel time on the three street segments, respectively. It is easy to see that joining these two results streams could also implement *Q3*, thus, these two CESs can be integrated as a valid composition plan. However, simply comparing the geographical location would not be sufficient for identifying these two CESs to use them in a composition plan.

Facilitating automatic service planning is an important contribution of introducing semantics to service descriptions [25]. State-of-the-art SWS planning and composition approaches model service tasks as a tuple $ST = (I, O, P, E)$, where I, O, P, E are the input, output parameters, preconditions and effects, respectively. For example, a hotel booking service may be modelled as in Listing 2.4. In this IOPE SWS modelling paradigm, predicates are used to define preconditions and effects and rule-based reasoning can be used to find possible composition plans that provides all inputs (using the intermediate outputs generated from the plan) for the target task while fulfilling all preconditions (by applying intermediate effects). Typically, the reasoning procedure is carried out in a backward chaining style, i.e., starting from the target objective or effect, find possible tasks that fulfil part of the required preconditions. An exemplifying research of this kind can be found in [61].

However, such IOPE-based service planning cannot be easily applied to CESs because it is not straightforward to define the precondition and effect of an event detection task.

```
HotelBooking = (
I = {customer, date, nights, creditCard},
O = {bookingReceipt},
P = {hasVacancy(date,nights), validCard(creditCard)},
E = {paymentMade, roomReserved} ).
```

Listing 2.4: A hotel booking service modelled in IOPE paradigm

The precondition of a complex event may be described by incorporating temporal logics and specifying comprehensive rules for temporal reasoning, as in [82]. However, event detection and data analysis process do not have any effects on the real-world except for the complex events derived as information entities. Even if the creation of the complex event types is described as effects, e.g., suppose the service composition for *Q3* has an effect of creating the event with type *AvgCongestionAndSumTravelTime*, it is not practical to use the type information for matching effects and preconditions because the type information alone does not identify the semantics of a complex event. Therefore, a pattern-based event service discovery and composition is necessary for automatic event service planning.

2.3.2.2 Event Service Planning based on Non-functional Aspects

While pattern-based event service planning addresses the functional requirements of service consumers, their non-functional requirements should also be addressed. In this thesis, only non-functional requirements regarding the quality-of-service parameters (e.g., latency, accuracy, completeness etc., detailed list of QoS parameters discussed in Chapter 7) are investigated. Consider the example of composing for *Q3*, a valid composition plan using 3 PESs is functionally equivalent to another plan using 2 CESs. However, the latency, completeness and cost etc., of the two plans may be different, and a user should be able to find composition plans that best suit his/her individual needs. QoS-aware service composition has two major challenges: 1) defining an appropriate QoS aggregation schema to calculate the QoS for the composition plans and 2) finding QoS-optimised composition plans efficiently.

QoS aggregation. Event service planning based on QoS parameters also needs to address these two challenges. A QoS aggregation schema contains a set of QoS parameters to be considered, a set of QoS aggregation rules on the parameters and a utility/cost function to calculate and compare different composition plans based on the aggregated QoS parameters. QoS aggregation schema has been discussed extensively, e.g., in [83–85]. Existing works have covered a broad range of QoS parameters and many have used a utility function based on *Simple Additive Weighting* (SAW [86]) to calculate the performance based on users' preferences, which are modelled as numerical weights. However, the aggregation rules in the existing work focus on conventional web services rather than CESs, which has a different QoS aggregation schema. For example, the event engine also has an impact on the QoS aggregation, which is not considered in conventional QoS aggregation. Also, the aggregation rules for some QoS properties based on event composition patterns is different to those based on workflow patterns (as in [84]).

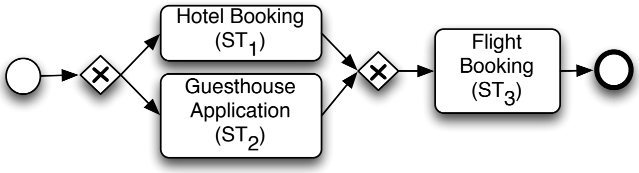


FIGURE 2.4: A travel planning service workflow.

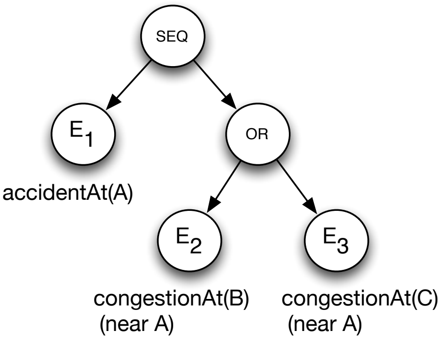


FIGURE 2.5: A severe traffic accident is detected if an accident happens and it causes congestion in the nearby regions.

Consider a simple service workflow for travel planning as shown in Figure 2.4 and a severe traffic accident event pattern in Figure 2.5, the energy cost of the sub-workflow of the travel planning process that correlates ST_1 and ST_2 (denoted C_1) is given by $C_1 = cost(ST_1)|cost(ST_2)$, where $cost(ST_i)$ gives the cost of the service task ST_i, i.e., the value of C_1 depends on which task is executed at run-time. However, the cost of the sub-pattern of the traffic accident (denoted C_2) is given by $C_2 = cost(E_1) + cost(E_2)$, because the energy consumption of event detection tasks start when subscriptions are made, and the cost exists even when no events are detected, the same applies for other computational resources. Also, the latency of the workflow (denoted L_1) is given by $L_1 = (L(ST_1) + L(ST_3))|(L(ST_2) + L(ST_3))$, because the service tasks are executed in sequence. However, the event pattern does not describe the execution plan of the event detection task, i.e., the latency will be the last event instance detected and picked to complete the pattern detection task at run-time. The latency and cost are just two examples of the differences of QoS aggregation rules between service workflows and event patterns. To the best of my knowledge currently there are no suitable QoS aggregation schema for CESs.

Composition plan creation and optimisation. The composition plans must be created and optimised efficiently to enable on-demand service planning. To solve this problem, numerous heuristic algorithms are proposed. There are two prominent strands: Integer Programming (IP) (e.g., [83, 87, 88]) and Genetic Algorithm (GA) (e.g., [89–92]) based solutions. In [87] the limitation of local optimisation and the necessity of global planning are elaborated. The authors explained why brute force enumeration and comparison are not realistic to achieve global optimisation of service composition. The authors propose to address this problem by introducing an IP-based solution with an SAW based utility function to determine the desirability of an execution plan. This approach is extended in [88] with more heuristics to promote efficiency. In [83] a hybrid approach of local and global optimisation is proposed, in which global constraints are delegated to local tasks, and the constraint delegation is modelled as an IP-based optimisation problem. The problem with IP-based solutions in general is that they require the QoS metrics to be linear, and they do not address the service re-planning problem.

Genetic Algorithm (GA) based solutions are therefore proposed to address these issues, e.g., [85, 89–92]. However, these GA-based approaches can only cater for IOPE based service compositions. CES composition is pattern-based, as explained earlier. Therefore, novel algorithms based on genetic evolution, including suitable genetic encoding for event patterns, crossover and mutation operations, must be developed.

2.3.3 Automatic and Adaptive SES Implementation

Service composition plans are abstract documents which give instructions on integrating different services. In order to implement the service composition, the composition plans must be implemented as concrete and executable programs. An automatic implementation of composition plans is desirable to reduce the human intervention in the service life cycle and to cope with the volatile business and physical environments. Also, it is desirable that the service implementation can be adaptive to QoS constraint violations at run-time. In the following the limitations of current approaches with regard to automatic and adaptive event service implementation are discussed.

2.3.3.1 Automatic Event Service Implementation

Conventional web service composition can be implemented by various workflow/process engines, e.g., YAWL [93], BPEL[4], BPMN 2.0[5] etc. Implementing a conventional web service composition requires enabling the control and data flow in the composition plan. Implementing the control flow is trivial, since the composition plans are imperative commands. The data flow can be sometimes problematic, because there can be semantic or syntactical mismatches between service inputs and outputs. In such cases, service mediation might be needed [94]. Semantic web service composition also provides solutions

[4] WS-BPEL, version 2.0: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, last retrieved: Mar., 2015.

[5] Business Process Modeling Notation Specification, version 2.0: http://www.omg.org/spec/BPMN/2.0/, last accessed: Mar., 2015.

for mismatches in the messages used in service compositions, e.g., in BPEL4SWS [95], the message *lifting* and *lowering* mechanisms supported in SAWSDL [96] are used to transform XML data representations to RDF graphs and vice versa.

Implementing CES composition is different to conventional service composition. Composition plans for CESs are descriptive event/stream queries rather than imperative workflows, the actual workflow of query operators, i.e., query execution plans, are determined at run-time by event/stream processing engines. The mismatches in the event messages are resolved in semantic event services, since the messages are provided as RDF graphs directly. However, the event/stream queries must be specified manually. Also, the queries are tightly coupled to the implementation of the engine, i.e., current composition plans and queries for event services are platform-dependant. Different engines may support different sets of operators, and the semantics of the query pattern might be slightly different. For example, *Q3* can be transformed into ETALIS [19], CQELS [37] and CSPARQL [38] queries while the event query in Figure 2.5 can only be implemented in ETALIS and CSPARQL, because the sequence operator is not supported in CQELS. Also, there is a small difference in the syntax and semantics in *Q3* while it is implemented in CQELS and CSPARQL, i.e., the grammar of the query definition is different, also in CQELS results are produced whenever a new triple arrives, while in CSPARQL, the results are accumulated during a time interval and delivered to result listeners periodically. Currently, there are no standardised event/stream query languages, and because of the differences in the supported operators, language syntax and semantics, an automatic CES implementation requires a query transformation technique to deploy a federated CES query over different event/stream engines.

2.3.3.2 QoS-aware Event Service Adaptation

QoS-aware service adaptation is a desired feature in Smart City applications. In most of the existing solutions matchmaking between the requirements expressed by event consumers and available event providers is carried out at design-time and remain unchanged during the life span of the query deployed in the application. This approach is often far from optimal and its deficiencies become even more obvious in smart city scenarios due to their inherently dynamic stream properties. For example, the accuracy of sensor readings may vary depending on the network availability, weather conditions, energy reserved in battery etc. Therefore, providing the mechanism to detect QoS constraint violations and recover from those violations will significantly improve the reliability of smart city applications.

QoS-aware support in existing event-based systems is typically provided on the data level or the routing network level. On the data level, the QoS-aware optimisation is done via query rewriting, e.g., in [37, 57, 97]. Query rewriting techniques define a cost model of event processing for query operators and use the cost as a heuristics in determining which operator is to be executed at run-time. A frequently used cost is the number of intermediate results produced by the operator. For example, in Figure 2.6, two query execution plans for a variant of *Q7* produce the same results but one is preferable than the other because less intermediate results are produced. On the network level, the cost of an event subscription is calculated based on the link analysis of the hops used in a route, e.g., in [52–55]. When a broker node in the network is overloaded or the quality of a link has decreased significantly, the routing algorithm will dynamically create new routes for the subscription to ensure the quality of the event network.

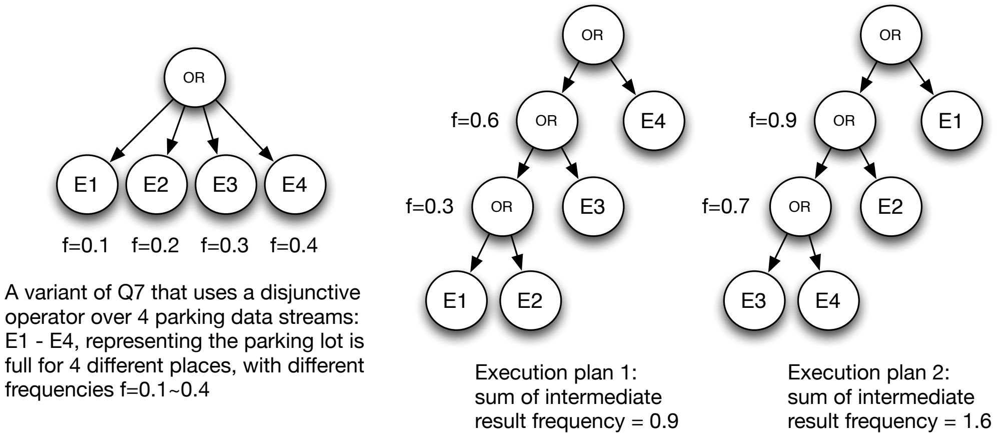

OR
E1
E2
E3
E4
f=0.1 f=0.2 f=0.3 f=0.4
A variant of Q7 that uses a disjunctive operator over 4 parking data streams: E1 - E4, representing the parking lot is full for 4 different places, with different frequencies f=0.1~0.4
OR
f=0.6
OR
E4
f=0.3
OR
E3
E1
E2
Execution plan 1: sum of intermediate result frequency = 0.9
OR
f=0.9
OR
E1
f=0.7
OR
E2
E3
E4
Execution plan 2: sum of intermediate result frequency = 1.6

FIGURE 2.6: Query rewriting example for variant of Q7: reducing intermediate results to optimise query performance.

However, QoS support is not provided on the service level for CESs, i.e., replacing the erroneous or low quality event services or partial service compositions with better candidates. For example, if a user defines a NFP constraint that the accuracy of *Q3* must be greater than 90%, and a composition plan CP_1 consisting of three primitive event services PES_1, PES_2, PES_3 is deployed, in which the accuracy of the PESs are $Acc(PES_1) = 98\%, Acc(PES_2) = 95\%, Acc(PES_3) = 97\%$, respectively. Then, the user's constraint holds within CP_1, assuming the aggregated accuracy is simply the product of the event services in the composition plan. However, if $Acc(PES_3)$ drops to 80% at run-time, the constraint no longer holds within CP_1 and a replacement for PES_3 (or other parts of the composition plan) is needed to maintain the QoS performance. Service adaptation is an active area of research, e.g., in [98–101]. However, these approaches consider only conventional web services and rely on type-based or IOPE based service discovery and composition techniques, which are not suitable for CESs.

2.3.4 Efficient SES Execution

When an SES composition plan is implemented as a semantic event query and executed by an RDF Stream Processing (RSP) engine, it is crucial that the RSP engine can handle the queries efficiently, i.e., with minimal query result latency and memory consumption. Existing RSP engines have provided different techniques for optimising the query performance, e.g., in CQELS [37] and C-SPARQL [38], and since RSP is still in its early stages, there are a lot of room for future improvements. In this thesis, optimising the internal processing mechanisms of various RSP engines is out of scope. However, since ACEIS aims to support different existing RSP engines, it is important for a developer that intends to use or extend ACEIS to understand the different processing capabilities of RSP engines, so that he/she can make the better choice in the type of RSP engine to be used. Benchmarking systems have been used in conventional database systems to investigate the performance issues regarding query processing, data storage and other data management operations [102, 103]. Similarly, LS Bench [39] and SR Bench [40] have provided benchmarks on RSP engines. However, both LS and SR bench use synthetic datasets and cannot show exactly how the engines perform under real-world scenarios and datasets. Moreover, they use static configurations for the testbeds and do not allow flexible configurations over query complexity, stream rate, query concurrency or background knowledge size etc. To have a better understanding of what may happen in real-world scenarios, it is important to have control over these parameters and test the performance with the configuration that best describes the user's actual problem domain. For example, for $Q3$ the number of streams involved depends on the specific route provided by the user, i.e., longer routes may need to integrate more streams, resulting in more complicated queries. Also, the size of the background knowledge used in $Q3$ could be different for different cities.

The performance of composition plan execution is not only impacted by the choice of the RSP engine type. In cases where multiple engine instances are deployed in parallel, the distribution of workload also affects the query performance, i.e., the specific engine instance to evaluate the query needs to be decided carefully. Load balancing techniques are discussed in the literature extensively to address this problem [41–43, 64, 65]. It is necessary to implement and evaluate different load balancing strategies for ACEIS to determine the optimal strategy.

2.4 Summary and Discussion

In this chapter, the research of this thesis is motivated with concrete Smart City scenarios. The concept of Smart City is introduced and three examples of Smart City applications are elaborated. Sample queries from three different application scenarios are introduced. The features of these queries are analysed to justify the need for using Semantic Event Services (SESs) in the application scenarios as an integration of Service Oriented Architecture (SOA), Complex Event Processing (CEP) and Semantic Web (SW) technologies. The necessity of providing event processing capability as semantic event services for Smart City applications are explained by analysing the requirements of the queries used in smart city applications. Moreover, the advanced requirements of managing semantic event service life-cycle are analysed and gaps in the state-of-the-art are discussed with examples, emphasising the need for a user-centric SES modelling, automatic and customised SES planning, automatic and adaptive SES planning, as well as an efficient SES execution. In the next chapter, previous developments of the basic building blocks for SES, i.e., SOA, CEP and SW, is presented as the background for this thesis.

Chapter 3

Background

In this chapter, the basic concepts and methodology related to this thesis are introduced. In particular, it describes the relevant terminology and paints in broad strokes the landscape in Semantic Web (in Section 3.1), Service Oriented Computing (in Section 3.2) and Complex Event Processing (in Section 3.3).

3.1 Semantic Web

It is quite fascinating to see how the World Wide Web (WWW), which is an invention of Sir Tim Berners-Lee only 26 years ago, has changed the lifestyle of people in such a profound way. Statistics[1] show that, since the first ever website[2] was launched, today there are more than 9.38×10^9 websites, and the number of Internet users has reached to more than 3 billion, about 40% of the total population on Earth. In 2001, Sir Tim Berners-Lee described his vision on the next generation of WWW, called the Semantic Web [104]. The Semantic Web (SW) is about networking **knowledge**, instead of documents, as in the traditional WWW. Just like using hypertext documents to abstract from physical and network layers of the internet, the Semantic Web use machine-accessible knowledge to abstract from web documents and applications. Various Semantic Web techniques are used in this thesis. In the following, the basic concepts in Semantic Web are introduced, the basic principle of publishing linked data is presented and the recent application of knowledge representation for sensor devices are discussed.

[1] http://www.internetlivestats.com, last accessed: Apr., 2015.

[2] http://info.cern.ch, last accessed: Apr., 2015.

3.1.1 Basic Concepts and Standards in Semantic Web

In Semantic Web, knowledge is encoded as ontologies in the form of interlinking graphs, where nodes represent ontological terms and concepts and edges represent relations. An ontology is a knowledge base implemented as a graph database. From a technical point of view, a variety of Semantic Web techniques and standards are designed to model, represent and query the information provided by those graph databases. Among these techniques, three technical standards are the cornerstones of the Semantic Web: Resource Description Framework[3] (RDF), Web Ontology Language[4] (OWL) and Simple Protocol and RDF Query Language[5] (SPARQL).

- **Resource Description Framework** specifies the *data model* for the Semantic Web. RDF describes the information on the Web using triples consisting of subjects, predicates and objects. Figure 3.1 shows a graphical representation of a triple, where the subject and object are represented as ovals and the predicate as a directed link. An RDF triple states that the relation represented by the predicate exists between the subject and object, hence it is also known as an RDF statement. A set of RDF triples constitutes an RDF graph. Resources in RDF graphs are identifiable by Internationalised Resource Identifiers (IRIs), which are Unicode strings. Literals in RDF graphs are used to describe concrete data values, which have data types, e.g., integers, strings and float values. Apart from IRIs and Literals, blank nodes are also used in RDF graphs to indicate that something exists, without identifying specific resources. Given a set of IRIs I, blank nodes B and literals L, a valid RDF triple t represents a 3-ary relation: $t \in ((I \cup B) \times I \times (I \cup B \cup L))$.

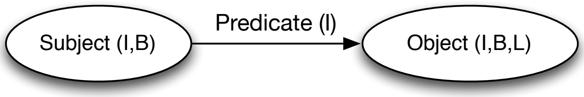


FIGURE 3.1: Visual representation of the RDF triple format

RDF Schema[6] (RDFS) provides an extension to basic RDF vocabulary, which allows defining taxonomies (i.e., class or property hierarchies), property domains and ranges, as well as containers etc. Semantics are provided to RDF and RDFS so that a machine can not only interpret the ontology specified using the vocabularies but also infer or entail implicit RDF statements from explicitly specified

[3] RDF 1.1 concepts and abstract syntax: http://www.w3.org/TR/rdf11-concepts/, last accessed: Apr., 2015.

[4] OWL language overview: http://www.w3.org/TR/owl-features/, last accessed: Apr., 2015.

[5] SPARQL 1.1 standard: http://www.w3.org/TR/sparql11-overview/, last accessed: Apr., 2015.

[6] RDFS 1.1: http://www.w3.org/TR/rdf-schema/, last accessed: Apr., 2015.

ones. Figure 3.2 shows an example of RDFS inferencing based on class types and `subClassOf` relations.

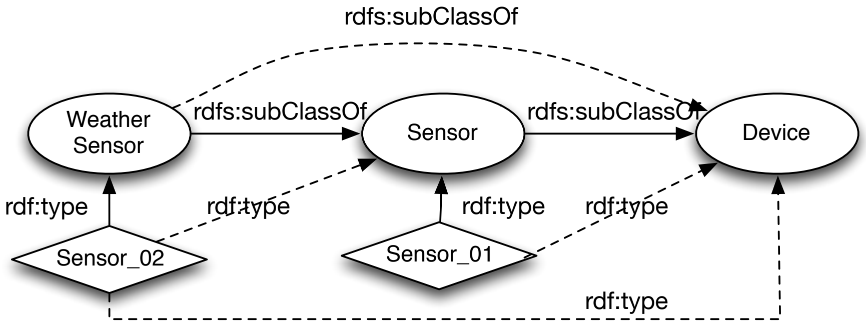


FIGURE 3.2: An example of RDFS inference: dashed links represent inferred relations.

- **Web Ontology Language** is designed to be the *knowledge representation* language for the Semantic Web. It provides additional vocabulary and a formal semantics to RDF and RDF Schema. In doing so, OWL offers higher expressiveness and better machine interoperability when used to describe ontologies. OWL contains three sub-languages: OWL Lite, OWL DL and OWL Full. As the names suggest, OWL Lite has the lowest expressiveness and OWL Full has the highest.

- **Simple Protocol and RDF Query Language** provides the *language* and *protocol* for querying RDF graphs. It uses a syntax similar to SQL. Listing 3.1 provides a basic SPARQL query example over the graph shown in Figure 3.2. The execution results of this query depends on what kind of inferencing and reasoning support is provided by the SPARQL engine implementation. A SPARQL query engine with no reasoning support will only give `Sensor_01` as the result, while a query engine with RDFS level reasoning support will provide both `Sensor_01` and `Sensor_02` as results.

```
Select ?sensorId  Where {
    ?sensorType rdfs:subClassOf :Device.
    ?sensorId rdt:type ?sensorType.
    }
```

LISTING 3.1: A basic SPARQL query example

3.1.2 Linked Data

Linked Data [18] is a set of best practices for publishing and interconnecting structured data on the Web. Linked Data provides explicit links between data from diverse domains such as social networks, organisational structures, government data, statistical data and many others. The ultimate benefit of following the Linked Data paradigm is the increased machine-readability of published and interconnected data. Linked Data is published using RDF where URIs are the means for connecting and referring between various entities on the Web. Tim Berners-Lee defines the following steps [18] for publishing Linked Data:

- **Assign URIs to the entities.** Published entities should have their URIs which map over HTTP protocol to their RDF representation. For example, each sensor should have a unique URI, which links to its information in RDF.
- **Set RDF links to other entities on the Web.** Published entities should be linked with other entities on the Web. For example, when providing the list of sensor functionalities, they should link to the URIs which describe the details of them in RDF.
- **Provide metadata about published data.** Published data should be described by the means of metadata to increase its usefulness for data consumers. Data should contain information on its creator, creation date and creation method. Publishers should also provide alternative means for accessing their data.

Over the last years, an increasing adoption of Linked Data principles and an explosion of datasets specified in RDF can be observed. Early adopters included mainly academic researchers and developers. However, more recently there is a considerable interest from organisations in publishing their data in RDF. Some of the most prominent examples include BBC music data[7], British government data[8] or Library of Congress data[9]. At the same time, an increasing number of public vocabularies (ontologies) and their interconnectedness are created, which forms a Linked Data Cloud[10].

3.1.3 Semantic Web and Sensor Networks

A number of modelling methods for formally representing sensors and sensor networks using Semantic Web technologies have been proposed, such as [105] and [106]. OntoSensor [105] describes an ontology-based description model for sensors based on the SensorML [107] which provides self-descriptive metadata for the transducer elements, however, sensor observation and measurement data is not modelled. The work in [106] presents the SensorData Ontology which is based on Observations & Measurements and

[7] British Broadcasting Company: http://www.bbc.co.uk, last accessed: Apr., 2015.
[8] British government: http://data.gov.uk, last accessed: Apr., 2015.
[9] Library of Congress: http://id.loc.gov, last accessed: Apr., 2015.
[10] LoD cloud: http://lod-cloud.net, last accessed: May, 2015.

SensorML specifications defined by the OGC Sensor Web Enablement (SWE) [107]. The ontology captures the semantic relationships and operational constraints of heterogeneous sensor data.

The W3C Semantic Sensor Networks Incubator Group (SSN-XG)[11] has developed a semantic description framework for Semantic Sensor Networks (SSN)[12] based on the concepts of systems, processes, and observations. The goal of SSN is to begin the formal process of producing ontologies that define the capabilities of sensors and sensor networks, and to develop semantic annotations for service-based sensor networks. Sensor ontology is still in an early stage, but it has taken current standards (e.g SWE, SensorML [107]) into the account and strives to model sensor-based systems with all relevant information.

3.2 Service Oriented Computing

Service Orientation is a set of design principles for computer software development. Analogously to Object Orientation (OO) which uses "Object" as the fundamental element in programming and enables the inheritance, polymorphism and encapsulation, Service Oriented Computing (SOC) uses "Service" as the basic building block for developing distributed programs in a platform-independent and collaborative way [108]. To facilitate SOC implementation and benefit from its features, a basic software design and development style is proposed as the Service Oriented Architecture (SOA).

3.2.1 Service Roles and Activities in SOA

In traditional SOA, there are three major roles: service consumers who utilise the services, service providers who provide the web services and service registries which store service descriptions for web services and conducts matchmaking between service requests and service descriptions. When such matchmakings are made by a service registry, a service consumer can interact with the web services by exchanging messages encoded in agreed formats, e.g., SOAP[13] messages. Service computing is adopted by many organisations and developers because it decouples service consumers from providers, enabling a platform-independent approach for parallel and distributed computing, thus greatly improves the reuse of software applications [11]. Figure 3.3 illustrates the concept of SOA.

[11] SSN-XG: http://www.w3.org/2005/Incubator/ssn/, last accessed: Aug., 2015.

[12] Semantic Sensor Networks Ontology: http://www.w3.org/2005/Incubator/ssn/ssnx/ssn, last accessed: Mar., 2015.

[13] Simple Object Access Protocol: http://www.w3.org/TR/soap/, last accessed: May, 2015.

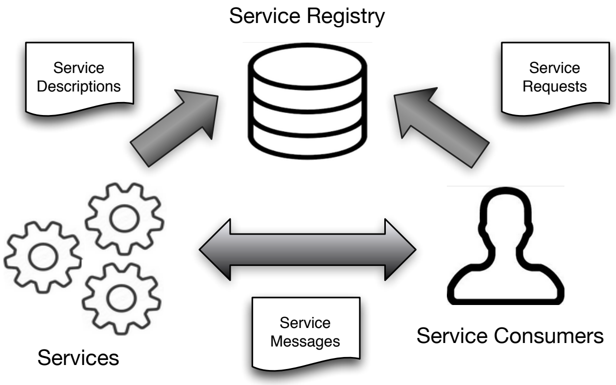


Figure 3.3: Service Oriented Architecture

3.2.2 Web Service and Service Description

Various standardisation organisations have provided definitions for services. The World Wide Web Consortium (W3C[14]) defines a (web) service based on the Web Service Description Language (WSDL) as

> *"... a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialisation in conjunction with other Web-related standards"*

In the SOA Reference Model[15] proposed by the Organisation for the Advancement of Structured Information Standards (OASIS) , a more general definition is provided, in which a service is defined as

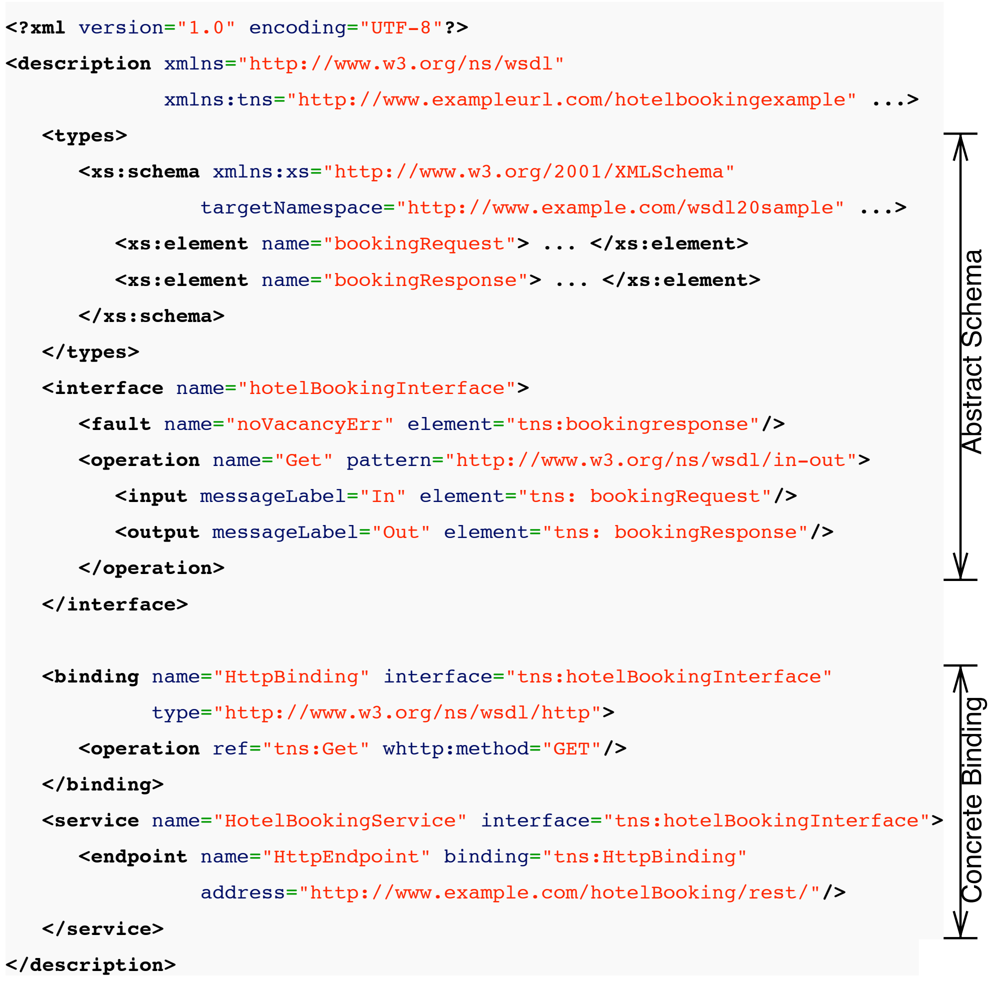
> *"... a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description. "*

In this thesis, the more general definitions on services (as in the OASIS reference model) is adopted, since the services discussed in this thesis are not restricted to the protocols and formats specified in the W3C web service definition.

[14] W3C Web Service Glossary: http://www.w3.org/TR/ws-gloss/#webservice, last accessed: Mar., 2015.

[15] SOA-RM: http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf, last accessed: Mar., 2015.

It is evident that, in the above definitions for services, a key element is the *Service Description*. A service description is a structured documentation for the capabilities offered by a service and it serves as a basis for enabling machine-to-machine interoperability in SOC. Each atomic capability of the service is modelled as a *Service Operation* in the description. Service Description is used for two major objectives: 1) it serves as an advertisement for a service in the service registry, so that service consumers can find the relevant service capabilities they need and 2) it is used as guidelines for developers to build software and communicate with the services provided. Hence, it contains an abstract part describing the types and structure of service operations and the input/output messages of the operations, as well as a concrete part describing the messaging protocols and access locations (URLs) of the service endpoints. Figure 3.4 illustrates the structure of a service description in WSDL.



```
<?xml version="1.0" encoding="UTF-8"?>
<description xmlns="http://www.w3.org/ns/wsdl"
             xmlns:tns="http://www.exampleurl.com/hotelbookingexample" ...>
   <types>
      <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
                 targetNamespace="http://www.example.com/wsdl20sample" ...>
         <xs:element name="bookingRequest"> ... </xs:element>
         <xs:element name="bookingResponse"> ... </xs:element>
      </xs:schema>
   </types>
   <interface name="hotelBookingInterface">
      <fault name="noVacancyErr" element="tns:bookingresponse"/>
      <operation name="Get" pattern="http://www.w3.org/ns/wsdl/in-out">
         <input messageLabel="In" element="tns: bookingRequest"/>
         <output messageLabel="Out" element="tns: bookingResponse"/>
      </operation>
   </interface>

   <binding name="HttpBinding" interface="tns:hotelBookingInterface"
            type="http://www.w3.org/ns/wsdl/http">
      <operation ref="tns:Get" whttp:method="GET"/>
   </binding>
   <service name="HotelBookingService" interface="tns:hotelBookingInterface">
      <endpoint name="HttpEndpoint" binding="tns:HttpBinding"
                address="http://www.example.com/hotelBooking/rest/"/>
   </service>
</description>
```

Figure 3.4: Example of WSDL service description for hotel booking

3.2.3 Service Invocation, Orchestration and Choreography

When a requested service is discovered and bound for a specific task, Service Operations can be invoked to interact with the application or service deployed for the service consumer. The message exchange between service operations can be *synchronous* (e.g., request-response operation in WSDL 1.1[16] and In-Out message exchange pattern (MEP) in WSDL 2.0[17]) or asynchronous (e.g., notification operation in WSDL 1.1 and Out-Only MEP in WSDL 2.0). Asynchronous messaging is ideal for publishing and subscribing events, recall in Chapter 1, a Complex Event Service (CES) is also defined as an asynchronous notification service.

Completing a service consumer's task often requires multiple services (and hence the underlying service operations) to interact with each other in a collaborative way. These services thus formulate a service *composition*. The coordination of the composed services is typically modelled as a workflow. Service workflows can be formally specified using various techniques, e.g., Pertri Net [109], Event Process Chain (EPC) [110], Yet Another Workflow Language (YAWL) [93] and Business Process Execution Language (BPEL). These workflow models are considered executable due to their formal semantics specified, and the automated service workflows are sometimes referred to as Service Orchestrations.

However, service orchestration languages impose heavy restrictions on the technical details and are not flexible enough to address the requirements of modelling processes at a strategic level for business users, e.g., providing easily understandable graphic notations for different stakeholders [111] or modelling human tasks[18]. For modelling business processes, the Business Process Model and Notation[19] (BPMN) has been proposed and standardised by the Object Management Group (OMG). Service composition at the business process level (as in BPMN) between multiple parties (without focusing on internal service implementations) are sometimes referred to as Service Choreography [112]. Early versions of BPMN were criticised for lacking formal semantics and causing ambiguities [113]. Later revisions of BPMN (version 2.0, released in 2009) provided execution semantics and allowed mapping to BPEL directly. Figure 3.5 illustrates service collaborations on different levels.

[16] WSDL 1.1: http://www.w3.org/TR/wsdl, last accessed: March, 2015.

[17] WSDL 2.0: http://www.w3.org/TR/wsdl20/, last accessed: March, 2015.

[18] Human tasks in BPEL are addressed in the extended version: BPEL4PEOPLE: http://docs.oasis-open.org/bpel4people/bpel4people-1.1.html, last accessed: May, 2015.

[19] BPMN: http://www.bpmn.org/, last accessed: May, 2015.

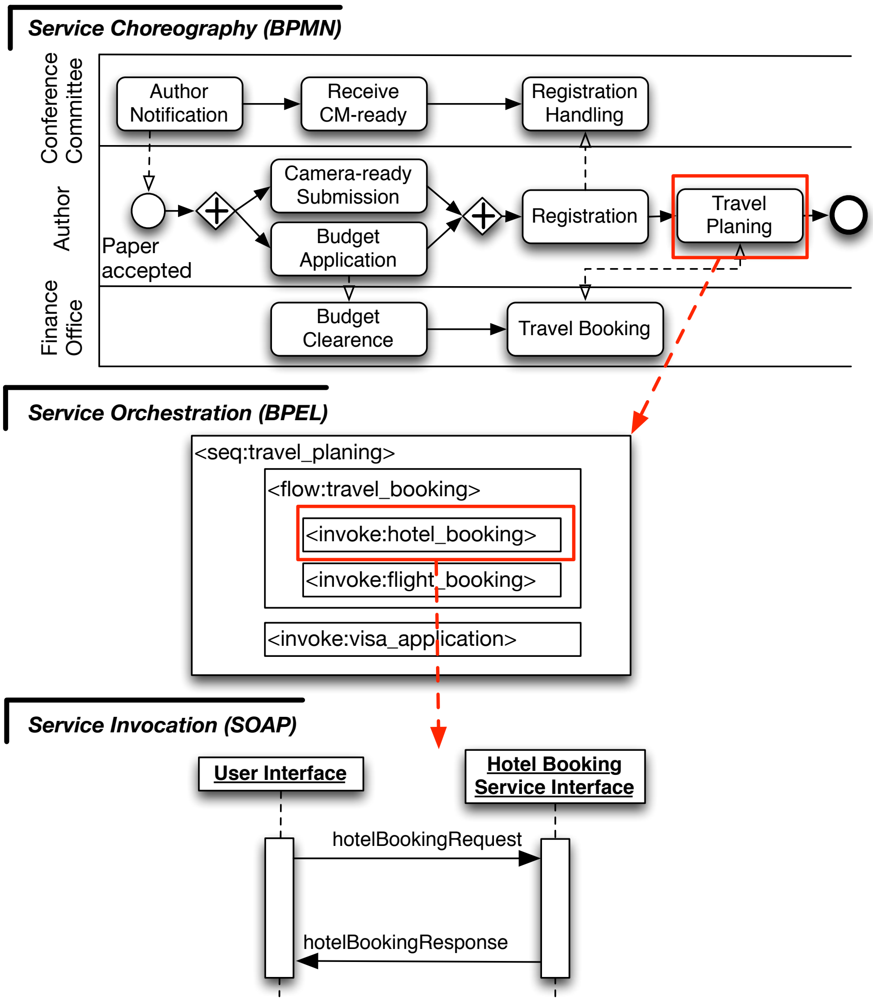


FIGURE 3.5: Example of service interactions of the process of attending an academic conference at different levels

3.2.4 Semantic Web Service

According to [114], WSDL concepts are familiar to software engineers thus they can easily implement and access services using WSDL. However, WSDL services are notorious for the lack of automated support for service discovery and composition [115, 116], because of lacking the semantic description on service capabilities and consumers' goals as well as the reasoning ability over the capabilities and goals. Semantic Web Service (SWS) is a research area that brings together web service and Semantic Web technologies. SWS enriches web services with knowledge representations and reasoning techniques. Semantic enrichments for service descriptions, including SAWSDL[20], WSMO[21] and OWL-S [117] and others, are used to facilitate automatic service discovery and composition. In

[20] Semantic Annotations for WSDL: http://www.w3.org/2002/ws/sawsdl/, last accessed: Mar. 2015

[21] Web Service Modelling Ontology: http://www.w3.org/Submission/WSMO/, last accessed: Mar., 2015.

SAWSDL, *modelReference* can attach to *portTypes* and message data types to indicate the category of operations and messages. Lifting and lowering schema are used to transform input and output data. In this way composing web services based on the semantics of IO messages are made possible. However, it does not go beyond providing semantics to the service interface. In WSMO and OWL-S, the semantics of input, output, precondition and effects are captured by using ontologies and axioms. Non-functional properties (service profile) are also captured.

Service discovery and indexing based on semantic similarity between a service request and a service description can be found in [26, 118–121]. Semantic service composition based on Artificial Intelligence (AI) planning and forward/backward chaining algorithms can be found in [122–127]. The above mentioned semantic service discovery and composition takes into account only the functional aspects of services. QoS aware service composition and optimisation is NP-hard [88]. Various techniques, e.g., [83, 88, 128–130], have proposed different heuristics to solve the problem efficiently.

3.3 Complex Event Processing

Complex Event Processing (CEP) is a technique for analysing real-time streams of information and generating (near) real-time insight on current situations. In particular, CEP performs a set of operations, including read, create, aggregate, discard etc., on events and derives conclusions as complex events. [1]. The research on CEP has several roots, e.g., discrete event simulation [131] and active database management [132]. Compared to traditional Databased Management Systems (DBMS) in which data is persistently stored and answers queries synchronously for users at run-time, in CEP systems the user queries are registered statically (for a period of time) and the data is continuously arriving in streams, and the query is answered in an asynchronous fashion [133], as depicted in Figure 3.6.

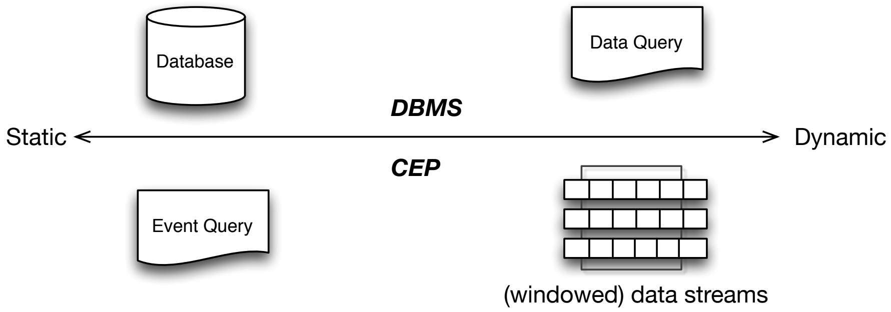


FIGURE 3.6: Differences between CEP and DBMS: static or dynamic information

Since the first CEP engine was introduced in 1995 [7], the methodology has been adopted in many application domains. In fact, CEP has been proved useful both in empowering traditional industry, e.g., financial services [134], supply chain management [135], health care [136] etc., as well as in enabling newly founded areas like Smart City applications [137]. A reference architecture for designing CEP systems is proposed in [138].

3.3.1 Basic Concepts in CEP Systems

A CEP system is organised with a set of entities related to *Events*. Figure 3.7 illustrates the high-level architecture of a CEP system. Central to this architecture, an *Event Processing Agent* (EPA) receives events produced and provided by *Event Providers*, evaluates in real-time the event processing logic (i.e., event pattern) against the input events and generates as output *Complex Events* consumed by *Event Consumers*. In the following, the basic concepts in CEP Systems are introduced.

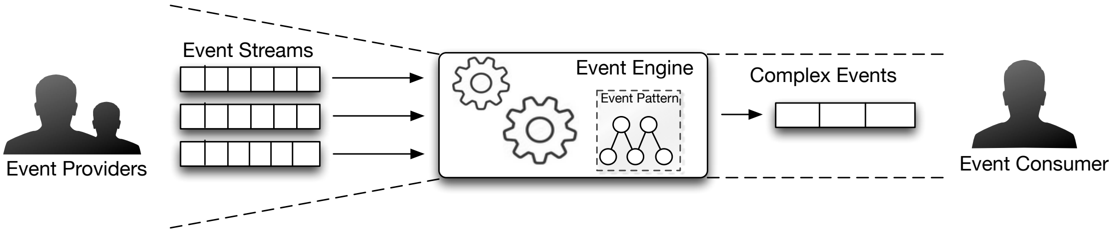


FIGURE 3.7: High-level architecture of CEP systems

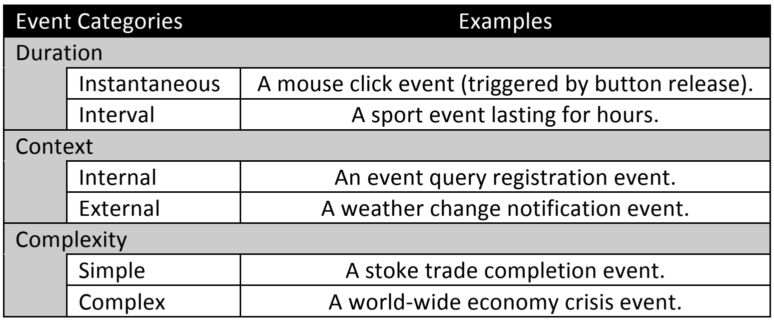
3.3.1.1 Events

According to the Event Processing Glossary[22] published by the Event Processing Technical Society (EPTS), an event is something that happens, or contemplated as happening. An event can be seen as a significant change in the state of the universe [139]. It signifies a notable thing that happens in or outside your business, e.g., a problem or impending problem, an opportunity, a threshold, or a deviation [14]. An event can also be a thing that did not happen at all, signifying the absence of the occurrence [140]. In [7] an event is defined as a computer processable object that formed with particular attributes (e.g., event id, content, timestamp etc.), signifies an activity with the attributes and relates to other events by time, causality and aggregation.

Events can be categorised into different types using different dimensions. On the temporal duration dimension, an event can be instantaneous, i.e., happens at a time point,

[22] Event Processing Glossary, version 2.0: http://www.complexevents.com/wp-content/uploads/2011/08/EPTS_Event_Processing_Glossary_v2.pdf, last accessed: Mar., 2015.

or interval, i.e., happens for a time period. On the contextual dimension, an event can be internal, i.e., happens within the CEP system, or external, i.e., happens outside the CEP system. On the complexity dimension, an event can be simple (primitive), i.e., consists of an atomic change in state, or complex (composite), i.e., consists of a set of changes in state. Table 3.1 shows examples of different event types.



Event Categories		Examples
Duration		
	Instantaneous	A mouse click event (triggered by button release).
	Interval	A sport event lasting for hours.
Context		
	Internal	An event query registration event.
	External	A weather change notification event.
Complexity		
	Simple	A stoke trade completion event.
	Complex	A world-wide economy crisis event.

Table 3.1: Event categorisation in different dimensions

The event types in Table 3.1 are agnostic to application domains. There are also domain-specific event types that describe events with similar meaning and structure. Hereafter, the notion of *event type* is used to represent domain-specific event types unless stated otherwise. Each event object is considered an instance of an event type. A type of event may re-occur in the history of time, e.g., the raining event may happen several times in a week or even in a day. On the contrary, an *event instance*, e.g., it was raining yesterday at 4:00 PM in Galway city, is considered unique in the history and can never happen again.

3.3.1.2 Event Processing Agent and Event Processing Network

An EPA can be generally defined as an entity that process events (as in the EPTS definitions). It is a node in an Event Processing Network (EPN) that receives one or more events as inputs, process them, and creates one or more events as output [141]. According to this definition, a Complex Event Service (CES) is an EPA wrapped into a service. Based on the event processing logics specified by an event consumer, an EPA may perform different kinds of computation on events such as filtering, transforming, and detecting *Event Patterns.* Figure 3.8 illustrates a taxonomy of EPAs introduced in [1]. Multiple EPAs and can be used in a coordinated way to achieve CEP tasks. *Event Channels* are used to deliver events between EPAs. Collectively, the coordinating EPAs and event channels connecting the EPAs form an EPN. Conceptually, a whole EPN can be seen as an EPA in a higher level EPN, if its internal communication is ignored [138]. Technically, an EPN can be implemented as a centralised runtime artefact or distributed runtime artefacts with a messaging system [1].

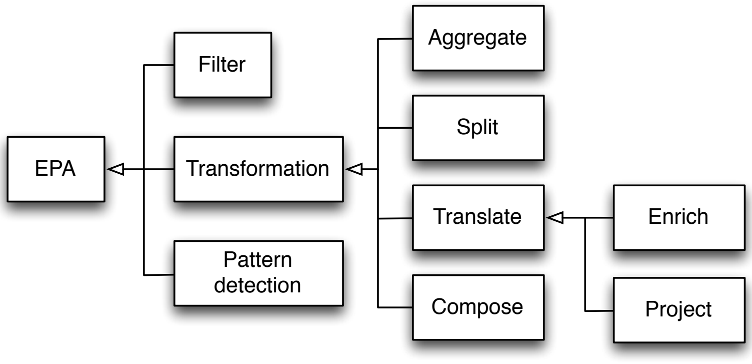


FIGURE 3.8: Taxonomy of EPA (from [1])

3.3.1.3 Event Provider and Consumer

Intuitively, event providers introduce events to EPAs (or EPNs) and event consumers accepts events from EPAs (or EPNs). Event providers and consumers are roles that could be taken by hardware, human-interaction and software [1]. An EPA can be an event provider and/or consumer. According to the CES definition in Chapter 1, a CES can be an event provider and consumer at the same time.

3.3.2 Event Channel and Routing

An event channel can be as simple as a one-to-one messaging without making routing decisions between two EPAs, or it can route events to specific receivers and it can have multiple sources and destinations. If all EPAs share a single channel with only one hop, i.e., direct links are available for any sources and destinations, the event channel is called an Event Bus [1]. For multi-hop channels, nodes in the channel can be EPAs publishing and consuming events and/or broker nodes forwarding events on behalf of other nodes. Various routeing mechanisms have been developed for such publish/subscribe systems, which can be categorised into *broadcast, multicast, peer-to-peer* and *rendezvous-based.*

3.3.2.1 Broadcast

All events entering the channel is forwarded to all destinations of the channel. Broadcasting is easy to configure and maintain and it works well in centralised EPNs or small-scale networks (e.g., Local-area networks) where there is little bandwidth restriction. However, broadcasting is not applicable for large-scale distributed EPNs, because it will impose excessive transportation overhead, i.e., unnecessary network traffic caused by irrelevant events received by EPAs.

3.3.2.2 Multicast

All events entering the channel are partitioned into non-overlapping *event groups*. For each event group, a *server group* is constructed that groups destination nodes in the channel (i.e., EPAs) together. Each EPA in a server group has an event subscription scope which overlaps with the events in the corresponding event group. Service groups may have overlaps. Internally in each server group has a routeing tree that spans all EPAs for all events in the event group. Upon receiving an event from a source, it is mapped to an event group and delivered to every member of the corresponding server group. Examples of multicast event routeing can be found in [142, 143]. Compared to broadcast, multicast creates less duplicated messages and reduces the network traffic overhead, meanwhile the routeing table is not very complicated [144]. Clustering algorithms like k-means [145] can be used to partition event groups so that the network overhead are better managed.

3.3.2.3 Peer-to-peer

In peer-to-peer (P2P) networks, all nodes are equal and have the knowledge of their neighbour nodes. Each node expresses its interests over events using filter-based subscriptions. Filters used in subscriptions can be type-based or content-based (i.e., event attribute based). Each node receives subscriptions from its neighbours and sends its own subscriptions to the neighbours. Subscriptions sent to neighbour nodes can be different: when node A sends its subscription to node B, A will send the subscriptions as the union of the events consumed by A and the subscriptions from all the neighbours of A excluding B. When the P2P network is initialised, the subscriptions of the nodes are broadcasted through the network, and then when an event arrives at a node, it matches the event to the subscriptions (i.e., filters) in its routeing table to determine which direction the event should be forwarded to. Examples of P2P filter-based event routing can be found in SIENA [44, 146], Gryphon [147] and REBECA [148]. P2P event routeing has higher network efficiency than multicast. However, the subscription broadcasting overhead grows super-linearly against the total subscriptions in the channel, also, the management and processing of the routeing tables is costly [144].

3.3.2.4 Rendezvous-based

In rendezvous-based routing, the event space is modelled as a n dimensional space, where n is the number of attributes in the events. The event space is divided into non-overlapping sub-spaces, and each sub-space is assigned to one rendezvous node in the network. A rendezvous node N_r (sometimes called proxy node, as in [144]) is thus responsible for matching the events in the assigned event sub-space to the subscriptions hosted on N_r. A subscription is hosted on N_r if the subscription scope (i.e., n dimensional "cube" formed by the filters in the subscription) intersects with the event sub-space assigned to N_r. When an event is received, it is firstly forwarded to the unique corresponding rendezvous node. Then, it is matched against the subscriptions hosted on the rendezvous node and forwarded to subscribers of the matching subscriptions. Examples of rendezvous-based routing can be found in [144, 149–152]. Rendezvous-based routeing has the advantages of both multicast and P2P filter-based routeing: simple routeing tables management as well as high network efficiency [144]. However, it does not cope well with the dynamicity in the network topology: when nodes are joining or leaving the network, the event sub-spaces and subscriptions must be re-arranged. Also, it does not cope well with un-ordered, discrete attributes, e.g., string values [153].

3.3.3 Event Pattern

Event pattern detection is the core functionality of event processing [1]. It is a computational process in which a set of temporally ordered events, i.e., an Event Instance Sequence (EIS), is evaluated to check whether they satisfy a pre-defined *event pattern* [141]. The process of event pattern detection can have three steps: a *filtering* step that filters out irrelevant events, a *matching* step that selects a subset of the relevant events and a *derivation* step that creates output events (i.e., a complex event) using the matched events. An event pattern provides instructions on how these steps should be carried out by an EPA over event streams. In [1], event pattern is defined as:

> *"An event pattern is a template specifying one or more combinations of events."*

In particular, an event pattern describes the *relevant event types*, *pattern type*, *pattern parameters* and *pattern policies* [1], where the relevant event types describe the type of events involved and monitored by the pattern, the pattern type describes the type of correlation between the set of events satisfying the pattern, pattern parameters give a set of values used in the pattern and pattern policy gives the instructions on which subset of events should be selected (filtered) and consumed. A similar description of event pattern is provided in [141], with additional explicit notations for the *predicate* between pattern parameters as well as for the *derived events.*

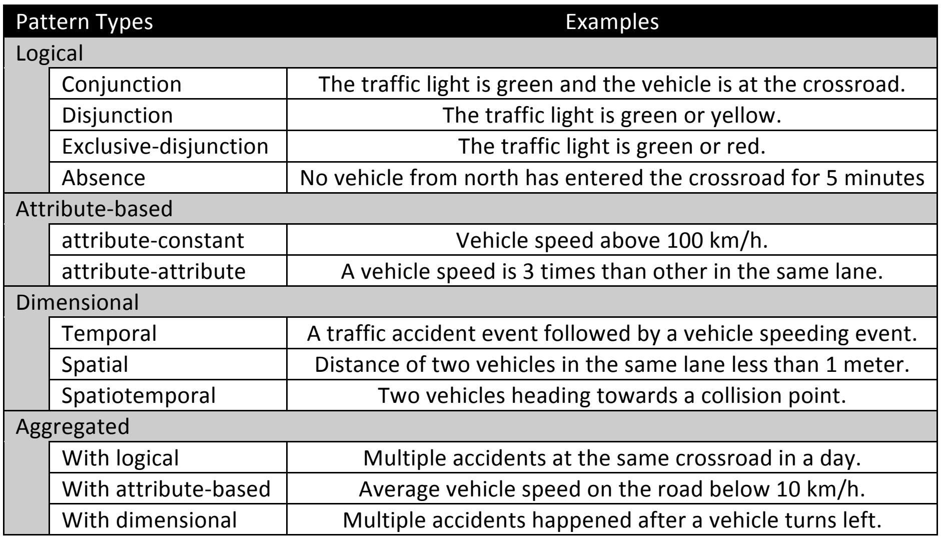
In [1], various event pattern types are discussed, which can be broadly categorised into *Logical*, *Attribute-based*, *Dimensional* and *Aggregated* patterns.

- **Logical Pattern:** Describes the existence (or non-existence) of relevant event types using logical operators, including *conjunction* (all relevant events must occur), *disjunction* (at least one relevant event types should occur), *exclusive-disjunction* (at least one relevant event type should occur but not all) and *negation* (the relevant event types should be absent).

- **Attribute-based Pattern:** Describes the rules to be evaluated over event attributes (if the event occurs) in order to detect the pattern. Such rules can be expressed as filters over event attributes. The expression of the filters describes the relation between event attributes with constants or other event attributes.

- **Dimensional Pattern:** Describes the temporal (e.g., the sequential order of event occurrences), spatial (e.g., distance between events) and spatiotemporal correlations (e.g., moving in a direction) between the relevant event types.

- **Aggregated Pattern:** Describes the rules over a set of event instances occurred. It can be used together with other patterns to provide aggregation functions, e.g., counting the occurrences of events (with a logical conjunction), calculating the average value of an event attributed (with attribute-based pattern) or detecting repetitions of an event sequences (with temporal pattern).



Pattern Types		Examples
Logical		
	Conjunction	The traffic light is green and the vehicle is at the crossroad.
	Disjunction	The traffic light is green or yellow.
	Exclusive-disjunction	The traffic light is green or red.
	Absence	No vehicle from north has entered the crossroad for 5 minutes
Attribute-based		
	attribute-constant	Vehicle speed above 100 km/h.
	attribute-attribute	A vehicle speed is 3 times than other in the same lane.
Dimensional		
	Temporal	A traffic accident event followed by a vehicle speeding event.
	Spatial	Distance of two vehicles in the same lane less than 1 meter.
	Spatiotemporal	Two vehicles heading towards a collision point.
Aggregated		
	With logical	Multiple accidents at the same crossroad in a day.
	With attribute-based	Average vehicle speed on the road below 10 km/h.
	With dimensional	Multiple accidents happened after a vehicle turns left.

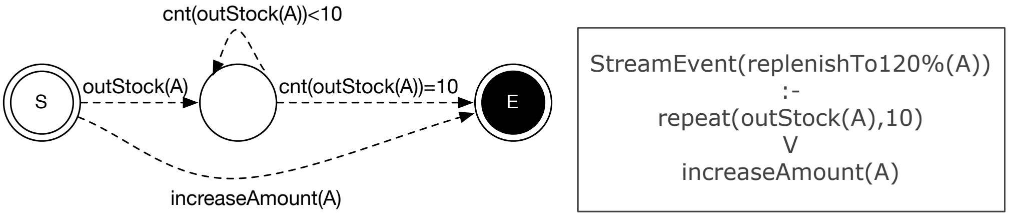
Table 3.2: Event pattern types and examples

Table 3.2 shows examples for different pattern types. An event pattern can be specified in a declarative way using Event Pattern Languages (EPLs). In [7] several basic requirements for EPLs are defined: 1) an EPL should be *expressive* enough to cover different application domains, 2) it should use *simple notations* to allow easy definition of event patterns, 3) it should have a mathematically *precise* semantics to ensure correctness in pattern detection and 4) the design of an EPL should be *scalable* so that it does not have any negative effects on the performance of pattern detection. In [154] the authors discuss in details the criteria for a successful EPL design which allows an easier generation of new CEP applications. Examples of EPL include Rapide [77], Borealis [78], RuleCore [79], SASE+ [80], Cayuga [81] etc.

3.3.4 Semantic Event Processing and Stream Reasoning

Despite various CEP solutions that have been developed (such as the EPLs and engines introduced in Section 3.3.3), there are no shared definitions for the syntax or semantics of complex events. This will lead to inconsistencies or conflicts between used terminologies and ultimately “silo” CEP solutions [154, 155]. Moreover, processing events on a syntactical level lacks the ability to integrate the real-time event streams with higher-level knowledge representation and reasoning [19, 20]. For example, syntactical event processing cannot detect events based on reasoning on their type hierarchy relations or on their relations to concepts from other application domains [156]. To address these issues, a recent research concept of Semantic (Complex) Event Processing (SCEP) has been proposed and studied to provide reasoning capability over semantically annotated events.

According to [156], there are two mainstream styles for CEP implementation: *rule-based* (e.g., [22]) or *Nondeterministic-Finite-Automata-based* (NFA-based, e.g., [157]). In rule-based implementations, events are injected to a logic programming system as facts, and event patterns are specified as goals with rules. In NFA-based implementations, a state model is used to describe the event pattern and the state transitions are controlled by an engine based on incoming events. Figure 3.9 illustrates examples of an event pattern modelled as NFA and rules (time windows are omitted).



```
StreamEvent(replenishTo120%(A))
:-
repeat(outStock(A),10)
V
increaseAmount(A)
```

(a) NFA

(b) Rule-based

FIGURE 3.9: Example of a complex event pattern: if 10 out of stock events are captured for a product in the supermarket in the past week, or a request for increasing the amount of product is received, an replenishment event notifying the need of increasing the amount of the product in the purchase order is produced.

Most existing SCEP systems adopts the rule-based approach, e.g., [21–24]. Indeed, it is natural to support semantic reasoning over events in rule-based systems, via encoding the background knowledge and event patterns as rules in domain ontology and event ontology, respectively.

A closely related research area to SCEP is called stream reasoning [158] or RDF Stream Processing (RSP), which evolved from Data Stream Management Systems (DSMSs) [159] and provides reasoning capability over data streams. A data stream can be considered as an event stream if each data item is considered an event [160], hence a data query pattern can be seen as an event pattern [155]. A recent study (in 2014) argues the research area of stream reasoning "remains vastly unexplored" [161], despite the research efforts made, such as EP-SPARQL [19], CQELS [37], CSPARQL [38] and SPARQL$_{stream}$ [162]. These approaches provide different extensions to the SPARQL[23] syntax and semantics to implements different stream operators, enabling querying and reasoning over RDF streams.

Most of the stream reasoning engines employ an underlying DSMS/CEP engine (NFA-based) for handling basic streaming operations, e.g., CQELS and CSPARQL use Esper[24] and SPARQL$_{stream}$ uses SNEEql [163]. EP-SPARQL is the only stream reasoning engine so far that attempts to support the full set of CEP operators using a rule-based engine (Prolog [164]). A recent research proposes to provide a hybrid approach for SCEP combining NFA and rule-based systems to achieve both high expressiveness and high throughput of the system [165]. However, a concrete implementation of the system has yet to be developed.

3.4 Summary and Discussion

In this chapter, relevant concepts and techniques in the SW, SOC and CEP are introduced as the research background of the thesis. In particular, for the SW, the data model (RDF), knowledge representation language (OWL) and data query language (SPARQL) are introduced. The term Linked Data is explained as interlinking public vocabularies. The usage of semantic annotations in IoT and sensor networks is discussed. For SOC, the service oriented architecture is introduced. The concept of service description documents is explained. Service interactions on different levels are elaborated with an example. The usage of the SW in services is also explained, resulting in Semantic Web Services that allows automatic discovery and composition. For CEP, the concepts of events, event processing agents and event channels are introduced. These CEP concepts constitute a distributed event processing network. The routing mechanisms in event

[23] SPARQL Query Language for RDF: http://www.w3.org/TR/rdf-sparql-query/, last accessed: Mar., 2015.

[24] Esper home: http://www.espertech.com/esper/index.php, last accessed: Mar., 2015.

channels are detailed. Different event patterns describing logical and temporal relations between events are explained. The state-of-the-art in Semantic CEP is discussed. In this thesis, we leverage these three techniques and integrate them to realise Semantic Event Services. In the next chapter, an overview of the SES management middleware is presented. This middleware is an integration of the solutions developed in this thesis.

Part II

Core - Semantic Event Service Management

Chapter 4

Overview of the Automatic Complex Event Implementation System

In this chapter, a high-level overview of the Automatic Complex Event Implementation System (ACEIS) is presented. ACEIS is a middleware for managing the activities in the life cycle of Semantic Event Service (SES). It is an integration of the techniques studied and developed in this thesis. In the following sections, the functional design of ACEIS and how it fulfils the requirements in Chapter 2 is elaborated (Section 4.1). The architecture of ACEIS and the interactions between ACEIS components are described (Section 4.2) and the deployment of ACEIS in a Smart City framework is presented (Section 4.3). Finally, a summary is provided (Section 4.4).

4.1 ACEIS Key Functionality Design

ACEIS is designed to address the requirements for handling the modelling, planning and adaptive implementation of SESs. In particular, Chapter 2 analysed three main requirements in managing SESs: *user-centric SES definition*, *automatic SES planning* and *automatic and adaptive SES implementation*. In order to address these requirements, ACEIS is designed to realise four key functionalities, which are explained in the following. Figure 4.1 illustrates how these functionalities are designed to address the requirements.

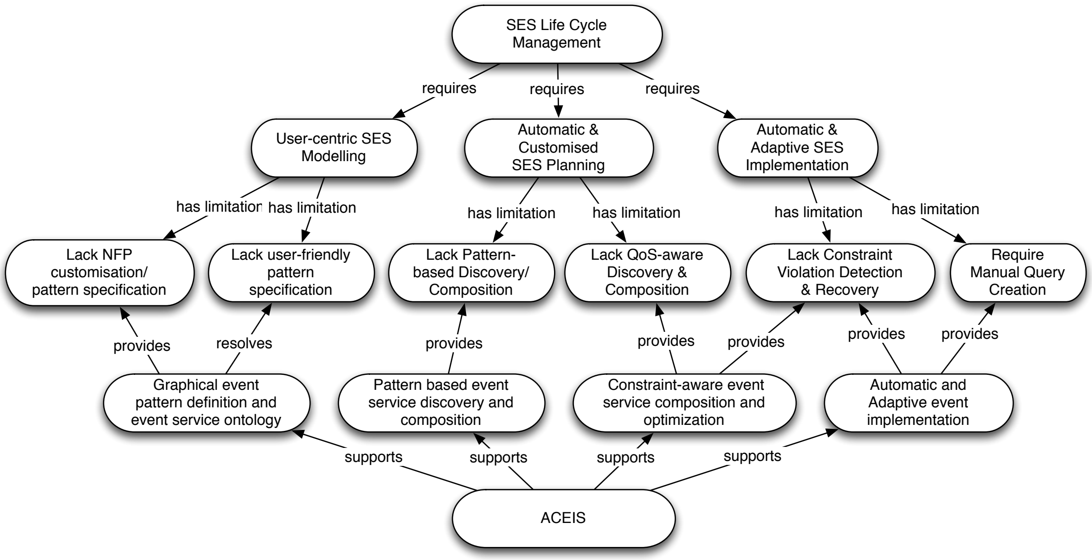

SES Life Cycle Management
requires
requires
requires
User-centric SES Modelling
Automatic & Customised SES Planning
Automatic & Adaptive SES Implementation
has limitation
has limitation
has limitation
has limitation
has limitation
has limitation
Lack NFP customisation/ pattern specification
Lack user-friendly pattern specification
Lack Pattern-based Discovery/ Composition
Lack QoS-aware Discovery & Composition
Lack Constraint Violation Detection & Recovery
Require Manual Query Creation
provides
resolves
provides
provides
provides
provides
provides
Graphical event pattern definition and event service ontology
Pattern based event service discovery and composition
Constraint-aware event service composition and optimization
Automatic and Adaptive event implementation
supports
supports
supports
supports
ACEIS

FIGURE 4.1: Functional design of the Automatic Complex Event Implementation System

4.1.1 Event Service Annotation and Event Pattern Definition

In ACEIS, a Complex Event Service Ontology (CESO) is developed as an extension of OWL-S [117] to annotate Complex Event Services (CESs). CESO extends the *service profile* in OWL-S and defines an event profile to specify the features of the CESs. Event profiles in CESO contains recursively defined event patterns to specify the functional aspects of the CES and reuses the concept of *service parameters* in OWL-S to define the non-functional aspects. Moreover, CESO provides concepts to define event service requests with requested event patterns, non-functional constraints/preferences. This way it allows customising the descriptions of the event services/requests for each individual service provider/consumer and facilitates *user-centric SES modelling.*

The syntax and semantics of the event patterns defined within CESO are provided as a revision of Business Event Modelling Notations (BEMN [3]), called BEMN$^+$. The formal semantics of event patterns are the basis for realising *automatic SES planning and implementation.* And by adopting the graphical and process-model-compatible notations in BEMN, the event pattern definition can be provided in a user-friendly way for business/non-technical users.

4.1.2 Pattern-based Event Service Discovery and Composition

Discovering and composing event services based on event patterns addresses the functional aspect in *automatic SES planning.* In order to compare the semantics of event patterns, means to derive canonical forms of event patterns are provided. Then, the problem of semantic equivalence or subsumption relation between event patterns can be transformed into graph/sub-graph isomorphism problems, which is NP-hard [166]. To improve the efficiency of the composition algorithms, an event pattern reusability index is developed based on comparing event pattern semantics, i.e., a CES is reusable to another if the pattern of the latter subsumes the former. Leveraging the index and the heuristic of minimising the network traffic demand of composition plans, the automatic SES planning is realised to allow on-demand composition of event services.

4.1.3 Constraint-aware Event Service Discovery and Composition

In order to fully support *customised SES planning*, the pattern-based event service composition algorithms are extended to support Quality-of-Service (QoS) optimisations. In particular, a QoS model is designed to capture different QoS parameters to be considered in event service compositions. A QoS aggregation schema is designed to estimate the QoS performance of a composition plan based on the QoS descriptions of the event services involved. A multi-dimensional QoS utility function is created to compare and rank candidate composition plans based on their estimated QoS performance and user-defined QoS constraints/preferences. Finally, a Genetic Algorithm (GA) is designed to efficiently derive near-optimal composition plans, using the QoS utility as the fitness function.

4.1.4 Automatic Event Service Implementation and Adaptation

Existing RDF Stream Processing (RSP) engines (e.g., CQELS [37] and CSPARQL [38]) provide the capability of reasoning over semantically annotated event streams. The query patterns supported by the RSP engines is a subset of the patterns supported in the CESO and BEMN$^+$. Query transformation algorithms are developed to transform event patterns in the composition plans into executable RSP queries, so that the *automatic SES implantation* over different RSP platforms is realised. To ensure the correctness of the query transformation, the semantics of the RSP query operators and event operators in BEMN$^+$ are aligned.

Leveraging the capability of *automatic SES implementation*, an *adaptive SES implementation* can be realised. In particular, a monitoring component in ACEIS received QoS updates for relevant composition plans and determines if a QoS update will cause the violation of user-defined QoS constraints. If so, an adaptation strategy is applied to recompose fully or partially of the composition plan, in order to recover the QoS performance degrade of the composition plan.

4.2 ACEIS Architecture

Figure 4.2 illustrates the architecture view of ACEIS. The architecture consists of four main components: *Knowledge Base*, *Application Interface*, *Semantic Annotation* and *ACEIS Core* component. These components implement the key functionalities described in Section 4.1 as well as additional functionalities, e.g., subscription management, RSP result handling, data storage and streaming etc. The components are colour-coded to indicate the chapters that describe the details of the components.

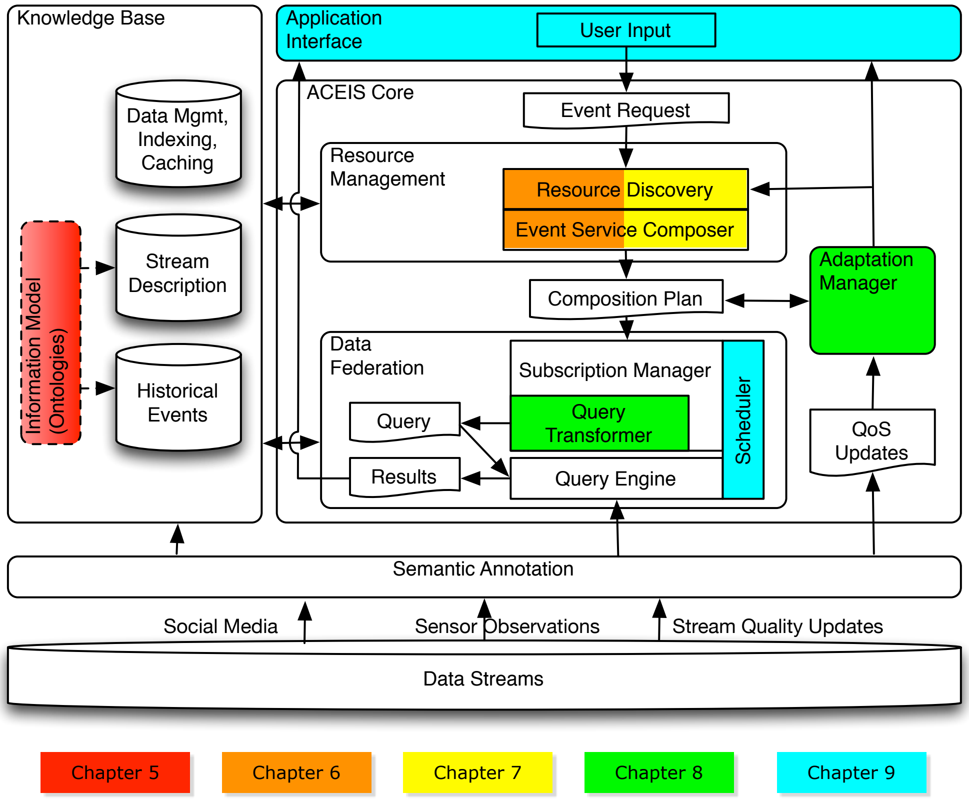


FIGURE 4.2: ACEIS architecture

4.2.1 Knowledge Base

The knowledge base stores the semantic annotations for the static description of event services as well as the indexing structure for accelerating the discovery and composition of event services. The knowledge base can also store historical events and data items for analytical purposes. An information model that consists of an event service ontology, a sensor ontology and domain ontologies are used as guidelines for annotating the data in the knowledge base. In the current implementation the Semantic Sensor Network

(SSN) ontology[1] is used (together with domain ontologies) for describing sensors and sensor observations. The domain ontology is application-specific and is expected to be provided by domain experts. The event service ontology is described in Chapter 5.

4.2.2 Application Interface

The application interface interacts with end users as well as ACEIS core modules. It allows users to provide inputs required by the application and presents the results to the user in an intuitive way. It also augments the users' queries, requirements and preferences with some additional, implicit constraints and preferences determined by the application domain or user profile. For example, in a travel navigation scenario, a user may specify only the start and target location on the map, with a constraint on the travel time t, because she needs to get there on time. The application may add additional constraints on the IoT data streams used to calculate the travel time. For example, the frequency of the data streams should be more than $1/t$, otherwise the user may not receive any updates on the traffic condition during her trip and the detour suggestions for traffic jams will never happen.

These augmented user inputs are transformed into a semantically annotated complex event service request (event request for short). The event request is consumed by ACEIS core components to discover and integrate urban streams with regard to the functional and non-functional constraints specified within the event request. The application interfaces of the prototype implementations for the motivation scenarios are presented in Chapter 9.

4.2.3 Semantic Annotation

The semantic annotation component receives IoT/data streams (e.g., ODAA real-time traffic sensors data[2]) as well as static data stores (e.g., ODAA traffic sensors metadata[3]) as inputs. It annotates syntactical information with semantic terms defined in ontologies. The outputs of semantic annotation include semantically annotated dynamic information (e.g., sensor observations, events) as well as static event service descriptions. With semantic annotations of both static resource and dynamic data, ACEIS gains additional data interoperability both at design time for event service discovery/composition and at run-time for semantic event detection.

[1] SSN-XG: http://www.w3.org/2005/Incubator/ssn/, last accessed: Aug., 2015.

[2] Realtime Traffic Data in Aarhus: http://ckan.projects.cavi.dk/dataset/bliptrack-alpha/resource/d7e6c54f-dc2a-4fae-9f2a-b036c804837d, last accessed: Aug., 2015.

[3] Traffic Sensor Metadata: http://ckan.projects.cavi.dk/dataset/bliptrack-alpha/resource/e132d528-a8a2-4e49-b828-f8f0bb687716, last accessed: Aug., 2015.

4.2.4 ACEIS Core

The ACEIS core module serves as a middleware between low-level IoT data streams and upper-level Smart City applications. ACEIS core is capable of discovering, composing, deploying and adapting event services. The ACEIS core consists of three major components: resource management, data federation and adaptation manager. In the following, their functionalities and interactions are introduced.

4.2.4.1 Resource Management

The resource management component is responsible for discovering and composing event services based on static service descriptions. It receives event requests generated by the application interface containing users' functional and non-functional requirements and preferences. Then, it creates composition plans for event requests, specifying which event services are needed to address the requirements in event requests and how they should be composed.

The resource management component contains two sub-components: a resource discovery component and an event service composer. The resource discovery component uses conventional semantic service discovery techniques to retrieve IoT services delivering primitive events. It deals with the primitive event requests specified within event requests. The event service composer creates service composition plans to detect the complex events specified by event requests based on event patterns. Chapter 6 provides methods for discovering and composing event services based on the functional requirements, while chapter 7 focuses on discovering and composing event services based on non-functional requirements.

4.2.4.2 Data Federation

The data federation component is responsible for implementing the composition plan over event service network and processing complex event logics over federated data streams. The composition plan is first used by the subscription manager which will make subscriptions to the event services involved in the composition plan. Later, the query transformer transforms the semantically annotated composition plan into a set of stream reasoning queries to be executed on an RDF stream processing engine. Different query transformation algorithms can be implemented in the query transformer to cope with different query engines. The soundness of query transformation and two different query transformation algorithms are discussed in Chapter 8.

Leveraging the service-oriented nature of ACEIS, the query results streams can also be wrapped as event services. Thus the event service compositions can be deployed over distributed query engine instances to improve the performance of the query processing. To balance the load between different engine instances, a scheduler is implemented to determine workload distribution at run-time. Chapter 9 presents the different load balancing strategies and the performance evaluations in prototype implementations.

4.2.4.3 Adaptation Manager

The adaptation manager monitors the QoS updates for the event services and determines if the QoS properties of a deployed event service composition have violated the non-functional constraints specified in the event request. When a QoS constraint violation is detected, the adaptation manager makes an attempt to automatically find replacements for parts or whole of the deployed composition plan in order to keep the QoS performance at an acceptable level. If no possible adaptation is available, a notification is sent to the user interface, which informs the user that the QoS constraint has been violated and the attempt of automatic recovery has failed. Different adaptation strategies and their performance evaluation are discussed in Chapter 8.

4.3 ACEIS Deployment in Smart City Framework

In order to provide general guidelines and structures for developing Information and Communication Technology (ICT) systems that realise Smart Cities, a Smart City Framework (SCF) is proposed in [2]. The high-level design of the architecture of SCF is shown in Figure 4.3, in which the components in red are relevant to ACEIS. The architecture in Figure 4.3 combines groups of functionalities with programmable/informational interfaces. The SCF consists of five main function groups. The *Large-Scale Data Analysis* functionality integrates a large amount of heterogeneous data and event sources producing real-time streams. Through data virtualisation, federation and aggregation, coarse-grained and semantically-enriched IoT streams are derived from fine-grained information collected from IoT devices, social media streams etc., and are utilised by the *Real-Time Intelligence* functionality, where the ability to adapt to changing situations based on the real-time information is provided and a user-centric and context-aware decision making is realised. The *Real-Time Intelligence* functionality also provides APIs for the *Smart City Applications*. The *Large Scale Analysis* and *Real-time Intelligence* functionalities are supported by *A-priori Knowledge* in the form of a knowledge base in order to carry out the reasoning capability. They are also empowered by the *Reliable Information Processing* functionality, so that the reliability of the whole system is ensured.

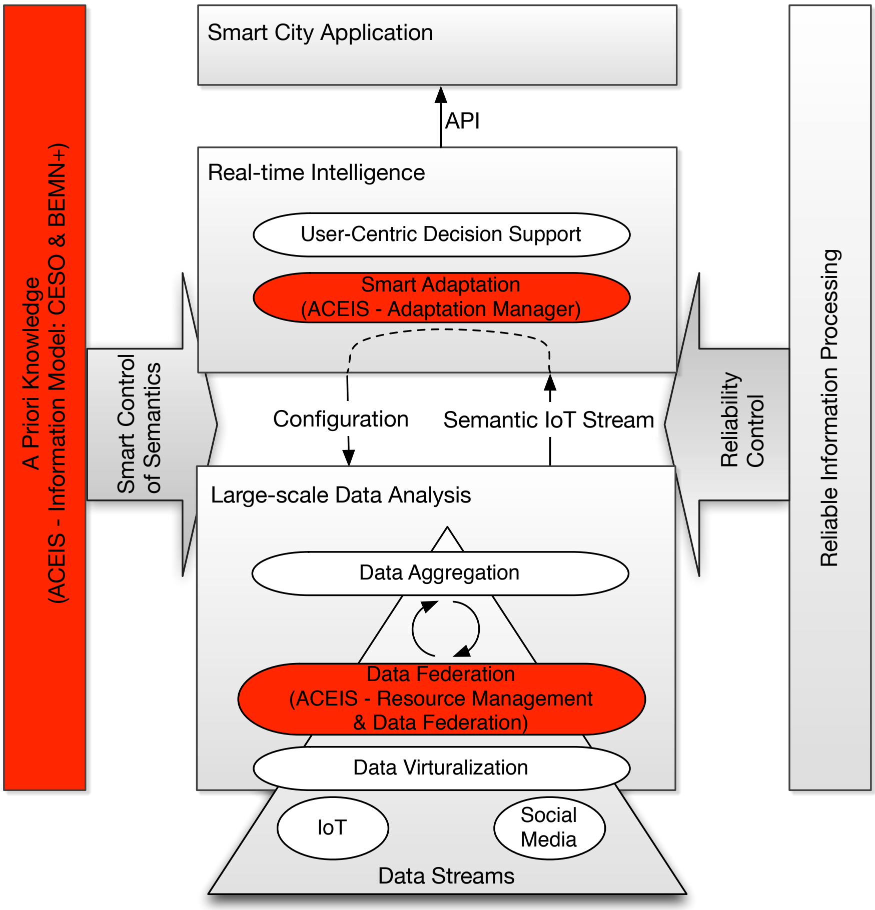


Figure 4.3: Smart City Framework: high-level architecture (from [2])

ACEIS is mainly used as an implementation for the *Data Federation* module in the *Large-scale Data Analysis* functionality in SCF. The event service discovery and composition mechanisms in Chapter 6 and 7 are utilised to create and optimise data/event stream federations. The query transformation mechanisms in Chapter 8 implements the data/event stream federations as executable RDF stream queries. The information model used in ACEIS (Chapter 5) also allows semantically annotate stream meta-data in the *Data Virtualisation*. Moreover, the technical adaptation mechanism in Chapter 8 realises partially the *Smart Adaptation* functionality designed in the *Real-time Intelligence*.

4.4 Summary and Discussion

In this chapter, an overview of the Automatic Complex Event Implementation System (ACEIS) is presented. The functional design of ACEIS is elaborated and its relevance to the requirements of SES modelling, planning and adaptive implementation is discussed. Then, the architecture of ACEIS is illustrated and described in details. The functionalities of ACEIS modules and their interactions are presented. Finally, the bigger picture of a Smart City Framework (SCF) is introduced as general guidelines and structures for providing Smart City applications. The roles and responsibilities of ACEIS in the SCF are explained. In the following chapters (Chapter 5 to Chapter 8), the design, implementation and evaluations of the key components of ACEIS are elaborated.

Chapter 5

Event Service Ontology and Event Pattern Definition*

An event service needs to be described and registered to a service repository in order to be discovered and reused. The functionality of a Complex Event Service (CES) is determined by the semantics of the complex events delivered by the service. The event semantics are specified as event patterns, and defined by users. It is important to ensure that the event service and event pattern description is both human-understandable, so that users can design them effortlessly and collaboratively, as well as machine-processable, so that activities in the event service life cycle can be handled in an automatic manner.

In this chapter, the information model used in ACEIS is described, as shown in Figure 5.1. The information model consists of an event service ontology for semantically annotating (complex) event service descriptions, and a graphical language with execution semantics for defining event patterns. The event service ontology provides guidelines to the *Semantic Annotation* component in ACEIS for creating annotated documents, which are processable by the *ACEIS Core*. The execution semantics of the event patterns specified in the pattern definition language provide guidelines for the pattern-based event service composition and query transformation algorithms in the *ACEIS Core*.

The event service ontology is a service model for event services. According to [169, 170], a service model should describe the mechanisms for accessing and interacting with the service interfaces, so that automatic service invocations are possible. Meanwhile, the service model should capture the functional and non-functional aspects of the service capability, so that a service consumer can choose the services based on specific requirements [171–173], i.e., different users may be interested in different events, or same events with different performance or quality expectations. In [174, 175] the authors explain the

*Part of the content in this chapter is published in [167, 168].

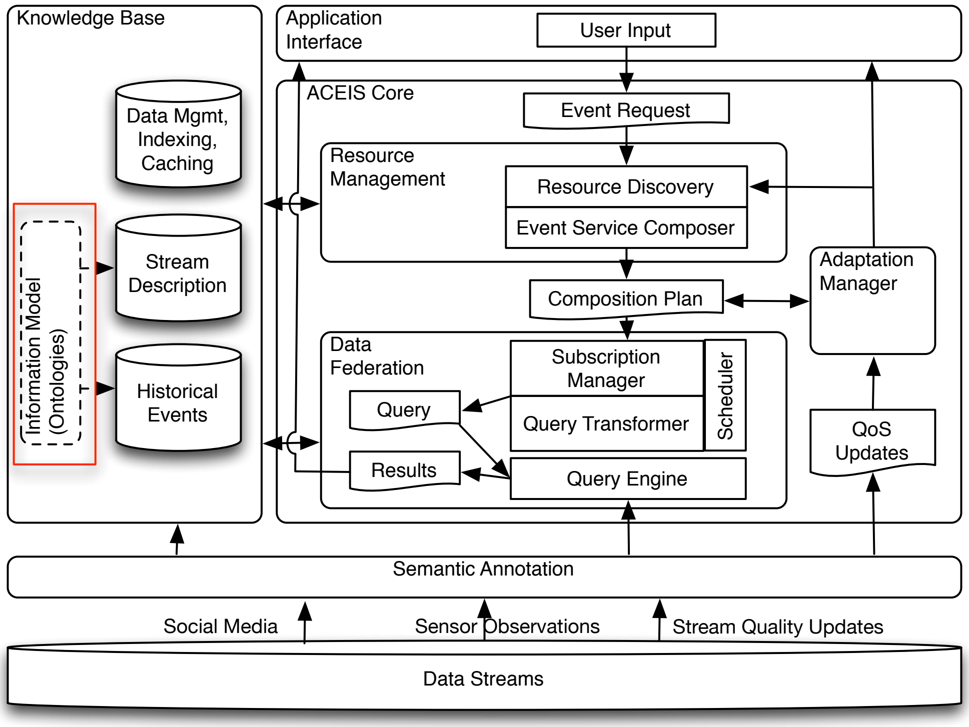


Figure 5.1: The information model in ACEIS

need for describing event patterns within business processes, to incorporate complex events in business processes and facilitate event-driven Business Process Management (ed-BPM). In [3] the authors propose to realise a seamless integration of event patterns into graphical process modelling languages with graphical event pattern representations. In [62], the authors discuss the necessity of defining the formal semantics of complex event with a small but expressive set of event operators. In summary, the event service ontology and event patterns definition language has the following requirements.

1. The event service ontology should provide the access mechanisms for automatic service invocations.

2. The event service ontology should provide the service capability descriptions for automatic and customised service discovery and composition.

3. The event pattern language should be process-model-compatible so that business users can define patterns with minimal learning overhead.

4. The event pattern language should be equipped with execution semantics that covers a wide range of scenarios where CEP is used.

This chapter presents the Complex Event Service Ontology[1] (CESO) that fulfils the first two requirements, as well as the extended Business Event Modelling Notations [3] (BEMN$^+$) that addresses the third and fourth requirements. The *Grounding* concept in CESO provides the access mechanism for event services, and the *Event Profile* in CESO is used to describe the functional and non-functional aspects of event service capabilities. BEMN$^+$ adopts the graphical notations of BEMN [3] and are made compatible with the Business Process Model and Notation[2] (BPMN). The execution semantics of BEMN$^+$ is elaborated and compared with existing RDF Stream Processing (RSP) languages and the original BEMN.

The remainder of this chapter is organised as follows. Section 5.1 introduces the CESO and elaborates the concepts defined in CESO. Section 5.2 introduces the BEMN$^+$ used as the language for defining the event patterns specified in CESO. The visual design, syntax, constraints and formal semantics of BEMN$^+$ are detailed. Section 5.3 discuss the related work on event ontology and graphical event pattern definitions before Section 5.4 summarises.

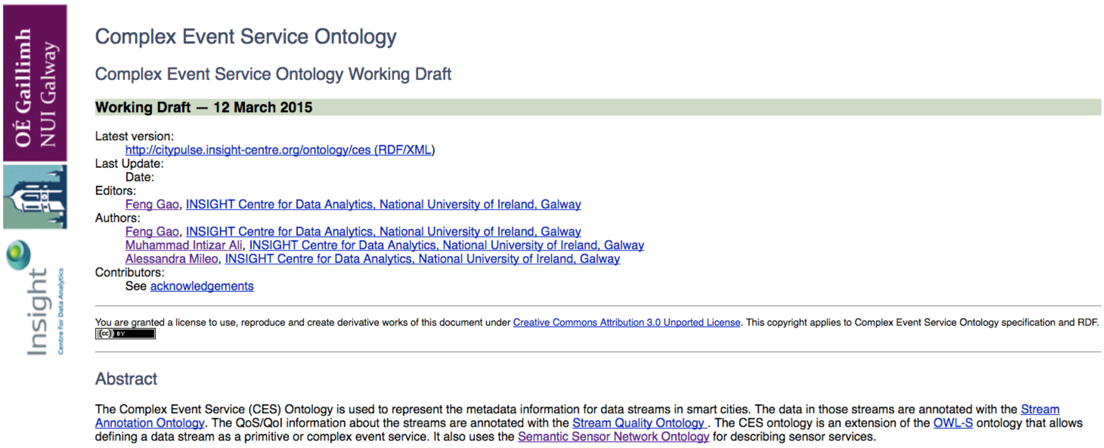
5.1 Complex Event Service Ontology

In order to facilitate on-demand, cross-platform and semantic discovery and federation of event streams, the Service Oriented paradigm is followed and event streams are considered as services transmitting events. Atomic events like sensor observations delivered in event streams are considered as primitive events without pattern descriptions, and query results over federated streams as complex events with patterns describing the temporal and logical correlations between the set of events constituting the complex events. Based on whether event patterns are described in event service descriptions, event services can be categorised as Primitive Event Service (PES) and Complex Event Service (CES). Current approaches have discussed extensively on PES modelling using traditional service description frameworks, e.g., sensor services discussed in [176–181]. However, CES description, discovery and composition remain largely unexplored. CESO is designed to address the requirements in event service discovery and composition. CESO caters for both PES (e.g., sensor data streams) and CES (e.g., federated sensor data streams). A screenshot of the web page hosting CESO is shown in Figure 5.2.

CESO is an extension of OWL-S [117], because of its extensibility for service profiles and its native support for QoS parameters. However, we do not exclude the possibility

[1] CES Ontology: http://citypulse.insight-centre.org/ontology/ces/, last accessed: May, 2015.

[2] BPMN: http://www.bpmn.org/, last accessed: May, 2015.



Complex Event Service Ontology

Complex Event Service Ontology Working Draft

Working Draft — 12 March 2015

Latest version:
http://citypulse.insight-centre.org/ontology/ces (RDF/XML)
Last Update:
Date:
Editors:
Feng Gao, INSIGHT Centre for Data Analytics, National University of Ireland, Galway
Authors:
Feng Gao, INSIGHT Centre for Data Analytics, National University of Ireland, Galway
Muhammad Intizar Ali, INSIGHT Centre for Data Analytics, National University of Ireland, Galway
Alessandra Mileo, INSIGHT Centre for Data Analytics, National University of Ireland, Galway
Contributors:
See acknowledgements

You are granted a license to use, reproduce and create derivative works of this document under Creative Commons Attribution 3.0 Unported License. This copyright applies to Complex Event Service Ontology specification and RDF.

Abstract

The Complex Event Service (CES) Ontology is used to represent the metadata information for data streams in smart cities. The data in those streams are annotated with the Stream Annotation Ontology. The QoS/QoI information about the streams are annotated with the Stream Quality Ontology . The CES ontology is an extension of the OWL-S ontology that allows defining a data stream as a primitive or complex event service. It also uses the Semantic Sensor Network Ontology for describing sensor services.

Figure 5.2: Screenshot of the CESO web page

of extending other, more "light-weighted" (and perhaps currently more popular) service ontologies, e.g., SA-WSDL[3] and SA-REST[4]. However, because they are more light-weighted and only provides annotations on taxonomical information (e.g., types for interfaces, operations, messages etc.), the concept of *Service Capability* [182] is missing. Therefore, using them to capture the capability of CESs will require defining more terms.

CESO imports concepts from Semantic Sensor Network (SSN) ontology[5] to describe sensor capabilities for PESs when those PESs are provided by sensors. CESO is designed to be used in combination with the Stream Annotation Ontology (SAO)[6] and the Stream Quality Ontology (SQO)[7]. The data in those streams are annotated with the SAO and SSN. The QoS/QoI information about the streams are annotated with the SQO. The relations between CESO, SAO and SQO is depicted in Figure 5.3.

We validated our ontology together with all reused ontologies using Jena 3.0[8] (RDFS reasoner) and Pellet 3.0[9] (OWL2 DL reasoner). The validity reports showed no inconsistencies. We refer readers to [183] for a study on evaluating ontologies. However, a detailed evaluation of CESO is considered out of the scope. In the following the basic concepts in CES are introduced.

[3] SA-WSDL: https://www.w3.org/2002/ws/sawsdl/, last accessed: June, 2016.

[4] SA-REST: https://www.w3.org/Submission/SA-REST/, last accessed: June, 2016.

[5] Semantic Sensor Network ontology: http://www.w3.org/2005/Incubator/ssn/ssnx/ssn, last accessed: May, 2015.

[6] Stream Annotation Ontology: http://iot.ee.surrey.ac.uk/citypulse/ontologies/sao/sao, last accessed: Mar., 2015.

[7] Stream Quality Ontology: https://mobcom.ecs.hs-osnabrueck.de/cp_quality/, last accessed: Mar., 2015.

[8] Jena: https://jena.apache.org/index.html, last accessed June, 2016.

[9] Pellet reasoner: https://github.com/Complexible/pellet, last accessed: June, 2016.

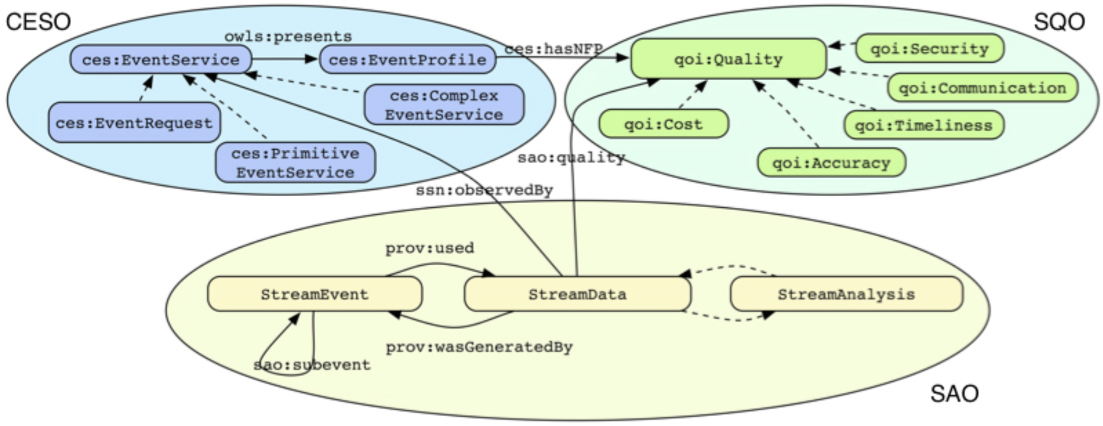


FIGURE 5.3: Relations between ontologies used in ACEIS

5.1.1 Overview

An EventService is described with a Grounding and an EventProfile. The concept of Grounding in OWL-S informs an event consumer on how to access the event service. It provides the technical details on the service protocols and message formats etc, so that a machine or program can use this information and make automatic service subscription/invocation. An EventProfile is comparable to the ServiceProfile in OWL-S, which describes the semantics of the events delivered by the service as well as the properties of the service itself, so that the capabilities of event services are identifiable and reusable by service consumers. Figure 5.4 illustrates the overview of the CES ontology.

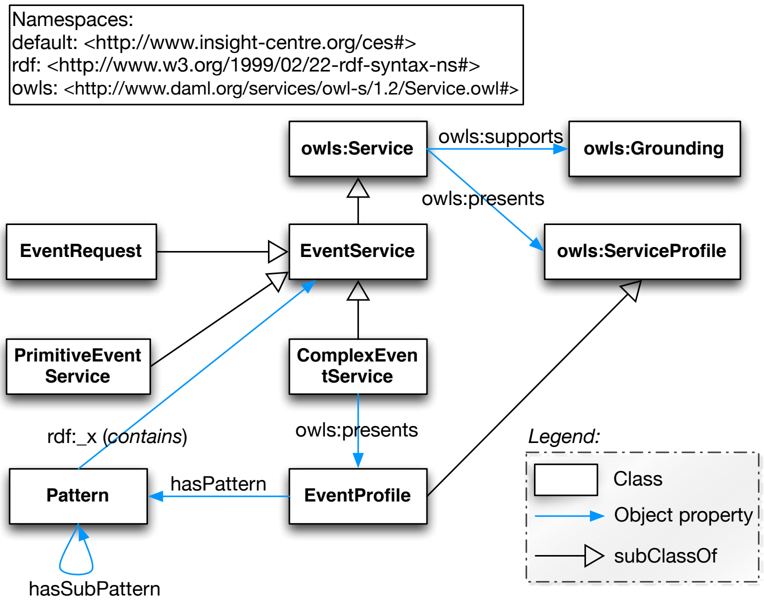


FIGURE 5.4: Complex Event Service Ontology: overview

5.1.2 Event Profile

An `EventProfile` describes a type of event with a `Pattern` and `Non-Functional Properties` (NFP). A `Pattern` describes the functional aspects of event services, i.e., the semantics of the complex events using a set of event operators. An event pattern may have sub-patterns or other event services as member event services. An event profile without a `Pattern` describes an simple event service, otherwise it describes a complex event service. `NFP` refers to the QoI and/or QoS metrics, e.g., accuracy, latency, energy consumption etc., which are modelled as sub-classes of `ServiceParameter` in OWL-S. Figure 5.5 shows the ontology for describing event profiles.

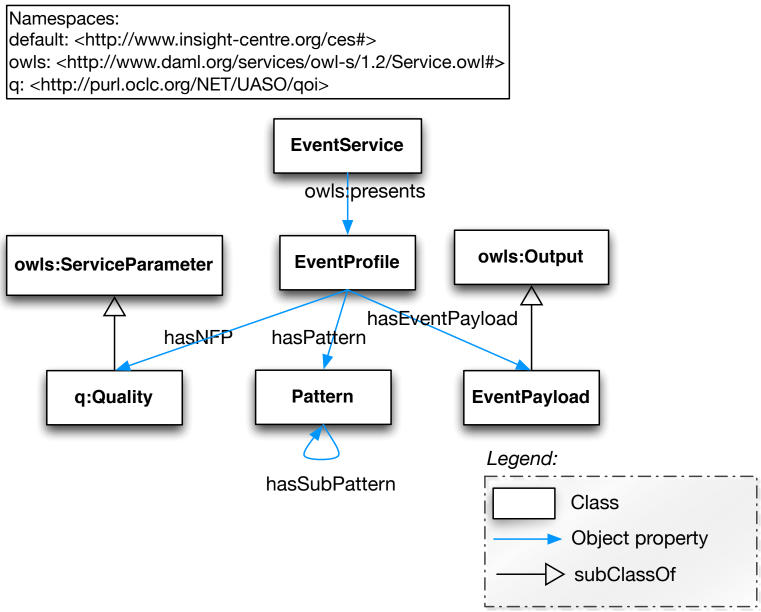


FIGURE 5.5: Complex Event Service Ontology: event profile

5.1.3 Event Pattern

The temporal relationships captured by an `EventPattern` has three basic types: sequence, parallel conjunction and parallel alternation/disjunction. If two events (or event patterns) are correlated by a sequence pattern, one should occur before the other, in parallel conjunction, both should occur and in parallel alternation, at least one should occur. Hence three types of patterns are defined respectively: `Sequence, And` and `Or`. A special case of `Sequence` is that the sequence repeats itself for more than once, in this case the sequence can be modelled by a `Repetition` pattern, with a cardinality indicating the number of repetition. A repetition can be an overlapping or non-overlapping repetition, specified by the `isOverlapping` property. Besides the temporal relationships, `Filters` and `Selections` can be used to specify attribute-based patterns and a sliding `Window` can be used to specify the window applied over event streams. `Aggregation` is a subclass of `Filter` which can be used to specify aggregated event patterns. A transitive `hasSubPattern` property is defined to describe the provenance relation between patterns and their sub-patterns and member event services. Also, we insert the rule in Listing 5.1 into the ontology, to allow reasoners to entail sub-pattern relationships for analysing the causal relations between events delivered in ESN. Notice that `rdfs:member` is the super-property for the container membership property (i.e., `rdf:_1, rdf:_2 ...`) in RDF Schema version 1.1. Figure 5.6 reveals more details on the event pattern model. Listing 5.2 gives the example of a semantically annotated event pattern in Figure 3.9 using CESO (in Turtle[10] format, prefix omitted).

```
[Rule1: (?x rdfs:member ?y) -> (?x ces:hasSubPattern ?y)]
```

Listing 5.1: Entail sub-patterns via RDF containers.

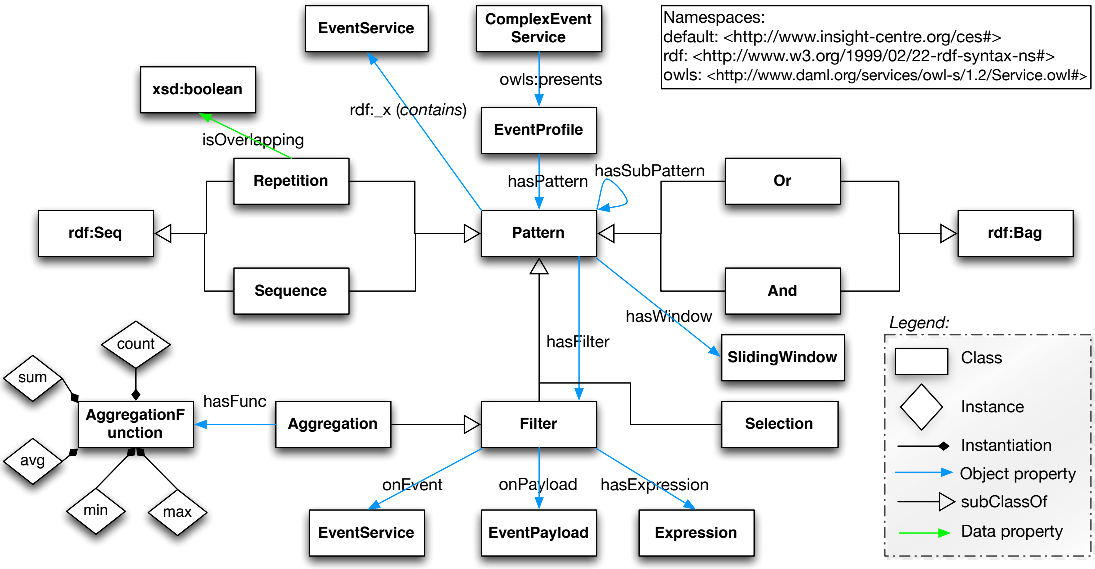


Figure 5.6: Complex Event Service Ontology: event pattern

By comparing the above sub-classes under `EventPattern` to the pattern types identified in Table 3.2, it is evident that the pattern types modelled in CESO covers all the basic types listed in the Table but not all pattern types in each sub-category, e.g., logical operators including exclusive-disjunction and absence are not supported, neither are the spatial and spatiotemporal patterns, or aggregations with attribute-based patterns. Indeed, the focus of this thesis is not providing and developing yet another a comprehensive and highly expressive event pattern language, instead this work aims to prove

[10] Turtle - Terse RDF Triple Language: http://www.w3.org/TeamSubmission/turtle/, last accessed: May, 2015.

```
:IncreasedReplenishmentProfile a ces:EventProfile;
        ces:hasPattern :Pattern_1.

:Pattern_1    a  ces:Or, rdf:Bag;
        rdf:_1 :IncreasePurchaseOrderService;
        rdf:_2 [    a ces:Repetition,rdf:Seq;
                rdf:_1 :OutOfStock_1;
                ces:hasCardinality "10"^^xsd:integer];
        hasWindow [     a owl-time:Interval;
                owl-time:hasDurationDescription
                    [    owl-time:day "7"^^xsd:gDay]].
```

Listing 5.2: CESO annotations for the event pattern in Figure 3.9

that different types of patterns used in existing event pattern languages can be captured in event service descriptions and used in event service discovery and compositions. A more expressive set of event pattern types can be studied in future research.

5.1.4 Event Request

Using CESO, a user can also perform some basic discovery function using SPARQL. Listing 5.3 shows a sample SPARQL query that identifies sensor services by querying the properties they measure. The query results shall contain the service observing `ces:AverageSpeed`, if the `ces:AverageSpeed` is annotated as a sub-class of `ces:Speed`. Besides this simple SPARQL-based discovery, CESO provides the concept of `EventRequest` to model more complicated discovery and composition requests.

```
prefix ces:    <http://www.insight−centre.org/ces#>
prefix ssn:    <http://purl.oclc.org/NET/ssnx/ssn>

SELECT ?sensorService
WHERE { ?sensorService ssn:observes ces:Speed. }
```

Listing 5.3: Simple discovery via SPARQL

An `EventRequest` captures the users' requirements on the event services. It can be seen as an incomplete `EventService` description without concrete service bindings for the member event services. `Constraints` are used to declare users' requirements on the NFPs in `EventRequests` with an `Expression`. `Preferences` are used to specify a weight between 0 to 1 over different quality metrics representing the users' preferences on QoS metrics: higher weight indicates the user cares more on the particular QoS metric. Together the `Constraints`, `Preferences` in `EventRequest` and the `NFPs` in `EventProfile` description are used to address the *Customised Implementation* requirement in Section 1.2.1 in the *Service Description* and *Request Definition* phases. Figure 5.7 shows the ontology for describing event requests.

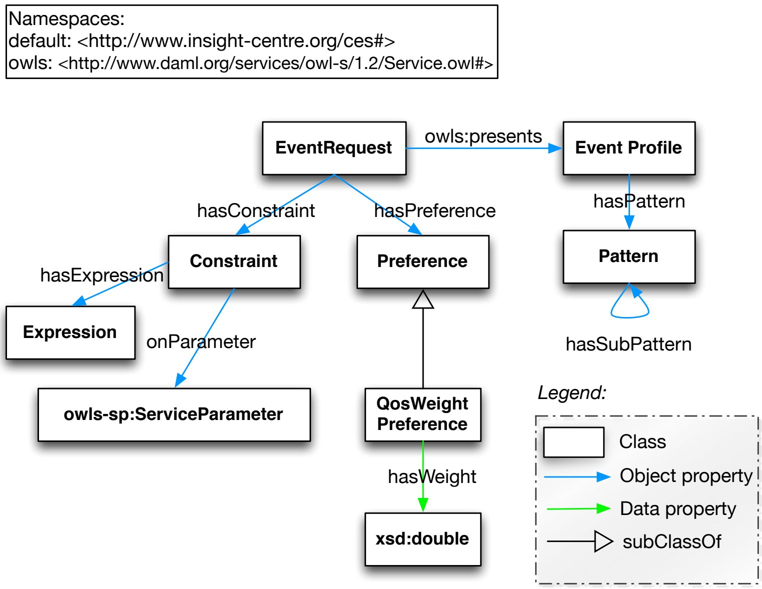


FIGURE 5.7: Complex Event Service Ontology: event request

5.1.5 Traceability between Event Services

Leveraging the sub-pattern property in CES ontology, one can query the provenance relations (within the same `EventProfile`) specified in composition plans (as well as other event patterns) using the query specified in Listing 5.4, with OWL reasoners (augmented with the rule in Listing 5.1). In order to track provenance relations among different event profiles, additional rules in Listing 5.5 must be used.

```
prefix ces:   <http://www.insight−centre.org/ces#>
prefix ssn:   <http://purl.oclc.org/NET/ssnx/ssn>

SELECT ?subpattern
WHERE { :SampleService owls:presents ?sampleProfile.
       ?sampleProfile    ces:hasPattern ?pattern.
       ?pattern    ces:hasSubPattern ?subPattern.  }
```

LISTING 5.4: Tracking pattern provenance via SPARQL

```
[Rule2: (?ep1 ces:hasSubPattern ?s)
         (?s owls:presents ?p)
         (?p ces:hasPattern ?ep2)
         -> (?ep1 ces:hasSubPattern ?ep2)]
```

LISTING 5.5: Entail sub-patterns among different event services.

5.2 Extended Business Event Modeling Notations

In order to facilitate effortless and precise modelling of event patterns described in the CES description, an extension of the graphical event modelling language called Business Event Modelling Notations (BEMN) [3] is made. In this section, BEMN is briefly introduced first. Then, the advantages and disadvantages of the BEMN are discussed before the extended version provided in this thesis is presented, which is called BEMN$^+$. The formal semantics of the event patterns in BEMN$^+$ are also elaborated.

5.2.1 Overview of Business Event Modeling Notation

BEMN can be used to define complex/business events from their member events using event composition models. An event composition model represents the rules and conditions to be evaluated for deriving the complex event. Each event composition model consists of an *event pattern description*, specifying combinations of events that are captured in the complex event, as well as a set of *output event declarations* indicating the complex/business events to be produced as results. Figure 5.8 shows the constructs of BEMN.

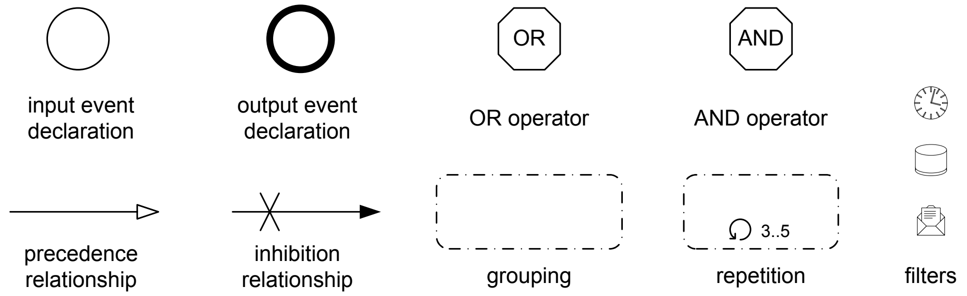


FIGURE 5.8: Business Event Modelling Notation constructs (from [3])

An *event pattern description* describes the combination of events using *input event declarations*, *logical operators*, *temporal relations*, *groupings* and *filters*. *Input event declarations* indicate the member events of the event composition model. *AND* operators and *OR* operators logically relate parts of the event pattern: the member events occurred must match the event patterns specified in all or at least 1 branch connected to an AND or OR operator, respectively. Input event declarations and logical operators are called objects in event pattern descriptions. *Precedence* and *inhibition* relationships specify temporal relations between two objects. Precedence relationships indicate that the source objects must have occurred before the target objects. Inhibition relationships indicate that the source objects must not have occurred before the target objects.

Groupings are sets of objects with additional constraints. *Filters* attached to groupings specify constraints for the input events declarations contained in the grouping. There are different types of filters: time-related, event-data-related, environment-data-related and other filters. Each filter comes with an expression (in natural language) describing the constraint. Filters can also be directly attached to individual event declarations, which is an abbreviation for a grouping only containing one event declaration. Furthermore, groupings can be typed as repetitions. A repetition grouping indicates that the event pattern contained in the grouping must occur multiple times. Finally, input event declarations can be attached to groups ("exception event declarations"). In this case, a corresponding event serves as an inhibitor for the groupings.

A complete description on the syntax, constraints and formal execution semantics can be found in [3]. As part of the execution semantics, core event composition models are introduced as a special class of BEMN models. The idea is that the semantics of such a model can be directly given and that core models represent the unit of execution: Matching of an event rule represented in a core model will result in a single transactional step, while matching of non-core models might involve several steps. Non-core models therefore represent a set of core models. An example of an event composition model in BEMN is shown in Figure 5.9.

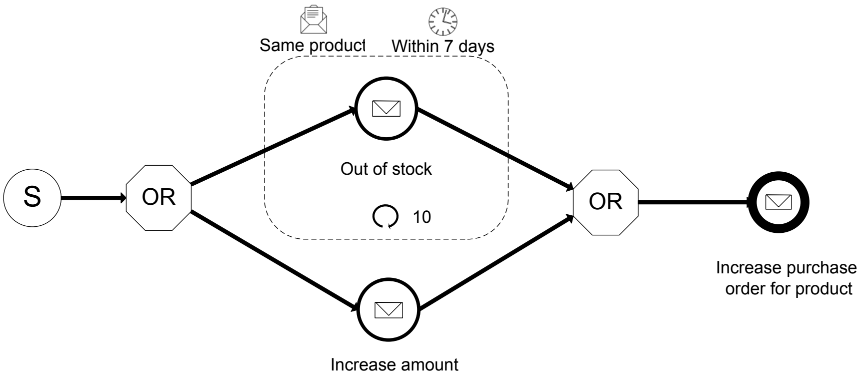


Figure 5.9: Example of event pattern in Business Event Modelling Notations (same semantics as the example in Figure 3.9).

5.2.2 Advantages and Limitations of BEMN

BEMN intend to provide a graphical representation for event compositions beyond conventional textual language, e.g.: Rapide [77]. BEMN diagram can be integrated into BPMN process models seamlessly to facilitate complex event description in business processes. BEMN is able to describe the business event patterns identified in [175] and it is executable because the formal semantics are defined. However, there are some limitations in BEMN, which are elaborated on in the following sections.

5.2.2.1 Flexibility and Query Efficiency

BEMN language did not take into account how to align the execution semantics of the event composition models with stream queries, so that the composition models (and the underlying event patterns) can be executed by existing stream query engines. Instead, event composition models in BEMN are executed by checking 3 conditions defined by the authors: match condition ensuring the temporally ordered occurrences of required event instances, inhibition condition ensuring the absence of inhibitor event instances and filter condition ensuring all filters evaluates to true.

By checking these conditions, BEMN can only support the direct execution of the restrained core composition models. General (non-core) models are translated into core models before execution, which introduces overhead. Moreover, no instructions on how these conditions can be implemented by or adapted to existing CEP/Stream processing engines (potentially equipped with more advanced query optimisation techniques) are provided, which limits the flexibility and efficiency of the approach. In BEMN$^+$, event semantics are designed to be aligned with query semantics in stream reasoning engines, so that query transformation algorithms can be developed and the event patterns specified in BEMN$^+$ can be evaluated by different stream processing engines. The details of the query transformation are elaborated in Section 8.1.

5.2.2.2 Granularity and Semantic Interoperability of Information

BEMN does not specify how an event declaration is structured, e.g., what data does an event contain and how such data are used in filters. This part is left to the programmers who implement the event rules represented by the composition models. In this way, the technical details are hidden from the users targeted by the graphical language (business users), but it will require more effort to implement composition models. To this end, BEMN$^+$ is extended to allow more specific definition on event declarations and filters.

Also, it is unrealistic for the users to create executable event composition models without knowing what the member events really mean. BEMN$^+$ uses semantic technology to help business users discover the primitive events they need (based on sensor capabilities), as well as creating filters upon the event data.

5.2.2.3 Event Instance Selection Policy

BEMN specified the event pattern types and the consumption policies it supports. However, it is not clear what is the selection policy for the event instances triggering/firing an output event instance. More details on the selection policy is discussed in Section 5.2.4.1 and Section 5.2.4.2.

5.2.2.4 Aggregation

Aggregation patterns are important in CEP but not fully supported in BEMN, i.e., only aggregation with temporal pattern (repetition) is supported. BEMN$^+$ also supports aggregation with attribute-based patterns.

5.2.3 BEMN$^+$: the Revised Constructs, Syntax and Constraints

In order to overcome the limitations of the original BEMN language described in the previous section, the language is revised. In the following sections, the revised language constructs, abstract syntax and language constraints are introduced.

5.2.3.1 Language Constructs

BEMN$^+$ reuses most of the graphical notations introduced in BEMN, with additional notations for *event declaration with payload*, *aggregated* grouping and *instance-based window*. Changes on the language constructs in BEMN$^+$ is are depicted in Figure 5.10.

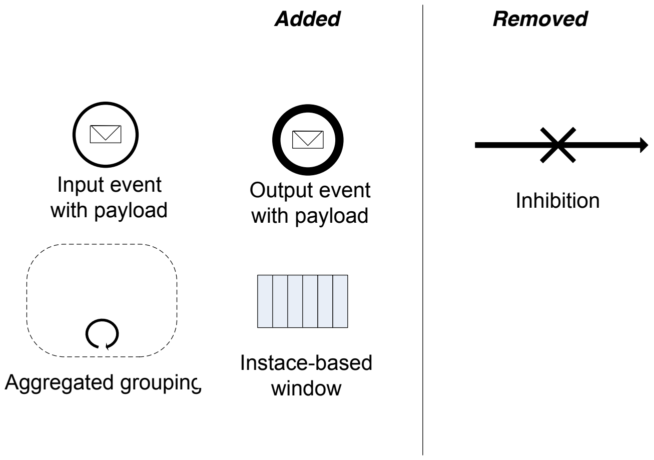


Figure 5.10: Extended Business Event Modelling Notation constructs: changes from BEMN

- **Event declarations with payloads** indicate that one or more event payloads (messages) are needed in the pattern evaluation. When event declarations with payloads are used, users are asked to provide a reference (e.g., URI in a domain ontology) to the payload type (e.g., a sensor observation property type in SSN) and domain of the payload (e.g., a feature-of-interest in SSN).

- **Aggregated Grouping** provides functions to aggregate the information of the event declarations in the group, these functions are similar to the SQL functions including count() and sum() etc. Aggregations will specify an aggregation variable at design time and calculate an aggregation value for it at run time. These aggregation values can serve as payload data for the output event declarations or as inputs for filters.
- **Instance-based window** provides a windowing function based on the number of event instances kept in memory. When used, the user is asked to specify an integer for the number of event instances stored.
- **Inhibition** is removed from BEMN constructs, because it is not supported currently in CESO (see Section 5.1.3 and Section 5.2.4.2 for explanations).

5.2.3.2 Abstract Syntax

Using the CES ontology and the event semantics defined above, an event service provider can describe event services and store these service descriptions in a service repository; an event service consumer can formulate an event service query to specify his requirement on event services. In this section the abstract syntax of event patterns described in the CES ontology is given.

An *Event Declaration* describes a CES without considering the NFPs. It is a tuple

$$E = (src, type, ep, D)$$

where *src* is the service location where the events described by *ed* are hosted, *type* is the term for the domain specific service type, *ep* is the event pattern for E and D is its data payload as a set of event properties sets, e.g., timestamps, event identifier, message contents, etc. E is an output event declaration in BEMN.

An *Event Pattern* describes the detailed semantics of a complex event. It is a tuple

$$ep = (\mathcal{E}, OP, Gr, R, Pr, Sel, F, Pol, W), \text{where:}$$

- $\mathcal{E}$ is a set of member event declarations involved in *ep*, for a member event declaration $E' \in \mathcal{E}$, D' represents the payload of E';

- OP is a set of operators, $op \in OP = (t_{op}, r)$ where $t_{op} \in \{Seq, Or, And, Rep_o, Rep_n\}$ is the type of operator (Rep_o and Rep_n are overlapping and non-overlapping repetitions, respectively), $r \in \mathcal{N}^+$ is the cardinality of repetition, $r > 1$ for repetition operators, and $r = 1$ otherwise;

- $Gr \subseteq \wp(OP \cup \mathcal{E})$ is a set of sets of objects (i.e., operators and event declarations), so called groupings, $gt : Gr \to \{n, r_o, r_n, agg\}$ is a function stating a grouping is a normal one, overlapping repetition, non-overlapping repetition or aggregated grouping;

- $R \subset (OP \times (OP \cup \mathcal{E}))$ is a set of asymmetric and transitive relations on operators and member events, it captures the provenance (i.e., causal) relation within *ep*, $\forall (op, n) \in R$, the execution of the operator node *op* relies on the execution result of another operator node n when $n \in OP$, or the occurrence of an event declaration node n when $n \in \mathcal{E}$;

- $Pr \subset (OP \cup \mathcal{E}) \times (OP \cup \mathcal{E})$ is a set of asymmetric relations on operators and member events, it gives the temporal order (precedence relations) within *ep*, $\forall (n_1, n_2) \in S, \exists n \in OP \wedge (n, n_1), (n, n_2) \in R \wedge n.t_{op} = (Seq|Rep_o|Rep_n)$ where n_1, n_2 are two nodes in *ep*, also, the occurrence of n_1 (if $n_1 \in \mathcal{E}$) or the last member event instance that completes the execution of n_1 (if $n_1 \in OP$) should happen before the occurrence of n_2 (if $n_2 \in \mathcal{E}$) or the first member event instance that completes the execution of n_2 (if $n_2 \in OP$);

- $Sel : \bigcup\limits_{E' \in \mathcal{E}} E'.D' \to D$ is a mapping function that selects the payloads of member events as the payloads of the output event, where D' is the payloads of E';

- F is a set of filters evaluating constraints over event properties in member events (i.e., $\bigcup\limits_{E' \in \mathcal{E}} E'.D')$. A filter $f \in F$ is to be evaluated as true or false at query execution time according to the event property values and the arithmetic expression described in f. $F_{gr} : F \to Gr$ is a function that attaches filters to groupings. $Agg \subseteq F$ is a special set of filters evaluating constraints over multiple occurrences of the properties of the same event type. These occurrences are aggregated by an aggregation function $Func_{agg} \in \{count, sum, avg, min, max\}$.

- *Pol* is the set of event instance selection policies over the input event streams, $Pol(E)$ gives the selection policy on E. $\forall E \in \mathcal{E}, Pol(E) \in \{last, cumulative\}$ where *last* picks only the latest event instances and *cumulative* picks all matching instances.

- W is a set of sliding windows specified for ep over the input event streams, each $w \in W$ is considered as a time duration or a number of events to be kept, $W(E)$ gives the time window on E.

Given the abstract syntax of BEMN$^+$ it is trivial to map relevant elements in BEMN$^+$ to CESO concepts. Appendix B gives an XML serialisation of BEMN$^+$. It is worth noticing that not all elements in the abstract syntax are visible in BEMN$^+$ notations. For example, the selection policy is specified simply as an attribute of the pattern, the selections can be specified as a set of selected event-property pairs. The sequence and repetition operators in OP are not visible as nodes like conjunctive and disjunctive operators, instead they are captured implicitly via the precedence relations and repetition groupings. Also, the provenance relations are not visible in BEMN$^+$.

The provenance relation and the operators can be used in another visual representation of event patterns, which is the *Event Syntax Tree* (EST). ESTs can be constructed by recursively appending operator and event declaration nodes as child/leaf nodes to a root operator node when there is a provenance relation between them. The visual representation of ESTs omits many elements in BEMN$^+$, such as output event declarations, groupings, time window, event payloads and selections. Because of the brevity of ESTs, throughout the thesis ESTs are used to illustrate event patterns. Figure 5.11 illustrates the EST for the product replenishment event pattern described in Figure 3.9. The detailed definition and use for ESTs are postponed to Section 6.1.

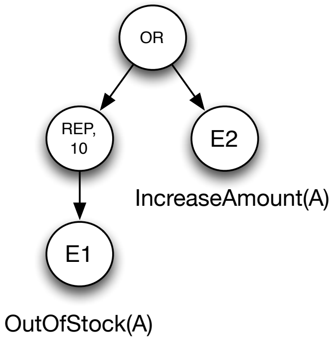


FIGURE 5.11: Event syntax tree for the increased replenishment event (see Figure 3.9)

5.2.3.3 Language Constraints

To ensure the models created by users are valid, a set of constraints are applied to BEMN$^+$. Before presenting the constraints, some basic notions used in the constraints need to be clarified: a temporal relationship is referred to as an incoming relationship for its target and as an outgoing relationship for its source; the auxiliary functions in, out

as defined in the original BEMN are reused: $in,\ out := \mathcal{E} \cup Op \longrightarrow \wp(\mathcal{E} \cup Op)$, where $in(o) = \{x \in \mathcal{E} \cup Op | x\ Pr\ o\}$ and $out(o) = \{x \in \mathcal{E} \cup Op | o\ Pr\ x\}$. The following constraints are to be satisfied by an event composition model:

1. every input event declaration has at most 1 incoming and exactly 1 outgoing relationship and has a path of temporal relationships to the output event declaration, i.e.$\forall e \in \mathcal{E}[|in(e) \leq 1| \wedge |out(e)| = 1 \wedge (e(Pr)^{+}E)]$,
2. every output event declaration has at exactly 1 incoming and exactly 0 outgoing relationships, i.e.$in(E) = 1 \wedge out(E) = 0$,
3. there is only 1 start and output event declaration, i.e. $|E_s| = 1$, $|E| = 1$,
4. every operator has at least 1 incoming and 1 outgoing relationship, i.e.$\forall e \in Op[|in(o)| \geq 1 \wedge |out(o)| \geq 1]$,
5. in each grouping there is at most one object that has an incoming relationship with a source outside the grouping and at most one object that has an outgoing relationship with a target outside the grouping, i.e.$\forall g \in Gr[|\{o \in g | in(o) \backslash g = \emptyset\}| \leq 1 \wedge |\{o \in g | out(o) \backslash g = \emptyset\}| \leq 1]$,
6. if two groupings contain the same events then one grouping must be fully contained in the other, i.e. $\forall g1, g2 \in Gr[g1 \cap g2 = \emptyset \Rightarrow g1 \subset g2 \vee g2 \subset g1]$,
7. Pr is acyclic, i.e.$\nexists(e1, e2) \in\ Pr[(e2, e1) \in\ Pr^{+}]$.

5.2.4 BEMN^{+}: Formal Semantics of Event Pattern

In order to enable automatic complex event pattern evaluation and ensure correct event service composition, it is necessary to define the formal semantics of the event patterns specified in the CESO. In this section the formal semantics are presented. First a meta-model for complex event semantics is introduced. Then, this meta-model is used to compare and analyse the semantics of event patterns (or query semantics) in existing CEP and semantic stream processing approaches, including the designed semantics of event patterns in the BEMN^{+}.

5.2.4.1 Meta-model of Event Semantics

In [62] a meta-model is proposed for defining the formal semantics of complex events, i.e., what does a complex event pattern mean and how to detect this event pattern over an *Event Instance Sequence* (EIS). According to [62] the semantics of complex events can be defined by answering three basic questions: 1) how to use a limited set of operators, constructs and descriptors to specify various complex event types (i.e., complex event patterns) unambiguously, 2) how to determine which subset of the EIS belongs to a complex event type when there are more than one subsets satisfying the constraints specified by the complex event types and 3) whether an event instance can be used in multiple EISs mapping different complex event types. Therefore three basic dimensions for describing event semantics are identified: *Event Type Pattern, Event Instance Selection* and *Event Instance Consumption*, for answering these three questions, respectively. On top of these three basic dimensions, an additional dimension is whether events are considered instantaneous or lasting for an interval. This dimension is called *Event Duration*. The details of each dimension are elaborated below:

- An **Event Duration** can be categorised into instantaneous or interval-based. The fundamental difference between instantaneous and interval-based events is whether 1 or 2 (i.e., start and end) timestamps are necessary for describing an event instance. Also, instantaneous events can be seen as special cases of interval-based events which have identical start and end timestamps.

- The **Event Type Pattern** can be divided into 3 sub-dimensions: *Operators, Coupling* and *Context Condition*. The operators specify temporal constraints over EISs, including binary operators: *Sequence (;), Simultaneous (==), Conjunction* $(\wedge)$*, Disjunction* $(\vee)$, unary operator *Negation* $(\neg)$ and n-ary operator *Repetition*.

  For two event types E_1, E_2, $;(E_1, E_2)$ indicates the timestamps of event instances of type E_1 are older than the timestamps of event instances of type E_2[11]; $==(E_1, E_2)$ indicates the timestamp(s) of the event instances are equal; $\wedge(E_1, E_2)$ and $\vee(E_1, E_2)$ indicate both and at least one of the instances of E_1 and E_2 should occur regardlessly of the temporal order, respectively. It is evident that sequence, simultaneous, conjunctive and disjunctive operators are associative.

  For an event type E_3, $\neg(E_3)$ indicates the absence of instances of E_3. Note that although negation is, in theory, a unary operator, in practise it is normally used within the interval determined by its previous and next operands.

  For n event types $E_1, ..., E_n$, $(;(E_1, ..., E_n))^r$ indicates that the sequence of instances of $E_1, ..., E_n$ must repeat for r times. Repetitions have two modes: *overlapping* and *non-overlapping*, denoted $\wedge(;(E_1, ..., E_n))^r$ and $;(;(E_1, ..., E_n))^r$, respectively. For example, for two event types $E_3 := \wedge(;(E_1, E_2))^2$, $E_4 :=;(;(E_1, E_2))^2$, $EIS_1 : (e_1^1, e_1^2, e_2^1, e_2^2)$ triggers E_3 but not E_4, while $EIS_2 : (e_1^1, e_2^1, e_1^2, e_2^2)$ triggers both E_3 and E_4 (e_i^j is the jth instance of event type type E_i). It is evident that

[11] When considering overlaps for interval-based events the sequence operator can have more variants e.g.: meets, finishes and participates etc. see [184]

overlapping repetition can be transformed into a conjunction of sequences, while the non-overlapping repetition can be transformed into a sequence of sequences. The *Window* operator specifies how many events are to be kept in memory. The length of the window can be specified as a temporal duration or the number of events pertained.

The *Coupling* sub-dimension has two types: *Continuous* and *Non-continuous*, indicating whether an EIS for an event type allows irrelevant event instances. For example, $EIS_3 : (e^1_1, e^1_3, e^1_2)$ can trigger an event pattern $E_5 :=; (E_1, E_2)$ if E_5 is non-continuous. However, if E_5 is continuous, it cannot be triggered by EIS_3.

The *Context* sub-dimension specifies if the event pattern is triggered under conditions on *Environment* (e.g., applications, users, transactions, etc.), *Data* (e.g., event properties, message contents, etc.) or executions of certain *Operations* (e.g., database record insert, delete, etc.)

- **Event Instance Selection** has three modes: *first* and *last* modes pick the oldest and youngest mapping event instances in an EIS respectively. *Cumulative* mode picks all instances in an EIS satisfying the constraints.

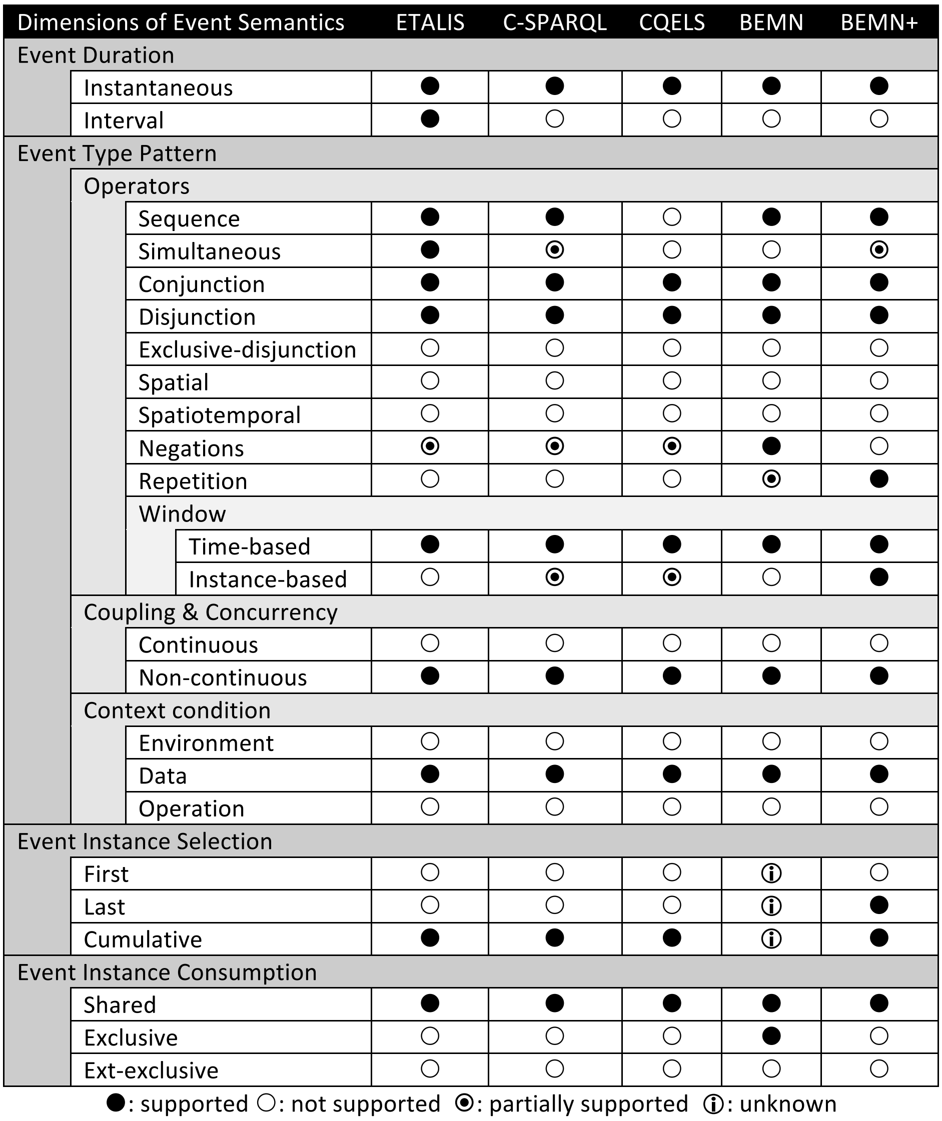
- **Event Instance Consumption** has three modes: *Shared*, *Exclusive* and *Ext-exclusive*. In shared mode all subscriptions can share event instances, i.e., event instances are kept until they expire in the time window. In the exclusive mode the event instances are removed once they are used to trigger an event type. In the ext-exclusive mode when e^j_i is used to trigger E_a, all e^k_i in the EIS before the *terminator* (i.e., last event instance in EIS triggering E_a) are removed.

5.2.4.2 BEMN$^+$ Semantics In Comparison with Existing Approaches

In this section the semantics of BEMN$^+$ event patterns is given in comparison with the event/query semantics in existing CEP/stream processing systems. The dimensions used in the comparison is derived from the complex event meta-model presented in Section 5.2.4.1. In [133] a thorough survey has been conducted on existing *Information Flow Processing* systems, however it does not describe the features of recent semantic stream processing systems. A survey on stream reasoning engines can be found in [161]. In Table 5.1 the event semantics used in ETALIS [19], CSPARQL [38], CQELS [37], BEMN [3] and BEMN$^+$ are compared.

- **Event Duration** All investigated approaches support using instantaneous events, i.e., annotating events and triples with a single timestamp. Only ETALIS fully supports interval-based events, since it allows triples to be annotated with a start

Table 5.1: Comparison of Event Semantics



Dimensions of Event Semantics	ETALIS	C-SPARQL	CQELS	BEMN	BEMN+
Event Duration					
Instantaneous	●	●	●	●	●
Interval	●	○	○	○	○
Event Type Pattern					
Operators					
Sequence	●	●	○	●	●
Simultaneous	●	⦿	○	○	⦿
Conjunction	●	●	●	●	●
Disjunction	●	●	●	●	●
Exclusive-disjunction	○	○	○	○	○
Spatial	○	○	○	○	○
Spatiotemporal	○	○	○	○	○
Negations	⦿	⦿	⦿	●	○
Repetition	○	○	○	⦿	●
Window					
Time-based	●	●	●	●	●
Instance-based	○	⦿	⦿	○	●
Coupling & Concurrency					
Continuous	○	○	○	○	○
Non-continuous	●	●	●	●	●
Context condition					
Environment	○	○	○	○	○
Data	●	●	●	●	●
Operation	○	○	○	○	○
Event Instance Selection					
First	○	○	○	ⓘ	○
Last	○	○	○	ⓘ	●
Cumulative	●	●	●	ⓘ	●
Event Instance Consumption					
Shared	●	●	●	●	●
Exclusive	○	○	○	●	○
Ext-exclusive	○	○	○	○	○

●: supported ○: not supported ⦿: partially supported ⓘ: unknown

and end timestamp. CSPARQL partially supports intervals for complex events, i.e., events consists of multiple triples with different timestamps. To capture the interval for such complex events in CSPARQL one must use the *f:timestamp* function provided by CSPARQL language to retrieve all timestamps and get the oldest and youngest timestamps.

- **Event Type Pattern.** The *Sequence* operator is supported by all investigated approaches except for CQELS. The *Simultaneous* operator is directly supported by ETALIS using the *EqJoin* operator extended from SPARQL *join* and indirectly supported by CSPARQL and BEMN$^+$ by comparing timestamps of events and triples. The *Conjunction* and *Disjunction* operators are supported by all investigated approaches. *Exclusive-disjunction, Spatial* and *Spatiotemporal* patterns identified in Section 3.3.3 are not supported by any investigated approaches. *Negation* is directly supported by BEMN using *Inhibition* and indirectly supported by ETALIS, CSPARQL and CQELS using the combination of *LeftJoin* operator and *bound* filters. Currently, BEMN$^+$ does not support negations as it will introduce complexity in event stream federation, but it is on the agenda of future work. *Repetition* is partially supported in BEMN with only *overlapping* mode, it is fully supported in BEMN$^+$ in both *overlapping* and *non-overlapping* modes. A time-based *Window* operator is supported by all approaches, while an instance-based window is partially supported by CSPARQL and CQELS, since they allow triple-size-based windows. BEMN$^+$ supports both kinds of windows. All approaches support *non-continuous* coupling, i.e., irrelevant events and triples will not affect the results derived from relevant ones. All approaches support context conditions on *data* using filters.

- **Event Instance Selection.** ETALIS, CSPARQL and CQELS support only a *cumulative* event instance selection policy because their language semantics are extended from SPARQL, in which all mapping variable bindings are returned as results. In BEMN, the selection policy is not explicitly explained. BEMN$^+$ is designed to support both *cumulative* and *last* selection, in order to be compatible with existing stream reasoning engines which extends SPARQL semantics (i.e., using cumulative selection), while in some traditional CEP systems, a minimum event instance selection policy (e.g., last pick) is desired due to performance concerns (see section 4.3 in [133]).

- **Event Instance Consumption.** Existing semantic event/data stream engines like ETALIS, CSPARQL and CQELS allow registering multiple queries at the same time. Also, they do not remove triples from the stream unless these triples expire in the window. Therefore, they support only a *shared* event instance consumption mode. BEMN supports *shared* and *exclusive* consumption modes by configuring the event type definitions and subscription scopes. In this thesis, a decentralised system is designed in which queries are evaluated by different event engines on distributed servers and the messages are delivered via publish-subscribe systems, therefore only *shared* event instance consumption is supported in BEMN$^+$.

5.3 Related Work

In this section, the related works in event ontologies and graphical event pattern definition languages are discussed.

Approaches	Differences to CESO
Moser et al. [185]	Only semantic equivalence, subsumption and relations on simple events are described
Li et al. [24]	Temporal operators not supported for complex events
Rinne et al. [160]	Extends Event-F ontology to describe complex events, support event payloads and multiple timestamps, but execution semantics not given.
Liu et al. [82]	Integrate temporal logics with OWL DL to semantically describe complex events, however it needs dedicated reasoning engine to evaluate rules, no integration with existing CEP solutions are provided.

Table 5.2: Related works in complex event ontology

5.3.1 Event Ontologies

Moser et al. [185] elaborate ways to extend syntactical event correlation to semantic event correlations. They consider three kinds of semantic event correlations, i.e., basic semantic correlation to identify semantic equivalence despite syntactical differences, inherited correlation to resemble terminology hierarchies and relation base correlation to define relations between terms.

Li et al. [24] define an event ontology to define event rules, called the Smart Space Event Ontology (SSEO). SSEO captures the causal relations between events and can be used to infer complex situations at runtime. However, temporal-based CEP operators are not modelled in SSEO.

Rinne et al. [160] extend the Event-F ontology to provide specific semantics for complex events. In particular, it supports describing simple and complex event objects as well as their relations. Also, it supports describing event payloads and multiple timestamps. However, it does not provide the formal semantics of the complex events.

Liu et al. [82] extend OWL DL with an Event Description Logic to provide syntax and semantics to define complex events. However, the authors do not elaborate how this ontology can be integrated with existing CEP systems or provide concrete tools to parse and query event ontologies defined in OWL DL. Table 5.2 summarises comparison of the related works to CESO.

5.3.2 Graphical Event Pattern Definition Languages

Besides BEMN, there exist a few works that aim to provide graphical notations for event rules. Sen et al. [47] use graphical notations to model Event-Condition-Action rules (ECA). In their usecase tweets are collected from Twitter as complex events and are decomposed to simple events according to their predefined event schema. Then these simple events are forwarded to the Esper engine where ECA rules are executed.

Karampiperis et al. [33] surveyed some commercial CEP products, which are not open source and can only support either rule-based or query-based language. The authors aim to provide a graphical language that can support both rule and query language and a tool to design event rules using the graphical language. In their current implementation, they can translate the event rules into SQL and query on a static event repository, i.e., no window operators or temporal orders are supported.

Sasa et al. [186] propose an architecture for Complex Event Service implementation using OWL ontologies. A translator is deployed in a CES, which translates event data to instance-level OWL statements, and an OWL reasoner is responsible for combining the semantic event data with the event ontology and inferring if new complex events are detected. However, they used a general purpose reasoner which is not optimised for CEP applications with high throughput, also the temporal-based event reasoning is not elaborated.

Taylor et al. [23] aim to detect events in real time that arise from complex correlations of measurements made by independent sensing devices. This approach transforms the semantic complex event descriptions into EPL statements and executes them on CEP engines. The authors claim this approach is capable of reusing previously configured event streams within the middleware. However, they did not discuss how to reuse the streams outside the scope of the middleware. Also, since the actual event detection is done by a conventional CEP engine, it is not possible to carry out semantic inferencing tasks at runtime.

In SARI [60], event models are composed by event conditions, event patterns and response events. Event conditions specify rules on triggering the detection of a type of event, an event pattern describes the event rules for the event detection, a response event specifies an event generated as output. Event conditions, event patterns and response events are connected by logical operators to describe their logical correlations. The event modelling language in SARI is intended for business users. This approach uses both rule-based systems to evaluate event preconditions and a state-machine based CEP engine (i.e., Esper[12]) to evaluate event patterns. However, there is a functional overlap between event preconditions and event patterns, i.e., both describe situations are define the complex event. Also, although the rule-based part is made service-oriented, the pattern evaluation part is not. As a result, the composition of rule-based event services are possible but only limited to precondition dependencies, i.e., the composition based on temporal orders is not realised.

[12] Esper engine: http://www.espertech.com/products/esper.php, last accessed: Aug, 2015

Approaches	**Differences to BEMN$^+$**
Sen et al. [47]	ECA-based approach, only simple twitter events allowed, Esper engine used for evaluation.
Karampiperis et al. [33]	Translates to SQL statements over static repositories, no window operators supported.
Sasa et al. [186]	Relies on OWL-reasoning, temporal reasoning not elaborated, not optimised for continuous reasoning over data streams with high velocity.
Taylor et al. [23]	Transforms semantic complex events into EPL statements, allows streams to be reused within the middleware, but the approach is not service-oriented, i.e., streams cannot be reused outside the middleware.
SARI [60]	Business-user-oriented, rule based event services, but event service discovery and composition based on temporal orders are not possible.
All above	Only suitable for one particular CEP engine, not compatible with existing BPM tools (except for SARI)

Table 5.3: Related works in graphical event pattern language

The above-mentioned approaches are platform dependent and are difficult to migrate to other CEP systems. Moreover, they can only work on predefined event streams, adding new event streams or alternating existing ones will take considerable development efforts. Furthermore, they do not consider how can to make event models compatible with existing business process modelling technique (except for SARI). The event model used in this chapter extends the work in [3], which can integrate complex event modelling with BPMN. Table 5.3 summarises the comparison of the existing graphical event pattern languages with BEMN$^+$.

5.4 Summary and Discussion

In this chapter, the information model used to define and describe complex events and complex event services are elaborated. A Complex Event Service Ontology (CESO) is introduced to describe Complex Event Services (CES). CESO is an extension of the standardised semantic web service ontology OWL-S and uses similar structures to describe CES profiles (service properties) and groundings (access mechanisms). CESO uses a stream quality ontology to describe the quality aspects of CES and it uses a stream annotation ontology to describe the stream data. The event semantics are annotated as `EventPattern` in CESO. The semantics are aligned to an extended version of the Business Event Modelling Notation, called BEMN$^+$. BEMN$^+$ provides graphical notations for complex event definitions. These notations are similar to Business Process Modelling Notations (BPMN) so that business users can use them with minimal learning overhead. The syntax and semantics of BEMN$^+$ are also introduced. The comparison between the BEMN$^+$ semantics and other RDF stream processing engines are elaborated. Leveraging CESO and BEMN$^+$, the next chapter describes how the event service composition can be realised based on the functional aspects of event services.

Chapter 6

Pattern-based Event Service Discovery and Composition[*]

In the context of Internet-of-Everything (IoE) and Smart City, end users, such as business analysts, city administrators and citizens are typically interested in complex events with business value and/or high-level meanings rather than primitive events representing simple changes of states. Complex Event Processing (CEP) is a suitable technique to detect the high-level events expressed with event patterns. Despite the extensive research on CEP, providing CEP applications as reusable services that allow the composition of complex event services based on event patterns efficiently is still a challenging task. Many existing event service description and discovery mechanisms are topic or content based, which is sufficient for reusing primitive/simple event services. However, it is impossible to reuse a complex event without knowing its exact semantics expressed in the pattern [15].

Providing complex events as services accelerates the implementation of CEP systems, because it allows the event consumers and providers to be loosely coupled in the system [108]. Moreover, by reusing business event services instead of subscribing directly to primitive event streams, the amount of events delivered through the network can be greatly reduced. This is important for CEP applications in general because it reduces the use of bandwidth and CPU, resulting in more efficient and in-time event detection. Take the severe traffic accident event pattern in Figure 2.5 as an example, the event detection task has two sub-tasks: 1) detecting traffic accidents and 2) detecting traffic congestions in the surrounding areas of the accidents. While the first task can be as simple as getting notified when someone calls the police or ambulance service and reports an accident on-site, the second task may require monitoring a set of traffic sensors in

[*] Part of the content in this chapter is published in [167].

the relevant areas and determining whether a congestion has occurred based on the readings. On the other hand, the sub-task of traffic congestion detection could already be implemented as a service available for various scenarios and applications because of its frequent use across domains. In this case, subscribing to the congestion detection service instead of monitoring and calculating sensor readings directly can reduce the event operators used and potentially require fewer messages to be transmitted over the network.

To provide business/complex events as reusable services and facilitate more efficient event processing systems, the following sequence of questions needs to be answered:

1. How to describe event services properly so that event service matchmaking based on event patterns can be realised?
2. How to determine if two event patterns are functionally equivalent (i.e., produce the same complex event notifications), provided that different event patterns may have identical meanings?
3. How to choose optimal event service composition plans that consume the least amount of input event data?
4. How to derive event service compositions efficiently for very complicated event patterns (i.e., with a lot of event rules) and in a large scale event marketplace?

Chapter 5 has answered the first question. This chapter provides answers to the other questions. By doing so, this chapter provides the means for implementing the *Resource Management* component in ACEIS and facilitates discovering and composing event services based on event patterns, as shown in Figure 6.1. When an event request (without quality-of-service constraints and preferences) is received at the *Resource Management* component, it will first query the service metadata store and tries to find direct matching Complex Event Services (CESs) for the event request by comparing event pattern semantics (if the event request contains event pattern, otherwise a conventional type-/attributed based discovery is performed). If no direct matchings are found, it will try to compose a set of Primitive Event Services (PESs) and/or CESs to address the event request, with the help of the event service index.

The remainder of the chapter is organised as follows. Section 6.1 answers the second question by presenting the operations Event Syntax Trees (ESTs) to create canonical forms of event patterns. And then, the semantic equivalence between event patterns can be examined by checking the isomorphism of canonical event patterns. Section 6.2 answers the third question, it provides the definition of *Network Optimised* event

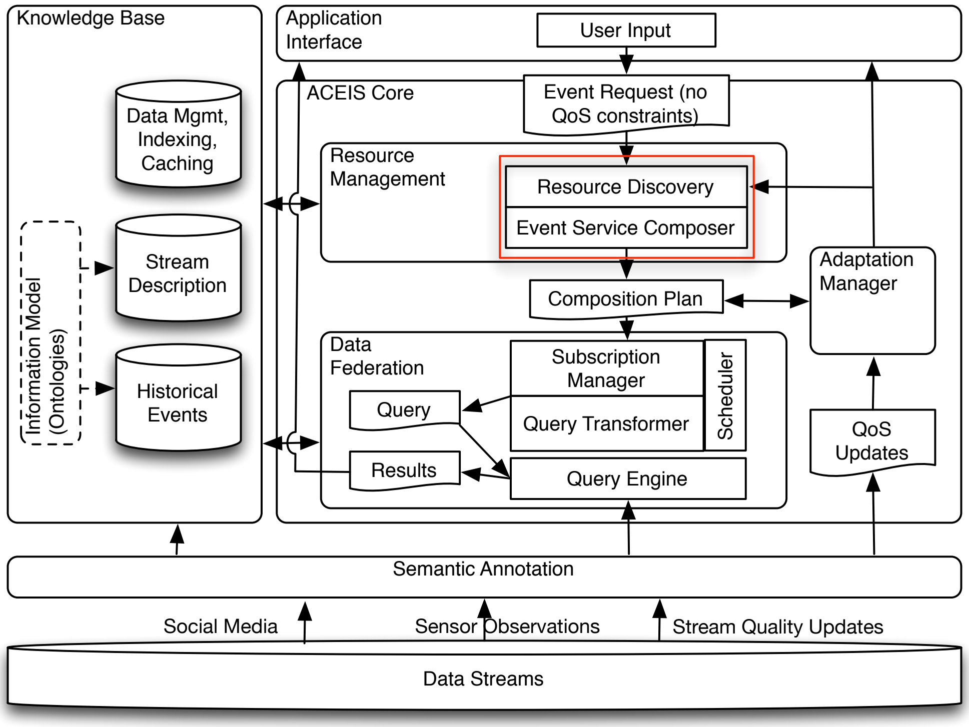


Figure 6.1: The resource management module in ACEIS

service compositions, which have the least Estimated (network) Traffic Demand (ETD). The calculation of ETD is provided as a measure for choosing optimal composition plans. Section 6.2 also addresses the fourth question. It presents two composition algorithms creating network optimised event service compositions: a slow algorithm based on event substitution and a fast algorithm based on the reusability index of event patterns. The reusable relations between event patterns are defined in order to create the reusability index. Section 6.3 demonstrates the performance of the proposed algorithms with experiments. Section 6.4 discusses related work before Section 6.5 summarises.

6.1 Canonical Event Pattern

In Chapter 5, the syntactical and semantic definition of an event pattern used in this thesis are presented. Recall that an event pattern is defined as a tuple consisting of a set of elements including event operators, event declarations etc. Discovery and composition of CESs relies on identifying the semantic equivalence/subsumption relations between event patterns (see Definition 6.1, 6.2). However, it is evident that the semantics of an event pattern cannot be uniquely defined with a tuple, i.e., different combinations of operators and sub-events may yield the same meaning. For example, a conjunctive event type pattern $E_1 := \wedge(E_2, E_3, E_4)$ is semantically equivalent to another event type pattern $E_5 := \wedge(E_6, E_4)$ where $E_6 := \wedge(E_2, E_3)$. In order to compare the semantics of event patterns, a canonical form of the event patterns is needed. In the following the methods for deriving canonical event patterns using operations over Event Syntax Trees (ESTs) are described.

Definition 6.1 (Semantic Equivalence between Event Patterns)**.** Given two event pattern ep, ep', an Event Instance Sequence (EIS) eis, a pattern evaluation function $eva : (P, EIS) \to EIS$, where P is a set of event patterns and EIS is a set of EISs. ep is semantically equivalent to $ep' \iff \forall eis \in EIS, eva(ep, eis) = eva(ep', eis)$, denoted $ep \doteq ep'$.

Definition 6.2 (Semantic Subsumption between Event Patterns)**.** Given two event pattern ep, ep', an EIS eis, a pattern evaluation function $eva : (P, EIS) \to EIS$, where P is a set of event patterns and EIS is a set of EISs. ep semantically subsumes $ep' \iff \forall eis \in EIS, eva(ep, eis) \subseteq eva(ep', eis)$, denoted $ep \succeq ep'$.

6.1.1 Definitions of Event Syntax Tree

An event syntax tree describes an event pattern with a tree. More formally, givent an event pattern $ep = (\mathcal{E}, OP, Gr, R, Pr, Sel, F, Pol, W)$, a syntax tree $T(v) = (V, R, F)$ is generated from ep, where $V = (OP \cup \mathcal{E})$ is the set of vertices representing operators and member events, $v \in V$ is the root node, i.e., $\nexists (v', v) \in R$, typically the root of an EST is an operator, and R is the set of directed edges representing the provenance relation. If $(v_1, v_2) \in R$, v_2 is called a child node of v_1, if $(v_1, v_2) \in R^*$, v_2 is called a descendant of v_1, where R^* is the transitive closure of R. The precedence relation is implicitly given by the left-to-right order of child nodes of sequence and repetition operators: the node to the left precedes the one to the right, i.e., if $(v_1, v_2) \in Pr$, then v_1 is placed to the left of v_2. Each node in V is labeled with its type, repetition cardinality (omitted if $r = 1$) and data payload (if any). F represents a set of filters, the filters in F on a single node v is denoted $F(v)$, which are attached to the node labeled by the payload ($F(v) = \emptyset$ if no filters are attached to v). A filter on two or more nodes (e.g., an aggregated filter) is attached to the lowest common ancestor (LCA) of the nodes. A mapping function $est : P \to T$ maps the set of event pattern P to their generated ESTs T.

Two ESTs are isomorphic if they have the same structure and each node/filter in a tree has a functional equivalent counterpart in the other tree at the same location. More formally:

Definition 6.3 (Isomorphic EST)**.** Given $t = (V, R, F), t' = (V', R', F')$, t and t' are isomorphic (denoted $isomorphic(t, t')$) $\iff$ there exists a bijective mapping $map : V \to V'$ such that $\forall v \in V, v \doteq map(v) \wedge \forall F(v) \in F, F(v) = F'(map(v)) \wedge \forall (v_1, v_2) \in R, (map(v_1), map(v_2)) \in R'$, where $v_1 \doteq v_2 \iff v_1, v_2$ are operators of the same type and cardinality or v_1, v_2 are event declarations providing the same kind of events, i.e., both primitive events with the same event type or both complex events with semantically equivalent event pattern semantics.

Definition 6.4 (Isomorphic event pattern)**.** Two event patterns are isomorphic (denoted $isomorphic(p, p')$) if and only if they produce isomorphic ESTs and there is a set of bijective mapping functions similar to *map* correlates selections, selection policies and windows in p to a functional equivalent counterpart in p'.

Lemma 6.5. *$isomorphic(p, p') \implies p \doteq p'$, where p, p' are event patterns.*

Given an EST $t = (V, R, F) \in T$, $v_l \in V$ is a leaf node *iff* $\nexists (v_l, v') \in R$. The set of leaf nodes V_l of t is given by a function $leaves : T \rightarrow \mathcal{V}$ where $\mathcal{V}$ is the set of all vertices. The root of t can be given by $root : T \rightarrow \mathcal{V}$; the *depth* of a node is the number of edges connecting the node to the root, denoted $depth(t)$; the *height* of a tree is the maximum depth of its leaves, denoted $height(t)$; the *degree* of a node v in t is the number of its child nodes, denoted $degree(v, t)$. Given a sequence or repetition operator v in an EST t and the set of node v's child nodes $child(v, t)$, a child node $v' \in child(v, t)$ is the head *iff* $\nexists v''$ is to the left of v', denoted $head(child(v, t))$. Similarly, a child node $v' \in child(v, t)$ is the tail *iff* $\nexists v''$ is to the right of v', denoted $tail(child(v, t))$

Given two ESTs $t = (V, R, F), t' = (V', R', F')$, t' is a sub-tree of $t \iff V' \subseteq V \wedge R' \subseteq R \wedge F' \subseteq F$, denoted $t' \subseteq t$. t' is a Direct Sub-Tree (DST) of $t \iff$

- $t' \subseteq t$ (t' is a sub-tree of t), and
- $(v, v') \in R$ where v, v' are the roots for t, t', respectively, and
- $\forall v'' \in V'' \mid (v, v'') \in R^* \implies v'' \in V'$ (t' contains all descendants of v in t, i.e., V''), and
- $r' \in R' \iff r' \in (R \cap (V'' \times V''))$ (t' contains all and only provenance relations on nodes in V''), and
- $f' \in F' \iff f' \in F(V'')$ where $F(V'')$ is the set of filters over nodes in V'' (t' contains all and only filters on V'').

The set of DSTs of an EST t is denoted $DST(t)$. Based on the above definitions for sub-trees and DSTs, it is evident that $ep \preceq ep' \implies est(ep)$ is a sub-tree of $est(ep')$ the definitions for a Direct Sub-Event-Pattern (DSEP) are provided in Definition 6.6.

Definition 6.6 (Direct Sub-Event-Pattern)**.** Given event patterns ep and ep', ep is a direct sub-event-pattern of $ep' \iff ep \preceq ep' \wedge est(ep) \in DST(est(ep'))$

6.1.2 Complete Event Pattern

When a leaf node (an event declaration) in an EST contains an event pattern, it means the event service represented by the leaf node delivers non-primitive events. Such a leaf node is called a *complex* leaf, otherwise it is called a *primitive* leaf. A complex leaf can expand into an EST to reveal the event pattern of its own. An event pattern is *complete iff* all the leaf nodes in the generated EST are primitive, and such an EST is called a complete EST. A formal definition for complete event patterns are given in Definition 6.7.

Definition 6.7 (Complete Event Pattern)**.** An event pattern *ep* is a complete event pattern $\iff \forall v \in leaves(est(ep)), v$ is primitive

By checking recursively the event pattern definitions of the leaves, it is trivial to build complete event patterns. A complete event pattern has complete information on the logical rules specified for a complex event. A pattern completion function $f_{complete}$ that creates the complete event patterns is defined as follows.

Definition 6.8 (Pattern Completion Function)**.** $f_{complete} : P \longrightarrow P_c$, P, P_c are sets of event patterns . $p \in P, p_c \in P_c \land p_c = f_{complete}(p) \iff p_c$ is a complete event pattern.

The $f_{complete}$ function is implemented as Algorithm 1.

Lemma 6.9 (Correctness of pattern completion)**.** *The pattern completion function does not alter the semantics of event patterns, i.e.,* $p \doteq p' \iff f_{complete}(p) \doteq f_{complete}(p')$*, where* $p, p' \in P$ *are two event patterns.*

6.1.3 Irreducible Event Pattern

Complete event patterns are still not sufficient for event pattern discovery and composition, because an event pattern can add redundant operators without altering its semantics. As such, a minimal representation is needed for describing an event pattern without redundant operators. Informally, an event pattern is *irreducible* if it contains the least number of nodes and edges while expressing the same semantics. More formally, it is defined as:

Definition 6.10 (Irreducible Event Pattern)**.** An event pattern p is an irreducible event pattern $\iff \nexists p' \mid p'$ is semantically equivalent to $p \land |OP' \cup \mathcal{E}'| < |OP \cup \mathcal{E}|$, where $OP' \cup \mathcal{E}'$ and $OP \cup \mathcal{E}$ are the nodes in $est(p')$ and $est(p)$, respectively.

Algorithm 1 Creates a complete event pattern.

```
Require: Original event pattern $p = (\mathcal{E}, OP, Gr, R, Pr, Sel, F, Pol, W)$.
Ensure: Complete event pattern $p_c$.
procedure COMPLETE($p$)
    $p_c \leftarrow \emptyset$
    $V_l \leftarrow$ LEAVES(est(p))
    for $v \in V_l$ do
        $p \leftarrow$ EXPAND($v, p$)
    end for
return $p_c \leftarrow p$
end procedure
Require: Original event pattern $p$, leaf node to expand $v = (src', type', p', D')$, where
    $p' = (\mathcal{E}', OP', Gr', R', Pr', Sel', F', Pol', W')$.
Ensure: Expanded event pattern $p_c$.
procedure EXPAND($p, v$)
    $p_c \leftarrow \emptyset$
    if $p' = nil$ then return $p_c \leftarrow p$
    else
        expand node $v$ with its sub-pattern $p'$ and add it to $p_c$
return $p_c$
    end if
end procedure
```

The reduction process of event patterns can be intuitively shown in changes in the ESTs derived from the patterns. Two types of reduction operations are considered over the ESTs to create irreducible event patterns: lift and merge. *Lifting* "vertically" removes the redundant event operators, resulting in a lower height of the tree. *Merging* "horizontally" removes overlapping event operators or declarations, resulting in a less degree of the nodes in the tree[1]. Different rules apply when performing lift and merge operations on different types of nodes. Examples of lifting and merging operations are shown in Figure 6.2. In the following the descriptions of the reduction operations are presented informally. The formal definition for sequential lift is provided as an example, definitions for other reduction operations are given in a similar fashion.

- **Sequential Lift:** when a node and its child are both sequence operators, the child node can be removed. Incoming edges on the child node are removed while all outgoing edges will attach their sources to the parent node. Filters on the lifted child nodes are relocated to the parent node. More formally, sequential lift is a function $lift_s : (V_{seq}, V_{seq}, P) \rightarrow P$ where P is the set of event patterns and V_{seq} is the set of sequence operators. Given $p = (\mathcal{E}, OP, Gr, R, Pr, Sel, F, Pol, W), p' =$

[1] The sequential and repetition merge exemplified in Figure 6.2 do not decrease the total number of nodes in the tree because they only merge two nodes, for conformance reasons such merging are still considered necessary to create irreducible trees.

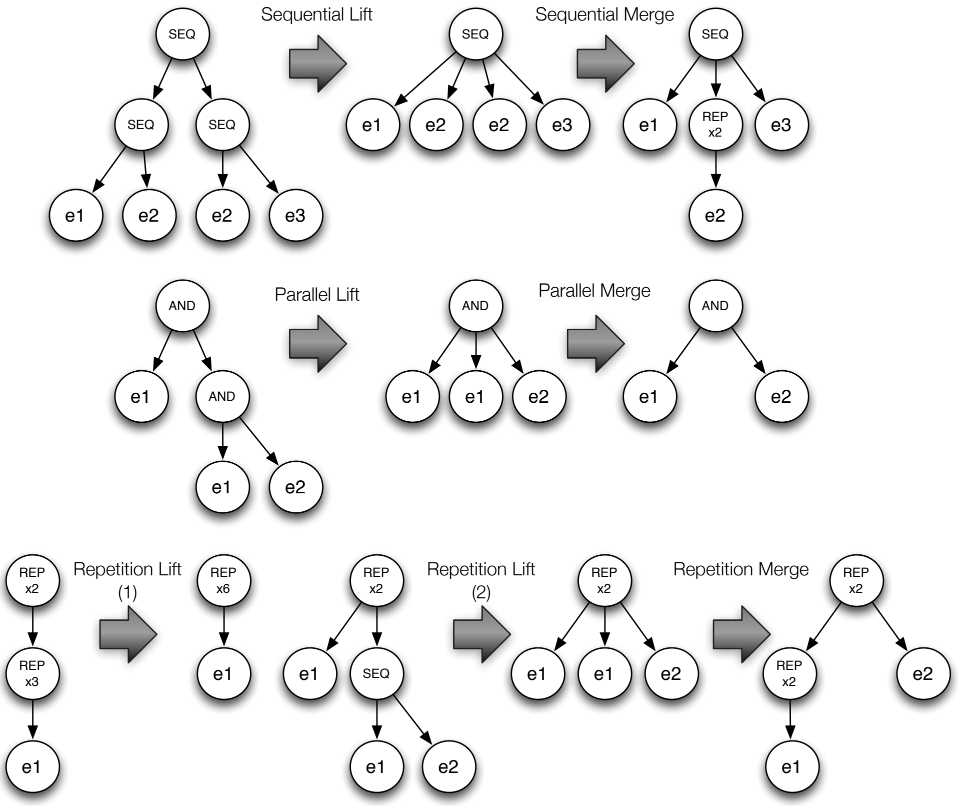


FIGURE 6.2: Examples of syntax tree reduction operations

$(\mathcal{E}, OP', Gr, R', Pr', Sel, F', Pol, W)$,

$$\begin{aligned}
p' = \mathit{lift}_s(v_1, v_2, p) \iff\ & (v_1, v_2) \in R \\
\wedge\ & OP' = OP \backslash v_2 \\
\wedge\ & R' = (R \cup \bigcup_{(v_2,v_3)\in R} (v_1, v_3)) \backslash ((v_1, v_2) \cup \bigcup_{(v_2,v_3)\in R} (v_2, v_3)) \\
\wedge\ & Pr' = Pr \cup (v_4, head(child(v_2, est(p)))) \cup \\
& (tail(child(v_2, est(p))), v_5) \backslash ((v_4, v_2) \cup (v_2, v_5)) \\
\wedge\ & F' = replaceElement(v_2, v_1, F)
\end{aligned}$$

where R is the provenance relations in p and $(v_4, v_2), (v_2, v_5) \in Pr$.

- **Sequential Merge:** when a node is a sequence operator and there is a repeating sequence in its child nodes (recurring primitive event or DST sequences), a non-overlapping repetition node is inserted as a child node of the sequence operator. Repeated sequences are merged into one and relocated under the inserted repetition node. The cardinality of the repetition node is determined by the number of occurrences of the sequence.

- **Parallel Lift:** when a node and its child are the same type of parallel operator (conjunction or alternation), the child node can be removed (as the sequential lift).

- **Parallel Merge:** when child nodes of a parallel operator have duplicates (recurring primitive events or DSTs), duplications are removed. When the only differences of two child nodes n_1, n_2 are the filters attached, and each filter in v_1 is *covered*[2] by the corresponding filter in v_2, then these two nodes (or DSTs) can be merged. For conjunction operators in this case, v_1 ($T(v_1)$) is kept, for disjunctive operators, v_2 ($T(v_2)$) is kept. Additionally, there is a special case for conjunctional merge: when a conjunction operator has two repetition DSTs with only different cardinalities, the DST with less cardinality is removed.

- **Repetitional Lift:** when a node is a repetition operator with cardinality n, and it has only one child node which is also a repetition of the same type (overlapping or non-overlapping) with cardinality m, the child node is removed and the cardinality of the parent node is changed into $n \times m$. Otherwise, if the child node is a sequence operator, the child node is removed.

- **Repetitional Merge:** merging operation for repetition nodes is the same as a sequential merge.

- **Special Lift:** when a sequence or parallel operator has only one child, this operator is removed. Such situations only happen during the reduction process.

Lemma 6.11 (Correctness of Atomic Pattern Reduction Operations)**.** *Pattern reduction operations $f_{atomicreduce} : (P, V) \to P$ do not alter the semantics of the patterns, where P is the set of event patterns and V is the set of removed nodes, i.e., $\forall v \in OP \cup \mathcal{E}, p \doteq f_{atomicreduce}(p, v)$, where OP and $\mathcal{E}$ are the operators and event declarations in p, respectively.*

Proof. The lifting operations do not alter the semantics because the sequence, conjunctive and disjunctive operations are associative. The correctness of merging operations are implied by the definitions of operator semantics (see Section 5.2.4.1 event type patterns)). □

6.1.4 Syntax Tree Reduction Algorithm

The algorithm to create irreducible syntax trees is shown in Algorithm 2. The algorithm traverses a syntax tree from the bottom to the top. The algorithm starts with lifting the whole tree to remove redundant operators. Then, it tries to merge sub-trees on the maximum depth, i.e., sub-trees whose root depths are equal to the height of the whole tree minus one. If these sub-trees are merged, the algorithm checks if they can be lifted

[2] f_1 covers $f_2 \iff P(f_1) \supseteq P(f_2)$, where $P(f_1), P(f_2)$ are the notifications produced by filters f_1, f_2, respectively.

again because merging could create further redundant operators. After merging and lifting all sub-trees on same depth, the algorithm decreases the depth and repeats the merging and lifting process until the whole tree is merged (and possibly lifted again).

Algorithm 2 Creates an irreducible syntax tree from a complete syntax tree ST.

```
Require: Event Syntax Tree $est$.
 1: procedure REDUCE($est$)
 2:     $height$ = getHeight($est$)
 3:     if $height < 1$ then
 4:         exit
 5:     end if
 6:     $root \leftarrow$ getRoot($est$)
 7:     LIFTTREE($root, est$)
 8:     for $height - 1 \rightarrow rootDepth \rightarrow 0$ do
 9:         $nodesToMerge \leftarrow$ getNodesByDepth($est, rootDepth$)
10:         for $node \in nodesToMerge$ do
11:             MERGE($node, est$)
12:             if $merged$ then
13:                 LIFTTREE($node, est$)
14:             end if
15:         end for
16:     end for
17: end procedure
```

In the algorithm, line 2 uses the method *getHeight* to compute the height (maximum maximal depth) of a syntax tree. Line 9 uses the method *getSubTreesByDepth* to retrieve all sub-trees within a syntax tree whose root is of a certain depth. The *merge* method used in Line 11 merges the DSTs of a certain node. The *liftTree* method in Line 7 and 13 carries out the lifting operations on a sub-tree.

Using the above algorithm the f_{reduce} function that produces irreducible patterns is defined as follows:

Definition 6.12. $f_{reduce} : P \longrightarrow P_i$, P, P_i are sets of event patterns. $p \in P, p_i \in P_i \wedge p_c = f_{reduce}(p) \iff p_c$ is an irreducible event pattern.

Lemma 6.13 (Correctness of Irreducible Pattern Creation)**.** *The irreducible event patterns do not alter the semantics of the original pattern, i.e.,* $p \doteq p' \iff f_{reduce}(p) \doteq f_{reduce}(p')$.

Proof. According to Algorithm 2, f_{reduce} is a composition of $f_{atomicreduce}$, i.e., $f_{reduce}(p) = f^{*}_{atomicreduce}(p)$, and by the correctness of the atomic pattern reduction (Lemma 6.11), the composite reduction is also correct. □

The canonical patterns are thus defined as a composition of pattern completion and pattern reduction as follows:

Definition 6.14. $f_{canonical} = f_{reduce} \circ f_{complete}$

Lemma 6.15 (Uniqueness of Canonical Pattern)**.** *two canonical event patterns are semantically equivalent if and only if they are isomorphic, i.e.,* $f_{canonical}(p_1) \doteq f_{canonical}(p_2) \iff isomorphic(f_{canonical}(p_1), f_{canonical}(p_2))$.

Proof.

Necessity: provided by Lemma 6.5.

Sufficiency:

suppose $f_{canonical}(p_1) \doteq f_{canonical}(p_2) \quad \not\Longrightarrow \quad isomorphic(f_{canonical}(p_1), f_{canonical}(p_2))$, then $\exists p_1, p_2 \mid f_{canonical}(p_1) \doteq f_{canonical}(p_2) \wedge \neg isomorphic(f_{canonical}(p_1), f_{canonical}(p_2))$.

By Definition 6.4:

$\neg isomorphic(f_{canonical}(p_1), f_{canonical}(p_2)) \implies$

$\neg isomorphic(est(f_{canonical}(p_1)), est(f_{canonical}(p_2)))$ `(case 1)`

$\vee (isomorphic(est(f_{canonical}(p_1)), est(f_{canonical}(p_2)))$

$\wedge f_{canonical}(p_1), f_{canonical}(p_2)$ has different windows, selections or selection policies) `(case 2)`

It is evident that case 2 contradicts $f_{canonical}(p_1) \doteq f_{canonical}(p_2)$, in the following case 1 is discussed. By Definition 6.3:

$\neg isomorphic(est(f_{canonical}(p_1)), est(f_{canonical}(p_2))) \implies$

$\nexists$ bijective mapping $map_v : V_1 \to V_2$ such that $\forall v_1 \in V_1, v_1 \doteq map_v(v_1)$ `(case 1.1)`

$\vee \exists map_v \wedge \exists v_1, v_2 \in V_1$ such that $(v_1, v_2) \in R_1 \wedge (map_v(v_1), map_v(v_2)) \notin R_2$ `(case 1.2)`

$\vee \exists map_v \wedge \exists v \in V_1, F_1(v) \in F_1, F_1(v) \neq F_2(map_v(v))$ `(case 1.3)`

If case 1.2 or case 1.3 holds, it contradicts $f_{canonical}(p_1) \doteq f_{canonical}(p_2)$ (same set of event operators and primitive events correlated by different temporal/logical relations or using different filters).

Suppose case 1.1 holds. If $|V_1| \neq |V_2|$, without loss of generality assume $|V_1| < |V_2| \implies f_{canonical}(p_2)$ is not irreduciable (Definition 6.10), which contradicts the definition of canonical patterns (Definition 6.12, 6.14). Otherwise, if $|V_1| = |V_2| \wedge \nexists map_v \implies |\mathcal{E}_1| \neq |\mathcal{E}_2|$, since p_1, p_2 are complete and irreducible, $f_{canonical}(p_1)$, $f_{canonical}(p_2)$ uses different primitive events or different temporal/logical relations are defined over primitive events, which contradicts $f_{canonical}(p_1) \doteq f_{canonical}(p_2)$.

□

Theorem 6.16 (Semantic Equivalence of Isomorphic Canonical Event Patterns)**.** *Two event patterns are semantically equivalent if and only if their canonical patterns are isomorphic, i.e.,* $p \doteq p' \iff isomorphic(f_{canonical}(p), f_{canonical}(p'))$.

Proof.

$$\begin{aligned} p \doteq p' &\iff f_{complete}(p) \doteq f_{complete}(p') && \text{(Lemma 6.9)} \\ &\iff f_{reduce}(f_{complete}(p)) \doteq f_{reduce}(f_{complete}(p')) && \text{(Lemma 6.13)} \\ &\iff f_{canonical}(p) \doteq f_{canonical}(p') && \text{(Definition 6.14)} \\ &\iff isomorphic(f_{canonical}(p), f_{canonical}(p')) && \text{(Lemma 6.15)} \end{aligned}$$

□

6.2 Event Pattern Discovery and Composition

With the capability of deriving canonical event patterns, it is possible to carry out CES discovery and composition. CES discovery finds a semantically equivalent CES for the queried event pattern based on finding isomorphic canonical patterns, while CES composition produces a set of event patterns as *composition plans* for the query.

A query pattern contains a complete EST created by complex event designer/modeller. It is assumed all the primitive events in a query have user-defined event types in their event declarations, without specifications of the event service groundings. The mission of event composition is to find out where should these primitive events come from. Of course, the mission can be accomplished by simply discovering PESs using the event types and then filling the source locations for the primitive event declarations in the query, but that will demand a lot of data traffic from the primitive event services. Therefore, it is necessary to reuse complex event services as well.

When a complex event service is reused, an appropriate portion (sub-tree or part of sub-tree) of the query EST is replaced with the event declaration of the complex event service, which transforms the portion into an event declaration node (a leaf node) with a complex event type and a service location. When all the leave nodes of a query have such type and location information, the query is said to be *bound*. When the query is bound, it is used as a composition plan. Apparently, if all leaves of a query can have at least one mapping primitive event service, a composition plan can be found. Otherwise, the composition will fail due to unable to fulfil the functional requirements. An example of a composition plan created with a query and a set of event services is shown in Figure 6.3. When the composition plan is generated, it can be implemented by transforming the plan into an event/stream query and get executed by event/stream processing engines.

The algorithms for event pattern composition have the following assumptions:

1. all events are instantaneous, which means each event has only one timestamp. In a complex event, the timestamp of the last detected member event is used as its timestamp;

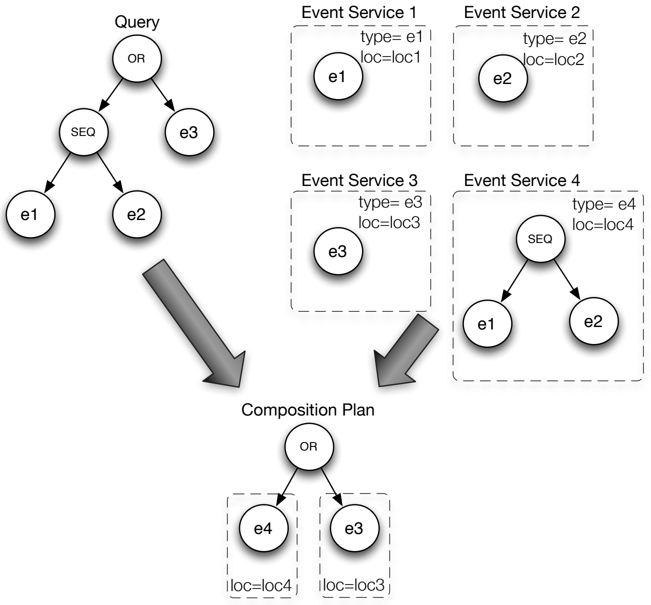


FIGURE 6.3: Example of a composition plan

2. all events delivered by event services are error-free, synchronised and complete;

3. all events have similar payload size;

4. in general, complex events are less frequently detected than their member events.[3]

The first two assumptions draw the scope for the discussion: only instantaneous events (while an event with a duration can be seen as a sequence of two instantaneous start and end events) are dealt with. The third and fourth assumptions allow using a heuristic for achieving the goal of reusing event patterns: to minimise traffic, a complex event service composition should contain as few as possible of the member event services, meanwhile, it should choose more coarse-grained member events. An event composition plan is said to be *network optimised* when it demands the least amount of data traffic over the network. Also, consuming the least amount of events will reduce the computation resource required for a query. Here, we only consider the optimisation from an event consumer's perspective, i.e., the composition plan created for the user should be optimised to produce results for a user efficiently. While this may lead to overloaded subscriptions for a set of service providers, this effect not discussed in this chapter.

In the following, first the method for determining whether a composition plan is network optimised is given. Then, a slow algorithm which derives optimised event compositions is presented. It traverses top-down in the query tree to find *substitutes* for its sub-trees. Finally, a fast event composition algorithm based on the event pattern *reusability index*

[3] This assumption does *not* hold for alternation event patterns and their member events.

is developed. The abstract workflows of the two composition algorithms are shown in Figure 6.4

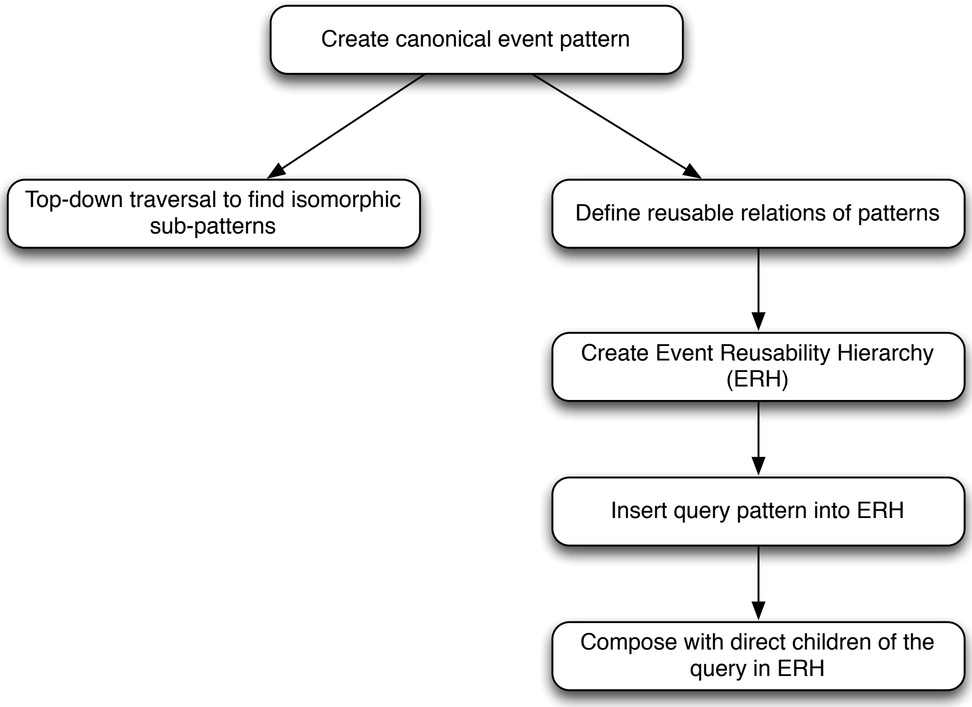

Create canonical event pattern
Top-down traversal to find isomorphic sub-patterns
Define reusable relations of patterns
Create Event Reusability Hierarchy (ERH)
Insert query pattern into ERH
Compose with direct children of the query in ERH

FIGURE 6.4: Workflow of pattern-based event service composition

6.2.1 Optimisation based on Network Traffic Estimation

Intuitively, to determine whether a composition plan is network optimised, the number of member event notifications consumed per unit time needs to be calculated. The simplest case is that the composition plan uses only PESs with pre-described frequencies, e.g., sensor sampling rates, then by summing up their static frequencies, the messages consumption rate can be derived. Otherwise, if all the member event services are up and running and they provide statistics on the frequencies of event notifications, the traffic demand of each composition can be derived by summing up the latest member event service frequencies. However, in realistic scenarios, one cannot assume all event services provide such frequency monitoring operations. Even if they do, there are cases when a user needs to deploy a batch of complex event services, in which some services may be used in others' compositions and they do not have any statistics on their frequencies. Therefore, the ability to estimate the traffic demands and notification frequencies of complex events is necessary.

Given an event declaration $E = (src, t, ep, D)$ and the EST $T_c(v) = est(f_{complete}(ep))$ where $v \in V$ is the root node, $\nu(n)$ denotes the frequency estimation of the member event represented by the sub tree $T_c(n) \subseteq T_c(v)$. Obviously, $\nu(v)$ is the frequency estimation of event described by *ed*. The traffic demand of *ep* is denoted $\mathit{Traffic}(ep) = \sum \nu(n)$ where $T_c(n)$ is the complete EST of a member event service directly used in the composition of *ep*. Given node $n \in V$, $m \in V'$ where V' is the set of child nodes of n, the relation of $\nu(n)$ and $\nu(m)$ is given by Equation 6.1.

$$\nu(n) \begin{cases} = freq(n) & \text{if n is a primitive event} \\ & \text{then its frequency is given} \\ & \text{directly by } freq(n) \\ = \sum \nu(m) & \mathit{type}(n) = \mathit{Or} \\ = \min\{\nu(m)\} & \mathit{type}(n) = \mathit{And} \\ \leq \min\{\nu(m)\} & \mathit{type}(n) = \mathit{Seq} \\ \leq \dfrac{\min\{\nu(m)\}}{r} & \mathit{type}(n) = \mathit{Rep}, r = \mathit{card}(n) \end{cases} \tag{6.1}$$

In the above equation, the *freq* function gives the frequency of a PES directly. The *type* function identifies the operator type for a node and the *card* function gives the cardinality of a repetition.

The equation calculates the maximum estimated frequencies for a set of member event services ($\max\{\nu(n)\}$), with which the maximum traffic demand estimation for an event composition plan that directly uses these services can be derived. Then by choosing the plans with the minimal estimation, the network optimised composition plans can be selected. However, there is a limitation of Equation 6.1: filters are not considered. Indeed, filters may have a strong impact on the frequency. Unfortunately, it is impossible to estimate the impact without knowing beforehand the value range of data payloads and their distributions over the range.

6.2.2 Event Pattern Composition based on Substitution

Based on the definitions of event pattern semantics and ESTs, the definition for the *substitute* relation between event patterns is provided as Definition 6.17.

Definition 6.17. *substitute* $\subset P \times P$ where P is a set of event patterns. $\mathit{substitute}(p_1, p_2)$ holds for $p_1, p_2 \in P \iff \mathit{isomorphic}(f_{canonical}(p_1) = f_{canonical}(p_2))$.

From Definition 6.16, if an event pattern is a substitute of another, they are semantically equivalent and can be seen as exact matches for each other during event service discovery. Intuitively, to create an event composition, a top-down approach that finds substitutes for the event pattern (or its sub-patterns) is necessary.

6.2.2.1 Substitution based Event Composition

The top-down event composition algorithm based on substitution (Algorithm 3) traverses a query tree from the root node to the leaves to find substitutes for sub-trees or different partitions of sub-trees.

The *getSubstitutes* method in line 2 is a key operation in Algorithm 3, it retrieves the complex event declarations whose patterns are substitutes to the query. The algorithm first tries to find an identical tree from a list of candidate canonical trees for the whole query tree. If there is a match, it will replace the query tree with the matching event declaration node.

When there's no direct match for a query, the algorithm tries to find substitutes for sub-trees (or sub-tree partitions) of the query. If the root node of the query is a repetition operator, it will first change the cardinality of the operator to its factors (starting from the biggest factor) and try to find substitutes for all factors (including 1, which makes the repetition a sequence), if it failed, the algorithm is recursively invoked for each DST of the root.

If the root is a sequence, conjunction or disjunction operator, the algorithm will create a set of non-overlapping partitions (using the *getDSTPs* method in line 34, an example of the operation is illustrated in Figure 6.5) with its DSTs. Then, the algorithm will try to find substitutes for each part in each DST partitions. Once all substitutes for a partition are found, the algorithm adds the composition plan for the partition to a list. After all possible partitions are investigated, the composition plan with the lowest traffic demand in the list will be picked (line 42) and corresponding replacements are made. If no partitions have complete substitutions, the composition algorithm is invoked on each DSTs of the root node.

6.2.2.2 Complexity Analysis

Algorithm 3 guarantees the creation of network optimised composition plans because all sub-trees and possible partitions of sub-trees are examined. However, it comes with the price of very high time complexity. The basic operation of the algorithm is the *getSubstitutes* method, which checks the *graph isomorphism* between a query and a

Algorithm 3 Creates optimal composition plans.

```
Require: Query Tree: $ST$, Candidate Trees: $cand$ Query Root: $root$.
procedure COMPOSE($ST$, $cand$, $root$)
    $matchingED \leftarrow$ getSubstitutes($ST, cand$)
    if $matchingED \neq \emptyset$ then
        replacePattern($ST, matchingED, root$)
    else if $root.type$ = REPETITION then
        $hasReplacement \leftarrow false$
        for $f \in$ getFactors($root.r$) $\cup 1, r > f >= 1$ do
            $newRoot \leftarrow$ createRepetition($root.r/f$)
            $root$.setCardinality($f$)
            $matching \leftarrow$ getSubstitute($ST, cand$)
            if $matching \neq \emptyset$ then
                $ST \leftarrow ST$.addRoot($newRoot$)
                replacePattern($ST, matching, root$)
                $hasReplacement \leftarrow true$
                $break$
            end if
        end for
        if $hasReplacement = false$ then
            for $dst \in$ getDSTs($root, ST$) do
                COMPOSE($ST, cand, dst.getRoot()$)
            end for
        end if
    else
        $hasMatchedPartition \leftarrow$ ANALYZEPARTITION($ST, cand, root$)
        if $hasMatchedPartition = false$ then
            for $dst \in$ getDSTs($root, ST$) do
                COMPOSE($ST, cand, dst.getRoot()$)
            end for
        end if
    end if
return $results \leftarrow$ getOptimal($results$)
end procedure

Require: Query Tree: $ST$, Candidate Trees: $cand$ Query Root: $root$.
Ensure: Boolean $result$.
procedure ANALYZEPARTITION($ST, cand, root$)
    $partitions \leftarrow$ getDSTPs($ST, root$)
    $replacements \leftarrow \emptyset$
    for $dstp \in partitions$ do
        $replacement \leftarrow$ getSubstitutes($dstp, cand$)
        $replacements \leftarrow replacements \cup replacement$
    end for
    if $replacements \neq \emptyset$ then
        $replacements \leftarrow$ getBestReplacements($replacements$)
        replaceAll($ST, replacements, root$)
return $true$
    else
return $false$
    end if
end procedure
```

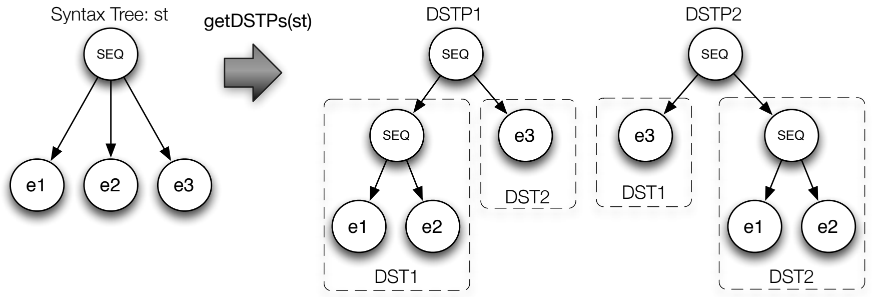


Figure 6.5: Example of creating direct sub-tree combinations

candidate. The *getSubstitutes* operation needs to be executed for every sub-tree and sub-tree partition for the query, comparing with every existing candidate. Given n candidates, for a query with average height[4] h and node degree d, the worst-case time complexity of the composition algorithm with regard to *getSubstitutes* is $\mathcal{O}((2^d)^h n)$. Clearly, the algorithm cannot scale and a much faster way to compose complex event services is needed.

6.2.3 Event Pattern Composition based on Re-usability Index

In order to accelerate the composition, a natural thought is to index the ESTs, so that for a certain sub-query (sub-tree or sub-tree combination), the number of examined candidates can be reduced. Additionally, if the index can tell which parts of a query can reuse existing syntax trees, the number of examined sub-queries can also be reduced. Therefore, a reusability index for event patterns is developed in this section. In the following the reusable relation between ESTs is provided. Then the reusable relation is used to organise event patterns into a hierarchy. Finally, how this hierarchy is used to accelerate the event composition is presented.

6.2.3.1 Reusability of Event Patterns

Informally, an event pattern is *reusable* to another, if the detection of the former can be used in the detection of the latter. An event pattern can be *directly reusable* or *in-directly reusable* to another. An event pattern ep_1 is directly reusable to ep_2, denoted $R_d(ep_1, ep_2)$, *iff* ep_1 is a direct sub-event-pattern (6.6) of ep_2, more formally:

Definition 6.18. $R_d \subset P \times P$ where P is a set of event patterns. $R_d(p_1, p_2)$ holds for $p_1, p_2 \in P \iff \exists T(v) \in DST(est(f_{canonical}(p_2))) \wedge p_2$ *subsumes* , where $T(v) = est(f_{canonical}(p_1))$. In this case p_1 is said to be directly reusable to p_2 on node v.

[4] For definitions of the height and depth of an EST please see paragraph 5, Section 6.1.1 (after Lemma 6.5).

An event pattern ep_1 is in-directly reusable to ep_2, denoted $R_i(ep_1, ep_2)$, *iff* ep_1 is not directly reusable to ep_2, but ep_1 can be transformed into ep'_1 using a sequence of operations on the canonical pattern of ep_1, as a result, it makes $R_d(ep'_1, ep_2)$ hold. These operations have four types: $F_{filter} : T \times F \longrightarrow T$ attaches filters to the roots of syntax trees; $F_{multiply} : T \times \mathcal{N}^+ \longrightarrow T$ multiplies the cardinality of repetition of the roots; $F_{append} : T \times T \longrightarrow T$ adds a sequence of DSTs to the sequential roots as prefixes or suffices; $F_{add} : T \times T \longrightarrow T$ adds a set of DSTs to the parallel roots. In the above function definitions, T is a set of ESTs, F is a set of filters. Definition 6.19 formally define in-directly reusable relation R_i.

Definition 6.19. $R_i \subset P \times P$ where P is a set of event patterns. Given $T = \{\text{the set of all ESTs}\}$, $\mathcal{N}^+ = \{\text{positive integers}\}$, $F = \{\text{the set of all filters}\}$, F_{filter}, $F_{multiply}$, F_{append}, $F_{add} = \{\text{sets of transformation functions}\}$;
$R_i(p_1, p_2)$ holds for $p_1, p_2 \in P \iff \neg R_d(p_1, p_2) \wedge \exists p'_1 \in P, T' \subset T, F' \subset F, n \in \mathcal{N}^+, T_1 = est(f_{canonical}(p_1)), T'_1 = est(f_{canonical}(p'_1)), r = \{\text{the root node of } T_1\}, f_f \in F_{filter}, f_m \in F_{multiply}, f_{add} \in F_{add}, f_{app} \in F_{append}$ such that $R_d(p'_1, p_2) \wedge$

$$T'_1 = \begin{cases} f_f(f_m(T_1, n), F') & type(r) = Rep \\ f_f(f_m(f_{app}(T_1, T'), n), F') & type(r) = Seq \\ f_f(f_m(f_{add}(T_1, T'), n).F') & type(r) = And|Or \end{cases}$$

Similarly, in this case p_1 is said to be in-directly reusable to p_2 on node r.

Formally, the reusable relation on event patterns R is defined in Definition 6.20. An example of reusable relations is depicted in Figure 6.6.

Definition 6.20. $R = (R_d \cup R_i)$

Corollary 6.21 (Reusable is the Inverse of Subsumption). $R(p_1, p_2) \iff p_2 \succeq p_1$

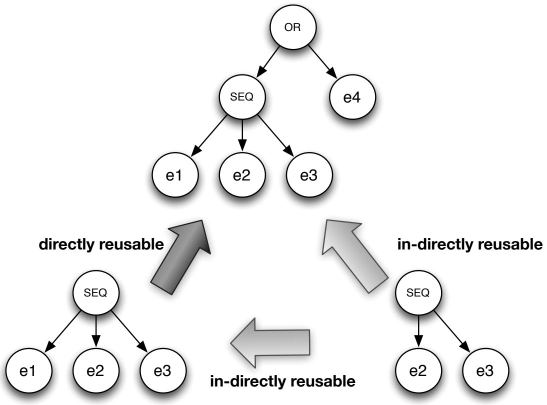


Figure 6.6: Example of event pattern reusability

6.2.3.2 Event Pattern Reusability Hierarchy

Using the reusable relation, a hierarchy of event patterns can be built, called an Event Reusability Hierarchy (ERH). An ERH is a Directed-Acyclic-Graph (DAG), denoted $ERH = (P, R)$ where P is a set of nodes (canonical event patterns) and $R \subset P \times P$ is a set of edges (reusable relations) connecting nodes. Formal definition of an ERH is provided in Definition 6.22.

Definition 6.22 (Event Reusability Hierarchy)**.** Given an ERH $erh = (P, R)$, $\forall (p_1, p_2) \in P$, $R(p_1, p_2)$ holds and $\nexists p_3 \in P$ such that $R(p_1, p_3) \wedge R(p_3, p_2)$.

According to Definition 6.22, if an ERH is built for the three event patterns in Figure 6.6, the edge at the top-right is ignored. The nodes that do not reuse any other nodes are called roots in the ERH, the nodes cannot be reused by other nodes are leaves.

Constructing an ERH requires iteratively inserting canonical event patterns into the hierarchy. If not all nodes can be inserted into a single ERH, a set of separated ERHs is derived, called an Event Reusability Forest (ERF). The algorithm that inserts a node into a given ERF is shown in Algorithm 4.

Algorithm 4 Insert an canonical pattern into an event reusability forest.

```
Require: Canonical Pattern $ep$, ERF $erf$.
 1: procedure INSERT($ep$, $erf$)
 2:     $roots \leftarrow$ getRoots($erf$), $leaves \leftarrow$ getLeaves($erf$)
 3:     $erf$.addNode($ep$)
 4:     $parents \leftarrow$ getReusable($roots$, $ep$)
 5:     drawEdges($parentse$, $p$)
 6:     $childNodes \leftarrow$ getChildNodes($parents$, $erf$)
 7:     $parents \cup$ getReused($childNodes$, $ep$)
 8:     drawEdges($parents$, $ep$)
 9:     remove redundant edges
10:     if $parent$ is modified then
11:         go to 6
12:     end if
13:     perform reversed operations on $leaves$
14: end procedure
```

The above algorithm takes the canonical event pattern ep and an event reusability forest as inputs. As the first step, it finds all $p \in P$ where P is the set of nodes in the forest such that $R(p, ep)$ holds, starting from the roots (line 4). Then the algorithm draws all edges for (p, ep) and removes the redundant edges. As the second step, it draws all necessary edges for (ep, p'), where $p' \in P \wedge R(ep, p')$ holds. During the navigation of nodes, if a pattern $p' \doteq ep$ is found, ep is merged into the same node representing p and the algorithm terminates. This step is omitted in Algorithm 4 for brevity.

As mentioned above, finding reusable components or substitutes for a certain pattern can be achieved by the first step of the node insertion algorithm. Compared to Algorithm 3 in which all nodes may need to be compared, it is now possible to use Algorithm 4 to prune the irrelevant parts of the hierarchy and reduce the number of comparisons required. Figure 6.7 shows an example of identifying the potential composition components of a query (the blue node) once inserted into an ERH, and prunes the branches that are not reusable to the query.

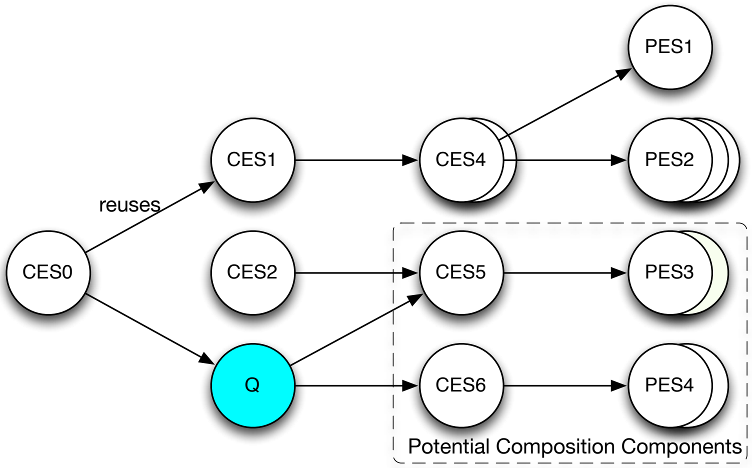


Figure 6.7: Example of pruning irrelevant branch via ERH

6.2.3.3 Event Composition Algorithm with Event Reusability Forest

Although the efficiency of the event composition can be improved by reducing the number of comparisons required, it comes with the price of more complicated comparisons. Reusability checking is a *sub-graph isomorphism* problem, which is a generalisation of substitute checking and is NP-complete. Moreover, the full potentials of the reusability index are not exploited.

In fact, once a query tree representing an event pattern is inserted into the ERF, the components needed in the composition plans of the event pattern are prepared, even if no isomorphic syntax trees are found for the whole query. All one needs to do is to gather the parent nodes of the inserted query and replace appropriate parts of the query with the event declarations of these parent nodes. In cases when in-directly reusable nodes/sub-trees are replaced, additional transformation functions are invoked. If the replacement results in a bound query, the composition plan is derived, otherwise, the composition fails due to the lack of required PESs. The algorithm that accomplish this task is given in Algorithm 5.

Algorithm 5 Event composition for query Q with ERF *erf*.

```
Require: Query Tree: $Q$, ERF $erf$.
Ensure: Composition Plan $Q$.
 1: procedure composeWithIndex($Q$, $erf$)
 2:     insert($Q$, $erf$)
 3:     if an isomorphic node is found then
 4:         $EDs \leftarrow$ event declarations of the isomorphic node return getOptimal($EDs$)
 5:     end if
 6:     $parents \leftarrow$ getParents($Q$, $erf$)
 7:     for $p \in parents$ do
 8:         if $R_d(p, Q)$ then
 9:             directlyReplace($Q, p$)
10:         else
11:             inDirectlyReplace($Q, p$,getDSTs($p$))
12:         end if
13:     end for
14:     if $Q$ is bound then return $Q$
15:     end if
16:     Fail
17: end procedure
```

The *directlyReplace* operation in line 9 replace the sub-tree in Q that is isomorphic to p with the optimal event declarations of p. When $R_i(p, Q)$ holds and $p' = F(p)$ is the transformed pattern, according to Definition 6.19, the set of DSTs of $f_{canonical}(p)$ is a subset of the DSTs of $f_{canonical}(p')$. The *inDirectlyReplace* operation will replace all DSTs (as well as all possible partitions of the DSTs) of p' in Q which are isomorphic to the DSTs of p with the event declarations of p, with necessary filter attachments and cardinality changes.

Compared to Algorithm 3 which requires $\mathcal{O}(n(2^d)^h)$ graph isomorphism checks to find proper substitutes, Algorithm 5 only needs $\mathcal{O}(2n)$ subgraph isomorphism comparisons (which is NP-complete in it self [187]) to find reusable components. The efficiency of event composition is greatly improved by Algorithm 5.

6.3 Experiment Evaluation

In this section, the performance of the proposed algorithms is evaluated with simulation datasets. Three sets of experiments are conducted to evaluate the performance of the event query reduction (Algorithm 1), event reusability forest construction (Algorithm 3) and event compositions (Algorithm 2 and 4). In the following, the general experiment settings are described, then, the detailed settings for each experiment and its results are elaborated.

6.3.1 General Experiment Settings

All experiments are carried out on a Macbook Pro with a 2.53 GHz duo core CPU and 4 GB 1067 MHz memory. The algorithms and experiments are developed in Java (JDK 1.7) and deployed on a JVM with 256 to 2048 MB heap size. Unless otherwise specified, the experiments in this thesis are carried out in this setup.

In order to have more accurate results, all results are averaged from 5 test iterations for each test setting. To ensure the test results are unbiased, an Event Pattern Generator (EPG) is developed to create random event patterns/queries. The EPG will choose the leaf nodes used in event patterns randomly from 10 different primitive events. The event operators are also randomly created as roots or intermediate nodes in query trees. It is worth mentioning that the EPG is designed to be generic enough to cover different types of queries that could be used in the smart city scenarios introduced in Section 2.1. To ensure the random event pattern creation stops at some point, and also to have some control on the size of patterns (number of nodes in the query tree) created, the EPG receives two parameters: height and degree, specifying the maximal tree height and degree of the output. The EPG is used to create simulated datasets in the experiments.

6.3.2 Performance of Event Query Reduction

Query reduction is a basic and important operation in CES discovery and composition. To test its performance, the EPG is used to generate 5000 event patterns. The query trees of these patterns have a maximum degree and height of 5. The query reduction algorithm is invoked on each pattern and the execution time is grouped by the size of patterns. In this way, the minimal, maximum and average reduction time are derived for event patterns of different sizes. An average group size of about 33 patterns is expected (5000/150), when group sizes are relatively small (e.g., only less than 10 patterns are created with 25 nodes), we do not include the result for this group (sample size too small). Figure 6.8 shows the results of the experiment.

The results show that most event patterns can be reduced to their canonical forms efficiently, in fact, 92% of the event patterns are reduced in less than 100 ms. However several "spikes" occur in the maximal reduction times, 2.4% of event patterns took more than one second to be reduced, and in extreme cases, it goes up to 8 seconds. After investigating the dataset, it is observable that these extremely long reduction time are due to the nested repetition nodes in the query tree. Since the repetition nodes are first transformed into sequence operators in order to identify possible merges with other sequences, they may significantly increase the total size of the pattern. As a result, the

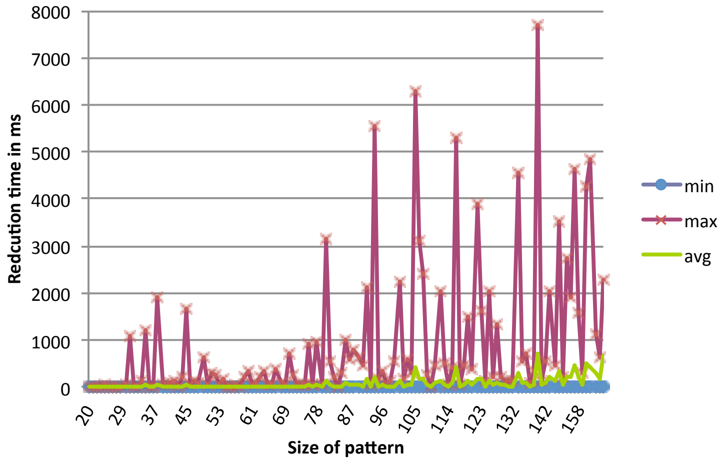


FIGURE 6.8: Execution time of query reduction

merge operation may take much more time. A partial solution to the problem is to use faster graph isomorphism algorithms and accelerate the merge operations. In conclusion, the query reduction algorithm is efficient for most event queries.

6.3.3 Performance of Event Reusability Forest Construction

The feasibility and efficiency of ERF construction are evaluated by measuring the time required. The EPG is used to create 100 to 1500 random event patterns with different maximal degree and height parameters. Then, the ERF construction algorithm is invoked on these sets of patterns to observe the time needed. Figure 6.9 shows the results of the experiment.

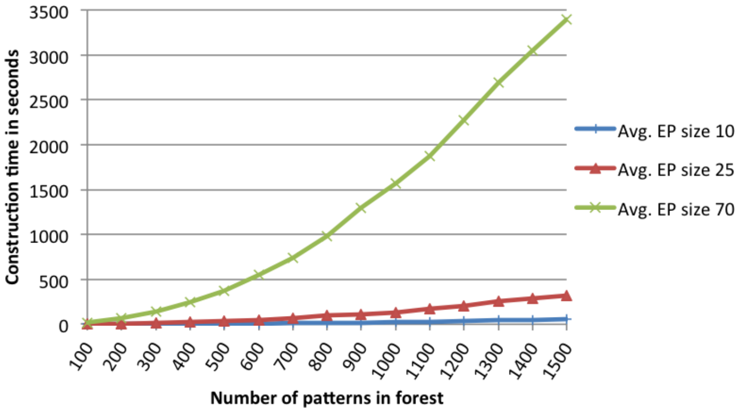


FIGURE 6.9: Execution time of hierarchy construction

In the above results, the lowest blue line indicates the time needed for constructing an ERF with sets of random event patterns with an average pattern size of 10 nodes.

For example, with 1500 event patterns, it took 58 seconds to complete the construction. Similarly, the red line in the middle represents the set of event patterns with 25 nodes on average and completes the construction in 323 seconds. Finally, the green line represents the set of event patterns with 70 nodes on average and take nearly an hour to construct the hierarchy. The results indicate that for event patterns with around 25 nodes, inserting it into a 1500-node forest could take hundreds of milliseconds. However, inserting a large and complex event pattern with about 70 nodes into a large forest with 1500 very complicated event patterns could take more than 2 seconds.

6.3.4 Performance of Event Composition

In order to evaluate the composition algorithms, fixed sets of queries are composed based on the fixed sets of candidate replacements/reusable components for both indexed and unindexed algorithms to compare their results. More specifically, the EPG is used to create 500 and 1000 event patterns with an average pattern size of 25 nodes as candidates. The EPG is then used again to create 3 sets of event patterns as queries. There are 100 event patterns in each query set and their average pattern size are 10, 14 and 25 nodes. Figure 6.10 shows the time needed for each query set against each candidate set using indexed or unindexed composition algorithms.

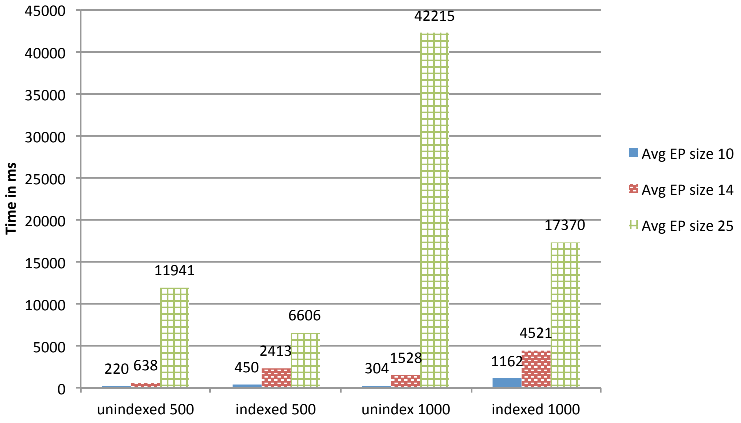


FIGURE 6.10: Execution time of composition: indexed vs. unindexed approaches

The results indicate that for small event patterns, the unindexed approach out-performs the indexed one, but for large event patterns the indexed algorithm is much faster. This is aligned with the discussions in Section 5: reusability checking is more complicated than graph isomorphism, but the number of subgraphs compared is much less for reusability checking in large graphs. In fact, the algorithms are also tested with 70-node queries, the indexed approach took 75 and 157 seconds to complete but the unindexed algorithm terminates before finish due to insufficient memory (with heap size set to 2 GB).

Another factor causing the slowly indexed composition on small patterns is that the shape of the forest is too "flat" for random patterns, i.e., very few candidate event patterns reuse others. In fact, it is observable that about 80% of the nodes in the random forests are roots which do not reuse other patterns. In such forests, the advantage of navigating the forest/hierarchy to avoid unnecessary comparisons is not significant. In real world scenarios, there are reasons to believe users may use existing event patterns as templates to create new ones, so that the probability of reusability can be higher than randomly created datasets. To evaluate the impact, a reuse probability is assigned from 10% to 90% to the EPG to make it reuse existing patterns with the assigned rate. Figure 6.11 demonstrates the impact of reuse probability. It shows the time required for composing 100 14-node queries on 1000 14-node candidates created with different reuse probabilities. The results indicate that even for simple event patterns, the indexed approach is faster than the unindexed one when the probability of reuse is above 70%.

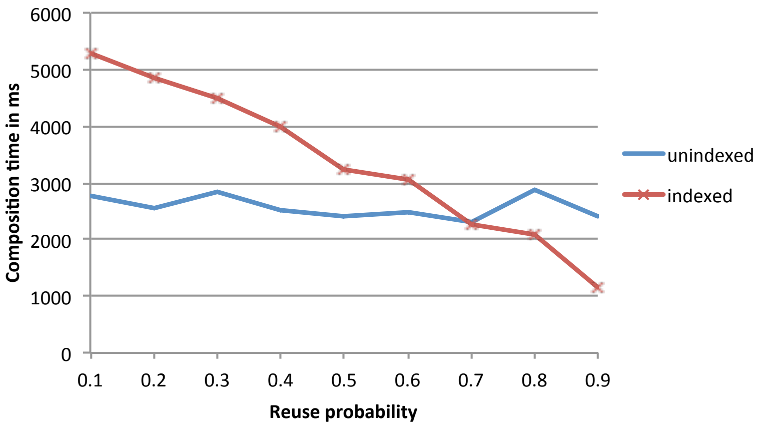


FIGURE 6.11: Impact of reuse probability on indexed composition

6.4 Related Work

Complex event service composition can be seen as a variant of service composition. However, current planning based service composition (e.g., the work done by Rao et al. [27]) only consider the matching of input/output message types and the evaluation of logical formulas in preconditions/effects. In complex event service composition, it is not straightforward to define preconditions and effects for event detection tasks, nor is it enough to create composition plans based on matchmakings between event types. Rather, comparing event patterns/queries in event service descriptions/requests is essential to determine their reusability. In database systems, various techniques including query subsumption (Deen et al. [68]), multiple query optimisation (Sellis et al. [188]) and y-filter [189] etc. are developed to share and reuse partial results among similar queries in static databases. The pattern-based CES composition in ACEIS is inspired by query subsumption: the subsumption (inverse of reusable) relationships are identified between canonical event patterns.

Reusing event queries/subscriptions is also discussed in many other event based systems, including content-based event overlay networks [44, 190–195] and CEP query optimisation [97, 196].

Hasan et al. [194] and Curry et al. [190] evaluate the reusability of event queries based on the similarity between event attribute types and values, no event patterns are considered. SIENA [44] and its extended version [192] reuse only simple attribute filters and sequential event patterns by defining a subscription covering relation. In the works done by Li et al. [191] and Long et al. [193], brokers are equipped with event engines which makes them capable of processing more event operators, however, the decomposition of event subscriptions follows a top-down traversal on the query tree without considering different partitions of sub-trees. Akdere etl al. [196] consider using shared events to reduce event detection cost, however, the complexity of finding shared events are not discussed. Schultz-Moller et al. [97] uses different measures for different operators to find reusable sub-events, i.e., for conjunctive and disjunctive operators they use a greedy algorithm with time complexity $\mathcal{O}(n \log n)$, while for sequential pattern they develop a dynamic programming algorithm with the time complexity $\mathcal{O}(n^3)$. When a query is deployed, a top-down traversal tries to find the largest already deployed component in the query, and builds the remaining part bottom-up. In this chapter, the reusability index is used to accelerate reusable sub-event discovery while ensuring network optimised plans.

E-Cube [197] also analyses the reusability between event patterns and organises them into a hierarchy. The main differences to the approach in ACEIS are 1) E-Cube only supports sequence and negation operators, whereas sequence, conjunction, disjunction and repetition operators is supported in this thesis, and 2) E-Cube has different rules in determining reusable relation between sequence patterns, e.g.: the event sequence $E_4 := (E_1, E_2, E_3)$ is reusable to $E_5 := (E_1, E_3)$ in E-Cube but not so in ACEIS. As a result, when E_5 tries to reuse E_4, an overhead of checking $\neg E_2$ is required. Table 6.1 summarises the comparison of the related work in this chapter.

6.5 Summary and Discussion

In this chapter, the discovery and composition methods are discussed for CESs described with the event pattern semantics and formats from Chapter 5. A canonical form of event

Approaches	**Differences to pattern-based discovery and composition in ACEIS**
Conventional service composition like Rao et al. [27]	Service matchmaking on IOPE, not suitable for complex event services.
Query reuse in database systems e.g., Deen et al. [68], Sellis et al. [188], Y-Filter [189]	Suitable for query over static databases, temporal logics not considered.
Hasan et al. [194] and Curry et al. [190]	Consider reusability based on similarity of event attributes not event patterns.
SIENA [44, 192]	Reuse only simple attribute filters and sequential event patterns.
Li et al. [191] and Long et al. [193]	Partitions of sub-trees in patterns not considered when finding reusable events.
Akdere etl al. [196]	The complexity of finding shared events are not discussed.
Schultz-Moller et al. [97]	Use dynamic programming and greedy algorithms to find optimal execution plans, but the optimal plan is not guarenteed.
E-Cube [197]	Supports only sequence and negation, overhead exists when reusing sequences that differ by negative event types.

Table 6.1: Related works in complex event pattern reuse

patterns can be derived by first creating a complete pattern and then removing all redundant nodes in the complete pattern. It is proved that by comparing the isomorphism of canonical event patterns, one can determine if two patterns are semantically equivalent, i.e., can be used as substitutes during CES discovery and composition. A top-down traversal algorithm is developed to compose CESs based on substitution. However, it is evident that this approach would not scale for complicated queries and large datasets. In order to accelerate CES composition, a CES index based on the reusability of event patterns is proposed, called an Event Reusability Forest (ERF). A CES composition algorithm using ERF is introduced to reduce the number of graph isomorphism checking operations needed during the composition. The query reduction and ERF construction algorithm are evaluated for their feasibility and efficiency. Both the indexed and un-indexed composition algorithms are evaluated for their efficiency. The results indicate that for randomly created queries and datasets, the indexed composition algorithm has a better performance that the un-indexed when the query is complicated and service repository is large. It is also observed that when more reusable relations can be found in the service repository, the performance of the indexed composition improves significantly. In the next chapter, the composition algorithm is extended to handle QoS constraints and preferences defined by users.

Chapter 7

Constraint-aware Event Service Discovery and Composition[*]

In Chapter 6 Complex Event Processing (CEP) applications are provided as reusable services and the reusability of those event services is determined by examining complex event patterns and primitive event types and attributes. Event service composition based on functional properties is therefore addressed. However, it is still difficult to determine which event service candidates (or service compositions) best suit users' and applications' multi-modal Quality-of-Service (QoS) requirements. A sub-optimal service candidate/composition may lead to inaccurate event detection, lack of system robustness or delay in event detection etc.

QoS-aware event service discovery, i.e., finding direct matchings (without service composition) for event service requests while fulfilling the QoS constraints is trivial. For example, given an event service described by a tuple $E = (src, t, ep, D, Q)$, where src describes the service endpoints, t is the domain-specific event type, ep is the event pattern describing the temporal and logical rules for detecting complex events, D is the data payloads of the event and Q is the QoS parameters of the event service. An event service request is a tuple $E_r = (t_r, ep_r, D_r, Const, Pref)$, where t_r is the requested event type, ep_r is the requested event pattern, D_r is the requested data payloads, $Const$ is a set of QoS constraints and $Pref$ is a set of QoS preferences. If $ep_r = \emptyset$, i.e., requested event service is a PES, E matches E_r if and only if: 1) t_r subsumes t, 2) $\forall d_r \in D_r, \exists d \in D$ such that $d_r = d$ or d_r subsumes d and 3) Q satisfies $Const$. If $ep_r \neq \emptyset$, the first condition is replaced by ep_r is semantically equivalent to ep, denoted $ep_r \doteq ep$. However, when event service compositions are considered, the problem gets more complicated.

[*] Part of the content in this chapter is published in [198].

The goal of the study in this chapter is to enable a QoS-aware event service composition and optimisation. By doing so, this chapter implements the constraint-aware event service discovery and composition functionalities in the *Resource Management* component in the Automatic Complex Event Implementation System (ACEIS), as shown in Figure 7.1.

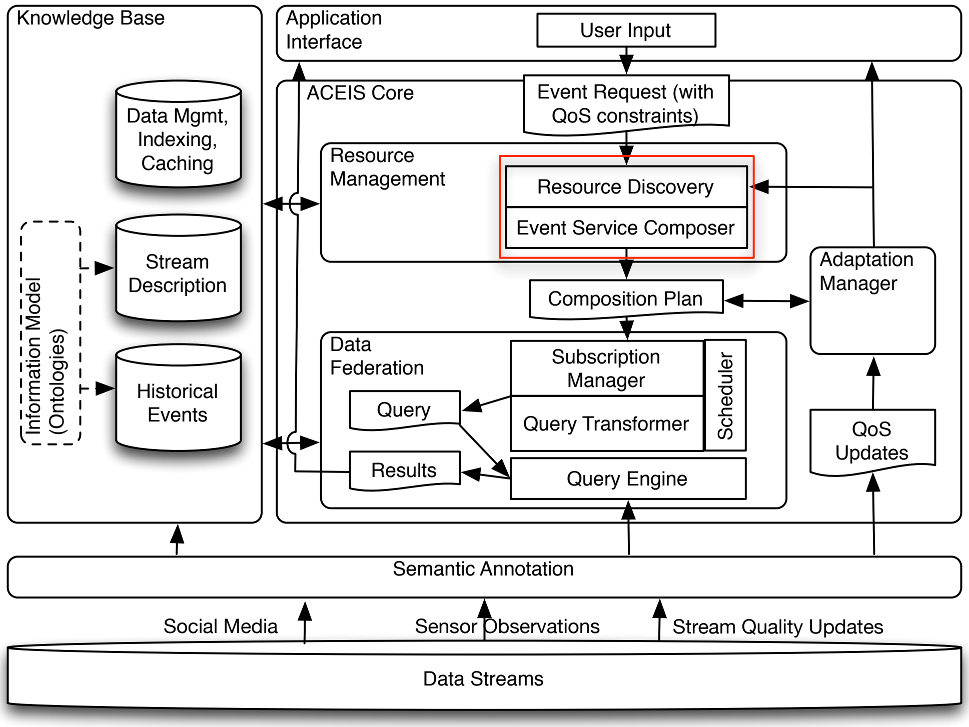


Figure 7.1: The resource management module in ACEIS

In order to facilitate constraint-aware event service discovery and composition, two issues should be considered: QoS aggregation and composition efficiency. The QoS aggregation for a complex event service relies on how its member events are correlated. The aggregation rules are inherently different to conventional web services. Efficiency becomes an issue when the complex event consists of many primitive events, and each primitive event detection task can be achieved by multiple event services. In summary, this chapter needs to address the following two issues:

1. How to create QoS aggregation rules and utility functions to estimate and assess QoS for event service compositions?

2. How to enable efficient event service compositions and optimisation with regard to QoS constraints and preferences based on Genetic Algorithms?

The remainder of this chapter is organised as follows. Section 7.1 addresses the first issue by developing a QoS aggregation schema over multiple QoS dimensions. Different aggregation rules are provided for QoS dimensions to estimate the overall QoS performance of an event service composition based on the composition plan and the QoS of the event services involved in the composition plan. A QoS utility function is provided to quantify the QoS performance and compare them. Section 7.2 tackles the second issue by designing a Genetic Algorithm (GA) for optimising event service compositions, using the QoS utility as the heuristic. Section 7.3 evaluates the performance of the GA and provide validations for the QoS aggregation schema using both real and synthetic datasets. Section 7.4 discuss the state-of-the-art in QoS-aware service composition before Section 7.5 summarises.

7.1 QoS Model Aggregation Schema

The QoS properties of event service compositions may vary depending on the set of member event services used in the compositions. Some QoS properties may propagate along the event service network. In this section, a QoS model is used to represent some sample QoS properties discussed in literature (e.g., [84, 85, 199–201]). Then the QoS aggregation schema is presented to estimate the QoS properties for complex event service composition plans. Finally, a utility function is introduced to evaluate the QoS performance under constraints and preferences.

7.1.1 QoS Properties of Event Services

The event QoS attributes describe the non-functional performance of event services (and service compositions). In [199, 200] a stream quality ontology is developed for modelling the quality of Internet-of-Things (IoT) data streams. In this chapter, the following QoS attributes are considered propagatable in event service compositions (when a data stream is modelled as an event service):

- *Latency* (L) describes the delay of an event service, i.e., the temporal difference between the time when the event consumer receives the notification and the time when the event actually happens (usually denoted by the timestamp of the event), latency is studied in [84, 85, 201–203],
- *Price* (P) describes the monetary cost for an event services, price is studied in [84, 85, 201],

Table 7.1: Overall quality-of-service calculation

Dimensions	**QoS Symbols**			**Overall Calculation**
	SI	CP	EE	
Latency	L_i	L_c	L_e	$L = L_i + L_c + L_e$
Monetary	P_i	P_c	n/a	$P = P_i + P_c$
Energy	Eng_i	Eng_c	Eng_e	$Eng = Eng_i + Eng_c + Eng_e$
Network	n/a	B_c	n/a	$B = B_c$
Availability	Ava_i	Ava_c	n/a	$Ava = Ava_i \times Ava_c$
Completeness	C_i	C_c	C_e	$C = C_i \times C_c \times C_e$
Correctness	Acc_i	Acc_c	Acc_e	$Acc = Acc_i \times Acc_c \times Acc_e$
Encryption	S_i	S_c	n/a	$S = \min(S_i, S_c)$

- *Energy Consumption* (Eng) describes the energy cost for an event service, energy cost is studied in [84, 204, 205],
- *Bandwidth Consumption* (B) describes the usage of network bandwidth for an event service, network usage is studied in [84, 201, 203]
- *Availability* (Ava) describes the possibility of an event service being accessible, it can be numerically represented in percentages, availability is studied in [84, 85],
- *Completeness* (C) describes the completeness of events delivered by an event service, it can be numerically represented as recall rates in percentages, completeness is studied in [85, 201],
- *Accuracy* (Acc) describes the correctness of events delivered by an event service, it can be numerically represented as percentages indicating the possibility of having correct results, accuracy is studied in [203, 206, 207], and
- *Security* (S) describes the security protocol used by event services numerically represented as integer encryption levels, higher numerical value indicate higher security levels, security is studied in [84, 203].

By the above definition, a quality vector $Q =< L, P, Eng, B, Ava, C, Acc, S >$ can be specified to indicate the performance of an event service in 8 dimensions.

7.1.2 Quality-of-Service Aggregation

The QoS performance of an event service composition is considered to be influenced by three factors: *Service Infrastructure*, *Composition Pattern* and *Event Engine*. The *Service Infrastructure* consists of computational hardware, service Input/Output (I/O) implementation and the physical network connection, it determines the inherent I/O performance of a service. The *Composition Pattern* refers to the event patterns evaluated locally by the event engine and the set of member event services directly involved.

Table 7.2: Quality-of-Service aggregation rules based on composition patterns

QoS Dimensions	**Aggregation Rules**	**Applicable Event Operators**
$P_c(E)$	$\sum P_c(e)$, where $e \in E_{ice}$	*And, Or, Sequence, Repetition*
$Eng_c(E)$	$\sum Eng_c(e)$, where $e \in E_{ice}$	*And, Or, Sequence, Repetition*
$B_c(E)$	$\sum B_c(e)$, where $e \in E_{ice}$	*And, Or, Sequence, Repetition*
$Ava_c(E)$	$\prod Ava_c(e)$, where $e \in E_{ice}$	*And, Or, Sequence, Repetition*
$Acc_c(E)$	$\prod Acc_c(e)$, where $e \in E_{ice}$	*And, Or, Sequence, Repetition*
$S_c(E)$	$min\{S_c(e) \vert e \in E_{ice}\}$	*And, Or, Sequence, Repetition*
$L_c(E)$	$L_c(e) \vert e =$ last event in $E_{dst}\}$	*Sequence, Repetition*
	$avg\{L_c(e) \vert e \in E_{dst}\}$	*And, Or*
$C_c(E)$	$\frac{min\{C_c(e) \cdot f(e), e \in E_{dst}\}}{card(E) \cdot f(E)}$	*And, Sequence, Repetition*
	$\frac{max\{C_c(e) \cdot f(e), e \in E_{dst}\}}{f(E)}$	*Or*

Indeed, the performance varies on which services are used to produce the member events and how they are logically correlated by event operators. For the event correlations, four event operators are considered: *And*, *Or*, *Sequence* and *Repetition*. The internal implementation of the *Event Engine* also has an impact on the event service composition performance. However, it can be difficult to assess or specify, because it depends on different implementations of event engines. Different QoS parameters have different characteristics and thus different aggregation rules can be applied to them.

Table 7.1 summarises how different QoS parameters of an event service composition are calculated based on the three factors. In this table, we assume event engines are free to use, always ready to accept new queries when deployed, and do not introduce security issues. When network consumption is measured in the number of event messages, we consider it irrelevant to the service infrastructure (connection types, messaging formats etc.). We do consider the effect of service infrastructure over accuracy because out-of-synchronization messages during data transmission may lead to incorrect results. For a primitive event service that is not equipped with CEP engines (e.g., a sensor service), its overall quality vector is identical to the quality vector of the *Service Infrastructure*.

The *Composition Pattern* is a key factor in aggregating QoS properties for event service compositions. A step-wise aggregation over Event Syntax Trees (ESTs) is adopted to aggregate QoS properties. More specifically, the aggregation rules are applied iteratively from the leaves to the roots on ESTs. Aggregation rules for different QoS dimensions can be event operator dependent or independent. Event operator dependent rules are defined based on the QoS properties of the set of Direct Sub-Trees (DSTs) of the entire event syntax trees. Event operator independent rules are defined based on the QoS properties of the set of Immediately Composed Event services (ICEs). Table 7.2 shows the detailed rules for each quality dimension. In the following the rationale for each rule is explained.

1. **Price**, **Energy Consumption** are operator independent properties. They can be specified in different manners, e.g., the price can be charged over subscription time or volume, similar for energy consumption. For simplicity, it is assumed that they are specified over time. The overall price or energy cost of an event service E is the sum of the price or energy cost of the *immediately composed event* services (denoted E_{ice}).

2. **Bandwidth Consumption** can be measured by the number of events consumed by an event composition, i.e., its traffic demand. It is an operator independent property, the aggregated bandwidth consumption is the sum of the product of the completeness and the frequencies of the services in E_{ice} (denoted $f(e), e \in E_{ice}$). Recall that the method for estimating frequencies of event services is given in Equation 6.1.

3. **Availability**, **Accuracy** and **Security** are operator independent properties. The availability and accuracy of E are the multiplication of event service availability and accuracy in E_{ice}. The rationale of using the multiplication is that a result is considered incorrect if one of the inputs used to calculate the result is incorrect. By the same logic, the availability is aggregated using multiplication. The security level is determined by the most vulnerable event service in E_{ice}.

4. **Latency** of event E is an operator dependent property. It is determined by the last event completing the event pattern of E [196] . Therefore, if the root operator of E is a sequence or repetition, the latency of E is the same as the last event in the *direct sub-events* of E (denoted E_{dse}). Since it is hard to predict when the last direct sub-event occurs in parallel operators, i.e., "And" and "Or", an approximation is made with the average of the latencies of the event services in E_{dse}.

5. **Completeness** is an operator dependent property. The completeness of E can be estimated based on its direct sub-event frequencies ($f(e), e \in E_{dse}$), and completeness ($C_c(e), e \in E_{dse}$). $\frac{min\{C_c(e) \cdot f(e) | e \in E_{dse}\}}{(card(E))}$ or $\frac{max\{C_c(e) \cdot f(e) | e \in E_{dse}\}}{(card(E) \cdot f(E))}$ represents how often all or any direct sub-event instances would occur under the influences of the completeness, where $card(E)$ gives the cardinality[1] of the root operator of E. The completeness of E is derived by dividing this frequency with the estimated frequency of E.

[1] For repetition operators their cardinalities are greater than 1, for other operators the cardinality is 1.

7.1.3 Event QoS Utility Function

In order to choose the best service composition under users' QoS constraints and preferences, a QoS utility function is needed. Defining such a utility function falls into the category of Multi-Criteria Decision Making (MCDM). In MCDM research, numerous methods have been proposed (e.g., see [208]). While defining a sophisticated utility function is not the focus of this chapter, the *Simple Additive Weighting* (SAW [86]) technique is used to define the service QoS utility. It is worth noting that by applying SAW the following three assumptions are made [209]:

1. **risk independence,** implying the uncertainty of QoS values are not considered, i.e., the probability p of a QoS attribute q has the value v is not modelled,
2. **preferential independence,** implying preferences over values of a set of QoS attributes do not depend on the values of other attributes and
3. **utility independence,** implying QoS attributes are independent of each other.

The first assumption is trivial because the probability of errors in the QoS aggregation is not considered. The second assumption also holds because a total order can be applied to all eight QoS dimensions discussed in Section 7.1.1, regardless of the values in other dimensions. For example, under any circumstances, it is safe to assume that lower latency is more desirable, as well as higher accuracy. The third assumption is a simplification of the real-world settings: sensors *may* use more energy to take more samples in order to achieve higher accuracy, and the bandwidth consumption of a composition plan *may* have a correlation with the completeness of the composition. In this chapter, the correlations between quality attributes are not modelled. The same assumptions are made in existing works on QoS-aware service composition (e.g., in [85, 87, 90, 201, 210]), since SAW is adopted in these works.

Given a quality vector of an event service composition $Q =< L, P, Eng, B, Ava, C, Acc, S >$ representing the service QoS capability, q denotes one of the eight quality dimensions in the vector (e.g., q_L for the latency of a composition plan, say 100 milliseconds), $O(q)$ as the theoretical optimum value in the quality dimension of q (e.g., for latency the optimum value is 0 seconds), $C(q)$ as the user-defined value specifying the hard constraints (i.e., worst acceptable value, say 10 seconds) on the dimension, and $0 \leq W(q) \leq 1$ as the weighting function of the quality metric, representing users' preferences (higher $W(q)$ means more important for the user). Further, QoS properties can have positive or negative tendency: $Q = Q_+ \cup Q_-$, where $Q_+ = \{Ava, R, Acc, S\}$ is the set of properties with the positive tendency (larger values the better), and $Q_- = \{L, P, Eng, B\}$ is the properties with the negative tendency (smaller values the better). The QoS utility U is derived by:

$$U = \sum \frac{W(q_i) \cdot (q_i - C(q_i))}{O(q_i) - C(q_i)} - \sum \frac{W(q_j) \cdot (q_j - C(q_j))}{C(q_j) - O(q_j)} \tag{1}$$

where $q_i \in Q_+, q_j \in Q_-$. Intuitively Equation (1) calculates a utility for a QoS parameter by measuring how good the QoS parameter in a composition plan is (difference to the worst case constraints) by comparing it (the aggregated QoS parameter) with how good it can be in theory (difference between constraints and theoretical optimum values). According to Equation (1) the best event service composition should have the maximum utility U. A normalised utility with values between $[0, 1]$ can be derived using the function $\bar{U} = (U + |Q_-|)/(|Q_+| + |Q_-|)$.

7.2 Genetic Algorithm for QoS-Aware Event Service Composition Optimisation

The detection of the complex event pattern of an event service composition can be achieved by monitoring different sets of member events on different granularity levels. Each member event detection task can be achieved by subscribing to a set of event services. If a complex event pattern can be detected by n different sets of sub-events, each set has an average size of m sub-events, and each sub-event detection task can be implemented by subscribing to l (on average) event service candidates, the total number of concrete composition plans is estimated to be $n \cdot l^m$. In large scale scenarios, it is highly inefficient to enumerate all possible compositions of event services and evaluate their overall performance. In this chapter, a heuristic method based on *Genetic Algorithms* (GA) is developed to derive global optimisations for event service compositions, without the need for enumerating all possible composition plans. The algorithm is intended to be deployed as an event service discovery/composition engine on a server.

Typically, a GA requires a genetic encoding for the solution space, as well as a fitness function to evaluate the solutions. A standard GA-based search iterates the procedure of *select*, *crossover* and *mutate* until termination conditions are met.

The GA approach in this chapter follows these steps. A workflow of the QoS-aware event service composition is shown in Figure 7.2. The "fitness" of each solution can be evaluated by the QoS utility function in Equation (1). Compared to traditional GA-based optimisations for service compositions, where a composite service is accomplished by a fixed set of service tasks, event service compositions can have variable sets of sub-event detection tasks. Determining which event services are reusable to the event service request is resolved in Chapter 6, where hierarchies (i.e., ERHs, introduced in

Section 6.2.3.2) of reusable event services are maintained, called an Event Reusability Forest (ERF). The termination of the GA can be configured with different conditions, e.g., terminates after a duration of time, after a number of generations, or when the differences among the individuals in the population are marginal, i.e., the results have converged.

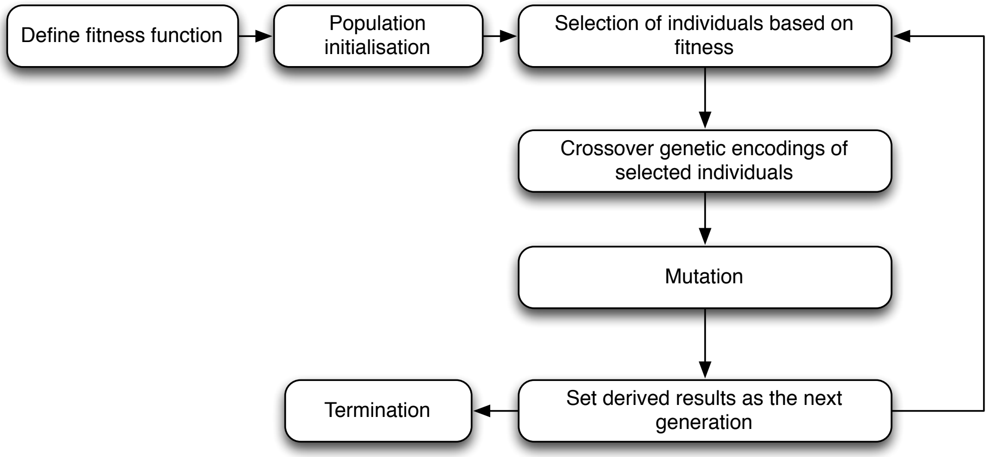

Define fitness function
Population initialisation
Selection of individuals based on fitness
Crossover genetic encodings of selected individuals
Mutation
Set derived results as the next generation
Termination

FIGURE 7.2: Workflow of QoS-aware event service composition

7.2.1 Population Initialisation

Given an event service composition request represented by a *canonical* event pattern *ep*, the initialisation of the population consists of three steps. First, enumerate all *Abstract Composition Plans* (ACPs) of *ep*. An ACP is a composition plan without concrete event service bindings. Second, pick randomly a set of ACPs. Third, for each chosen ACP, pick randomly one concrete event service binding for each sub-event involved. Then, a set of *Concrete Composition Plans* (CCPs) with random structure and service bindings are obtained. The second and third steps are trivial. In the following, the method for creating ACPs based on ERF is explained.

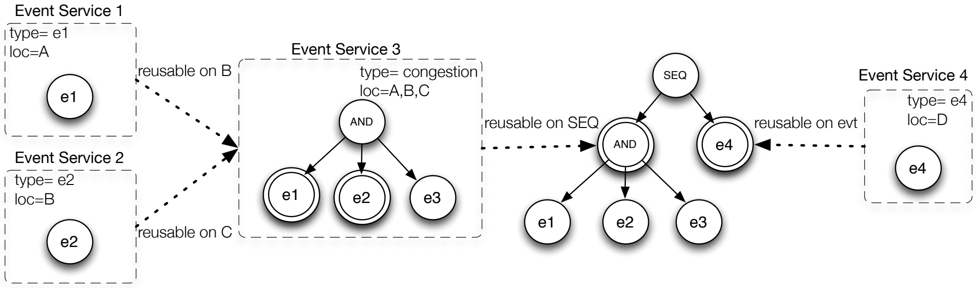


FIGURE 7.3: Marking the reusable nodes

When an event pattern *ep* is inserted into the ERF, the *reusable nodes* in the EST of *ep* are marked, denoted N_r: $N_r \subseteq f_{canonical}(ep) \wedge \forall n \in N_r, \exists ep^{'} \in ERF$, $ep^{'}$ is reusable to *ep* on n, as depicted in Figure 7.3. Obviously, a primitive event involved in *ep* has at most 1 ACP, which is subscribing to the primitive event services with the requested primitive event type. And the ACPs for any sub-event patterns of *ep* (including *ep* itself) can be enumerated by listing all possible combinations of the ACPs of their *immediate* reusable nodes. By recursively aggregating those combinations, the ACPs for *ep* are derived. The purpose of creating ACPs is to create "templates" for generating random concrete composition plans, i.e., different ACPs will have different structures in their ESTs.

It is worth noting that although it requires enumerating *all* ACPs to ensure the diversity in the structure of event compositions, the size of the different combinations of reusable sub-events is moderate, compared to the size of all concrete composition plans. The reusable relations can be efficiently retrieved from the ERF. Therefore, the enumeration of ACPs can be done efficiently.

7.2.2 Genetic Encodings for Event Syntax Trees

Individuals in the population need to be genetically encoded to represent their various characteristics. In a typical encoding for service compositions, each service task is encoded with a value indicating the concrete service implementing the task. These values are ordered in a sequence so that the positions of the values indicate to which service tasks they relate. Similarly, the event detection tasks (leaf nodes) in a CCP are encoded with values to indicate the service bindings used. However, the positions of the values (arranged in any tree traversal order) cannot represent which parts of the event detection task the reused event services contribute, since the CCPs are partially-ordered trees with variable structures. The only thing identifying an event detection task is the event pattern it detects.

Nevertheless, the sequence of ancestors of the nodes can give a hint about which roles they play in the entire event pattern and reducing the search space while finding their functional equivalent counterparts. Therefore, global identifiers are assigned to all the nodes in the CCPs and a leaf node in a CCP is encoded with a string of node identifiers as a prefix representing the path of its ancestors and a service identifier indicating the service binding for the leaf, as shown in Figure 7.4. For example, a gene for the leaf node "n13" in P_2 is encoded as a string with prefix "n10-n11" and a service id for the traffic service candidate for road segment B, i.e., "es3", hence the full encoding of the gene is

$n13 :< n10 - n11, es3 >$. The complete set of encodings for every gene constitutes the chromosome for P_2.

7.2.3 Crossover and Mutation Operations

After the population initialisation and encoding, the preparation tasks for GA based optimisation are completed. The algorithm iterates the cycle of select, crossover and mutation to find optimal solutions. The selection is trivial, individuals with better finesses (QoS utility derived by Equation (1)) are more likely to be chosen to reproduce. In the following the details of crossover and mutation operations are explained.

7.2.3.1 Crossover

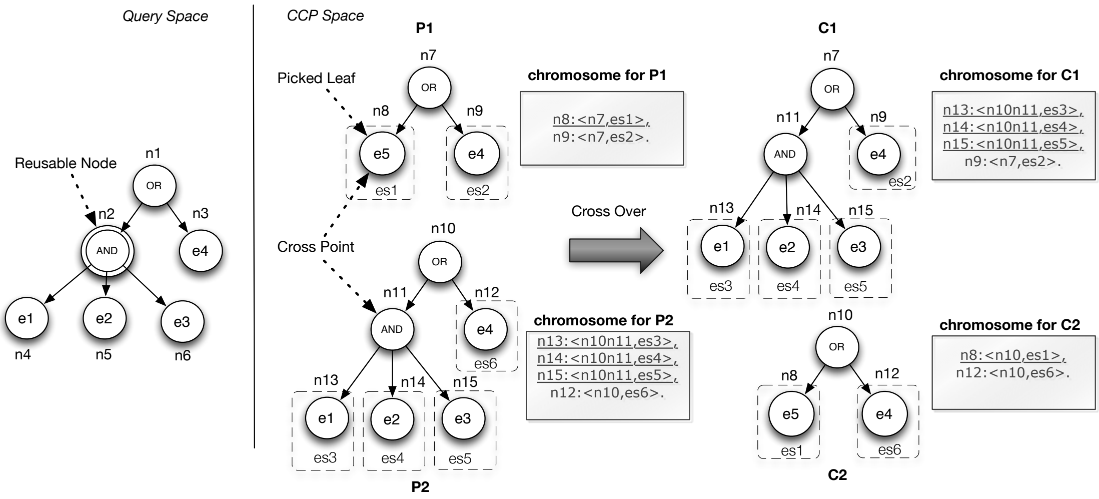


FIGURE 7.4: Example of genetic encoding and crossover operation

In order to ensure valid child generations are produced by the crossover operation, parents must only exchange genes representing the same part of their functionalities, i.e., the same (sub-) event detection task, specified by semantically equivalent (sub-) event patterns. An example of crossover is illustrated in Figure 7.4. The pseudo code of the crossover algorithm is shown in Algorithm 6.

Given two genetically encoded parent CCPs P_1 and P_2, the event pattern specified in the query Q and the event reusability forest ERF, the crossover algorithm takes the following steps to produce the children:

1. Pick randomly a leaf node l_1 from $est(P_1)$, create the node type prefix ntp_1 from the genetic encoding of P_1: $code_1$, as follows: replace each node id in the prefix of $code_1$ with the operator type,

Algorithm 6 Crossover Algorithm

```
Require: Event Pattern: $P_1, P_2$, ERF $erf, CrossoverRate : r$.
Ensure: Event Pattern after Crossover: $P'_1, P'_2$.
procedure CROSSOVER($P_1, P_2, erf$)
    $rand \leftarrow$ RAND$(0, 1)$
    if $rand \geq r$ then
return $P'_1 \leftarrow P_1, P'_2 \leftarrow P_2$
    end if
    $encodings_1 \leftarrow$ ENCODE$(P_1), encodings_2 \leftarrow$ ENCODE$(P_2)$
    $foundCrossPoints \leftarrow false, visited \leftarrow \emptyset$
    while $foundCrossPoints \neq true \wedge visited \neq encodings_1$ do
        if $code_1 \leftarrow$ GETRANDELEMENT$(encodings_1) \notin visited$ then
            $visited \leftarrow visited \cup code_1$
            $n_1 \leftarrow$ GETLEAF$(code_1), subTree_1 \leftarrow$ GETSUBTREE$(n_1, P_1)$
            $ntp_1 \leftarrow$ GETTYPEPREFIX$(code_1, P_1)$
            for $code_2 \in encodings_2$ do
                $n_2 \leftarrow$ GETLEAF$(code_2), subTree_1 \leftarrow$ GETSUBTREE$(n_2, P_2)$
                $ntp_2 \leftarrow$ GETTYPEPREFIX$(code_2, P_2)$
                if $ntp_1 = ntp_2$ then
                    if $subTree_1 \doteq subTree_2$ then
                        $foundCrossPoints \leftarrow true$
                    else if $R(subTree_1, subTree_2, erf) \vee R(subTree_2, subTree_1, erf))$
then
                        find $n_1 \in code_1, n_2 \in code_2$ such that
                        GETSUBTREE$(n_1, P_1) \doteq$ GETSUBTREE$(n_2, P_2)$
                        $foundCrossPoints \leftarrow true$
                    end if
                else if $ntp_1$ extends $ntp_2$ then
                    if REUSABLE$(subTree_1, subTree_2, erf) = true$ then
                        $n_1 \leftarrow$ FINDREPLACEABLE$(code_1, subTree_2)$
                        $subTree_1 \leftarrow$ GETSUBTREE$(n_1, P_1)$
                        $foundCrossPoints \leftarrow true$
                    end if
                else if $ntp_2$ extends $ntp_1$ then
                    if REUSABLE$(subTree_2, subTree_1, erf) = true$ then
                        $n_2 \leftarrow$ FINDREPLACEABLE$(code_2, subTree_1)$
                        $subTree_2 \leftarrow$ GETSUBTREE$(n_2, P_2)$
                        $foundCrossPoints \leftarrow true$
                    end if
                end if
            end for
        end if
    end while
    $P'_1 \leftarrow$ REPLACESUBTREE$(P_1, n_1, subTree_2)$
    $P'_2 \leftarrow$ REPLACESUBTREE$(P_2, n_2, subTree_1)$
return $P'_1, P'_2$
end procedure
```

2. For each leaf l_2 in $est(P_2)$, create the node type prefix ntp_2 from $code_2$ (i.e., encodings for l_2) and compare it with ntp_1. If $ntp_1 = ntp_2$ and the event semantics of l_1 and l_2 are equivalent, i.e., they are merged into the same node in the ERF (recall Algorithm 4 in Section 6.2.3.2) , then mark l_1, l_2 as the crossover points n_1, n_2. If $ntp_1 = ntp_2$ but the pattern of l_1 is reusable to l_2 or l_2 is reusable to l_1, then search back on $code_1, code_2$ until the cross points n_1, n_2 are found on $code_1, code_2$ such that $T(n_1) \doteq T(n_2)$, i.e., the sub-patterns of P_1, P_2 with n_1, n_2 as the root node of the ESTs of the sub-patterns are semantically equivalent.

3. If ntp_1 is an extension of ntp_2, e.g., $ntp_1 = (And; Or; Seq), ntp_2 = (And; Or)$ and the pattern of l_1 is reusable to l_2 in the ERF, then search back on $code_1$ and try to find n_1 such that the sub-pattern with EST $T(n_1)$ is equivalent to l_2. If such n_1 is found, mark l_2 as n_2.

4. if ntp_2 is an extension of ntp_1, do the same as step 3 and try to find the cross point n_2 in $code_2$.

5. Whenever the cross points n_1, n_2 are marked in the previous steps, stop the iteration. If n_1 or n_2 is the root node, return P_1, P_2 as they were. Otherwise, swap the sub-trees in P_1, P_2 whose roots are n_1, n_2 (and therefore the relevant genes), resulting in two new CCPs.

7.2.3.2 Mutation

The mutation operation changes the composition plan for a leaf node in a CCP. To do that one can simply select a random leaf node n in a CCP P, and treat the event pattern of n (possibly a primitive event) as a new event query that needs to be composed, then the random CCP creation process specified in the population initialisation (Section 7.2.1) is used to create mutated variants.

7.2.3.3 Elitism

An *Elitism* method is used in the GA. More specifically, after the selection in every generation, an exact copy of the best individual is added directly into the next generation without crossover or mutation (the original instance may still participate in the crossover and mutation). *Elitism* ensures the best individual is kept through the evolution until a better individual has occurred.

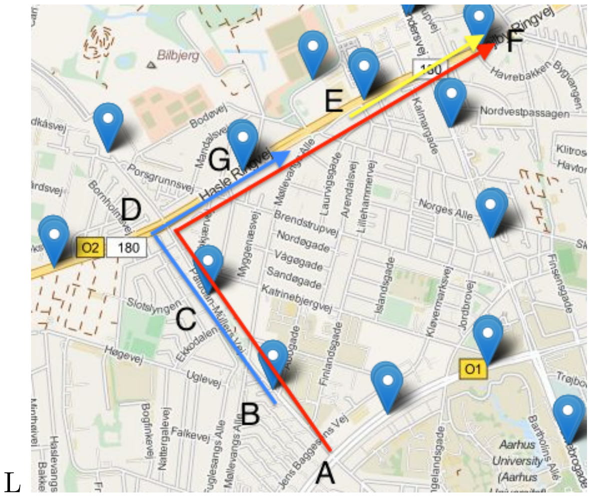


FIGURE 7.5: Traffic sensors in Aarhus City

7.3 Experiment Evaluations

In this section, the evaluation results of the proposed approaches are presented. The experiments scenario is the travel planning scenario using both real and synthetic sensor datasets. The evaluation has two parts: in the first part, the performance of the genetic algorithm is analysed. In the second part, the correctness of the QoS aggregation rules is demonstrated. Experiment results are averaged from 30 iterations.

7.3.1 Experiment Scenario: Travel Planning

The scenario of the experiment is a variation of the travel planning scenario in Section 2.1. The scenario leverages the sensor data streams related to urban traffic. The traffic sensors are paired as start nodes and end nodes. Each pair is capable of monitoring the average vehicle speed v and vehicle count n on a street segment (from the start node to the end node). Combined with the distance d between the two sensors, the estimated travel time $t = d/v$ and congestion level $c = n/d$ can be easily derived. Figure 7.5 shows some traffic sensor nodes on the Aarhus city map. Suppose a user, Alice, has an important appointment in 15 minutes, and she has to travel from home (point A in Figure 7.5 to her work place (point F in Figure 7.5) within the time frame. Alice decides not to pick a route randomly since it is rush hour and there is a good chance that she may experience traffic congestion. Instead, Alice uses a travel planner application on her smartphone to select the fastest route. Also, Alice would like to receive live traffic condition reports during her trip, in case some traffic incidents happen on the selected route and a detour is necessary. To do that, she specifies the start and end location of the travel, she also wants to be sure that the time estimation is accurate, so she sets some non-functional constraints such as the accuracy of the estimated travel time above 90%.

According to Alice's request, the backend system will calculate possible paths and query the sensor services for the latest traffic condition. Based on this information, the system calculates the fastest route (A-D-F) for Alice. Meanwhile, the backend system will have its own non-functional constraint, which is to create composition plans with the minimal network traffic demand for the requests. Taking into account the non-functional constraints from Alice and the system, composition plans reusing event services on different granularity are created. For example, as shown in the map in Figure 7.5, if other users, e.g., Bob and Charlie, are living in the neighbourhood and they have deployed some semi-permanent services monitoring the traffic conditions within some segments (B-C-G and E-F) on the route that Alice chooses, and have registered these services to the service registry, the backend system will recognise these services as reusable and try different combinations of reusable services to find the optimal solution. Meanwhile, the optimisation needs to be efficient to enable real-time re-planning and adapt to the fast changing service environment, simply enumerating all possible composition plans would not scale.

7.3.2 Part 1: Performance of the Genetic Algorithm

In this part of the evaluation, the QoS utilities and the scalability achieved by Brute-Force (BF) enumeration and GA based approach are compared. Moreover, the impact of different GA parameters is analysed and guidelines to fine-tune GA parameters are provided.

7.3.2.1 Datasets

There are 449 pairs of traffic sensors in Open Data Aarhus (ODAA), which are used in the experiments to answer requests on travel planning. ODAA also provides other types of sensors, which might be used in traffic monitoring and travel planning, e.g., air pollution sensors and weather sensors. These sensors are not actually relevant to requests like Alice's (Q_a in Figure 7.6) or Bob's (Q_b in Figure 7.7), i.e., they are noise to queries like Q_a, Q_b (but could be used in other travel related queries). In total 900 real sensors from ODAA are used, in which about half of them are noise. This dataset sensor repository is denoted R_0.

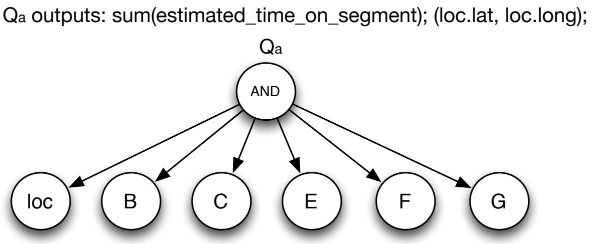

Qa outputs: sum(estimated_time_on_segment); (loc.lat, loc.long);
Qa
AND
loc
B
C
E
F
G

FIGURE 7.6: Traffic planning request for Alice (denoted Q_a): calculate the sum of the estimated travel time from A to F on the map when a traffic update from all segments are received, meanwhile, keep track of Alice's latest location.

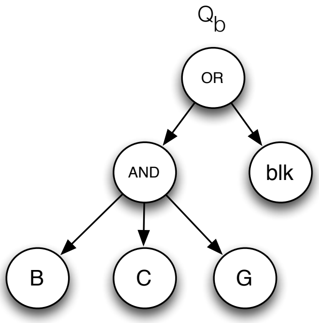


FIGURE 7.7: A variant of Bob's request (denoted Q_b): notify the user when congestion events detected on street B, C and G, or if a road construction has blocked the streets (labelled "blk")

TABLE 7.3: Simulated sensor repositories

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9
N	1	2	3	4	5	6	7	8	9
total size	1800	2700	3600	4500	5400	6300	7200	8100	9000

Each sensor in R_0 is annotated with a simulated random quality vector $< L, Acc, C, S >$ where $L \in [0ms, 300ms]$ is the latency, $Acc, C \in [50\%, 100\%]$ are the accuracy and completeness, respectively, $S \in [1, 5]$ is the security level. Price or energy consumption are not modelled in the experiments because their aggregation rules are similar to bandwidth consumption, for similar reasons availability is not modelled. The frequency of the sensor is annotated as $f \in [0.2Hz, 1Hz]$. To test the algorithms on a larger scale, the size of the sensor repository is further increased by adding N functionally equivalent sensors to each sensor in R_0 with a random quality vector, resulting in the 9 different repositories as shown in Table 7.3. In the experiments, a loose constraint[2] is used to enlarge the search space and all QoS weights are equally set to 1.0[3]. The queries used in the experiments are summarised in Table 7.4.

7.3.2.2 QoS Utility Results and Scalability

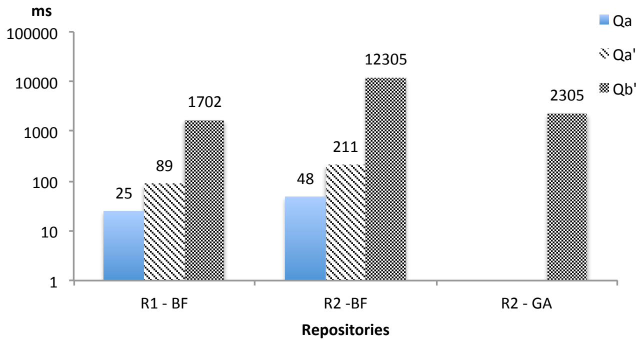
The scalability, i.e., efficiency of the GA is first compared to the BF enumeration. The results in Figure 7.8 indicate that the composition time of a Brute-Force (BF) enumeration grows exponentially with regard to the complexity of the query (number of sensors involved) and to the number of sensors in the repository. When we apply the GA over

[2] Constraint used in the evaluation: $(L \leq 3000ms, Acc \geq 0, C \geq 0, S \geq 1, B \leq 50)$

[3] In this thesis, the weights represent users' personal preferences and I do not differentiate between "good" or "bad" weight settings.

Table 7.4: Queries Used in Experiments

Queries	Description	Nodes
Q_a	Alice's query on estimated travel time on route, depicted in Figure 7.6.	1 AND operator, 6 streams
Q_b	Bob's query on whether a route has been congested or blocked, depicted in Figure 7.7.	1 AND operator, 1 OR operator, 4 streams
Q'_a	A variants of Q_a with more nodes.	1 AND operator, 3 random operators, 8 streams
Q'_b	A variant of Q_b with more nodes.	1 AND operator, 1 OR operator, 10 streams



L

Figure 7.8: Brute-Force vs. GA on R_1 and R_2

Q'_b on R_2, we save approximately 80% of the execution time. However, for Q_a and Q'_a, the solution space is too small for the GA evolution.

The usefulness of the GA is demonstrated by comparing the QoS utility of the composition plan derived by GA to a BF algorithm and a random pick algorithm. Figure 7.9 shows the experiment results for Q_a over R_3 to R_9, where Q_a has 6 service nodes and 1 operator. A more complicated variant of Q_a with 8 service nodes and 4 operators is also tested, denoted Q'_a. In particular, Figure 7.9 shows the utilities of the composition plans derived from BF, GA and a random picking approach for Q_a, Q'_a. In the experiments in Section 7.3.2.2, the GA has the following parameter settings: the initial population size is set to 200, crossover rate is 95%, mutation rate is 3%.

The best utility obtained by the GA is the highest utility of the individual in the last generation before the GA stops. Given the best utility from BF $\bar{U}_{bf}$, best utility from GA $\bar{U}_{ga}$ and the random utility of the dataset $\bar{U}_{rand}$, the degree of optimisation is calculated as $d = (\bar{U}_{ga} - \bar{U}_{rand})/(\bar{U}_{bf} - \bar{U}_{rand})$. From the results in Figure 7.9 it is observable that the average $d = 89.35\%$ for Q_a and Q'_a. In some cases the BF algorithm fails to complete, e.g., Q_a over R_8 and R_9, because of the scalability issue of brute-force enumeration. The results also show that for smaller repositories, d is bigger. This is because under the same GA settings, the GA has a higher chance of finding the global optimum during the evolution when the solution space is small and the *elitism* method described in Section

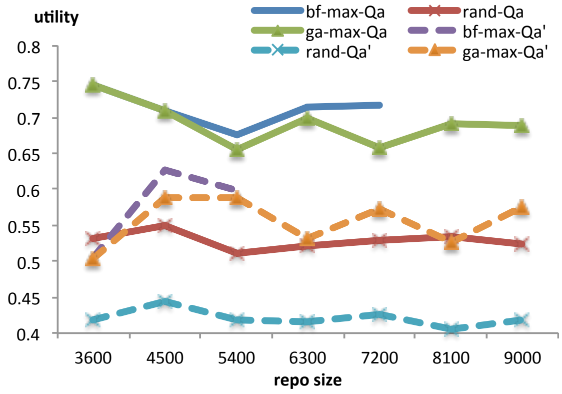


FIGURE 7.9: QoS utilities derived by brute-force, genetic algorithm and random pick

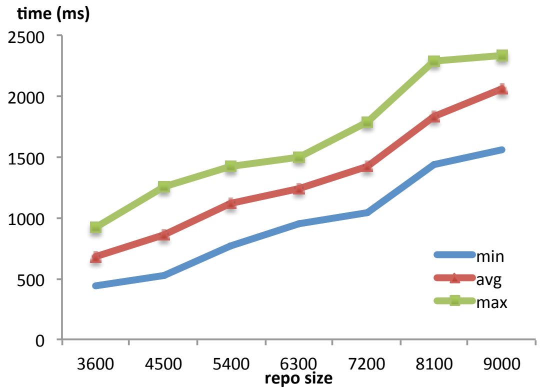


FIGURE 7.10: Genetic algorithm scalability over event service repository size

7.2.3.3 makes sure that, if found, the global optimum "survives" till the end of evolution, e.g., in the GA results for Q_a over R_3, R_4 in Figure 7.9.

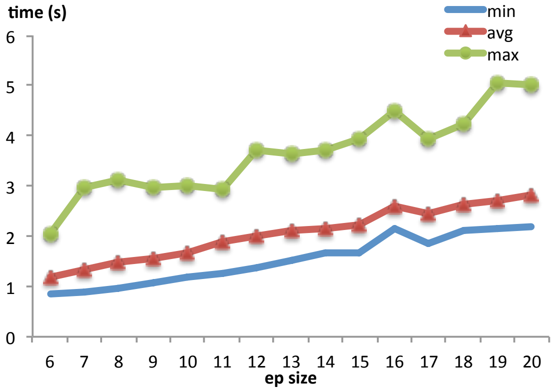


FIGURE 7.11: Genetic algorithm scalability over query pattern size

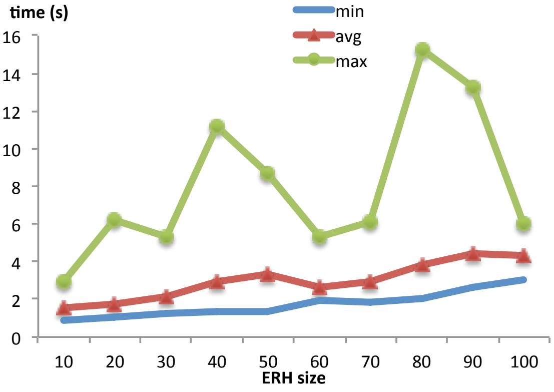


FIGURE 7.12: Genetic algorithm scalability over event reusability forest size

It is evident that a BF approach for QoS optimization is not scalable because of the NP-hard nature of the problem. The scalability of the GA is analysed using different repository sizes, queries with different sizes (the size of a query is the sum of the number of event operators and event service nodes), as well as different number of CESs in the ERF.

The results in Figure 7.10 show that the composition time of Q_a grows linearly for GA when the size of the repository increases. To test the GA performance with different event pattern sizes with different operators, the EST of Q_b is used as a base and replaces its leaf nodes with randomly created sub-trees (invalid ESTs excluded). Then the GA convergence time of these queries over R_5 is tested. The results from this experiment are detailed in Figure 7.11, and they indicate that the GA execution time increases linearly with regard to the query pattern size.

In order to test the scalability over a different number of CESs in the ERF (called ERF size), 10 to 100 random CESs are added to R_5, resulting in 10 new repositories. The GA is tested on a query created in the previous step (denoted Q'_b) with the size of 12 nodes (2 operators, 10 sensor services). The execution time of the GA is detailed in Figure 7.12. To ensure each CES could be used in the composition plan, all the CESs added are sub-patterns of Q'_b. From the results, it is observable that although the increment of the average execution time is generally linear, in some rare test instances there are "spikes", such as the maximum execution time for ERF of size 40 and 80. After analysing the results, I found that most (over 90%) of the time in those cases is used during population initialisation, and this is caused by the complexity of the ERF, i.e., the number of edges considered during ACP creation.

7.3.2.3 Fine-tuning the Parameters

In the experiments in Section 7.3.2.2, a fixed set of settings is used for the GA parameters, including crossover rate, mutation rate and population size. In order to find good settings of the GA in the given problem domain, the mutation rate (m), population size (p) and crossover rate (c) are fine-tuned based on the default setting used in Section 7.3.2.2 ($m = 0.03, c = 0.95, p = 200$), i.e., one parameter value is changed at a time while other parameters are kept unchanged.

In order to determine the effect of the parameter tuning, a Cost-Effectiveness score (i.e., CE-score) is defined as follows: given the random pick utility of a dataset $\bar{U}_{rand}$, the final utility derived by GA $\bar{U}_{ga}$ and the number of milliseconds taken for the GA to converge t_{ga}, *CE-score* $= (\bar{U}_{ga} - \bar{U}_{rand}) * 10^5/t_{ga}$. Two queries $Q_a, Q_{b'}$ are tested over two new repositories R'_5, R'_9, which are R_5 and R_9 with 50 and 100 additional CESs, respectively. Hence there are 4 test combinations on both simple or complex queries and repositories. The results for fine tuning the mutation rate, population size and crossover rate are shown in Figure 7.13, 7.14 and 7.15.

From the results of fine-tuning the mutation rate shown in Figure 7.13, it is observable that the optimal mutation rate is quite small for all tests, i.e., from 0% to 0.4%. Results of fine-tuning the population size are shown in Figure 7.14, they indicate that for smaller solutions spaces such as Q_a over R'_5 and R'_9, the optimal initial population size is smaller, i.e., with 60 individuals in the initial population. For more complicated queries and larger repositories, using a larger population size e.g., 100, is more cost-efficient. Results from Figure 7.15 indicate that for Q_a over R'_5, the optimal crossover rate is 35%, because the global optimum is easier to achieve and more crossover operations bring overhead. However, for more complicated queries and repositories, a higher crossover rate, e.g.,

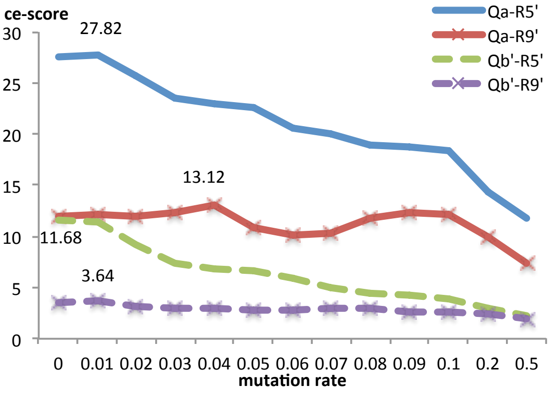


FIGURE 7.13: Cost-effectiveness score over mutation rate

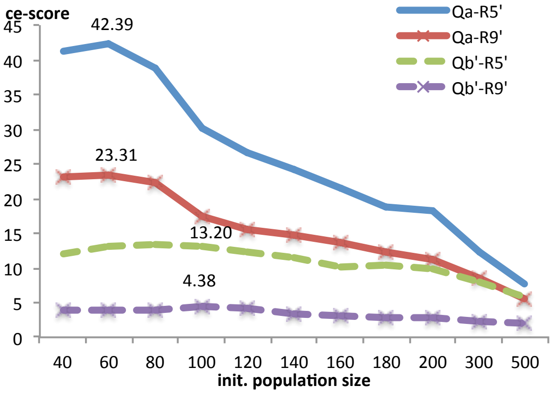


FIGURE 7.14: Cost-effectiveness score over population size

from 90% to 100%, is desired. It is worth noticing that in the results from Figure 7.13, 7.14 and 7.15, the changes of the score for Q_b' over R_9' is not significant. This is due to the fact that the GA spends much more time (about 90% of the time) trying to initiate the population, making the cost-effectiveness score small and the differences moderate.

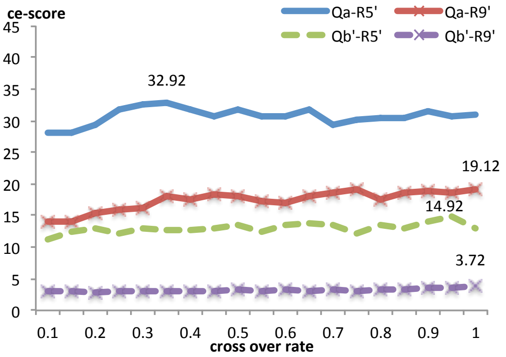


FIGURE 7.15: Cost-effectiveness score over cross over rate

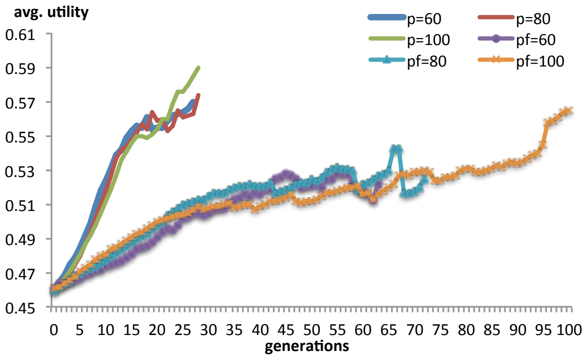


FIGURE 7.16: Average utility over generations using reducible (marked "p=x", "x" represents the initial size) and fixed (marked "pf=x", "x" represent the size) population

In the previous experiments in this chapter, every individual is chosen to reproduce once (except for the elite whose copy is also in the next generation). This will ensure the population will get smaller as the evolution progresses and the GA will converge quickly. This is desirable because the algorithm is called at run-time and is time sensitive. However, it is also possible to allow an individual to reproduce multiple times and keep a fixed population size during the evolution.

In order to compare the differences of having a fixed or reducible population size, the average utility (of Q_b' over R_9') over the generations is shown in Figure 7.16. The results show that the evolution iterations (number of generations before converged) for reducible population sizes are similar, while larger initial sizes achieve higher utilities. Meanwhile, the evolution iterations in fixed population sizes are very different: for a fixed population size of 60 the GA converges in about 60 generations and for the size of 100 it lasts more than 100 generations. Larger sizes also produce better final results in a fixed population, but it is much slower and the utilities are lower than those obtained from reducible populations. In summary, using a reducible population size is better than a fixed population size using the GA described in this problem setting.

7.3.3 Part 2: Validation of QoS Aggregation Rules

In this part of the evaluation, the validity of the QoS aggregation rules is demonstrated using simulated data.

7.3.3.1 Datasets and Experiment Settings

In order to demonstrate the effect of QoS aggregation and optimisation, two composition plans are generated by GA for Q_a over R'_9 using the same constraints as in Section 7.3.2.1: CP_1 is optimised for latency, with the weight of latency set to 1.0 and other QoS weights set to 0.1; while CP_2 is optimised for bandwidth consumption, with the weight of bandwidth consumption set to 1.0 and others 0.1. The reason these two plans are used is that the they are the most different in structure, as shown in Figure 7.17.

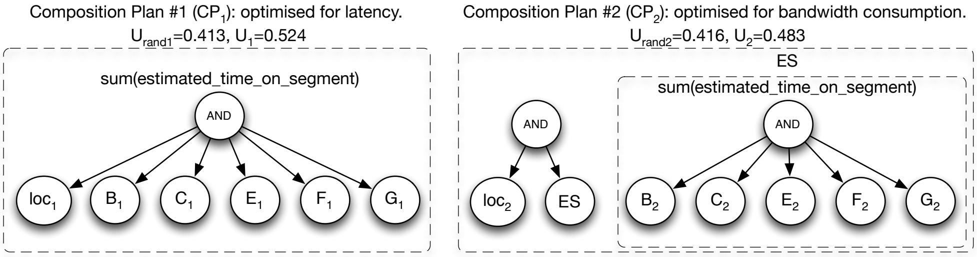
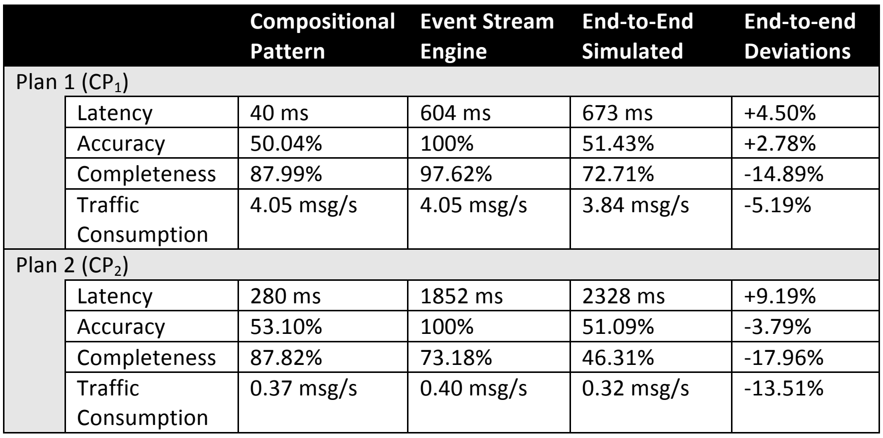

Composition Plan #1 (CP$_1$): optimised for latency.
U$_{rand1}$=0.413, U$_1$=0.524
sum(estimated_time_on_segment)
AND
loc$_1$
B$_1$
C$_1$
E$_1$
F$_1$
G$_1$
Composition Plan #2 (CP$_2$): optimised for bandwidth consumption.
U$_{rand2}$=0.416, U$_2$=0.483
ES
sum(estimated_time_on_segment)
AND
AND
loc$_2$
ES
B$_2$
C$_2$
E$_2$
F$_2$
G$_2$

FIGURE 7.17: Composition plans for Q_a under different weight vectors

When the two composition plans are generated, the composition plans are transformed into stream reasoning queries (i.e., CSPARQL query) using the query transformation algorithm defined in [211]. The queries are evaluated against the semantically annotated traffic data collected from ODAA sensors. According to the composition plan as well as the quality annotations of the event services (both sensor services and CESs) involved in the plans, the event streams are simulated on a local test machine, i.e., artificial delays, wrong and lost messages are created according to the QoS values in the quality vector, the sensor update frequency is set to be the frequency annotated (so as to affect the messages consumed by the query engine). Security is annotated but not simulated,

Table 7.5: Validation for QoS aggregation and estimation.



		Compositional Pattern	Event Stream Engine	End-to-End Simulated	End-to-end Deviations
Plan 1 (CP_1)					
	Latency	40 ms	604 ms	673 ms	+4.50%
	Accuracy	50.04%	100%	51.43%	+2.78%
	Completeness	87.99%	97.62%	72.71%	-14.89%
	Traffic Consumption	4.05 msg/s	4.05 msg/s	3.84 msg/s	-5.19%
Plan 2 (CP_2)					
	Latency	280 ms	1852 ms	2328 ms	+9.19%
	Accuracy	53.10%	100%	51.09%	-3.79%
	Completeness	87.82%	73.18%	46.31%	-17.96%
	Traffic Consumption	0.37 msg/s	0.40 msg/s	0.32 msg/s	-13.51%

because the aggregation rule for security is trivial, i.e., estimated to be the lowest security level. Notice that the simulated quality is the *Service Infrastructure* quality (see Section 7.1.2). The results and the query performance over these simulated streams are observed and compared with the QoS estimation using the rules in Table 7.1 and 7.2, to see the differences between the practical quality of composed event streams and the theoretical quality as per estimation.

7.3.3.2 Simulation Results

The results of the comparison between the theoretical and simulated quality of the event service composition are shown in Table 7.5. The first column is the quality dimensions of the two composition plans, the second column is the computed quality values based on the aggregation rules defined in Table 7.2. The rules take into account the *Composition Pattern* of the query as well as the *Service Infrastructure* quality of the composed services. This quality is denoted QoS_{cp}. However, this is not the end-to-end QoS, because the quality of the event stream engine needs to be considered. To get the stream engine performance, the queries are deployed with optimal *Service Infrastructure* quality, i.e., no artificial delay, mistake or loss, and the performance are recorded in the third column. The engine quality is denoted QoS_{ee}. The simulated end-to-end quality is recorded in the fourth column, denoted QoS_s. The theoretical end-to-end quality based on QoS_{cp} and QoS_{ee} is calculated using Table 7.1. This theoretical end-to-end quality is denoted QoS_t and calculate the deviation $d = (QoS_s/QoS_t) - 1$, which is recorded in the last column.

From the results it is observable that the GA is very effective in optimising latency for CP_1 and bandwidth consumption for CP_2: latency of the former is 1/7 of the latter and event messages consumed by the latter are less than 1/8 of the former.

It is also observable that the deviations of latency and accuracy are moderate for both plans, however, the completeness estimation is about 15% to 18% different to the actual completeness. For the bandwidth consumption in CP_1 the estimation is quite accurate, i.e., about 5% more than the actual consumption. However, the bandwidth consumption for CP_2 deviates from the estimated value by about 13.51%. The difference is caused by the unexpected drop in CSPARQL query completeness when a CES with imperfect completeness is reused in CP_2, which suggests that an accurate completeness estimation of a service could help improving the estimation of the bandwidth consumption for event service compositions using the service. Indeed this shows that the utility between completeness and network consumption are not independent, a workaround will be prohibiting users to specify non-zero preferences on both parameters, i.e., when the user specify the preference of either parameter to be zero, the utility contribution of completeness will not be calculated twice for network consumption (which leads to biased results).

Another interesting observation from Table 7.5 is that the end-to-end delay of CP_2 is about 1650 ms longer than CP_1, while the artificial delay imposed by the CES is no more than 280 ms. This is caused by the internal query mechanism of the CSPARQL engine: when composed queries are registered to the same engine instance, the engine will take much more time to process the query because of the concurrency. This suggests that a federated manner of query composition over distributed engine instances is desirable.

7.4 Related Work

In various event publish/subscribe system [212] architectures, Event Broker Networks (EBN) has gained much research interests in recent years because of the scalability of this architecture. Early works in EBN can be found in [44, 213]. However, these systems have very different characteristics and they do not provide a common ground for comparing and evaluating them in terms of service quality [203]. In [203], the authors identify important QoS metrics in EBN systems, and describe their application to the publish-subscribe model. A survey in [32] review a selection of eight EBN middlewares and analyse their support for five QoS metrics: latency, bandwidth, reliability, delivery semantics and message ordering. According to the survey, the support for QoS-aware event routeing is quite poor: three of the eight middleware do not provide any support for the five metrics. Jedi [206] supports message ordering, Gryphon [207] supports delivery semantics, Hermes [213] and Medym [214] supports reliability, IndiQoS [202] supports latency and bandwidth. The above QoS-aware EBNs (and many other peer works) develop various routeing policies to determine event distribution routes with the optimal performance. However, they don't provide a holistic view of QoS aspects for event services, also, most of them do not cater for complex events and event compositions. Moreover, the routeing decisions are made based on network analysis, e.g., link status and node resources, while in this chapter, the QoS propagation is also influenced by the event engine and composition pattern.

The first step of solving the QoS-aware service composition problem is to define a QoS model, a set of QoS aggregation rules and a utility function. Existing works have discussed these topics extensively, e.g., by Alrifai et al. [201], Jaeger et al. [84] and Mela et al. [85]. In this chapter some typical QoS properties are extracted from the existing work and analysed. However, the aggregation rules in existing works focus on conventional web services rather than complex event services, which need a different QoS aggregation schema. For example, the event engine also has an impact on the QoS aggregation, which is not considered in conventional service QoS aggregation. Also, the aggregation rules for some QoS properties based on event composition patterns is different to those based on workflow patterns (as in Jaeger et al. [84]), as explained in detail in Section 2.3.2.2 and 7.1.2.

As a second step, different concrete service compositions are created and compared with regard to their QoS utilities to determine the optimal choice. To achieve this efficiently, various heuristic approaches are developed. There are two prominent strands: Integer Programming (IP) (e.g., [87, 201, 210]) and Genetic Algorithm (GA) (e.g., [89–92]) based solutions. The studies in [215, 216] discuss some other heuristics used besides GA, e.g., Simulated Anealling, Tabu Search etc. In the following, mainly IP- and GA-based approaches are discussed, because of their relevance and popularity.

Zeng et al [87] elaborate the limitation of local optimisation and the necessity of global planning. The authors propose to address the complexity problem of global planning by introducing an IP-based solution with an SAW based utility function to determine the desirability of an execution plan. This approach is extended by Berbner et al. [210] with more heuristics to promote efficiency. Alrifai et al. [201] propose a hybrid approach of local and global optimisation, in which global constraints are delegated to local tasks, and the constraint delegation is modelled as an IP-based optimisation problem. The problem with IP-based solutions is that they apply only on a fixed set of service classes in a composition plan, without discussing how different service classes on different granularity levels are generated. Moreover, since the constraint allocation relies on the QoS distribution of the existing service candidates, the overhead of recomputing constraints are significant in dynamic environments [87], such as IoT-based event services studied in this chapter.

Approaches	Differences to QoS-aware discovery and composition in ACEIS
Event Broker Networks, e.g., Jedi [206], Gryphon [207], Hermes [213], Medym [214] and IndiQoS [202]	Content-based subscription partitioning, do not support pattern-based decomposition for complex events, routing algorithms based on network analysis, do not consider QoS influenced by event engine or composition pattern.
Integer programming based, e.g., Zeng et al [87], Berbner et al. [210], Alrifai et al. [201]	Do not scale well when number of service candidates increases (except for Alrifai et al. [201]), apply to a fixed set of service classes in a service composition plan, not suitable in dynamic environments.
Canfora et al. [90]	Use fixed length encodings.
Zhang et al. [91]	Use two-dimensional encodings to capture all execution paths, no validation for crossover.
Gao et al [92]	Use tree coding chromosomes with validations on crossover and mutation.
Zhang et al. [89]	Genetic encoding length not fixed, poor readability of the genomes.
Wu et al. [85]	Can compose services on different granularity levels using Generalised Component Services.
All existing GA	Use IOPE-based service matchmaking, not suitable for complex event services.

Table 7.6: Related works in QoS-aware service composition

Several GA based solutions are therefore proposed for address QoS-aware web service optimisation, some of them (e.g., [90, 217]) provides efficiency analysis showing that the GA-based solutions can provide on-demand, close to real-time solutions within milliseconds to seconds. Zhang et al. [89] encode the chromosomes of service compositions with binary bits representing whether a service is selected or not. The problem with this approach is that the readability of the genomes are poor and the chromosome length is not fixed during evolution. Canfora et al. [90] use a different encoding approach, which leads to a fixed chromosome length. In the work done by Zhang et al. [91], a two-dimensional genome encoding is proposed to express all execution paths while considering task relations, but crossover and mutation need validation. Gao et al [92] use tree coding chromosomes, crossovers operate on sub-trees and mutation operate on leaf nodes to avoid invalid reproductions. Karatas et al. [218] develop a GA-based approach that goes beyond QoS-aware composition and enables compliance-aware service composition.

The above GA-based approaches can only evaluate service composition plans with fixed sets of service tasks (abstract services) and cannot evaluate composition plans which are semantically equivalent but consists of different service tasks, i.e., service tasks on different granularity levels. A more recent work done by Wu et al. [85] addresses this issue by presenting the concept of Generalised Component Services (GCS) and developing the GA encoding techniques and genetic operators based on GCS. Results in [85] indicate that up to 10% utility enhancement can be obtained by expanding the search space. Composing events on different granularity levels is also a desired feature for complex event service composition. However, [85] only caters for *Input, Output, Precondition* and *Effect* (IOPE) based service compositions. Complex event service composition requires an *event pattern* based reuse mechanism [167]. As a result, it requires different genetic encoding mechanisms and crossover operations. Table 7.6 summarises the comparison of the related works in this chapter.

7.5 Summary and Discussion

In this chapter, the issue of enabling QoS-aware event service composition and optimisation using services is addressed. A QoS aggregation schema is proposed to calculate the overall QoS vector for an event service composition. Based on a user-defined constraint and weight vector, a QoS utility function is defined to calculate the performance event composition. A genetic algorithm for creating optimal event service compositions is developed and evaluated. The proposed approach is evaluated over a travel planning scenario with both real and synthetic datasets. The experimental results show that the genetic algorithm is scalable (the execution time increases linearly) over the service repository size, query pattern size and ERF size. It gives near-optimal (about 89% optimal) results efficiently, i.e., within 2.5 seconds for a repository with 9000 services/sensors. The fast convergence of the GA allows it to be used for on-demand optimisation of CESs, where the user may tolerate a waiting period of seconds for online applications but not minutes [219]. The study in [219] shows that the users rate the service quality as low when waiting for more than 10 seconds. Considering the GA in this chapter rarely spends more than 10 seconds in our scalability test, it is safe to say that it can be used for an online application.

Guidelines on how to fine-tune the GA parameters including mutation rate, crossover rate and population size in order to achieve the best performance of GA are provided. The experiment results show that the optimal mutation rate is between 0% to 0.4%, the optimal crossover rate is smaller (about 35%) in cases of simple queries and small repositories, and is greater (about 95%) when complicated queries and larger repositories are used. The experiments also show that having a reducible initial population size out-performs using a fixed population size. The experiments on the validation of QoS aggregation shows that the estimation model does not deviate too far from the practical results (± 2.78% to 17.96%). Notice that in this chapter, only standard GA algorithm with elitism is implemented, and we leave optimisations of the standard GA (e.g., Breeder GA [220]) for the future work. Also, when the QoS constraints cannot be fulfilled, we simply notify the user about the composition failure. Some future works can be done in this direction to negotiate the constraints with the users, similar to the SLA-based service negotiations discussed in [221].

The composition plan derived in this chapter is not yet executable, in the next chapter, the implementation of composition plans are discussed, i.e., how the executable RSP queries are generated from the composition plans is elaborated. Moreover, mechanisms allowing run-time adjustments for the composition plans are discussed.

Chapter 8

Automatic Event Service Implementation and Adaptation[*]

Using the methods described in Chapter 7 it is possible to efficiently create event service composition plans with good Quality-of-Service (QoS) performance. However, these composition plans cannot be executed directly, they need to be translated into event/-data stream queries executable by event/stream processing engines. Moreover, although these event service compositions have good QoS performance when they are created and deployed, their performance may change during execution, especially when using Internet-of-Things (IoT) services, whose performance may be affected by environmental changes [223]. In this chapter, the means for facilitating an automatic and adaptive semantic event service implementation are discussed, in order to implement the *Query Transformer* and *Adaptation Manager* in the Automatic Complex Event Implementation System (ACEIS), as shown in Figure 8.1.

ACEIS supports different event engines for evaluating event patterns defined with the extended Business Event Modelling Notation (BEMN$^+$). However, different event engines may support different sets of operators. In order to correctly implement the Complex Event Services (CESs), i.e., ensure the same event request implemented over different engines can produce the same set of results, it is necessary to compare and align the semantics of the event operators in BEMN$^+$ with the query operators used in the event engines. Then, different query transformation algorithms are needed for different event engines which translate the patterns described in BEMN$^+$ into queries executable on the target event engines. In Smart City applications, especially in the applications based on IoT services, it is unrealistic to assume an implementation of a CES (based on the query generated from the transformation algorithms) can always remain valid during

[*] Part of the content in this chapter is published in [211, 222].

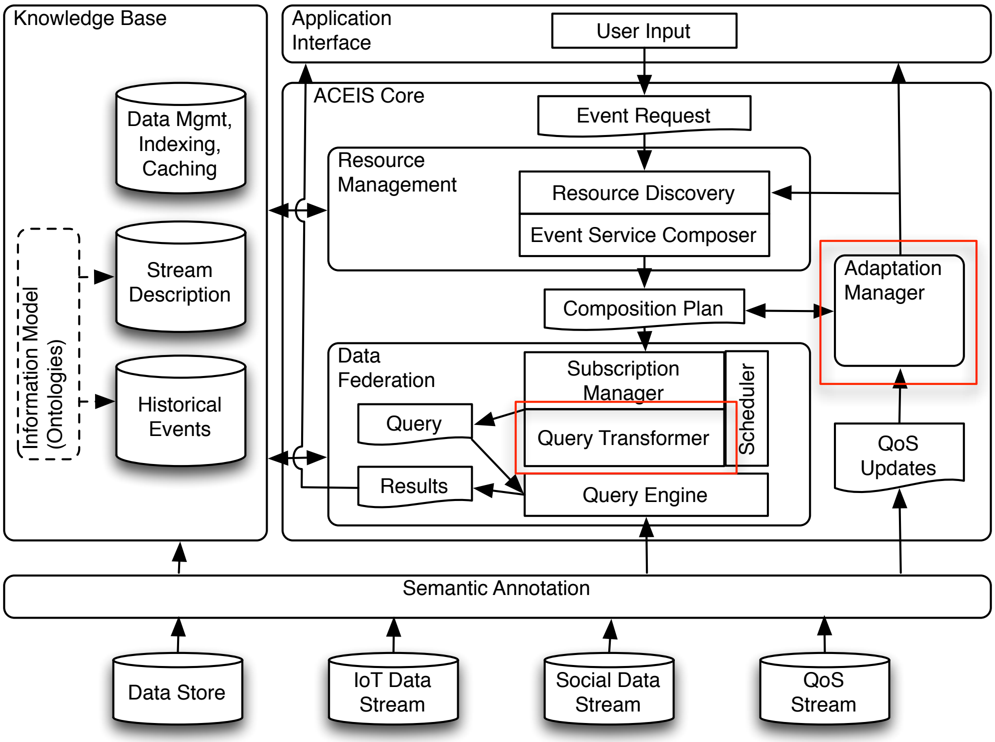


Figure 8.1: The query transformer and adaptation manager modules in ACEIS

the lifespan of the CES. IoT-based data streams are inherently dynamic in nature and somehow unreliable, therefore more prone to fluctuations in quality. For example, a wireless sensor could be offline due to network issues, the accuracy of a sensor might be affected by its battery level [224], air temperature, humidity [225] etc. In order to maintain the performance of a CES implementation, it is needed to continuously monitor the QoS of the event services involved in the composition and determine if an adaptation is required, when a QoS update is received. It is also important to determine the scope of changes when an adaptation is needed, since there is typically a trade-off between efficiency and effectiveness of the adaptation, depending on how much changes are needed. To summarise, an automatic and adaptive event service implementation needs to consider the following issues:

1. How to ensure the correctness of the stream query transformation over different stream query languages?
2. What are the suitable algorithms to create queries for stream processing engines?
3. How to determine whether an event service adaptation is necessary?
4. When an adaptation is required, what are the suitable adaptation strategies and process?

The remainder of this chapter is organised as follows. Section 8.1 addresses the first and second issues by aligning the semantics of event patterns in BEMN$^+$ and the query semantics of existing RDF Stream Processing (RSP) engines and presenting the query transformation algorithms. Section 8.2 addresses the third issue by using the QoS aggregation methods discussed in Section 7.1. Section 8.2 also addresses the fourth issue by elaborating different QoS-aware event service adaptation strategies as well as the adaptation process in ACEIS. Section 8.3 evaluates the adaptation strategies with both real and synthetic datasets in Smart City environments. Section 8.4 discusses the related works before Section 8.5 summarises.

8.1 Automatic Event Service Implementation

Implementing a Primitive Event Service (PES) is trivial, i.e., simply making a subscription to the PES is sufficient. The implementation of a CES is more complicated and it involves the following steps. Upon receiving an event composition plan for a CES, the *Subscription Manager* makes subscriptions to the relevant event sources using the service bindings provided in the composition plan. Then, the query transformer creates stream queries and registers the queries at the stream engine according to the event pattern specified in the composition plan. In this section, the algorithms for transforming composition plans into RSP queries are discussed.

```
:Observation_1  a  ssn:Observation;
                ssn:observedBy :sampleTrafficSensor
                ssn:observedProperty [ a ct:EstimatedTime];
                ssn:featureOfInterest :FoI_1;
                ssn:observationResult :observationResult_1.
:observationResult_1 ssn:hasValue
              [ ssn:hasQuantityValue "25"^^xsd:integer;
                muo:unitOfMeasurement muo:second].
```

Listing 8.1: Traffic sensor stream data in Semantic Sensor Network Ontology

In the current ACEIS implementation, CQELS and CSPARQL are used as the RSP engines. ETALIS engine is not integrated into the current prototype implementation but the semantic alignments for ETALIS operators are presented. These engines consume semantically annotated events. The query transformation algorithms in ACEIS depends on the schema of annotated events, i.e., the ontologies used. However, they can adapt to different event ontologies with little effort as long as the essential information (i.e., the source of event and event payload) is provided. Without loss of generality, the primitive events in the smart city context are assumed to be annotated as sensor observations in the SSN ontology. A sample traffic sensor reading annotated as *Observation* in SSN is shown in Listing 8.1. In the following the semantics alignments of the event operators in BEMN$^+$ and RSP query operators are discussed. Then the detailed query transformation algorithm for generating CQELS and CSPARQL queries based on the semantic alignments are presented.

8.1.1 Semantics Alignment

To ensure the query transformation creates queries that detect the right event patterns, it is required to map the semantics of event operators to query operators. Table 8.1 summarises how event operators in BEMN$^+$ can be implemented by query operators in CQELS, CSPARQL and ETALIS. In the following, the details are discussed.

TABLE 8.1: Semantics Alignment for Event Operators

BEMN$^+$	E	And	Or	Seq	Rep_o	Rep_n	Sel	Filter	Window
CQELS	SGP	$\bowtie$	⟕	-	-	-	BGP+$proj$	Filter	Window
CSPARQL	BGP	$\bowtie$	⟕	f_t	$\bowtie +f_t$	f_t^+	BGP+$proj$	Filter	Window
ETALIS	BGP	$\bowtie$	⟕	SeqJoin	$\bowtie$ +SeqJoin	SeqJoin$^+$	BGP+$proj$	Filter	getDuration()

8.1.1.1 EventDeclaration

An *Event Declaration* E in an event pattern ep indicate the occurrences of event instances of type E. As shown in Listing 8.1 the occurrences of sensor events are annotated as observations. If one uses SPARQL to query the occurrences of sensor observations, a single triple pattern can suffice:

$$t = (\text{?id}, \text{rdf:type}, \text{ssn:Observation})$$

Given a set of mappings Ψ, $u \in \Psi$ is a partial function from variables to values, such that $u(var(t))$ gives the mapping value, i.e., the Internationalized Resource Identifier (IRI) of the subject of the triple t where $var(t)$ is the set of variables in t. To get only the observations produced by E, a *BasicGraphPattern* (BGP)

$$P = (t \cup (\text{?id}, \text{ssn:observedBy}, E.src))$$

can be used, where $E.src$ is the source (i.e., service id) of E specified in the composition plan. Then, $\Psi(var(P))$ gives all the IRIs of sensor observations produced by E. $\Psi(P)$ gives the set of triples by replacing the variables in t with corresponding values from Ψ. These triples are called *event id triples* for E, denoted $T_{id}(E)$ and this pattern *event id pattern* for E, denoted $P_{id}(E)$. Indeed the existence of $T_{id}(E)$ indicates the occurrence of an event instance of type E in the dataset (i.e., event stream). Notice that $T_{id}(E)$ should contain only 1 sensor observation if E is primitive, otherwise it may contain more than 1 observations, which are the member event instances in the EIS triggering E. The engines in Table 8.1 reuse and extend the query semantics of SPARQL, therefore the same BGPs[1] can be used to query the occurrence of events instances of type E.

8.1.1.2 AND Operator

An *And* operator indicates instances of the connected 2 sub-event types E_1, E_2 should occur, i.e., Given $E_3 := \wedge(E_1, E_2)$, $T_{id}(E_3) = T_{id}(E_1) \cup T_{id}(E_2)$, where $T_{id}(E_1) \neq \emptyset \wedge T_{id}(E_2) \neq \emptyset$. This event operator can be implemented by *join* ($\bowtie$) in SPARQL. Given P_1, P_2, Ψ_1, Ψ_2 such that $\Psi_1(P_1) = T_{id}(E_1)$, $\Psi_2(P_2) = T_{id}(E_2)$, it is evident that Ψ_1 *join* Ψ_2 creates a new set of mappings $\Psi_3 = \Psi_1 \bowtie \Psi_2$ such that $dom(u_3) = dom(u_1) \cup dom(u_2)$ where $u_1 \in \Psi_1, u_2 \in \Psi_2, u_3 \in \Psi_3$. Notice that u_1, u_2 are always compatible because they are disjoint. Since u_3 is also a partial function, it must provide mapping values for each variable $v \in dom(u_3)$, i.e., $\Psi_3 = \emptyset \iff \Psi_1 = \emptyset \vee \Psi_2 = \emptyset$. The *Join* operator in SPARQL is reused in the semantic stream query engines so that the *And* operator can be implemented by *join*. However, using *join* is only correct for the *cumulative* event instance selection policy (recall Section 5.2.4.1 and 5.2.4.2), since all mappings, i.e., event instance sequences fitting the pattern, are picked. If the selection policy is configured as *last*, a result processing program is needed to filter out all variable bindings that appeared in previous query solutions.

8.1.1.3 OR Operator

An *Or* operator indicates at least one of its sub-events should occur, i.e., Given $E_4 := \vee(E_1, E_2)$, $T_{id}(E_4) = T_{id}(E_1) \cup T_{id}(E_2)$, where $\neg(T_{id}(E_1) = \emptyset \wedge T_{id}(E_2) = \emptyset)$. It can be implemented by using *LeftOuterJoin* ($⟗$) operator with *bound* filters in SPARQL. To do that a new set of mappings: $\Psi_4 = ⟗\ \Psi_1 ⟗ \Psi_2$ is created, where Ψ_4 satisfies the condition:

$$\forall u_4 \in \Psi_4, \exists v4 \in dom(u_4) \Rightarrow bound(v_4) = true$$

It is evident that Ψ_4 can be implemented by the *OPTIONAL* keyword and the condition can be implemented by a set of *bound* filters.

[1] In CQELS *StreamGraphPattern* (SGP) is used as an extension of BGP

8.1.1.4 SEQUENCE Operator

A *Sequence* operator requires all its sub-events to occur in a temporal order, e.g., $E_5 :=; (E_1, E_2)$. To implement E_5, it is required to join event id triples based on their timestamps. In ETALIS a *SeqJoin* operator is defined as an extension of SPARQL *join*. For brevity the readers are referred to [19] for detailed definition. In CSPARQL such an extension does not exist. However, CSPARQL provides a function f_t to query the timestamp of a variable mapping, denoted $f_t(v)$ where $v \in dom(u)$ is a variable in a mapping u. Using this function a set of mappings $\Psi_5 = \Psi_1 \bowtie \Psi_2$ is created such that: $\forall u_5 \in \Psi_5, u_5 = u_1 \bowtie u_2$ where $u_1 \in \Psi_1, u_2 \in \Psi_2$, $f_t(v_1) < f_t(v_2)$ holds for all $v_1 \in dom(u_1) \cap dom(u_5)$ and $v_2 \in dom(u_2) \cap dom(u_5)$. Intuitively, this condition ensures all event instances of type E_1 occurred before those of type E_2. Currently CQELS (public version 1.0.0) does not support *SeqJoin* or provide functions to access the timestamps of the stream triples, therefore *Sequence* is not supported in CQELS.

8.1.1.5 REPETITION Operator

Repetition is a generalization of sequence, recall definitions in Section 5.2.4.1, an overlapping (i.e., Rep_o) or non-overlapping (i.e., Rep_n) repetition can be transformed into a conjunction of sequence or a sequence of sequence, respectively. Therefore, repetition can be implemented in CSPARQL and ETALIS by combining the ways they implement $\wedge$ and ; event operators, while CQELS does not support repetition because sequence is not allowed in CQELS.

8.1.1.6 Selection

Selection retrieves event payloads from member event instances. If payload $p \in D$ where D is the set of payloads for event E is selected, information on p can be queried by adding triple patterns to $P_{id}(E)$:

(?id ssn:observationResult ?x. ?x ssn:hasValue ?v...)

and *project* the relevant variables into the query results. For brevity not all triple patterns required are listed here.

8.1.1.7 Filter and Window

Filter and *Window* operators in event patterns is be mapped to *Filter* and *Window* operators in the three engines, respectively. Notice that in ETALIS an explicit *Window* operator does not exist, the window operator is implemented by using a filter $F(getDuration() < \delta, \Psi)$ where *getDuration* is a function retrieving the duration all mappings in Ψ and δ is a time interval.

8.1.1.8 Data or Time Driven Query Execution

CQELS uses a data-driven approach to invoke query execution, i.e., whenever new data arrives in the window, the query is evaluated against the data in the current window. However, CSPARQL uses a time driven approach, in which a query is executed periodically, whenever the window slides. In order to have the same results produced by CQELS and CSPARQL engines, a post-processing filter is deployed on the CQELS result handler that reports only the results when the time window slides and simulates the time driven query execution.

8.1.2 Transformation Algorithm

The event pattern specified in a composition plan specifies the event query to be deployed and evaluated on the RSP engine. In this section the algorithms for parsing Event Syntax Trees (ESTs) of event patterns and creating semantic stream queries (i.e., CQELS and CSPARQL queries) based on the semantics alignments presented in Section 8.1.1 are described. Recall that in an EST, the nodes can be event operators in four types: *Sequence, Repetition, And* and *Or*, or they can be member event declarations; the edges represent the provenance relation in the complex event detection: the parent node is detected based on the detection of the child nodes.

8.1.2.1 CQELS Query Transformation

Using a top-down traversal of the event pattern tree and querying the semantics alignment table for each event operator encountered during the traversal, the event pattern in the composition plan is transformed into a CQELS query following the divide-and-conquer style. Algorithm 7 shows the pseudo code of the main parts of the query transformation algorithm.

Lines 1 to 6 in Algorithm 7 construct the CQELS query with three parts: a pre-defined query prefix, a select clause derived from the *getSelectClause()* function and a where

Algorithm 7 Transform event patterns into CQELS queries.

```
Require: Composition Plan: $comp$, Query Prefix String $prefixStr$
Ensure: CQELS Query String: $queryStr$
 1: procedure TRANSFORM($comp, prefixStr$)
 2:     $selectClause \leftarrow$ GETSELECTCLAUSE($comp.ep$)
 3:     $whereClause \leftarrow$ GETWHERECLAUSE($comp.ep$)
 4:     $queryStr \leftarrow prefixStr + "SELECT" + selectClause + "WHERE" + whereClause$
 5: return $queryStr$
 6: end procedure
Require: Event Pattern: $ep$
Ensure: Where Clause String: $whereClause$
 7: procedure GETWHERECLAUSE($ep$)
 8:     $root \leftarrow$ GETROOTNODE($ep$), $whereClause \leftarrow \emptyset$
 9:     if $root \in Op_{seq} \cup Op_{rep}$ then
10:         fail and terminate
11:     else if $root \in$ EventServiceDescription then
12:         $whereClause \leftarrow$ GETSGP($ep, root$)
13:     else if $root \in Op_{and}$ then
14:         for $subPattern \leftarrow$ GETSUBPATTERNS($ep, root$) do
15:             $whereClause \leftarrow whereClause +$ GETWHERECLAUSE($subPattern$)
16:         end for
17:     else if $root \in Op_{or}$ then
18:         for $subPattern \leftarrow$ GETSUBPATTERNS($ep, root$) do
19:             $whereClause \leftarrow whereClause +$ "optional" $+$ GETWHERECLAUSE($subPattern$)
20:         end for
21:         $whereClause \leftarrow whereClause +$ GETBOUNDFILTERS($ep$)
22:     end if
23:     if $filters \leftarrow$ GETFILTERS($ep$) $\neq \emptyset$ then
24:         $whereClause \leftarrow whereClause +$ GETFILTERS($filters$)
25:     end if
26: return "{" $+ whereClause +$ "}"
27: end procedure
```

clause derived from the *getWhereClause()* function. Lines 7-27 define the *getWhereClause()* function in a recursive way. It takes as input the event pattern in the composition plan (Line 7) and finds the root node in the event pattern (Line 8). Then, it investigates the type of the root node: if it is a *Sequence* or *Repetition* operator, the transformation algorithm terminates, currently transformation cannot be applied for *Sequence* or *Repetition* because of the limitations of the underlying query language (CQELS) (Lines 9-10). If the root node is an event service description, a *getSGP()* function creates the Stream Graph Patterns (SGP) in CQELS (Lines 11-12) describing the triple patterns of the observations delivered by the event service, and this SGP is returned as a part of the where clause. If the root node is an *And* or *Or* operator, the algorithm invokes itself on all sub-patterns of the root node and combines the where clauses derived from the sub-patterns (Lines 13-20). In addition, if the root is an *Or* operator, an *OPTIONAL* keyword is inserted for each *where* clause of the sub-pattern and a bound filter is created indicating at least one of the sub-patterns has bound variables (at least one sub-events occurs, Line 21). If there are filters specified in the event

```
Select ?locId ?es4 ?value1 ... Where {
    Graph <http://purl.oclc.org/NET/ssnx/ssn#>
        {?ob rdfs:subClassOf ssn:Observation.}
    Graph <http://sampleStaticKB>
         {?es4 ct:owner foaf:Alice}
    Stream <locationStreamURL> [range 5s]{
        ?locId rdf:type ?ob. ?locId ssn:observedBy ?es4.
         ?locId ssn:observationResult ?result1.
         ?result1 ssn:hasValue ?value1.
         ?value1 ct:hasLongtitude ?lon. ?value1 ct:hasLatitude ?lat.
         ?loc ct:hasLongtitude ?lon. }
    Stream <trafficStreamURL1> [range 5s] {
        ?seg1Id rdf:type ?ob. ?seg1Id ssn:observedBy ?es1.
        ?seg1Id ssn:observationResult ?result2.
         ?result2  ssn:hasValue ?value2.
         ?value2 ssn:hasQuantityValue ?eta1.}
    Stream <trafficStreamURL2> [range 5s] {...}
    Stream <trafficStreamURL3> [range 5s] {...} }
```

Listing 8.2: CQELS query example

pattern, a *getFilters()* function is invoked to add the filter clauses to the where clause (Lines 23-25). Finally, the where clause is returned with a pair of brackets (Line 26).

Listing 8.2 shows the transformation result for an event request similar to the query in Figure 7.6. Notice that the first graph pattern (?ob rdfs:subClassOf ssn:Observation) is used to join the SGPs in the query only because CQELS does not allow disjoint *join*. Also, *getSGP()* function can insert static graph patterns to combine the dynamic triples with static background knowledge, if such information is necessary (i.e., expressed in the event requests).

8.1.2.2 CSPARQL Query Transformation

The transformation algorithm for CSPARQL queries is shown in Algorithm 8. It has the same structure as Algorithm 7. It has a main method (i.e., *transform()*) taking the composition plan and query prefix mappings as input to produce the query string as output. The *transform()* method invokes several sub-methods to organise the *select, from* and *where* clauses of the query. Notice that while the "from stream.." clauses are retrieve by the *getSGP()* function in CQELS for each leaf node, in CSPARQL these from clauses are retrieved all at once by the *getFromClause()* because of the difference in syntax. The main difference in CSPARQL query transformation is that the recursively defined *getWhereClause* function supports sequence and repetition transformation (see Line 21-29). Indeed, many methods, e.g., *getSelectClause()*, *getSubPatterns()*, *getBoundFilters()* and *getFilters()* can be reused from Algorithm 7, this also demonstrates that developing a dedicated query transformation for yet another stream reasoning engine with a similar language syntax (i.e., extended from SPARQL) will not take much additional effort. An example of the transformation result is shown in Listing 8.3. Notice that in this CSPARQL query a sequence is specified for the user location update event and traffic report event, in order to demonstrate how sequence operator is implemented in CSPARQL using the timestamp filtering function f_t.

Algorithm 8 Transform event patterns into CSPARQL queries.

```
Require: Composition Plan: $comp$, Query Prefix String $prefix$
Ensure: CSPARQL Query String: $queryStr$
procedure TRANSFORM($comp, prefixStr$)
    $sClause \leftarrow$ GETSELECTCLAUSE($comp.ep$)
    $fClause \leftarrow$ GETFROMCLAUSE($comp.ep$)
    $wClause \leftarrow$ GETWHERECLAUSE($comp.ep$)
    $queryStr \leftarrow prefix + sClause + fClause + "WHERE" + wClause$
return $queryStr$
end procedure
Require: Event Pattern: $ep$
Ensure: Where Clause String: $wClause$
procedure GETWHERECLAUSE($ep$)
    $root \leftarrow$ GETROOTNODE($ep$), $wClause \leftarrow \emptyset$
    if $root \in$ EventDeclaration then
        $wClause \leftarrow$ GETBGP($ep, root$)
    else if $root \in Op_{and}$ then
        for $subPattern \leftarrow$ GETSUBPATTERNS($ep, root$) do
            $wClause \leftarrow wClause +$ GETWHERECLAUSE($subPattern$)
        end for
    else if $root \in Op_{or}$ then
        for $subPattern \leftarrow$ GETSUBPATTERNS($ep, root$) do
            $wClause \leftarrow wClause +$ "optional" + GETWHERECLAUSE($subPattern$)
        end for
        $wClause \leftarrow wClause +$ GETBOUNDFILTERS($ep$)
    else if $root \in Op_{seq}$ then
        for $subPattern \leftarrow$ GETSUBPATTERNS($ep, root$) do
            $wClause \leftarrow wClause +$ GETWHERECLAUSE($subPattern$)
        end for
        $wClause \leftarrow wClause +$ GETTIMESTAMPFILTERS($subPattern$)
    else if $root \in Op_{rep}$ then
        $subPattern \leftarrow$ EXPANDREPETITION($comp.ep, root$)
        $wClause \leftarrow wClause +$ GETWHERECLAUSE($subPattern$)
    end if
    if $filters \leftarrow$ GETFILTERS($ep$) $\neq \emptyset$ then
        $wClause \leftarrow wClause +$ GETFILTERS($filters$)
    end if
return "{" $+ wClause +$ "}"
end procedure
```

8.1.3 Event (Re-)Construction from Stream Query Results

The query solutions derived from evaluating the queries in Listing 8.2 and 8.3 are sets of variable bindings. To facilitate event stream composition on different abstract levels, i.e., allow the query results to be reused by other complex event requests, these results must be reconstructed into complex events. While the schema/ontology used to reconstruct

```
Select ?locId ?es4 ?value1 ...  Where {
    Graph <http://sampleStaticKB> {...}
    From Stream <locationStreamURL> [range 5s]
    From Stream <trafficStreamURL1> [range 5s]
    From Stream <trafficStreamURL2> [range 5s]
    ...
    {
    ?locId rdf:type ?ob. ?locId ssn:observedBy ?es4.
     ?locId ssn:observationResult ?result1.
    ?result1 ssn:hasValue ?value1.
    ?value1 ct:hasLongtitude ?lon. ?value1 ct:hasLatitude ?lat.
    }
    {
    ?seg1Id rdf:type ?ob. ?seg1Id ssn:observedBy ?es1.
     ?seg1Id ssn:observationResult ?result2.
    ?result2  ssn:hasValue ?value2.
     ?value2 ssn:hasQuantityValue ?eta1.
    } ...
 Filter(f:timestamp(?loiId)  < f:timestamp(?seg1Id))...}
```

Listing 8.3: CSPARQL query example

the complex events may vary depending on the applications, one can always reconstruct all the primitive events and forward them to the upper-level queries to ensure there is no information loss. However, this query-and-forward approach will demand more network traffic in the event service network.

8.2 QoS-aware Event Service Adaptation

Existing approaches to data stream processing address different issues related to streams and data processing such as stream data management, query processing, and data mining. However, it is still an open challenge to properly address issues that are more closely related to the quality of a federated stream, more precisely integration of the federated data/event streams while keeping their quality metrics in mind.

The quality metrics associated with the data streams can be defined as part of the users' or applications' non-functional requirements. These requirements may contain a set of constraints and preferences, as discussed in Section 7.1. Maintaining the quality of event stream is especially important and challenging in IoT environments [223]. To facilitate adaptability in quality-aware federation of IoT streams for smart city applications, the following requirements are considered in this thesis:

- **Monitor Quality Updates:** to monitor any updates in the quality metrics of the IoT streams involved in stream federation,

- **Evaluate Criticality:** to determine whether any particular quality update is critical and if there is any adaptation action that should be carried out, and
- **Perform Adaptation:** once a critical update is confirmed, perform necessary actions for automatic adaptation according to the new conditions/environment.

The adaptation manager can monitor the quality updates using pulling or pushing. In the current implementation pushing is used. The criticality evaluation (i.e., QoS aggregation and estimation) methods described in Section 7.1 are adopted for the adaptation manager. In the following the strategies for creating new service compositions at runtime are introduced. An abstract example is used for elaborating the adaptation. Then, the technical details of the adaptation manager and the adaptation process are presented.

8.2.1 Adaptation Strategies

Three different adaption strategies can be used for creating new composition plans: *local*, *global*, and *incremental* adaptations. All 3 approaches rely on querying the ERH to find valid candidate services. In the following these three strategies are elaborated on using an example ERH shown in Figure 8.2.

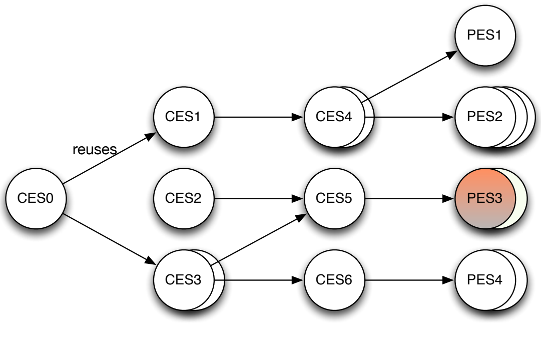


Figure 8.2: Example of an ERH

Recall that in Section 6.2.3.2, an ERH is described as a hierarchy that captures reusability between event services. In Figure 8.2 a (partial) example of an ERH (denoted $erh = (V, R)$) is presented, which contains 11 functionally different CES and SES nodes. Arrows in Figure 8.2 represent the R_r relations, and R_e relations are abstracted as stacked nodes, representing functional identical services. Suppose an instance of the ACEIS adaptation manager is currently monitoring the service CES_0, and its current composition plan contains a set of member event services $mes(CES_0) = \{CES_4, PES_3, CES_6\}$, the adaptation manager for CES_0 will subscribe to the QoS updates for all event services in $mes(CES_0)$. When a QoS update, say, for PES_3 is detected, it will recalculate the aggregated QoS metrics for CES_0 to see if it still complies with the user-defined constraint C. If the constraint C still holds for CES_0, it will do nothing except to update the QoS for SES_3 in the composition plan of CES_0 and publish a QoS update for CES_0 to all other interested adaptation manager instances. Otherwise it marks PES_3 as a *critical node* (marked in red), triggers an adaptation process, and tries to create a new composition plan for CES_0 that satisfies C using one of the following three strategies:

- **Local** adaptation that finds all functional equivalent services to PES_3, ranks them based on constraint C and preference P, and then substitutes PES_3 with the highest ranking replacement PES_3' in the current composition plan of CES_0;
- **Global** adaptation that recomposes a new composition plan entirely for CES_0 based on C, P and
- **Incremental** adaptation that follows the steps:
    1. try *local* adaptation, if failed i.e., no substitutes available or the substitution of PES_3 cannot satisfy C then,
    2. try to recompose the critical node, if failed, i.e., no composition possible, e.g., the *critical node* here is an PES, or replacing the *critical node* with the new composition cannot satisfy C then,
    3. find all Intermediate Nodes (denoted IN) between CES_0 to PES_3 in *erh* such that
    $$IN \subseteq V \mid \forall v \in IN \implies (v, PES_3) \in R_r^*$$
    where R_r^* is the transitive closure of $R_r \subseteq R$,
    4. starting from the node with shortest to the largest distance to PES_3 in IN, i.e., in the sequence of CES_5, CES_3 and CES_0, mark the node as the *critical node* and repeat step 1 and 2 on this node until a satisfying new composition plan is created and
    5. if all above steps failed to create a valid composition plan, exit with an adaptation failure notification to the users or the application, triggering the necessary recovery mechanisms.

Algorithm 9 provides the pseudo code for the adaptation algorithm, with a focus on incremental adaptation. Intuitively, each of the above three strategies has its own merits and drawbacks. *Local* adaptation causes the smallest changes to the composition plan and requires the least computational effort, however it has a relatively low chance of a successful adaptation, and even if it succeeds, the resulting new event service composition

Algorithm 9 CES adaptation algorithm

```
Require: Composition Plan: $comp$, QoS update: $qUpdate$, Constraint: $C$, Adaptation Mode: $mode$, Event Reusability Hierarchy: $erh$
Ensure: Adapted Composition Plan: $resultPlan$
procedure ADAPT($comp, qUpdate, C, mode, erh$)
    $needAdpt \leftarrow$ CHECKCONST($comp, qUpdate, C$)
    $resultPlan \leftarrow \emptyset$
    if $needAdpt = true$ then
        if $mode = local$ then
            $resultPlan \leftarrow$ LOCALADPT($comp, qUpdate, C$)
        else if $mode = global$ then
            $resultPlan \leftarrow$ GLOBALADPT($comp, qUpdate, C$)
        else if $mode = incremental$ then
            $resultPlan \leftarrow$
            INCREMENTALADPT($comp, qUpdate, C, erh$)
        end if
    end if
return $resultPlan$
end procedure
Require: Composition Plan: $comp$, QoS update: $qUpdate$, Constraint: $C$, Event Reusability Hierarchy: $erh$
Ensure: Adapted Composition Plan: $resultPlan$
procedure INCREMENTALADPT($comp, qUpdate, C, erh$)
    $resultPlan \leftarrow \emptyset$
    $IN \leftarrow$ GETIN($comp, erh$) $\cup$ $comp$
    for $criticalService \in IN$ do
        $resultPlan \leftarrow$ LOCALADPT($comp, qUpdate, C$)
        if $resultPlan = \emptyset$ then
            $subP \leftarrow$ GETSUBPATTERN($comp, criticalService$)
            $subResult \leftarrow$ GLOBALADPT($subP, qUpdate, C$)
            $resultPlan \leftarrow$ MERGERESULT($comp, subResult$)
            if CHECKCONST($resultPlan, qUpdate, C$) then
                $result \leftarrow \emptyset$
            end if
        end if
        if $resultPlan \neq \emptyset$ then
            $break$
        end if
    end for
return $resultPlan$
end procedure
```

may have a low overall QoS, since the substitute options are limited. *Global* adaptation ensures a high probability of success while it gets the best possible (with regard to the composition algorithm used) resulting QoS. However, it may change dramatically the structure of the original service composition and requires the same time of composing the original service in the service planning phase. As such, the time needed for global adaptation may be unacceptable for an adjustment during service execution phase, in which timeliness is crucial. *Incremental* adaptation is a more "balanced" choice between *local* and *global* adaptation [226], i.e., it changes the scope of adaptation when necessary, thus on average, it takes the intermediate time to adapt and creates the intermediate resulting QoS. Notice that in the case when an *incremental* adaptation is regressed into a *global* one, they produce the same quality results, but the *incremental* approach may take even more time than the *global*, due to the overhead of failed attempts. In Section 7.3 the above intuitions are verified with experiments.

8.2.2 Adaptation for Service Failures

It is worth mentioning that by adopting the QoS aggregation methods described in Section 7.1, ACEIS can handle adaptations for constraints over eight QoS metrics, including latency, accuracy, availability, completeness, security and energy/monetary/bandwidth consumption. However, service failures, such as server offline, or connection broken, are not directly supported. Nevertheless, service failures can be easily adopted as critical QoS updates and trigger the adaptation, as long as those service failures provide explicit notifications to the service consumers. In cases where no explicit notifications are provided, prediction methods based on statistics or patterns can be used to detect service failures and trigger adaptations [227]. These methods are not detailed in this chapter.

8.2.3 Adaptation Process

Figure 8.3 illustrates the structure of the adaptation manager and its interactions with the other ACEIS components. The adaptation manager consists of three components, namely: the *QoS Stream Discovery* component, the *QoS Monitor* component and the *Adaptation Handler*.

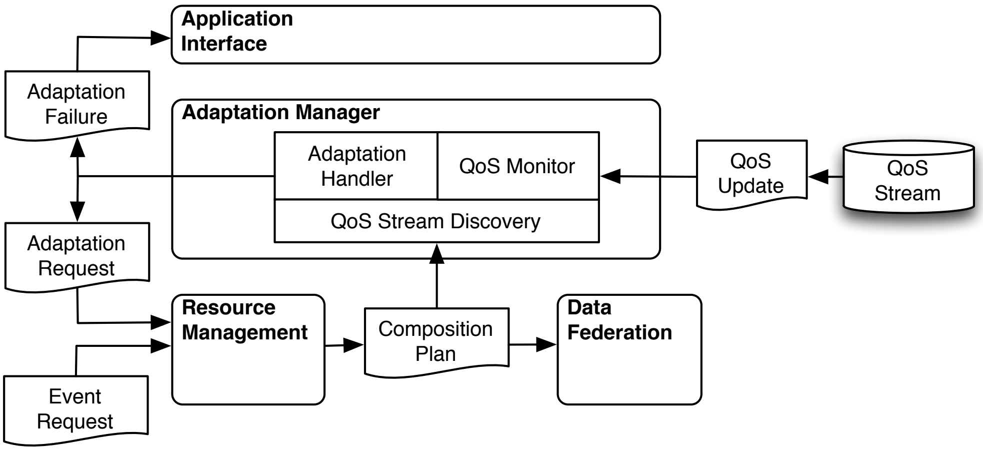


FIGURE 8.3: Structure and workflow of the Adaptation Manager in ACEIS

- **QoS Stream Discovery:** this module receives a composition plan with a reference to the event request for which the composition plan was generated. The event request contains user/application functional requirements as well as quality constraints and preferences. The discovery module finds the relevant quality update streams to subscribe for the event streams used in the composition plan. Notice that this discovery module is different to the resource discovery in the *Resource Management* component.

- **QoS Monitor:** this module generates a subscription request for any update in the quality scores of the relevant data streams. The monitoring module continuously monitors and verifies whether quality scores of all the contributing data streams in a composition plan are compliant to the event request.
- **Adaptation Handler:** this module is triggered if any of the user-defined non-functional constraints and preferences is violated, it utilises different adaptation strategies and tries to determine the scope of the adaptation. If an adaptation is possible, it invokes the *Resource Management* component to find replacements for parts of (or the entire) original composition plan. Then it creates the new composition plan via merging or replacing the original composition plan and sends it back to the *Data Federation* component for query re-deployment and other subsequent actions.

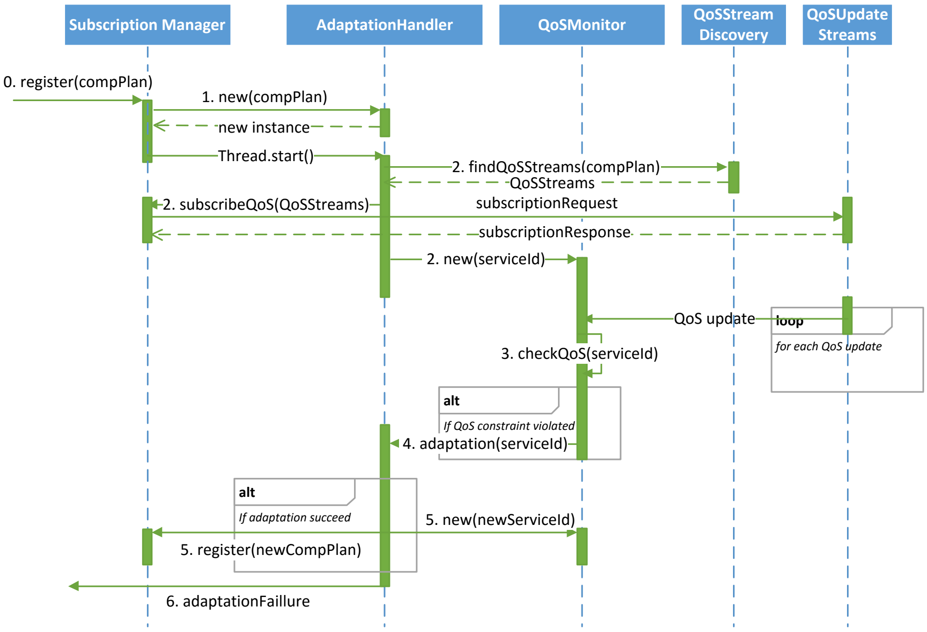


FIGURE 8.4: Sequence diagram of adaptation loop

Figure 8.4 depicts the system sequence diagram to showcase the interactions of the adaptation modules with each other as well as with the external components. A detailed step-by-step description is as follows:

1. When an event service composition plan `compPlan` for a user's request is registered by the *Subscription Manager*, an instance of the *Adaptation Manager* is instantiated as a new thread, and its `start()` function is called.

2. The newly instantiated *Adaptation Manager* receives the `compPlan` with a reference to the event request (for which the composition plan was generated) as a parameter. The *QoS Stream Discovery* module discovers relevant quality update streams for the subscription. Once discovered it invokes the `subscribeQoS()` function in the *Subscription Manager* to subscribe to those quality update streams. Meanwhile, *QoS Monitors* are instantiated to capture the quality updates.

3. Whenever there is an update in the stream quality, the listener receives the update from the *QoS Update Streams* and invokes the `checkQoS()` function in the *QoS Monitor* to check if the overall quality performance still conforms with the user/application constraints defined.

4. Whenever the quality constraints are violated. The *Adaptation Handler* invokes the `adaptation()` process with a parameter `serviceID` indicating which service performance is violating the quality constraints. The `adaptation()` process tries to create a new composition plan by invoking the event service discovery and composition components in the *Resource Management* module.

5. If the adaptation succeeds, the new composition plan is sent to the *Subscription Manager* and registers the new plan. The *Subscription Manager* then invokes the *Query Transformer* to create and deploy the new event query as well as renewing the subscriptions to the relevant streams. Meanwhile, the *Adaptation Manager* updates its quality monitors to listen to updates for new streams.

6. If the adaptation process fails, the *Adaptation Handler* sends a notification to the user/application. The notification message contains which event request is affected, and caused by which member event service.

8.3 Experiment Evaluation

In this section, the performance of the adaptation strategies is tested in a traffic monitoring scenario in the smart city context. In the following the scenario and dataset are described. The results of the experiments as well as the analysis of the results are presented.

8.3.1 Scenario and Datasets

The set of 449 traffic sensors in the City of Aarhus is used as the sensor repository in the experiments. The live traffic data, along with other sensor data on air pollution, weather etc. have been made publicly available via the Open Data Aarhus (ODAA) platform, as introduced in Section 9.2.1. The sensor data streams are wrapped as SESs that publish sensor data as sensor observation events. These events are fed to the ACEIS engines and consumed.

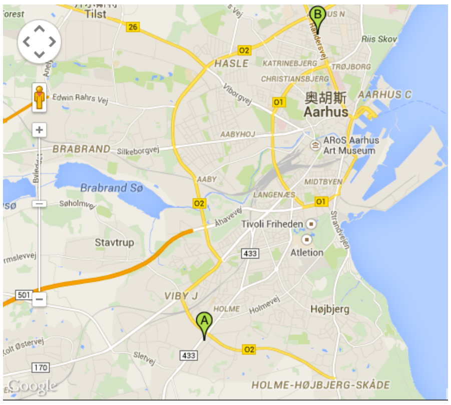


FIGURE 8.5: Traffic monitoring query on the map

To experiment on the adaptation capability, the QoS measurements are collected for the sensors used during August 2014 and play it back with the sensor observations to simulate the real-time quality updates[2]. Figure 8.6 and 8.7 show the QoS analysis for an event request over the selected month. In the experiments, a query of traffic congestion events over a specific route in the city is monitored (similar to the query in Figure 7.6). Figure 8.5 shows the start and end locations of the queried route, which consists of 10 street segments (10 traffic sensor services deployed on the route from point A to B in Figure 8.5). The accuracy of a sensor is calculated based on comparing each observation value to the real value and dividing the number of accumulated correct results with the total observation count periodically. All the accuracy reports of the 10 sensors during the selected month are recorded and the distribution of the accuracy values are shown in Figure 8.6. Figure 8.7 shows the trend of the aggregated accuracy for the query during the month, i.e., for each day of the month, the maximum, minimum and average of the aggregated accuracy of the query (multiplication of the accuracy of 10 sensors) are recorded. From the accuracy distribution and aggregated accuracy for each day it is observable that although for about 90% of the time the sensor observations are correct (100% accuracy), there are still some low accuracy results when investigating large queries using observations from many sensors.

[2] The CityPulse team at the University of Applied Sciences Osnabrück (UASO) is acknowledged for the QoS data assessment and collection.

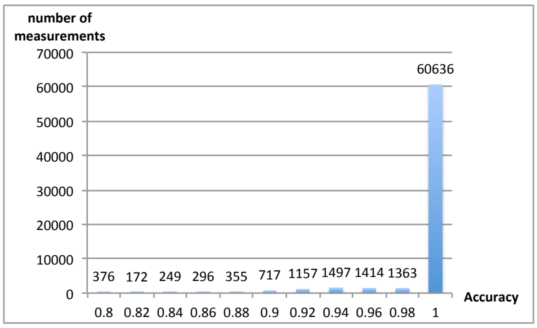


FIGURE 8.6: Accuracy distribution over a month

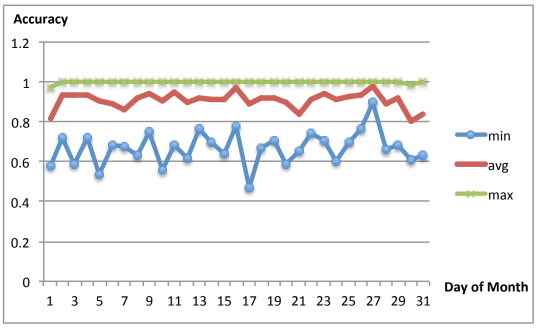


FIGURE 8.7: Accuracy trend over a month

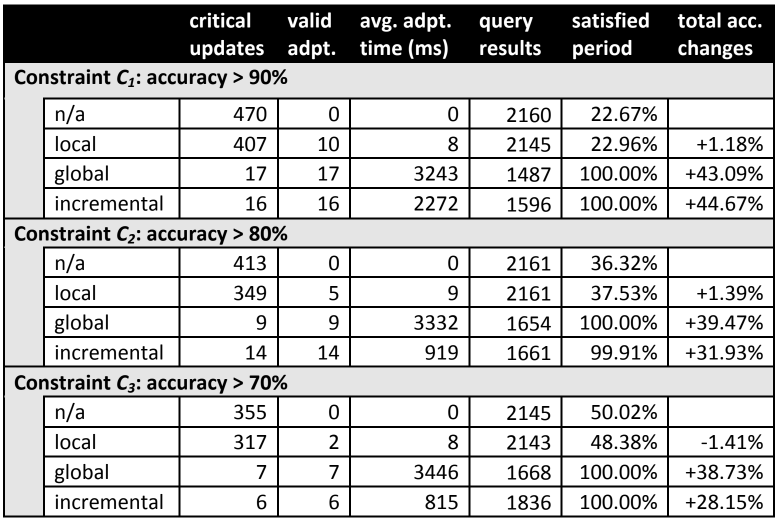
8.3.2 Performances of Adaptation Manager

To investigate the adaptation performance in more detail, the QoS updates are replayed in a random day of the month (e.g., 21st of August, 2014). The reactions of the adaptation manager to the quality updates are observed. In addition to the real-world datasets, for experimental purposes, synthesised datasets are created: for each sensor deployed in the city, 10 functionally equivalent virtual sensors are added, so that *local* adaptation is possible. QoS updates for these virtual sensors are also simulated: for each real sensor QoS update stream, ten different (and random) offsets are applied over the timestamps (e.g., +1 hour) of the updates to create 10 virtual sensor quality updates streams. Also, 100 CESs are deployed in the ERH, each CES is a random combination of the street segments used in the query in Figure 8.5. These CESs represents traffic monitoring queries over smaller regions and their results can be reused by the investigated query, so that the *incremental* adaptation is possible.

8.3.2.1 Comparison of Different Strategies

Table 8.2 shows an overall comparison of different adaptation strategies. The first column lists the adaptation strategies used ("n/a" stands for no adaptation) under 3 QoS constraints: a (relatively) strict constraint (C_1) requires the accuracy of query results above 90%, a medium constraint (C_2) above 80% and a loose constraint (C_3) above 70%. The second column lists the QoS updates that are considered critical, i.e., the updates causing constraint violations. It shows that while *local* adaptation can reduce some critical quality updates by switching data streams, *global* and *incremental* adaptation can reduce the amount to the minimum. The third column lists the number of successful adaptations. The results indicate that *local* adaptation has a much lower success rate than *global* and *incremental*. The fourth column lists the time required for the adaptations. From the results it is clear that *local* adaptation is very efficient while *global* may take more than 3 seconds to complete, and *incremental* adaptation takes less time than the *global* option and more than the *local* adaptation. The fifth column lists the number of query results (i.e., congestion events) obtained from the event stream engine. If the "n/a" option is used as the baseline (i.e., assuming the event engine does not create false positive/negatives), it is observable that the *global* and *incremental* adaptation suffers from high message loss ($\approx 30\%$ loss in the worst case) and *local* adaptation does not lose many event messages (less than 0.7%). More analysis on this is provided in Section 8.3.2.2. The sixth column shows the portion of time during the day that the constraints are satisfied using different adaptation strategies. The results show that the *global* and *incremental* adaptation can always keep the constraints satisfied, while the *local* adaptation provides slightly improved satisfactory time for constraint C_1 and C_2 and has worsened the situation for constraint C_3[3]. This effect is also reflected in the seventh column, where the summed accuracies of different strategies over the day are compared to the non-adapted approach.

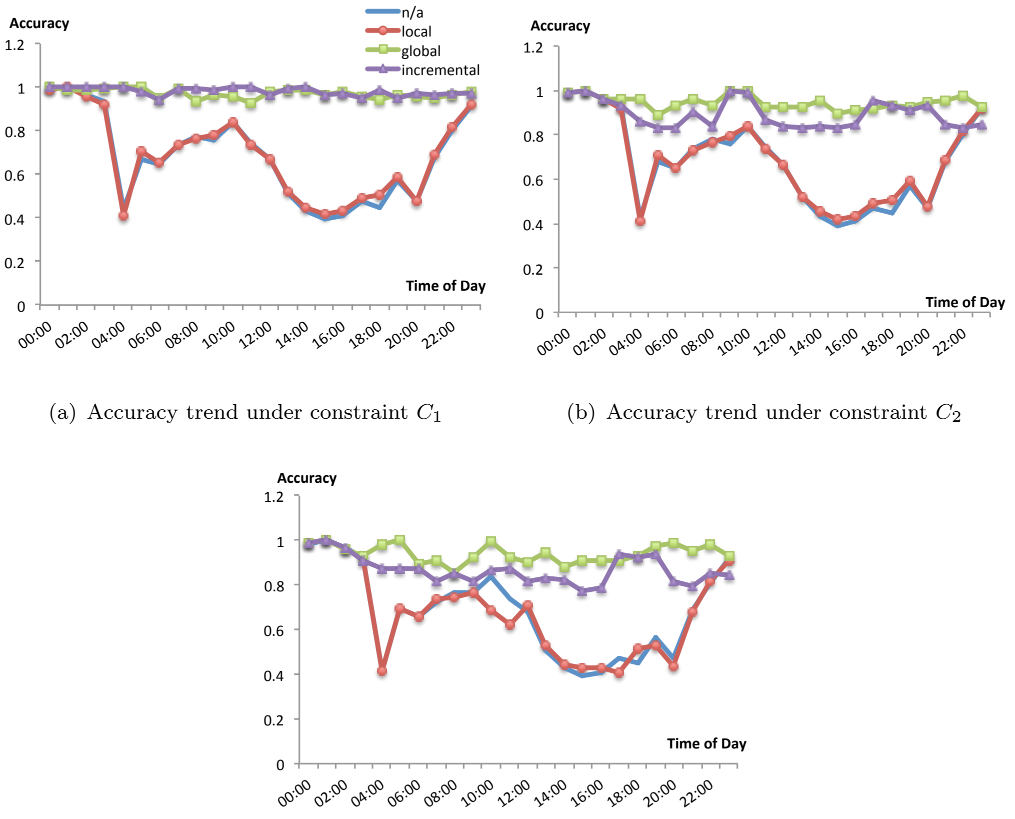
TABLE 8.2: Comparison of adaptation strategies



		critical updates	valid adpt.	avg. adpt. time (ms)	query results	satisfied period	total acc. changes
Constraint *C*$_1$: accuracy > 90%							
	n/a	470	0	0	2160	22.67%	
	local	407	10	8	2145	22.96%	+1.18%
	global	17	17	3243	1487	100.00%	+43.09%
	incremental	16	16	2272	1596	100.00%	+44.67%
Constraint *C*$_2$: accuracy > 80%							
	n/a	413	0	0	2161	36.32%	
	local	349	5	9	2161	37.53%	+1.39%
	global	9	9	3332	1654	100.00%	+39.47%
	incremental	14	14	919	1661	99.91%	+31.93%
Constraint *C*$_3$: accuracy > 70%							
	n/a	355	0	0	2145	50.02%	
	local	317	2	8	2143	48.38%	-1.41%
	global	7	7	3446	1668	100.00%	+38.73%
	incremental	6	6	815	1836	100.00%	+28.15%

In summary, the results in Table 8.2 show that the *local* adaptation is more efficient and has less message loss than the *global* and *incremental* adaptation due to the limited search space available. However, for the same reason it has a much lower success rate and quality improvement. The *local* adaptation success rate is likely to improve if there are more functional equivalent event services to adapt to. Users can choose the most suitable adaptation strategy according to their needs. To have a more intuitive representation of the accuracy trends over the day using different strategies, the hourly averaged accuracy of the query is plotted in Figure 8.8.

[3] Local adaptation may perform even worse than the original results without adaptation because, if the local replacement soon decreases its performance and there is no adaptations at the time that can restore the quality to above the threshold, the algorithm does nothing.



(a) Accuracy trend under constraint C_1

(b) Accuracy trend under constraint C_2

(c) Accuracy trend under constraint C_3

Figure 8.8: Accuracy trends under different constraints over a day using different strategies

8.3.2.2 Message Loss and Adaptation Time

From Table 8.2, it is evident that the message loss is positively related to the average adaptation time. Indeed, if more time is required to make the adjustments, there is a higher chance that a query result is lost. The frequency of query result update depends on the frequency of the input streams. In the experiments above the traffic conditions are reported every 3 seconds. To see the impact of stream frequency over the message loss rate, *global* and *incremental* adaptation are used under constraint C_2 using different streaming intervals. The results are shown in Figure 8.9. From the results, it is clear that slowing down the streaming rate can reduce the message loss for both strategies, but it cannot eliminate them. In fact, even when a streaming interval of 9 seconds is used, the message loss is still high: $\approx 15\%$ for both strategies. By further analysing the data, two more reasons are found for the message loss: 1) when a new event query is registered as a result of adaptation, a new event window is used and the previous events are discarded, thus, some query results may be lost, and 2) the semantic stream engine (e.g., CSPARQL) takes additional time (e.g., several seconds) to get the query results after the query is registered. To deal with these two causes one can deploy the

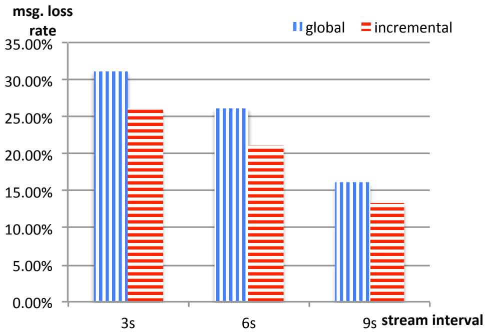


FIGURE 8.9: Message loss rate under constraint C_2 using different stream rates

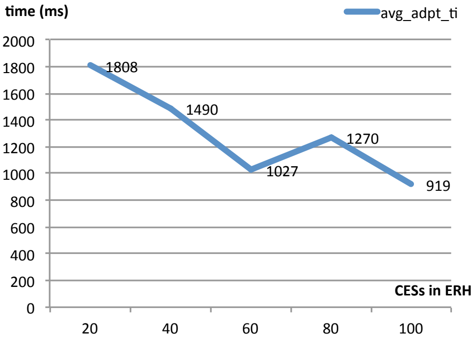


FIGURE 8.10: Avg. time used by incremental adaptation over different Event Reusability Hierarchies

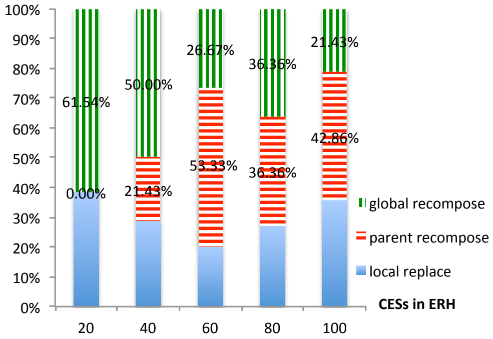


FIGURE 8.11: Distribution of incremental adaptation over different Event Reusability Hierarchies

new query along with the old one and keep the old query results until the new query is fully functional, i.e., the old time window has perished completely and the new query has started giving results. However, it causes an overhead and the system may receive low-quality query results.

The adaptation time is an important metric for evaluating the adaptation strategies. For *local* adaptation the time required simply depends on the number of R_e relations in the ERH. The *global* adaptation time depends on the efficiency of the event service composition algorithm. For *incremental* adaptation, the time required to create new composition plans largely depends on the structure and size of the ERH used. A successful *incremental* adaptation could be completed during 3 different phases in the adaptation procedure (recall the incremental adaptation steps in Section 8.2.1): *local replacement* (i.e., from steps 1 and 2), *parent replacement and recompose* (i.e., from steps 3 and 4 excluding global recomposition) and *global recomposition.* The *local replacement* and *global recomposition* take the least and most time to complete, respectively. Therefore, an ERH is ideal for *incremental* adaptation if the distribution of *global recomposition* can be minimised, i.e., it contains sufficient R_e relations to enable *local* *replacement* as well as sufficient R_r relations between the query and the critical node to enable *parent replacement*. The adaptation time and the distribution of successful *incremental* adaptations over ERHs with different sizes are tested under constraint C_2. The results are shown in Figure 8.10 and Figure 8.11. From the results it is observable that in general, the adaptation time is negatively related to the size of the ERH and is positively related to the percentage of *global recomposition*s occurred.

8.4 Related Work

Using query transformation techniques to create stream queries automatically is not novel. However, most of such query transformation techniques are platform-specific and can only be used for a specific kind of query engine, such as in [23]. To the best of my knowledge, the query transformation techniques in this chapter are the first attempt to align the semantics of different RSP engines and provide a platform-independent way of creating RSP queries with the same semantics over different types of RSP engines. The W3C RSP Working Group[4] is making an attempt to consolidate the query semantics and syntax of different RSP engines and create a unified RSP query language as a standard. However, the discussion is still on-going and no concrete outputs have been released yet. In the following, the related works on QoS adaptation in Event Broker Network (EBN) and Complex Event Processing (CEP) as well as service adaptation techniques are discussed.

8.4.1 QoS Adaptation in EBN and CEP

EBN has gained much research interests in the last decade because of their scalability, i.e., they provide sophisticated routeing techniques for Event Processing Networks (EPNs) [1] to enable efficient event transmission in wide-area networks(e.g., SIENA [44]). QoS support in EBN has been discussed extensively and is an active topic in current research. Mahambre et al. [53] propose an adaptive routeing algorithm to choose routes with the best-estimated reliability. Tariq et al. [228] leverage the knowledge of event traffic, user subscriptions and network topology to minimise the communication cost. Fischer et al. [229] use an automated workflow to create QoS-aware configurations at design time using event, domain and network profiles. In the works done by Koldehofe et al. [58] and Tariq et al. [59], the authors discuss how Software Defined Network (SDN) concepts can be used to improve the performance of EBNs. However, the above EBN based approaches are platform-dependant. Also, they evaluate the QoS of the network

[4] RDF Stream Processing Community: https://www.w3.org/community/rsp/, last accessed: July, 2015.

(and for a particular broker node) based on link analysis, which is not applicable in ESN, in which physical links are hidden.

Addressing non-functional aspects in CEP is an active area of research. Hasan et al. [230, 231] propose an approximate event processing to deal with inexact event type matchings. In the work done by Wasserkrug et al. [232] Bayesian Networks are used to calculate probabilities of detected events. These approaches deal with uncertainty in the stream data but they do not fully address the stream quality issues. Many CEP systems use query rewriting to determine the distributed operator execution plans to optimise the query processing time (e.g., Schultz-Molle et al. [97] and Rabinovich et al. [233]) but the rewriting does not consider stream source replacement. Wagle et al. [234] introduce a fault tolerance mechanism for System S [235], which deals with errors caused by stream connection lost using transactional database operations. Buys et al. [236] use replica deployment and selection to improve reliability for publish and subscribe services (i.e., WS-Eventing, WS-Notification etc). While the basic idea in [236] is similar to the adaptation manager, ACEIS provides QoS adaptability not only for conventional WS-* services but also for CESs. Table 8.3 summarises the comparison of existing QoS adaptation in EBN and CEP to the adaptation technique discussed in this chapter.

Approaches	**Differences to QoS-aware service adaptation in ACEIS**
Mahambre et al. [53]	Adaptive routing for optimising reliability.
Tariq et al. [228]	Using dynamic information on event traffic, subscriptions and network topology to minimise communication cost.
Fischer et al. [229]	Use automatic work flow to update and optimise configuration of EBNs at design-time.
Koldehofe et al. [58] and Tariq et al. [59]	Propose Software Defined Network to address end-to-end QoS for EBNs.
Above EBN-based approaches	Platform-dependant, evaluate QoS based on link analysis, not applicable in ESN.
Hasan et al. [230, 231]	Facilitate inexact event matching, do not address QoS.
Query rewriting in CEP systems, e.g., Schultz-Molle et al. [97] and Rabinovich et al. [233]	Dynamically adapt execution plans and changes the sequence of operators evaluated, do not discuss data source replacements.
Wagle et al. [234]	Addresses errors for System S using transactional database operators.
Buys et al. [236]	Use replica deployment and selection to optimise event services, but do not address complex event services.

Table 8.3: Related works in QoS-aware event broker networks and complex event processing engines

8.4.2 Adaptive Service Composition

QoS-aware adaptive service composition and self-recovery have been discussed extensively in the service computing community over the past decade. Mei et al. [237] use a multi-tier ranking system to categorise services based on link analysis over a snapshot of the service network. This approach can select popular services in the network based on dynamic bindings, however it can only recover from service failures, i.e., severe service problems. Also, only service re-discovery at an atomic level is realised. Qu et al. [99] use Back Propagation (BP) neural network to evaluate and detect service deficiencies from the service network. Service failure rate, response time, quality complaint rate and etc. are used as evidence in the BP network to decide whether a service composition becomes unreliable. If so, a global service re-composition is triggered. This approach demands a learning time for the BP network, so it is not very ideal for the IoT services where sensor services can be added or removed from the network frequently. Yu et al. [101] use reinforcement learning to optimise service compositions, however, like [99] and other learning based approaches, the learning phase is needed.

Joshi et al. [238] propose to use ontology-based solutions to manage the service life-cycle in the cloud, including service discovery, negotiation, composition and monitoring. A fuzzy-logic-based framework is used to monitor service quality. However, it does not provide sophisticated recovery mechanisms, other than re-defining new service requirements and iterate a new life-cycle. Wang et al. [239] discuss different service recovery strategies and a Dynamic Local Backup Recovery Algorithm (DLBRA) is proposed for ubiquitous services. DLBRA is quite similar to the approach in [237], only that the backups are proactively searched by the service monitor instead of taking snapshots when the recovery mechanism is triggered (as in [237]), also the quality analysis is based on a quality utility aggregated from multiple quality metrics instead of link analysis. However, proactively searching backups will introduce an overhead, also local recovery has a lower success rate because the choices are very limited. Bucchiarone et al. [226] propose a context-aware adaptive composition over IoT services. It adopts an incremental adaptation strategy, which is similar to the idea to the incremental adaptation strategy developed in ACEIS. However, it focuses on adapting contextual changes rather than quality constraint violations. Moreover, existing approaches in adaptive service computing rely on imperative workflows, which is inherently different to declarative event pattern definitions in complex event services. Therefore, they cannot be applied directly to the ESNs proposed in this chapter. Table 8.4 summarises the comparison of existing QoS-aware service adaptation techniques to the adaptation technique discussed in this chapter.

Approaches	**Differences to QoS-aware service adaptation in ACEIS**
Mei et al. [237]	Only support service failures, only re-discover service at an aotmic level.
Qu et al. [99] and Yu et al. [101]	Use learning algorithms to decide when an adaptation by global re-planning is needed, demands learning phase.
Joshi et al. [238]	Use statistics of service quality to re-define service requirements.
Wang et al. [239]	Proactively search service backups, introduces overhead.
Bucchiarone et al. [226]	Use incremental adaptation similar to ACEIS to adapt contextual changes rather than QoS constraint violations.

Table 8.4: Related works in QoS-aware service adaptation

8.5 Summary and Discussion

In this chapter, the means of transforming event patterns specified in BEMN$^+$ into RSP queries are provided. In particular, the semantics of event operators in BEMN$^+$ and the query operators in CQELS, CSPARQL and ETALIS are aligned. Based on these semantics alignments, the algorithms for generating CQELS and CSPARQL queries from BEMN$^+$ event patterns are developed and some example results are presented. Leveraging the automatic query transformation and deployment technique, it is possible to carry out automatic adaptation for event service compositions. Then, a service-oriented approach for quality-aware adaptive event stream federation is presented. The details of the event service adaptation mechanisms, including three different adaptation strategies: *local*, *global* and *incremental* adaptation, are discussed and the adaptation process in ACEIS is described. Finally, the different adaptation strategies are evaluated in a Smart City scenario with both real and synthesised datasets and analyse the evaluation results. The experiments show that local adaptation is the most efficient (takes several milliseconds to complete) and suffers least from message loss (about 0.7%), however its contribution in improving the QoS is insignificant (up to 1.4%). Global adaptation takes much more time (more than 3 seconds) to complete and causes a lot of message losses (about 30%), however, it can greatly improve the QoS (38.73% to 43.09%). Incremental adaptation uses less time than global adaptation (from 815 to 2272 milliseconds) and causes fewer message losses (from 14.4% to 26.1%). The average improvement of QoS caused by incremental adaptation is less than the global adaptation (28.15% to 44.67%). The results reveal that there is no global optimised strategy for the event service adaptation and users should choose different strategies based on their requirements as well as the characteristics of the datasets (i.e., service repositories). The experiments also show that using lower stream rate and less adaptation time can reduce the message loss rate. The adaptation module completes the life-cycle of SESs. In the next chapter, the prototypes using SES are demonstrated and the feasibility of using SESs in smart city scenarios is analysed.

Part III

Finale - Usage, Conclusion and Future Research

Chapter 9

Prototype Implementation and Query Performance Analysis[*]

The validity and feasibility of Semantic Event Service (SES) modelling, planning, implementation and adaptation in the Automatic Complex Event Implementation System (ACEIS) are discussed in previous chapters (Chapter 5 to Chapter 8). In this chapter, the practicality of ACEIS is showcased by demonstrating the usage of ACEIS in real-world applications. In particular, the user interfaces of the aforementioned smart city applications are presented and their functionalities are described, in order to demonstrate the contribution of ACEIS in smart city scenarios. Moreover, this chapter investigates the performance of ACEIS with regard to SES execution in realistic scenarios. Since ACEIS offers the SES composition as semantic event queries evaluated by RDF Stream Processing (RSP) engines, the performance of these engines (i.e., CQELS [37] and C-SPARQL [38]) has a great impact on the user experience of the smart city applications. A benchmarking system is developed to evaluate the performance of these two engines under different settings. The benchmarking results help an ACEIS user/developer to decide which type of RSP engine best suits his/her requirements. In addition, since both RSP engines show limited capabilities in processing a large number of concurrent queries in the benchmarking results, multiple instances of RSP engines are deployed in ACEIS in parallel. Load balancing techniques are applied to distribute the workload over different RSP engine instances, in order to improve the query execution performance. In summary, this chapter validates the ACEIS approach by answering the following questions:

1. How is ACEIS deployed in real-world scenarios and how does it fulfil the application requirements?

[*] Part of the content in this chapter is published in [70].

2. How to evaluate the performance of the RSP engines used in ACEIS and how RSP engines perform using the realistic datasets and queries?
3. Can existing RSP engines used in ACEIS provide efficient SES execution at large-scale? If not, how to optimise the performance while handling concurrent queries?

The remainder of this chapter is organised as follows. Section 9.1 answers the first question and describes the usage of ACEIS in prototypical applications in Smart City environments. Section 9.2 answers the second question and discusses the design of an RSP engine benchmark together with the analysis of the benchmarking results. Section 9.3 answers the third question and investigates the load balancing techniques used for optimising RSP query execution performance in ACEIS when using multiple RSP engine instances.

9.1 Usage in Smart City Application Prototypes

As part of the on-going activities in the CityPulse project, two prototypes are collaboratively developed by the teams involved in the project, namely the Smart Travel/Parking Planner (STPP) application and the Smart City Dashboard (SCD) application. Recall that in Section 2.1, three different scenarios are proposed: travel planner ($S1$), parking space finder ($S2$) and city administration console ($S3$). $S1$ facilitates the optimisation of the users' travel paths based on their preferences for route type, health and travel cost, as well as the real-time information (including traffic, weather, parking availability and so on) that can impact this optimisation. $S2$ is designed to facilitate car drivers in finding a parking spot combining parking data streams and predicted parking availability based on historical patterns. $S3$ facilitates city administrators by notifying them of the occurrence of specific events of interest. STPP covers part of the functionalities described in $S1$ and $S2$, SCD covers part of the functionalities in $S1$ and $S3$. STPP and SCD are implemented with datasets from the city of Aarhus.

Both STPP and SCD are built in a Client/Server fashion. The server side hosts a *Smart City Framework*, which is the output of the CityPulse project consisting of a set of interacting modules including *Data Virtualisation*, *Data Quality Analysis*, *Decision Support* and *Data Federation*. The *Data Virtualisation* module provides semantically annotated data graphs and data streams for the *Data Federation* module. This thesis is mainly used in the *Data Federation* and we acknowledge the CityPulse team for implementing the other components, including the User Interfaces.

The *Data Quality Analysis* module provides quality associated annotations for the data. ACEIS is deployed as the *Data Federation* module in the framework and is responsible for creating federated queries over data streams based on users' functional and non-functional requirements and deploying/maintaining the queries. The *Decision Support* module offers real-time decision making based on results from the *Data Federation* module as well as the context of the user. The architecture of the applications is shown in Figure 9.1. The client side of STPP is a mobile application on Android and the client of SCD is a web-based application. In the following, the user interfaces of the client applications are presented and the functionalities of STPP and SCD are described.

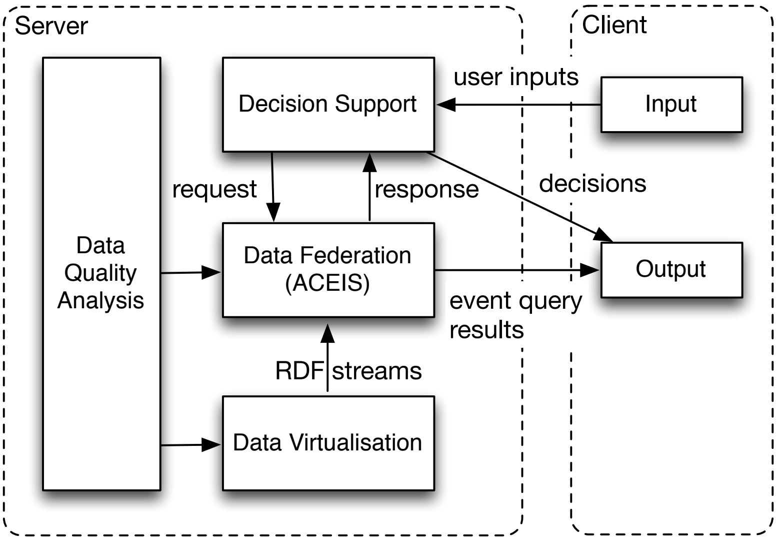


FIGURE 9.1: Architecture of the smart city application prototypes

9.1.1 Smart Travel/Parking Planner

When the user starts the client application of STPP, the city map is displayed and the user can select on the map a start and end point of his/her journey. The user can also specify whether the type of transportation such as walking, by car or by bicycle, as shown in Figure 9.2. After selecting the transport type, the user can also specify some preferences on the recommended routes, such as fastest, shortest or cleanest, as shown in Figure 9.3. Using these user inputs the DS component will first query the geographical database to create routes on the map and then filter these routes based on real-time sensor data (e.g., current traffic congestion levels, current air pollution levels) retrieved from ACEIS. In order to obtain the real-time sensor data, a one-time event request is sent to ACEIS and ACEIS will discover and subscribe to relevant sensors to get data snapshots. As a result, the recommended routes are displayed on the interface, as shown in Figure 9.4.

When the user selects a route and starts the navigation, the *Decision Support* component will deploy a set of event queries to detect events that may affect the user's

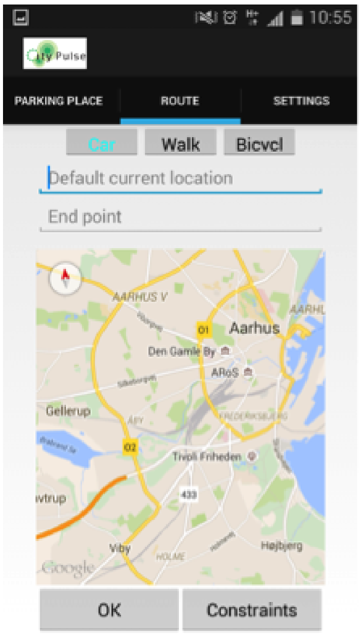
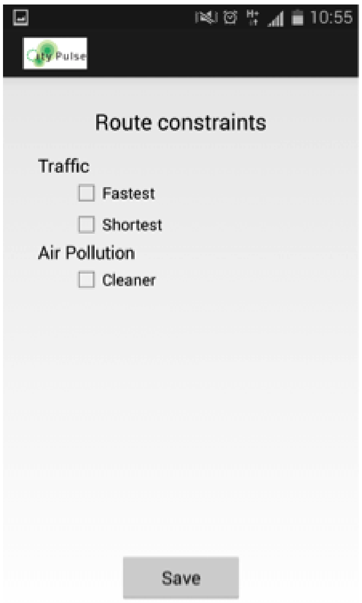


FIGURE 9.2: STPP route selection



Route constraints

Traffic

☐ Fastest

☐ Shortest

Air Pollution

☐ Cleaner

FIGURE 9.3: STPP route constraints

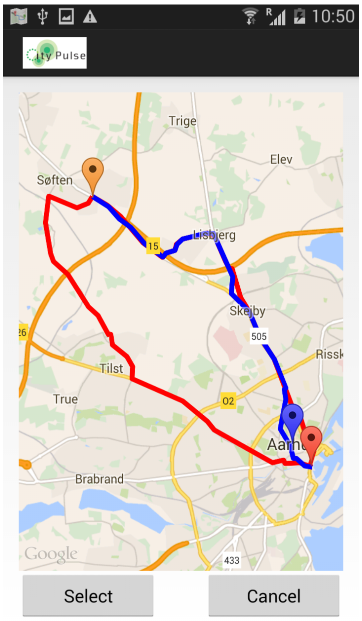


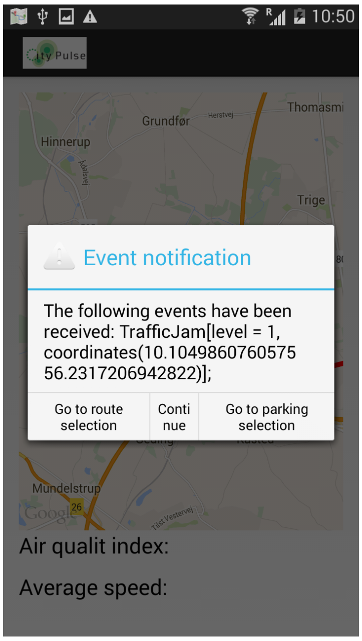
FIGURE 9.4: STPP selected routes

journey, such as traffic accidents happening on the user's current route. Also, ACEIS will deploy continuous queries on the average speed and the air pollution index on the user's current route and show the results on the client, as shown in Figure 9.5. When the user is travelling by car and is approaching the destination, he/she may switch to the "parking space" tab and specify an area to find parking spaces within. This area can be centered by the destination point or by the user's current location (shown as a blue circle on the map), as shown in Figure 9.6. The user may also choose to look for parking spaces with the cheapest toll or shortest walking distance to the destination, and the *Decision Support* will retrieve the real-time information about the parking spaces from ACEIS, including the price and parking availability. Based on the real-time parking information, recommendations for the best parking spaces can be made. An example of the recommended parking space is shown in Figure 9.7 as the red pin.

Table 9.1 summarises the functionalities required by STPP application with regard to integration and processing of data/event streams, and relevant solutions provided by ACEIS.

9.1.2 Smart City Dashboard

In the SCD application, a user is a city administrator, who is interested in monitoring various physical properties on different routes in the city, including the traffic conditions, air pollution and weather information. To use SCD, first the user needs to login to the



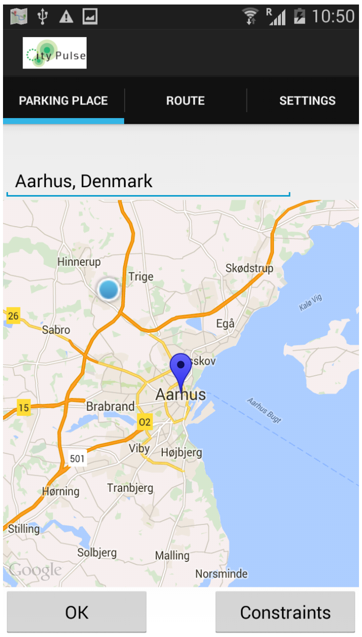
Event notification

The following events have been received: TrafficJam[level = 1, coordinates(10.1049860760575 56.2317206942822)];

Air qualit index:

Average speed:

FIGURE 9.5: STPP event notification and continuous query results



Aarhus, Denmark

FIGURE 9.6: STPP parking request

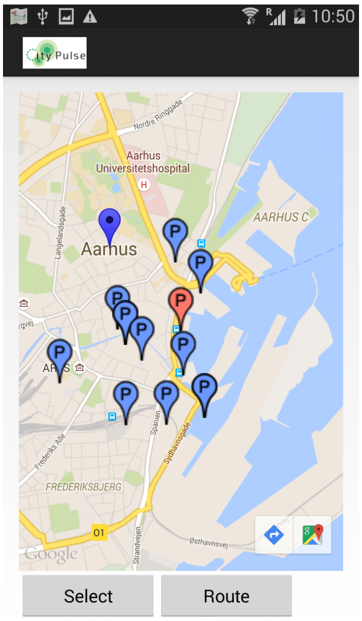
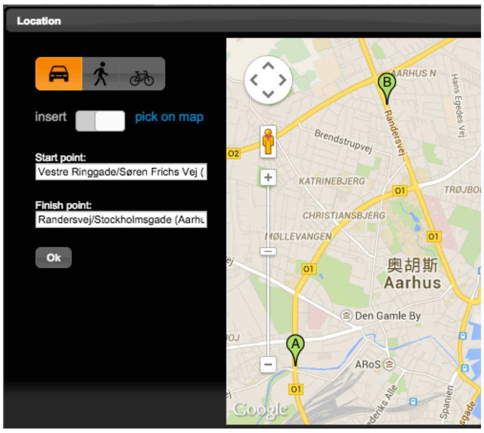


FIGURE 9.7: STPP selected parking spaces

Required Functionalities	**Solutions Provided by ACEIS**
Static description of traffic and parking data streams	Semantic event service annotations using *Complex Event Service Ontology* (Section 5.1)
Finding traffic and parking streams based on geographical information	Semantic *Resource Discovery* (Chapter 6) component
Reusing existing complex event services based on implicit functional constraints	Pattern-based event service composition provided by *Event Service Composer* (Chapter 6 and 7)component
Reusing existing complex event services based on implicit non-functional constraints	Constraint-aware event service composition provided by *Event Service Composer* component
Create and deploy user queries over RSP engines	Automatic RSP query creation algorithms implemented within the *Query Transformer* (Section 8.1) component
Query evaluation and result monitoring	Real-time RDF stream processing provided by the *Query Engine* used in ACEIS
Compliance of non-functional constraints during run-time	QoS-aware event service adaptation techniques developed in the *Adaptation Manager* (Section 8.2) component
Performance optimisation for multiple users	Load balancing techniques developed in the *Scheduler* (Section 9.3) component

TABLE 9.1: Requirements and solutions in Smart Travelling/Parking Planner with regard to data stream integration

web server that hosts the SCD web application and open the “location” tab on the page to specify the start and end locations of the route as well as the transport type, as shown in Figure 9.8. Additionally, the user can specify which set of functional properties (i.e., properties observed by sensors) is monitored on the route. The user can choose from traffic, air pollution or weather properties and apply aggregations and/or thresholds on the monitored properties as constraints, as shown in Figure 9.9. The user can also specify non-functional constraints and preferences as shown in Figure 9.10. It is worth mentioning that the top half of the interface in Figure 9.10 controls the data replay rate and duration, for historical data playback function.



Location

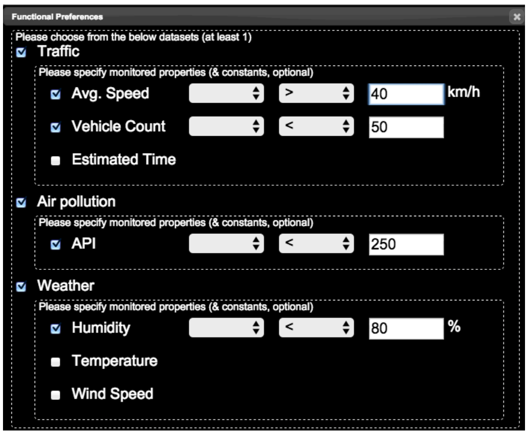
insert pick on map

Start point: Vestre Ringgade/Søren Frichs Vej (

Finish point: Randersvej/Stockholmsgade (Aarhu

Ok

FIGURE 9.8: SCD start and end locations



Functional Preferences

Please choose from the below datasets (at least 1)

☑ Traffic

Please specify monitored properties (& constants, optional)

☑ Avg. Speed > 40 km/h

☑ Vehicle Count < 50

□ Estimated Time

☑ Air pollution

Please specify monitored properties (& constants, optional)

☑ API < 250

☑ Weather

Please specify monitored properties (& constants, optional)

☑ Humidity < 80 %

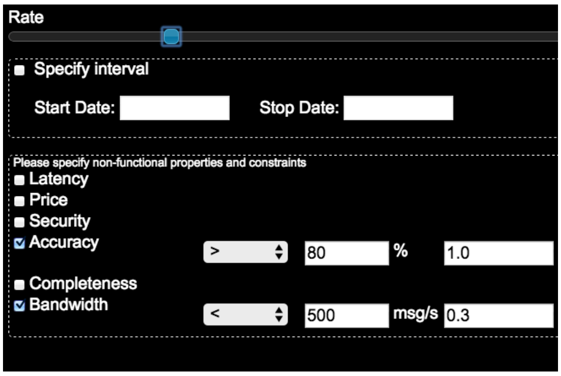
□ Temperature

□ Wind Speed

FIGURE 9.9: SCD functional properties

When the user has specified all the inputs, he/she can start the monitoring by clicking a button on the client to send a request to the server. On the server side, the *Decision Support* component invokes a routeing algorithm to generate a route. Using the route information together with the functional/non-functional requirements, an event request is generated and sent to ACEIS. As a result, an event service composition plan is created by ACEIS and a federated RSP query is deployed. The RSP query results are sent back to the client and displayed on the user interface. Meanwhile, the *Decision Support* component will detect some contextual events on the route, such as traffic accidents, or heavy air pollutions etc using pre-defined event rules, and display those events as bubbles on the 3-dimensional city map. The user can zoom in/out and navigate on the map to find out the details of the geographical information of those events.

Table 9.2 summarises the functionalities required by SCD application with regard to integration and processing of event streams, and relevant solutions provided by ACEIS. Compared to Table 9.1, it can be observed that the STPP and SCD applications share many requirements, e.g., both need semantic description and discovery for heterogeneous data streams, as well as creation, deployment and execution of semantic event queries over RSP queries. Also, both applications can reuse existing event services deployed on different servers leveraging the service-oriented nature of ACEIS and the event service


Rate

□ Specify interval

Start Date: Stop Date:

Please specify non-functional properties and constraints

□ Latency

□ Price

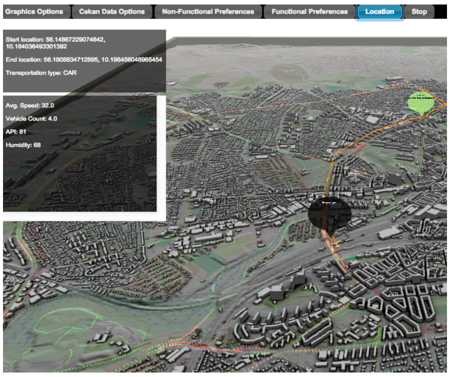
□ Security

☑ Accuracy > 80 % 1.0

□ Completeness

☑ Bandwidth < 500 msg/s 0.3

FIGURE 9.10: SCD non-functional requirements



Start location: 56.14867229074642, 10.184036493301392

End location: 56.1808834712895, 10.198456048965454

Transportation type: CAR

Avg. Speed: 32.0

Vehicle Count: 4.0

API: 81

Humidity: 68

FIGURE 9.11: SCD result panel

composition mechanisms offered within. For example, a traffic monitoring query deployed on SCD servers could be utilised by the STPP application to create a travel plan for a user. The major difference between the two applications is that the STPP server expects a large number of "casual" users, e.g., citizens using the application in their daily lives, while the SCD server expects a small group of professional users monitoring the situations happening in the city. For this reason, STPP provides implicit settings for the functional/non-functional constraints when performing event service compositions, and SCD allows explicit constraint specifications, considering the users of SCD may have more advanced requirements and constraints. Also, because of the potential high concurrency of users in STPP, the load balancing techniques are deployed by integrating a *Scheduler* component in ACEIS.

Required Functionalities	**Solutions Provided by ACEIS**
Static description of traffic, weather and pollution data streams	Semantic service annotations using *Complex Event Service Ontology*
Finding traffic, weather and pollution streams based on geographical information	Semantic *Resource Discovery* component
Reusing existing complex event services based on explicit functional constraints	Pattern-based event service composition provided by *Event Service Composer* component
Reusing existing complex event services based on explicit non-functional constraints	Constraint-aware event service composition provided by *Event Service Composer* component
Create and deploy monitoring queries over RSP engines	Automatic RSP query creation algorithms implemented within the *Query Transformer* component
Query evaluation and result monitoring	Real-time RDF stream processing provided by the *Query Engine* used in ACEIS
Compliance of non-functional constraints during run-time	QoS-aware event service adaptation techniques developed in the *Adaptation Manager* component

TABLE 9.2: Requirements and solutions in Smart City Dashboard with regard to data stream integration

9.2 Benchmarking RSP Engines with Realistic Datasets

During the prototype development, the query performance issue of the RSP engines, i.e., CQELS and CSPARQL, is noticed. More specifically, the query delay and the usage of resources (e.g., CPU time and memory) may increase significantly when using more complicated queries or when experiencing higher levels of concurrency, which may pose threats to the scalability of the system. To analyse the query performance, a benchmarking system for RSP engines using realistic datasets from cities is developed, called the CityBench. In the following the design of CityBench and some benchmarking results are discussed. The source code and reproducible experiment results are provided in a public repository[1].

9.2.1 Datasets and Queries

Various datasets can be collected from an urban environment and used in Smart City applications as event sources. Leveraging the outcomes of the CityPulse project, the datasets collected from the city of Aarhus, Denmark[2] are used. In the following, each of the datasets is briefly described.

- **Vehicle Traffic Dataset.** This dataset contains traffic data. The City administration has deployed 449 pairs of sensors over the major roads in the city. Traffic data is collected by observing the vehicle count between two points over a duration of time. Observations are currently generated every five minutes. Each pair of traffic sensors reports the average vehicle speed, vehicle count, estimated travel time and congestion level between the two points set over a segment of road. This dataset is provided via CKAN[3] server in CSV[4] format.

- **Parking Dataset.** Parking lots in Aarhus are equipped with sensors and capable of producing live data streams indicating the number of vacant places. The Parking Dataset consists of observations generated by 8 public parking lots around the city. This data is available from a RESTful web service[5] in XML format.

- **Weather Dataset.** Currently, there is only a single weather sensor available in the city to collect live sensor observations about the weather condition. Weather

[1] CityBench GitHub repository: https://github.com/CityBench/Benchmark/, last accessed: Dec., 2015.

[2] The CityPulse consortium team is acknowledged for the provision of Datasets http://iot.ee.surrey.ac.uk:8080/datasets.html (last accessed: Apr., 2015)

[3] CKAN data portal homepage: http:ckan.org, last accessed: Apr., 2015.

[4] Comma separated values, RFC 4180 standard: https://tools.ietf.org/html/rfc4180, last accessed: Apr., 2015.

[5] Aarhus live parking web service: http://pgsaar.dyndns.org:4000/consumer/webServices/getGarageData.aspx, last accessed: Apr., 2015.

sensor data provides observations related to dew point (°C), humidity (%), air pressure (mBar), temperature (°C), wind direction (°), and wind speed (kph). This dataset is available from a web interface in JSON[6] format.

- **Pollution Dataset.** Pollution is directly related to the traffic level, however due to unavailability of the pollution sensors, a synthesised pollution data for the city of Aarhus is made available to complement the traffic dataset. Observation points for traffic sensors are replicated to create mock-up sensors for pollution at the exact same location as traffic sensors. An observation for air quality index is generated every 5 minutes using a pre-selected pattern. Details regarding the procedure followed to synthesised pollution data are accessible at: `http://iot.ee.surrey.ac.uk:8080/datasets/pollution/readme.txt`. This dataset is streamed via web sockets in CSV format.
- **Cultural Event Dataset.** This dataset is quasi-static and contains cultural events provided by the municipality of Aarhus. The dataset is periodically updated to reflect the latest information related to planned cultural events. Updates can be delivered in data streams (a notification service notifies of any updates in the dataset) or it can be used as background knowledge that updates on a daily or weekly basis. This dataset can be crawled from the city event website[7].
- **Library Events Data.** This dataset contains a collection of past and future library events hosted by libraries in the city. A total collection of 1548 events is described in this dataset. Similarly to the Cultural Events Dataset, updates in the Library Events Dataset are also not frequent, therefore, the dataset is considered quasi-static. This dataset can be crawled from the city library website[8].
- **User Location Stream.** Most of the IoT-enabled Smart City applications are designed to be location-aware, therefore they strongly rely on updates on the location of mobile users. A User Location Stream is simulated to mock-up the movements of a user. This data stream contains periodic observations with geo-location coordinates of a fictional mobile user. The user location data is typically provided by an API in Software Development Kits for smartphones, e.g., Android SDK[9].

It is worth mentioning that apart from the cultural and library event datasets, all the other above datasets contain metadata describing the data/event sources. For example,

[6] Javascript Object Notation: `http://json.org/`

[7] Aarhus cultural events: `https://www.billetlugen.dk/`, last accessed: May, 2015.

[8] Aarhus library website: `https://www.aakb.dk/`, last accessed: May, 2015.

[9] Android devkit: `https://developer.android.com/sdk/index.html`, last accessed: Apr., 2015.

for traffic sensors the metadata describes the locations of the sensors by giving the GPS coordinates and street names they are deployed, for parking datasets the metadata describes the locations of the parking lots as well as their maximum parking vacancies etc. Below we list 13 different queries for the three scenarios ($Q1$ to $Q5$ for $S1$, $Q6$ to $Q9$ for $S2$ and the rest for $S3$) described in Section 2.1, that utilize the above datasets:

Q1: What is the traffic congestion level on each road of my planned journey?

This query monitors the traffic congestion from all traffic sensors located on the roads which are part of the planned journey. This query uses the traffic dataset.

Q2: What is the traffic congestion level and weather conditions on each road of my planned journey?

Q2 is similar to *Q1* with an additional type of input streams containing weather observations for each road at the planned journey of the user. This query uses the traffic and weather datasets.

Q3: What are the average congestion level and estimated travel time to my destination?

This query includes the use of aggregate functions and evaluates the average congestion level on all the roads of the planned journey to calculate the estimated travel time. This query uses the traffic dataset.

Q4: Which cultural event happening now is closest to my current location?

Q4 consumes user location data streams and integrates it with background knowledge on the list of cultural events to find out the closest cultural event happening near his current location. This query uses the traffic and cultural event datasets.

Q5: What is traffic congestion level on the road where a given cultural event X is happening? Notification for congestion level should be generated every minute starting from 10 minutes before the event X is planned to end, till 10 minutes after.

Q5 is a conditional query which should be deployed at the occurrence of an event and have predefined execution duration. This query uses the traffic and cultural event datasets.

Q6: What are the current parking conditions within the range of 1 km from my current location?

This query represents a most common query issued by users of a parking application to easily find a nearby parking place. This query uses the traffic and parking datasets.

Q7: Notify me whenever a parking place near to my destination is full.

Q7 is a combination of the travel planning and parking functionalities, where a user wants to be notified about parking situation close to the destination. This query uses the traffic and parking dataset.

Q8: Which parking places are available nearby library event X?

This query combines parking data streams with the static information about the library events to locate parking spaces nearby the library. This query uses the library event and parking datasets.

Q9: What is the parking availability status nearby the city event with the cheapest tickets price?

Similarly to *Q8*, this query monitors parking availability near a city event which has the cheapest ticket price. This query uses the cultural event and parking datasets.

Q10: Notify me every 10 minutes, about the most polluted area in the city.

Q10 is an analytical query executed over the pollution data streams to find out which area in the city in most polluted and how this information evolves. This query uses the pollution dataset.

Q11: Notify me when no observation from weather sensors have been generated in the last 10 minutes.

This query helps to detect any faulty sensors which are not generating observations or networking issues. This query uses the weather dataset.

Q12: Notify me whenever the congestion level on a given road goes beyond a predefined threshold more than 3 times within the last 20 minutes.

This query helps in early detection of areas where traffic conditions are becoming problematic. This query uses the traffic dataset.

Q13: Increase the observation monitoring rate of traffic congestion if it surpasses a pre-specified threshold.

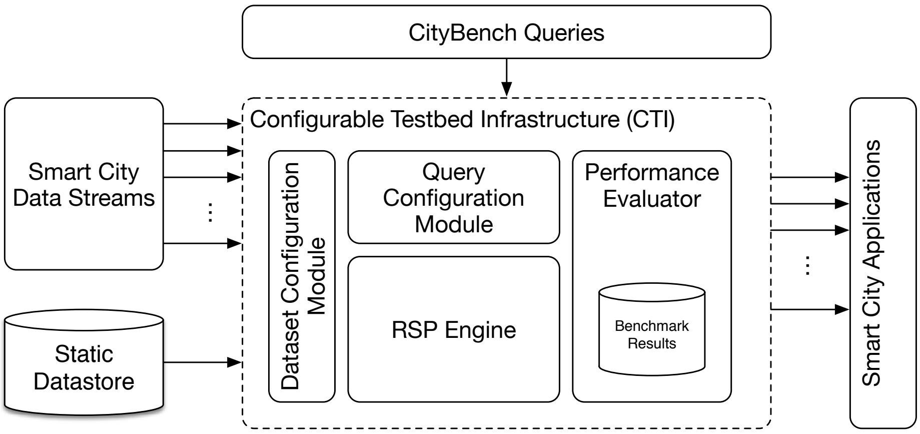


Figure 9.12: An overview of the configurable testbed infrastructure

This query provides a more frequent status update on congestion levels in critical conditions such as traffic jams or accidents. This query uses the traffic dataset.

9.2.2 CityBench Design

The performance of RSP engines does not depend only on language features but also on dynamic metrics related to the data and to the application. To evaluate the performance of RSP engines according to the dynamic requirements of smart city applications, a *Configurable Testbed Infrastructure* (CTI) containing a set of APIs is provided to set up the testbed environment. CTI allows its users to configure a variety of metrics for the evaluation of RSP engine. Figure 9.12 provides an overview of the CTI, there are three main modules, (i) *Dataset Configuration Module:* allows configuration of stream related metrics, (ii) *Query Configuration Module:* allows configuration of query related metrics, and (iii) *Performance Evaluator:* is responsible for recording and storing the measurements of the performance metrics. The CTI allows the following configurations:

- **Changes in Input Streaming Rate:** The throughput for data stream generation can be configured in CityBench. For example, a rate $r \in [1, inf]$ can be configured to set up the streaming rate to the real interval between observations ($r = 1$ means replay at original rate), or a frequency f can be used to set a different streaming rate.
- **PlayBack Time:** CityBench also allows to playback data from any given time period to replay and mock-up the exact situation during that period.
- **Variable Background Data Size:** CityBench allows to specify which dataset to use as background knowledge, in order to test the performance of RSP engines

with different static datasets. Duplicated versions (with varying size) of two static datasets, *Cultural Event Dataset* and *Library Event Dataset* (described in Section 9.2.1), are also provided. Any version of the given background datasets can be loaded to test RSP engines with different size of background data.

- **Number of Concurrent Queries:** CityBench provides the ability to specify any number of queries to be deployed for testing purposes. For example, any number of queries can be selected to be executed concurrently any number of times. Such situation will simulate a situation where a number of simultaneous users are executing the same query using any application.
- **Increase in the Number of Sensor Streams within a Single Query:** In order to test the capability of the RSP engine to deal with data distribution, CityBench makes it possible to configure the various size of streams involved within a single query. For example, for a query similar to traffic condition monitoring over a given path, to increase the number of streams involved within the query, one can simply increase the length of the observed path.
- **Selection of RSP Query Engines:** CityBench allows to seamlessly use different query engines as part of the testing environment. Currently, CQELS and CSPARQL are supported. However, users are encouraged to extend the list of RSP engines by embedding their engines within CTI.

9.2.3 Benchmarking Results

In order to showcase the feasibility of CityBench and highlight the importance of configuration parameters, experimental evaluations over CQELS and CSPARQL engines using CityBench are conducted. A testbed is set up with multiple configurations of *CTI* performance metrics. The two RSP engines are evaluated with respect to (i) Latency, (ii) Memory Consumption, and (iii) Completeness. It is worth mentioning that the overhead caused by the benchmark is insignificant and does not pose any threats to the validity of the results, i.e., for latency it costs several milliseconds to annotate a CSV row as an RDF graph, for memory consumption the benchmark uses up to 10 MB for tracking the results produced, for completeness the benchmark do not introduce any overhead.

9.2.3.1 Latency

Latency refers to the time consumed by the RSP engine between the input arrival and output generation. The latency of RSP engines is evaluated by increasing the number

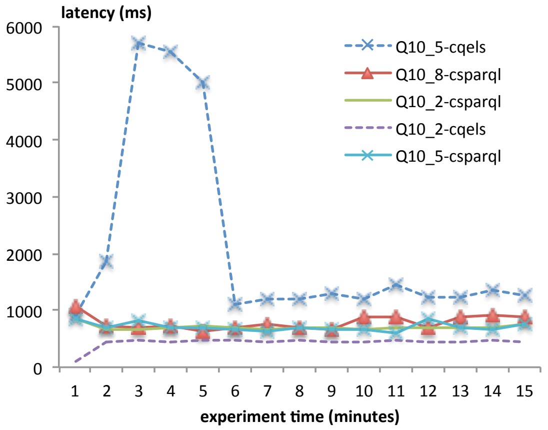


FIGURE 9.13: Latency over increasing number of data streams

of input streams within a query and by increasing the number of concurrent queries executed.

Increasing the Number of Input Streams. Three variations of query *Q10*[10] are designed to generate an immediate notification about polluted areas in the city with three configurations for the number of input data streams (2, 5 and 8). Results shown in Figure 9.13 depict that the overhead for CSPARQL was minimal with increasing number of streams, however CQELS suffer from abnormal behaviour for the query with 5 input streams and was unable to process 8 input streams within a single query. In summary, CSPARQL is the better choice for optimising query latency when the query involves many streams.

Increasing the Number of Concurrent Queries. The scalability test is performed by executing *Q1*, *Q5* and *Q8* over both engines. Queries are executed with three different configurations (1, 10, and 20) for the number of concurrent queries. Figure 9.14 and Figure 9.15 show the effect over latency with increasing number of concurrent queries for CQELS. A closer look at the results reveals that CQELS has a substantial delay, when the number of concurrent queries is increased from 0 to 10 for all three queries. However, CQELS performance is not much affected over subsequent increase from 10 to 20. As depicted in Figure 9.16 and Figure 9.17, CSPARQL seems to have a constant size of overhead for delay with the increasing number of concurrent queries in contrast to CQELS. In summary, CQELS is the better choice for optimising query latency when handling multiple concurrent queries.

[10] Different queries are selected for each experiment based on their suitability for the corresponding configuration metric. A comprehensive report containing complete results for all queries is available at CityBench website: https://github.com/CityBench/Benchmark/tree/master/result_log/samples, last accessed: Dec., 2015.

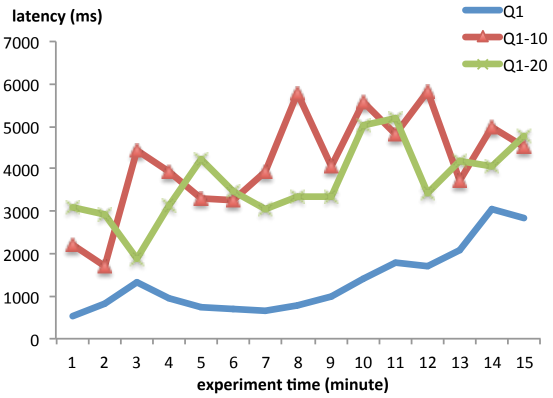


Figure 9.14: Latency over concurrent queries (*Q1* over CQELS)

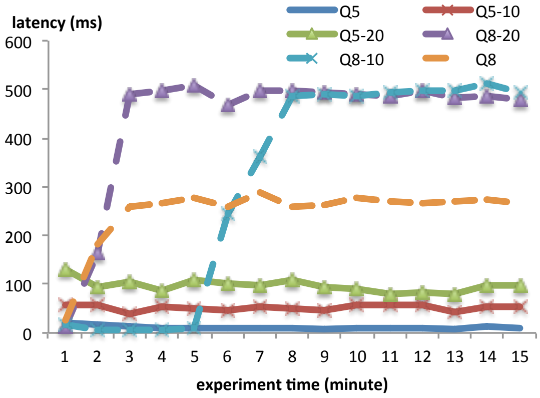


Figure 9.15: Latency over concurrent queries (*Q5* and *Q8* over CQELS)

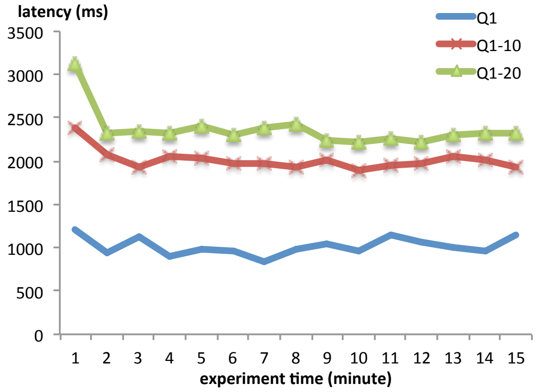


Figure 9.16: Latency over concurrent queries (*Q1* over CSPARQL)

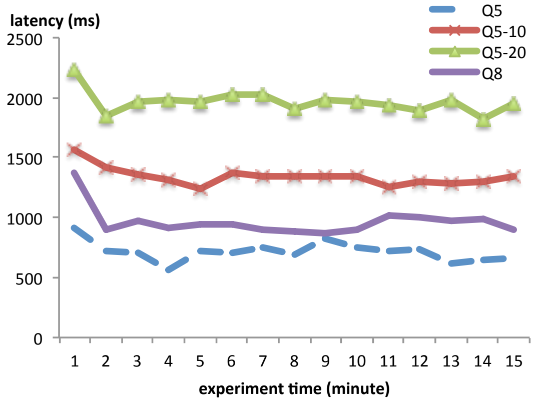


Figure 9.17: Latency over concurrent queries (*Q5* and *Q8* over CSPARQL)

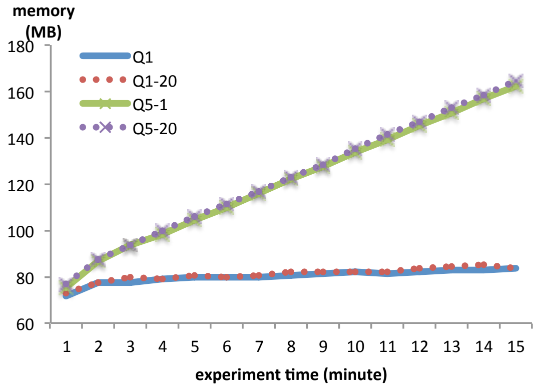


Figure 9.18: Memory consumption over concurrent queries (CSPARQL)

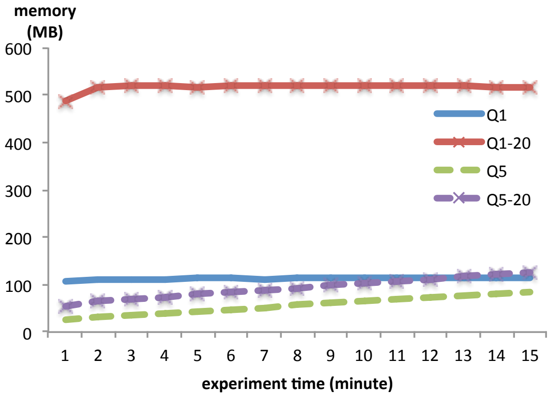


Figure 9.19: Memory consumption over concurrent queries (CQELS)

9.2.3.2 Memory Consumption

The memory usage of RSP engines is observed during the concurrent execution of an increasing number of queries and increasing the size of the background data.

Increasing the Number of Concurrent Queries. CityBench measures the memory consumption for *Q1* and *Q5* during 15 minutes of execution of each query. As shown in Figure 9.18, with an increasing number of concurrent queries, CSPARQL has a minimal impact on memory consumption for both queries. However, as time elapses during query execution, there is a constant increase in memory consumption for *Q5*, the rate of increase in memory is similar for both single query execution and 20 concurrent queries execution. In contrast, CQELS seems to have increasing memory consumption issue for *Q1*, there is also a substantial increase in memory consumption for *Q1* after an increase in the number of concurrent queries from 1 to 20. As depicted in Figure 9.19, CQELS has better performance regarding the stability of the engine over the time period of 15 minutes execution of *Q5*. Also, it is noticeable that the memory consumption of *Q5* increases linearly and it would eventually reach the memory limit and crash the engine. The reason for the abnormal behaviour is perhaps the cross-product join on the static data in *Q5* creates a lot (millions) of intermediate results and are not cleared from the cache properly in both engines. In summary, the CSPARQL is more memory efficient when increasing the number of concurrent queries.

Increasing the Size of Background Data. CityBench analyses memory consumption while increasing the size of background data. Three versions of background data required for the execution of query *Q5* are generated, with varying size of 3MB, 20MB and 30 MB. Figure 9.20 shows that CQELS seems to be better at memory management with background data of increasing size.

9.2.3.3 Completeness

The completeness of results generated by RSP engines is evaluated by executing Query *Q1* with variable input rates of data streams. Each stream is allowed to produce x observations and the benchmark counts y unique observation IDs in the results, hence the completeness c is given by: $c = y/x$. Note that the streams do not stop immediately when they finished sending triples but waited for a period of time until no new results are generated, this ensured that the stream engines have enough time for query processing. Figure 9.21, shows that CQELS completeness level has been dropped up to 50%, while CSPARQL continues to produce results with a completeness ratio of above 95%. The most probable cause of the completeness drop in CQELS is the complexity and concurrency of join over multiple streams. In summary, CSPARQL out-performs CQELS with regard to query result completeness.

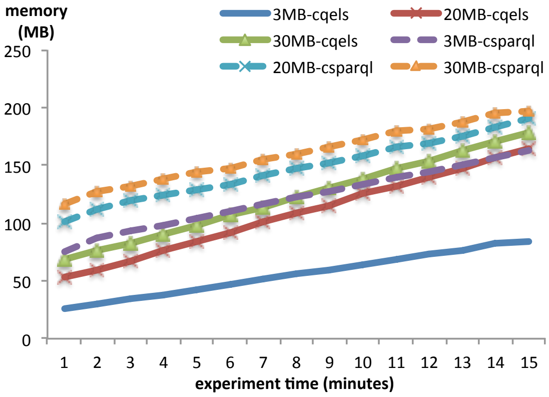


FIGURE 9.20: Memory consumption for increasing size of background data ($Q5$)

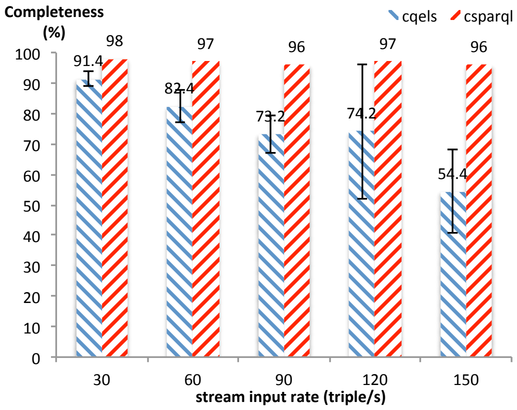


FIGURE 9.21: Completeness of results with increasing rate of input stream

9.2.4 Comparison to Existing Benchmarks

There exist several prominent works in Benchmarking RDF stores, including the Berlin SPARQL benchmark [240], LUBM [241] and SP2 [242]. However, these benchmarks are designed for static RDF storing and querying systems. LS Bench [39] and SR Bench [40] are two well-known efforts for benchmarking RSP engines. SR Benchmark is defined on weather sensors observations collected by Kno.e.sis[11]. The dataset is part of the Linked Open Data Cloud and contains weather data collected since 2002[12]. All sensor observations are semantically annotated using the SSN ontology. Beside weather streams, SR contains two static datasets (GeoNames[13] and DBPedia[14]) for integration of streaming data with background knowledge. The benchmark contains the verbal description of 17 queries covering stream flow, background integration and reasoning features. However, due to the lack of a common RDF stream query language, some of the queries are not supported by the existing engines and, therefore, cannot be executed.

LS Benchmark is a synthetically generated dataset on linked social data streams. The dataset contains 3 social data streams, namely (i) Location (GPS coordinates) stream of a social media user, (ii) stream of micro posts generated or liked by the user, and (iii) a stream of notification whenever a user uploads an image. LS Bench also provides a data generator to synthesised datasets of varying size. LS Bench contains 12 queries, covering the processing of streaming data as well as the integration of background data.

Both LS and SR benchmarks focus on evaluating RSP engines to demonstrate their query language support, process query operators and performance in a pre-configured static

[11] Wright State University homepage: http://knoesis.wright.edu, last accessed: Aug., 2015.

[12] Linked Sensor Data: http://wiki.knoesis.org/index.php/LinkedSensorData, last accessed: Aug., 2015.

[13] GeoNames dataset: http://datahub.io/dataset/geonames, last accessed: Aug., 2015.

[14] DBPedia homepage: http://wiki.dbpedia.org/, last accessed: Aug., 2015.

testbed. Best practices to design a benchmark are discussed by Gray et al. [102] and Scharrenbach et al. [103]. The real-world environment for the applications using RSP is however dynamic. Duan et al. [63] demonstrate that synthesised benchmark datasets do not portray the actual dataset requirements and therefore might produce unreliable results. CityBench extends the existing benchmarks and takes a new perspective on the evaluation of RSP engine which relies on the applications requirements and dynamicity of the environment to draw a picture that is closer to reality. Table 9.3 summarises the related work in RSP benchmarks.

Approaches	**Differences to CityBench**
Berlin [240], LUBM [241] and SP2 [242]	Applicable for static RDF repositories.
SR Bench [40]	Tests 17 verbally described queries over weather data streams and static datasets, not all are supported in existing engines, uses synthetic weather datasets and predefined configurations.
LS Bench [39]	Tests 12 queries over social media streams and user location data streams, provides data generator to create datasets of varing size, all datasets are synthetic, uses predefined configurations.

Table 9.3: Comparison to existing RSP benchmarking

9.3 Optimisation for Concurrent Queries

Benchmarking results in Section 9.2.3 show acceptable query latency and memory consumption for CQELS and CSPARQL. However, their capability for handling concurrent queries are limited, thus limits their use in city-scale applications. In order to investigate the feasibility of using RSP engines in large-scale applications, performance evaluation and optimisation with regard to concurrent queries is required. In Section 9.2.3 some initial results for handling concurrency are presented. However, only duplicates for a same query are used as concurrent queries and only 20 queries are tested at one time over a single engine instance. In the following, the performance of single CQELS and CSPARQL engines is analysed when processing multiple different queries. Then, the optimisation technique of using multiple engine instances in parallel and evaluate the improvement in query performance is discussed. Finally, the experiment results of the stress tests are presented in order to find out the capability of the server that hosts RSP engines using the aforementioned optimisation techniques. The experiments on concurrent queries are deployed over a machine running Debian GNU/Linux 6.0.10, with 8-cores of 2.13 GHz processor and 64 GB RAM. The queries used in the experiments are randomly created with 2-4 streams and 8 - 16 triple patterns[15]. The stream rate is

[15] This setting is the typical situation in the STPP and SCD scenario

configured to 15 triples per second per stream. Thus, for a single query, the input rate is 30 to 60 triples per second.

9.3.1 Multiple Different Queries over Single Engine Instance

Figure 9.22 and 9.23 show the performance of CQELS and CSPARQL engines when dealing with multiple queries. In the result data series, the letter "p" denotes the number of engine instances deployed and "q" represents the number of queries deployed.

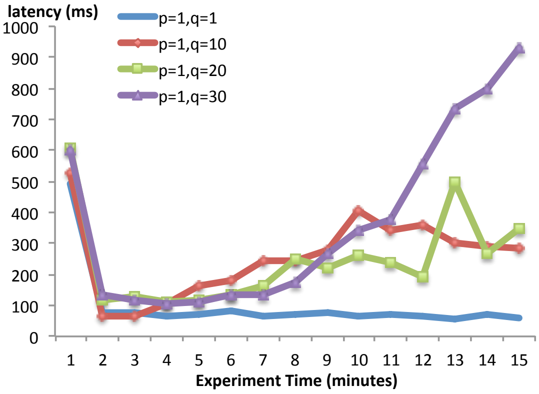


FIGURE 9.22: Latency of multiple different queries over single CQELS engine

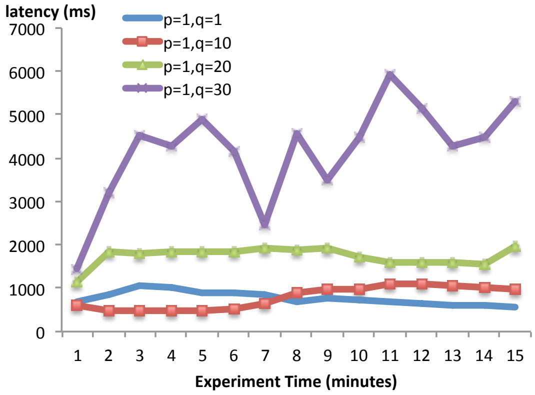


FIGURE 9.23: Latency of multiple different queries over single CSPARQL engine

The results in Figure 9.22 and 9.23 indicate that for both types of engines, the query latency increases when handling more queries. Notably, the latency of CQELS decreases significantly in the begining of the experiments. This is perhaps caused by the caching mechanisms implemented in CQELS result decoder. CQELS is relatively more efficient when handling multiple queries. Also, when the number of concurrent queries exceeds 30, the query latency is not *stable*, i.e., does not converge to stable values and will stop producing results after a period of time.

9.3.2 Optimisation using Multiple Engine Instances

One natural thought in handling many concurrent queries is to deploy multiple engine instances in parallel and distribute the workload over different engines. Thanks to the service-oriented nature of ACEIS, queries can reuse results from different engine instances and even from different types of engines. However, a load balancing strategy is needed to determine at run-time which queries are going to be deployed on which engine instances. For this purpose, an additional Scheduler module is developed, which consists of a query dispatcher and a performance monitor. The performance monitor gathers real-time status of the query engine instances, such as query latency, number of queries deployed and overall memory consumption etc. When a composition plan arrives at the subscription manager, the subscription manager queries the dispatcher for the current best engine instance for the composition. The dispatcher calculates the best engine instance based on the status of the engines reported by the performance monitor and send the identifier of the engine instance to the subscription manager. The subscription manager then deploys the query derived from the composition plan to the best engine instance. When necessary, the dispatcher will create new engine instances. The interactions between the scheduler and other components in ACEIS is depicted in Figure 9.24

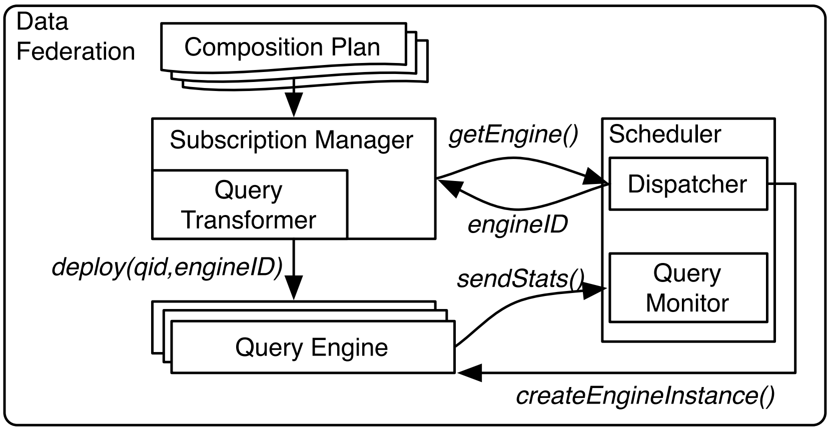


Figure 9.24: ACEIS concurrent query scheduler

The scheduler performs load balancing using different strategies. The simplest strategy is to initialise a fixed amount of engine instances in the beginning and keep the same amount of queries on each engine instance. This strategy is called the Equalised Query (denoted "EQ") strategy. Another strategy is to dynamically create new instances based on the current system load. This strategy is called the Elastic (denoted "EL") strategy. Since the experiments on a single engine instance suggest that an engine instance may become unstable when dealing with more than 30 concurrent queries, in EL, a new engine instance is deployed when the current engine reaches n queries, where $n \leq 30$ (n is set to 20 in the experiments below).

Figure 9.25 and 9.26 show the average query latency of multiple CQELS and CSPARQL engines, respectively. The results show that using two engine instances reduces the query latency for both CQELS and CSPARQL compared with single engine instance. However, deployment of more engines does not necessary result into better query performance, e.g., when 4 engines are used for 30 queries, the latency can sometimes be higher than using a single engine. Meanwhile, the elastic approach performs better than equalised

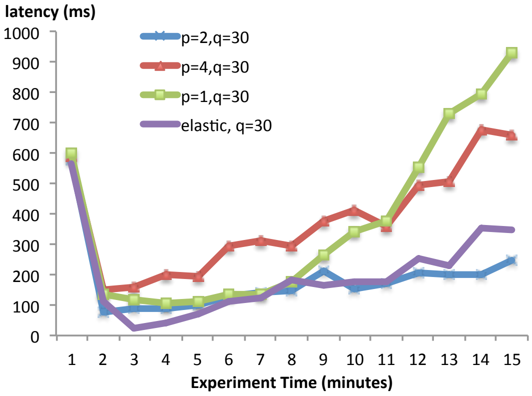


FIGURE 9.25: Latency of CQELS engines using EQ

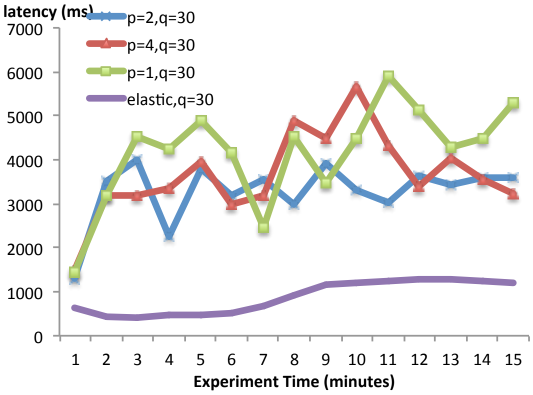


FIGURE 9.26: Latency of CSPARQL engines using EQ

queries in this experiment. Indeed, using multiple engines demands more resources such as memory and initialising all engines upfront creates overhead. Figure 9.27 and 9.28 show the memory usage for CQELS and CSPARQL under different number of concurrent queries and engine instances, respectively.

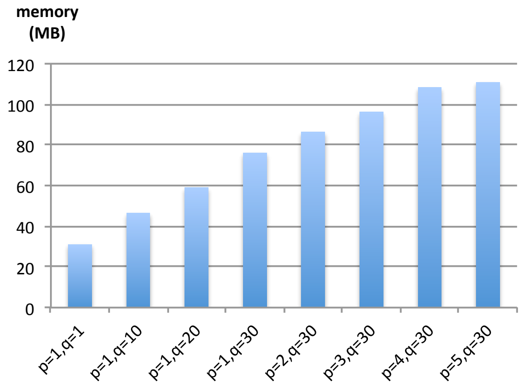


FIGURE 9.27: Memory consumption of multiple CQELS engines

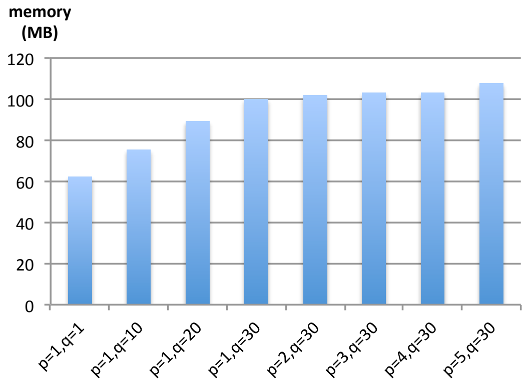


FIGURE 9.28: Memory consumption of multiple CSPARQL engines

From the results in Figure 9.27 and 9.28, it is clear that the memory consumption increases with the increasing number of concurrent queries as well as the number of engine instances. Also, CQELS uses less memory than CSPARQL when dealing with fewer queries but the memory growth rate over the increasing number of queries and engine instances are faster than CSPARQL.

Since the memory availability is limited in any system, the elastic approach will have to stop creating new engine instances at some point. Then it will regress to the equalised queries approach. An alternative way is to deploy queries on the engine that has the lowest average query latency. This strategy is called the Balanced Latency (denoted"BL") strategy. The results in Figure 9.29 and 9.30 show that the balanced latency strategy

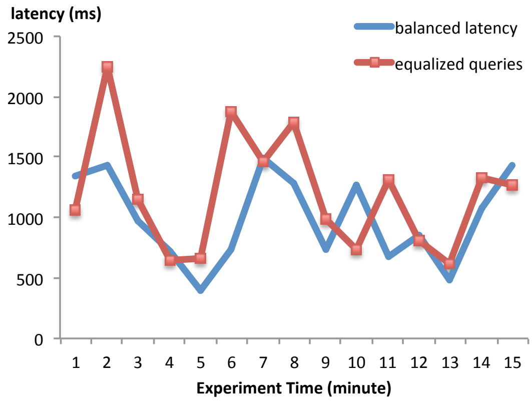


FIGURE 9.29: Latency of CQELS engines using EQ and BL while p=5,q=50

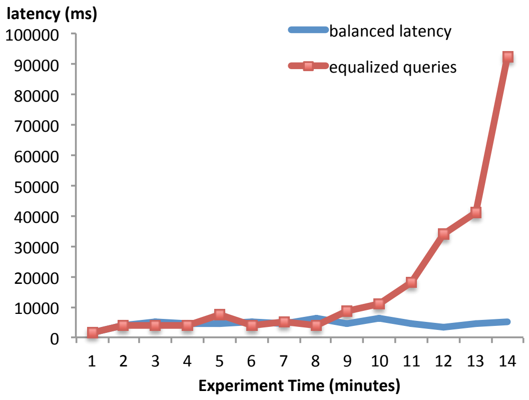


FIGURE 9.30: Latency of CSPARQL engines using EQ and BL while p=5,q=50

outperforms equalised query on both CQELS and CSPARQL when dealing with 50 concurrent queries over 5 instances. In particular, CSPARQL is unstable when using the EQ strategy but is stabilised when using BL. The results in Figure 9.31 show the improvement of query latency distribution when using BL instead of EQ. From the results, it is observable that, for CQELS engines, the number of query results with latency less than 500 milliseconds is 76% and 69% when using BL and EQ respectively. For CSPARQL engines, the number of query results with latency less than 5000 milliseconds is 49% and 36% when using BL and EQ respectively. The combined strategy of using the elastic approach in the beginning and switch to balanced query strategy when the memory limit has been reached is called Elastic-Balanced-Latency strategy (denoted “EBL”).

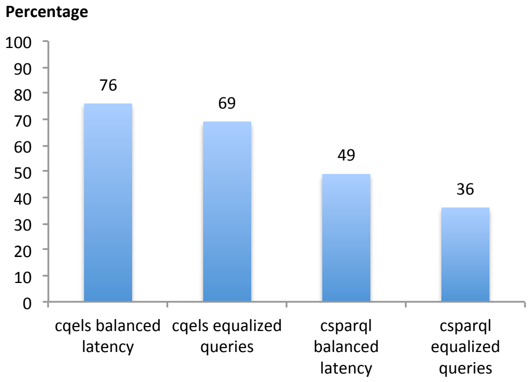


FIGURE 9.31: Query latency distribution, p=5,q=50

9.3.3 Stress Tests

In order to further investigate the feasibility of running federated RSP queries in large scale, i.e., with high input rate, large amount of input streams, and high volume of concurrent users, stress tests are conducted to evaluate the system with hundreds to thousands of queries (deploying a new query every 1-3 seconds) with the EBL load balancing strategy. The query latencies over an hour for CQELS and CSPARQL engines are shown in Figure 9.32 and 9.33.

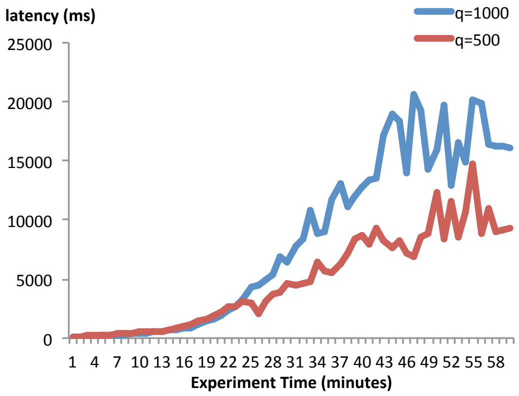


FIGURE 9.32: Latency of CQELS engines using EBL

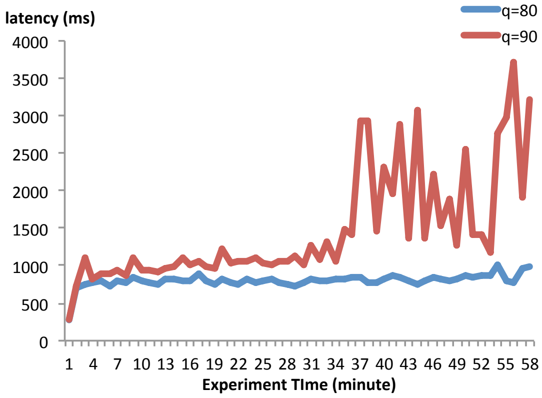


FIGURE 9.33: Latency of CSPARQL engines using EBL

The stress test results show that CQELS can handle around 1000 concurrent queries with a 15-20 second delay, while CSPARQL has a much more limited capacity of processing no more than 100 queries in a stable status. It is also worth mentioning that during the experiments it is observable that CQELS tends to use all CPU time when the workload is heavy, but CSPARQL does not use more than 30% of the CPU time even the concurrency and query delays are high. It is not clear whether this behaviour of CSPARQL is by design or an implementation issue.

9.4 Summary and Discussion

In this chapter, the usage of ACEIS prototype in real-world scenarios is introduced. The user interfaces of a smart travel/parking planning application and a smart city dashboard application are presented and the functionalities of these two applications are discussed. In order to evaluate the performance of RSP engines in real-world scenarios, a benchmarking tool is developed based on real-world datasets, called CityBench. The evaluation results of CQELS and CSPARQL engines using the datasets and queries introduced in Section 2.1 are presented and discussed. The benchmarking results indicate that both CQELS and CSPARQL have their own merits, e.g., CSPARQL is better for handling joins over multiple streams and CQELS is better at handling multiple queries etc. Leveraging the benchmark results, a developer using ACEIS can decide which type of RSP engine would be the best choice to execute the composition plans created by ACEIS, given the specific computing resources, datasets and queries he/she may have.

Benchmarking results from CityBench indicate that both RSP engines have limited capability in handling concurrent queries, which could hinder their usage in large-scale applications.

The practical problem of processing a large amount of queries concurrently using existing RSP engines (i.e., CQELS and CSPARQL) is further analysed. Means have been provided in this chapter to improve RSP engine performance without revising the internal data processing algorithms of CQELS and CSPARQL engines. In particular, the service-oriented nature of ACEIS is utilised and RSP queries are deployed as service compositions over service instances provided by multiple query engine instances. Thus, data federation over different engine instances is made possible. On top of multiple engine instances, different load balancing techniques are developed, including the equalised query, balanced latency and elastic strategy to determine which query should be delegated to which engine at run-time. A scheduler is integrated within ACEIS to control the delegation of query tasks over multiple RSP engine instances. Experiment results show that combing the Elastic and Balanced Latency strategy can achieve the best performance. Stress tests are carried out on CQELS and CSPARQL engines using EBL load balancing and the results show that on a single server node, CQELS can handle about 1000 queries in parallel while the current version of CSPARQL cannot deal with more than 100 concurrent queries. The experiment results also suggest that CSPARQL has a better performance in more complicated queries with more input streams. While the scheduler have shown progress in improving RSP capacity, this is still not enough for a city-scale application, where thousands or even millions of users can be expected. Further optimisation is required for a distributed and scalable RSP.

Chapter 10

Conclusions and Future Work

The thesis set out to explore the management of Semantic Event Service (SES) and their usage in Smart City applications, where real-time and large-scale federation of heterogeneous data streams are required. The study has investigated how to manage different activities within the life-cycle of SESs, namely the modelling, planning, implementation and adaptation phases. The state-of-the-art research in Service Computing has not provided adequate models or methods for handling SESs with event patterns and the Complex Event Processing community provides only platform-specific solutions. The study sought to answer four main research questions:

1. What is the suitable information model for describing event services and event patterns?
2. How to facilitate efficient and customised event service composition?
3. How to realise automatic and adaptive event service implementation?
4. How to evaluate and improve the performance of semantic event service execution?

The research questions are put into the context of Smart City applications in Chapter 2. The concept of a Smart City is introduced, the participants in a Smart City application and the roles they take are discussed. Sample datasets from a real-world city (the City of Aarhus, Denmark) are listed and different queries from three possible scenarios, namely the travel planner, parking place finder and city administration console, are presented. The analysis of these queries shows a strong need in the federation of heterogeneous data interfaces, formats and semantics, as well as real-time data analysis at a large-scale. These requirements justify the use of SESs in Smart City applications as an integration of Service Oriented Architecture, Complex Event Processing and Semantic Web technologies. The background of these technologies is introduced in Chapter 3. The management issues of the life-cycle of SESs and the gaps between the state-of-the-art are then discussed with examples from the motivational scenarios.

The solutions for the first three research questions constitute the main functionalities of an Automatic Complex Event Implementation System (ACEIS). An overview of ACEIS is presented in Chapter 4, where the functional design and architectural components are elaborated.

The information model used in ACEIS is described in Chapter 5. The information model consists of a Complex Event Service Ontology (CESO) extended from OWL-S and an event pattern language extended from Business Event Modelling Notations (BEMN), called BEMN$^+$. The language constructs, abstract syntax and formal semantics of BEMN$^+$ are elaborated. The event semantics of BEMN$^+$ is compared with existing languages, e.g., ETALIS [19], CSPARQL [38], CQELS [37], and BEMN [3], using the event semantic meta-model proposed in [62].

Leveraging the information model, Chapter 6 provides algorithms for pattern-based complex event service discovery and composition, which is used in the Resource Discovery and Event Service Composer components in ACEIS. It first discusses how to derive canonical event patterns from general ones in order to compare the semantic equivalence between event patterns. Then, algorithms are developed to create event service compositions based on semantic equivalence/subsumption relations between event patterns. This chapter also carries out evaluations on the performances of the proposed algorithms.

The complex event service discovery and composition algorithms in Chapter 6 are extended in Chapter 7 to address non-functional constraints and preferences. It first introduces a QoS aggregation schema to estimate the QoS metrics for event service compositions. Then, a Genetic Algorithm (GA) is designed to optimise event service compositions using their estimated QoS as the heuristic. The performance of the GA and the validity of the QoS aggregation are evaluated over synthesised datasets. The GA is also used in the Event Service Composer in ACEIS.

The composition plans generated by the algorithms in Chapter 6 and Chapter 7 are transformed into stream queries using the algorithms developed in Chapter 8, so that the requested SES can be implemented as continuous queries over annotated event streams. Chapter 8 also provides a means to enable QoS-aware event service adaptation. in cases when the user-defined QoS constraints are violated. Evaluations on the feasibility and performance of the automatic adaptation are presented and discussed. Algorithms provided in Chapter 8 are used in the Query Transformer and Adaptation Manager in ACEIS.

The answers for the fourth research question are provided in Chapter 9. This chapter first validates the usage of ACEIS in real-world scenarios by presenting the prototypical implementation of the ACEIS in two sample Smart City applications, namely the Smart Travel/Parking Planner (STPP) and the Smart City Dashboard (SCD). The functional requirements of STPP and SCD, as well as the fulfilment of the requirements by ACEIS are discussed. Then, an RSP benchmark (CityBench) is designed and developed to evaluate the performance of the RSP engines (which are responsible for the SES execution) used in ACEIS based on realistic datasets and queries. Understanding the limitation of the capacity of existing RSP engines, load balancing techniques over multiple RSP engine instances are integrated into a Scheduler component in ACEIS, in order to improve the RSP performance in handling concurrent queries. The benchmarking and optimisation results are presented and analysed.

The remainder of this chapter is organised as follows. Section 10.1 draws the conclusions of the thesis by answering the research questions. Section 10.2 presents the core contributions of the thesis. Section 10.3 discusses the limitation of the study. Section 10.4 presents some possible future directions of the research. Section 10.6 concludes.

10.1 Answers to the Research Questions

In this section, the answers for the research questions are summarised based on the solutions provided in this thesis.

RQ1: What is the suitable information model for describing event services and event patterns?

(a) *How to semantically annotate the description for event services and requests?*

The Complex Event Service Ontology (CESO) extends OWL-S to provide semantic annotations for event services and requests. By extending the *Service Profile* concept in OWL-S, CESO allows semantically annotating event service capabilities with both functional (e.g., event patterns, event payloads) and non-functional properties (e.g., QoS properties). Also, leveraging the *Grounding* concept in OWL-S, CESO can describe the access mechanism for event services, including their streaming protocol, subscription endpoint and message formats. An event request in CESO is modelled as an incomplete event service without concrete service groundings. An event request can express its functional requirements by describing the requested event pattern (event services in requested patterns also do not have groundings). Constraints and preferences over QoS properties can be used in event requests for non-functional requirements.

(b) *How to define the formal semantics of complex event patterns?*

The abstract syntax and semantics of the event patterns used in CESO are defined as an extended version of the Business Event Modelling Notations (BEMN), called BEMN$^+$. The comparison on the event semantics in BEMN$^+$ with existing approaches shows that BEMN$^+$ covers most existing event operators, including sequence, repetition, conjunction, disjunction, filters and window operators, and can define event patterns for different scenarios.

(c) *How to graphically present the event semantics so that the business users can understand and define them easily?*

BEMN$^+$ reuses and extends the graphical notations defined in BEMN, which are compatible with the standardised Business Process Model Notations (BPMN). This way BEMN$^+$ allows non-technical/business users to define event patterns with minimal learning overhead.

RQ2: How to facilitate efficient and customised event service composition?

(a) *How to efficiently create event service compositions based on the functional aspects of event services?*

The functional aspects of event services are captured by the semantics of the event patterns. Canonical forms of event patterns can be provided by applying rules for removing redundant operators. Then, the problem of comparing the semantics of two event patterns is translated into the problem of comparing the isomorphism of two canonical event patterns. A top-down traversal algorithm based on substituting semantically equivalent (sub-) event patterns is provided to compose complex event services. However, this algorithm has a high time complexity and is not efficient. To improve the efficiency, an index of event patterns is proposed based on the reusability of event patterns, called Event Reusability Forest (ERF). An algorithm is developed to build ERFs by inserting reusable patterns as descendant nodes of existing nodes. An index-based composition algorithm is then developed and evaluated. Using the reusability index, the pattern-based composition can be carried out efficiently for large service repositories and complicated queries.

(b) *How to efficiently create event service compositions based on the non-functional aspects of event services?*

A QoS aggregation schema is proposed for different QoS parameters to estimate the overall QoS performance of an event service composition and a utility function based on the weighted sums of QoS values is defined to calculate and compare QoS performances.

The QoS aggregation model is validated with simulated datasets to show that the QoS-aware optimisation is effective. A Genetic Algorithm (GA) is developed to create near-optimal event service compositions efficiently, using the QoS utility function as the heuristic. The population initialisation and crossover operations of the GA utilise the ERF for ensuring the correctness and improving the efficiency. Guidelines on fine-tuning the parameters of the genetic algorithms are provided, in order to obtain better composition results. For example, depending on the size of the solution space, the mutation rate should be set to 0 to 0.4%, population size should be set to 60 to 100, crossover rate should be 35% to 95%. The experiments results show that having a reducible population is more efficient (takes less time) and effective (achieves better results) than using a fixed size population (by allowing two individuals to crossover more than once).

RQ3: How to realise automatic and adaptive event service implementation?

(a) *How to automatically deploy executable services according to composition plans?*

The semantics of event operators in event patterns and query operators in RDF Stream Processing (RSP) are aligned. Query transformation algorithms are thus developed for two RSP engines, CQELS and CSPARQL, in order to generate RSP queries from composition plans and achieve automatic deployment of event service composition plans.

(b) *How to efficiently redeploy event service compositions when constraint violations are detected at run-time?*

Leveraging the capability of automatic implementation of event service compositions, an automatic, QoS-aware event service adaptation is realised. Different adaptation strategies, including local, global and incremental adaptations, are discussed and evaluated. The experiment results show that although there is a trade-off between adaptation success rate and adaptation time and no single best adaptation strategy can be found, the incremental adaptation based on the reusability index of the event services is the balanced choice between efficiency (time) and effectiveness (success rate). Further experiments show that the adaptation time of the incremental adaptation can be reduced when the ERF contains more edges and nodes, i.e., there exists more reusable patterns in the index, since the need for global re-composition is reduced when performing the incremental adaptation.

RQ4: How to evaluate and improve the performance of semantic event service execution?

(a) *How to design a configurable benchmark for RDF stream processing engines?*

A configurable benchmark for RSP engines is designed, taking into consideration multiple factors that may affect the RSP performance, including input streaming rate, playback time, background data size, the number of concurrent queries and the number of streams within a query. The evaluation results cover the latency over a different number of input streams and concurrent queries, memory consumption over different number of concurrent queries and different amount of background data, as well as completeness over different stream rates. The benchmark uses realistic datasets collected from the City of Aarhus as well as synthetic datasets (for simulating user locations). The queries are implemented based on the ones described in Chapter 2.

(b) *What are the differences of the RDF stream processing engines in terms of query performance?*

The benchmarking results show that CSPARQL is better at optimising query latency when handling queries with more streams involved, but CQELS is better at optimising query latency when handling more concurrent queries. CQELS consumes more memory when handling concurrent queries, while for CSPARQL the difference in memory usage is not distinguishable for handling 1 and 20 queries. Both engines consume more memory when using a larger amount of background data, but CQELS uses less memory than CSPARQL. The results on completeness show that is better at maintaining a high completeness than CQELS, when dealing with higher stream rates.

(c) *How to improve the performance of RDF stream processing engines with regard to handling concurrent queries?*

Applying load balancing techniques over multiple RSP engine instances can increase the number of concurrent queries processable by the engines. However, deploying multiple engine instances has an overhead, and the experiments show that when deploying a small number (e.g., 30) of concurrent queries over many (e.g., more than 4) engine instances, the overhead sometimes outweighs the optimisation. Therefore, creating engine instances elastically is the better choice. Moreover, balancing the load based on the average latency outperforms distributing the same number of queries over all engine instances.

10.2 Main Contributions

The core contributions of this thesis are the provision of the integrated solutions (i.e., ACEIS) for managing the life-cycle of SESs together with a configurable RSP benchmark based on realistic datasets and queries. More specifically, the contributions can be categorised into the following five sub-sections.

10.2.1 User-centric Event Service Modelling

The **Complex Event Service Ontology** is a novel service ontology extended from OWL-S [117]. CESO describes the service capability with both functional and non-functional aspects as well as access mechanisms for event services. It is the first service ontology that explicitly defines the concepts for event patterns, making complex events first-class citizen in SOA. Event requests are also modelled in CESO, with the possibility to specify QoS constraints and preferences for service consumers.

The **Extend Business Event Modelling Notations** is a graphical event pattern definition language extended from BEMN [3]. The graphical notations are reused from BEMN to be compatible with process modelling languages. The semantics of BEMN is revised and extended to be aligned with the query semantics in existing RDF stream processing engines.

10.2.2 Pattern-based Event Service Composition

Canonical Event Pattern Derivation is made possible by applying a pattern complete and reduce function. It is proved that these functions create isomorphic patterns for semantically equivalent patterns without altering the semantics. Most canonical event patterns can be created efficiently, i.e., out of 5000 random patterns with 3 to 130 nodes, 92% of them are created in less than 100 ms while 2.4% of the event patterns took more than one second, and in extreme cases it goes up to 8 seconds. In the cases when the derivation took much longer time than average, the main cause is the expanding and merging operations for the repetition nodes with high cardinality.

The **Event Reusability Forest** is constructed as an index for event services, based on the reusable relation between event patterns. The evaluation of the ERF construction algorithm shows that it scales well with the increasing size of service repositories, i.e., the completion time increases almost linearly. Also, for 1500 patterns with 10 and 25 nodes in average, indexing them takes about 58 and 323 seconds. While for very complicated patterns, e.g., patterns with 70 nodes on average, it may take nearly an hour. Fortunately, the construction of the index only needs to be done once when the server initialises. Maintaining the index during run-time would be much more efficient, i.e., inserting a pattern with 70 nodes into the index with 1500 70-node patterns takes about 2 seconds.

An **Efficient Event Service Composition Algorithm** is developed based on the ERF. The evaluation of the indexed and non-indexed pattern-based event service composition algorithm show that using the index will cause an overhead, leading to lower composition performance when the solution space is small, but the benefit of using the index emerges when the solution space gets larger. For example, when composing queries with 14 nodes over 500 service candidates, the unindexed algorithm uses 638 milliseconds and the indexed algorithm takes 2413 milliseconds, when composing more complicated queries with 25 nodes, the unindexed approach takes 11941 milliseconds and the indexed approach takes 6606 milliseconds, out-performing the unindexed algorithm by using only 55% of the composition time. When dealing with even large repositories with 1000 service candidates, the indexed approach takes 41% of the time, compared to the unindexed approach. The evaluation also shows that when the number of reusable patterns increases in the repositories, the indexed approach will out-perform the unindexed approach even if the solution space is small, e.g., for composing queries with only 14 nodes over 1000 service candidates, in which more than 70% of them can be reused by at least one another event service, the indexed composition performance is better than the unindexed.

10.2.3 Constraint-aware Event Service Composition

A **QoS Aggregation Schema** is developed to estimate the QoS for event service compositions based on the QoS of the services involved in the compositions. Unlike conventional QoS aggregation approaches ([84, 85, 201]) which cater only for composition plans represented as service workflows, the propose QoS aggregation schema is designed for declarative event composition plans and it takes into account three factors that may affect the QoS of the composition, including service infrastructure, event engine and composition pattern and it covers eight typical QoS metrics. The evaluation of the QoS aggregation shows that the estimated QoS for composition plans do not deviate too much from the real values observed from simulations. The observed latency (from simulation) deviates from the estimated value by +4.50% to +9.19%, the accuracy deviates by -3.79% to $+2.78\%$ the network consumption deviated by -13.51% to -5.19% and the completeness deviates from -17.96% to -14.89%. The reason that the deviation of completeness is more than the other metrics is that the RSP engines lose additional messages when evaluating queries over the composed data streams.

A **Genetic Algorithm for Optimising Event Service Composition** is developed with genetic encodings, population initialisation, crossover and mutation operations designed for event services. The evaluation of the GA shows that it can achieve about 89% optimal results (by comparing to the global optimum and the random picking results) efficiently, i.e., for a 12-node query, composed over 3000 to 9000 service candidates takes about 0.5 to 2 seconds. The GA also scales well with regard to the size of the query pattern and the size of the ERF, i.e., experiment results show that the execution time increases linearly.

10.2.4 Automatic Event Service Implementation and Adaptation

RSP Query Transformation Algorithms are developed to transform composition plans created by ACEIS into RSP queries executable over different RSP engines. The query transformation component in ACEIS is the first attempt to incorporate multiple target event engines (RSP engines), unlike existing systems (e.g., in [23]) which cater for a single event engine. The basis of the transformation algorithms is the semantics alignment of BEMN$^+$ with RSP query semantics. The alignment ensures correctness of the query transformation. Current transformation algorithms are designed for CQELS and CSPARQL but Incorporating other RSP engines will not take too much efforts following this methodology, since they have a similar syntax based on SPARQL.

Development and Evaluation of Different Adaptation Strategies, namely the local, global and incremental adaptation, are provided for event services. Existing adaptation strategies in CEP systems consider mainly the query rewriting to re-organise operators to achieve better performance, e.g., in [97, 233]. The adaptation in ACEIS takes a different approach by replacing the data sources. The evaluation of the adaptation strategies shows that the local adaptation is the most efficient choice, using about 8 milliseconds to complete on average, while the incremental adaptation takes about 1335 milliseconds and the global adaptation takes about 3340 milliseconds. The improvement of QoS is lowest when using the local adaptation, i.e., about 0.4% improvement, while the improvement for the incremental and global adaptation is about 34.92% and 40.43%, respectively.

10.2.5 RSP Benchmarking and Performance Optmisation

CityBench is developed as an RSP benchmarking tool. Unlike existing RSP benchmarks (i.e., LS Bench [39] and SR Bench [40]) which use mostly synthetic dataset and queries, with static test configurations, CityBench results are generated from realistic datasets. CityBench also offers configurable parameters to simulate different problem settings. The benchmarking results help a developer (or an ACEIS user) to determine which type of RSP engines has the better performance considering specific problem settings. The results for the completeness reveal a possible implementation issue in CQELS regarding the triple window maintenance.

Load Balancing for RSP Engines are implemented in the Scheduler component in ACEIS. In particular, three load balancing techniques are implemented: Equalised Query (EQ), Elastic (EL) and Balanced Latency (BL). Evaluation results show that the EL strategies out-performs the EQ strategy, i.e., for handling 30 queries using EL, CQELS and CSPARQL query latency is about 300 and 800 milliseconds, respectively. Using EQ, CQELS latency is about 425 milliseconds (averaged from results using 2 and 4 engine instances), and for CSPARQL with EQ, the average latency is about 3250 milliseconds. Also, the results show that both EQ and EL strategy out-performs the single engine performance. The evaluation results show that BL also out performs EQ over a fixed number of engine instances used. For example, when handling 50 queries with 5 engine instances, for CQELS engines, the number of query results with latency less than 500 milliseconds is 76% and 69% when using BL and EQ respectively. For CSPARQL engines, the number of query results with latency less than 5000 milliseconds is 49% and 36% when using BL and EQ respectively. The combined strategy of EL and BL is called the Elastic-Balanced-Latency (EBL) strategy, in which the EL is applied first and switches to BL when reaching the memory limit. Leveraging EBL, the number of concurrent queries that could be handled by CQELS (in stable states) increases from 30 to 1000, and for CSPARQL, the number increases from 30 to 90.

10.3 Limitations of the Study

In the following, the limitations of this study is discussed:

- **Incomplete event semantics captured in BEMN$^+$.** In this thesis, not all the event semantics are incorporated in ACEIS. Rather, a set of typical event operators and execution semantics adopted by existing RSP engines are selected and analysed. Nevertheless, the queries in the motivational scenarios and the functionalities in the prototypes of Smart City applications show that ACEIS can address various needs in Smart Cities.

- **Datasets from a single city.** The scenarios and datasets used in the experiments in this thesis come from one single city, i.e., the city of Aarhus, Denmark. Integrating scenarios and datasets from multiple cities, preferably from different countries or event continents could improve the validity of the research and the feasibility of the approach. However, collecting data from cities presents a challenge in itself during the past few years. It is believed that with the prosperity of IoT-enable devices and Smart City initiatives, more datasets will be available in the future.

- **Performance limitation of existing RSP engines.** As discussed in Chapter 9, the performance of existing RSP engine are limited. Although some optimisation techniques can be applied, they still cannot compete with conventional CEP engines like Esper (according to the figures provided in [39]), especially when dynamic reasoning is used. Some efforts have been made (e.g., in [243]) to leverage parallel stream processing engines (e.g., Apache Storm[1]) to implement RSP in a elastic and scalable way. However, so far no stable releases of such approaches are provided.

- **Integration with other semantic event engines.** ACEIS is designed to work with different event engines. Currently, only CQELS and CSPARQL engines are implemented as the target systems, in order to demonstrate the mechanisms of using multiple RSP engines. The semantics alignments for ETALIS is also provided but the engine is not used in the current ACEIS implementation. There are other RSP engines in the literature, such as TA-SPARQL [244], tSPARQL [245], Streaming SPARQL [246] and SPARQLstream [162], which may worth exploring. In the current ACEIS implementation, CQELS and CSPARQL are used because of their maturity levels and availability for technical support. Apart from these RSP engines, some rule-based semantic event engines (e.g., in [22, 156]) could also be considered, however, the query transformation might be more complicated, since they do not necessarily follow a SPARQL-like syntax.

10.4 Future Directions

The following directions are recommended for future research:

- **Extending event semantics in ACEIS.** Incorporating more event operators and event pattern semantics could expand the usage of ACEIS in more scenarios and allow ACEIS to incorporate more event engines. Possible extensions to the event semantics can be adding support for negation operators, exclusive disjunctions and interval-based events etc. In order to implement these extensions, the canonical pattern creation algorithm needs to be updated, more specifically, the lifting and merging operations in the event pattern reduction function need to be extended to support new operators, or updated to support interval-based events. Then, the definitions for reusable relations need to be expanded. The creation of reusability index and the composition algorithms based on the index can remain unchanged. Finally, the transformation algorithms needs to be redesigned.

[1] Apache Storm: http://storm.apache.org/index.html, last accessed: Dec., 2015.

- **Decentralising event service composition algorithms.** Currently, the composition algorithms are implemented as centralised Java programs. These algorithms can be decentralised by dividing the composition task into a set of sub-tasks and distributing the sub-tasks to a set of autonomous and collaborating agents in a Multiple Agent System (MAS) [247]. This way the composition task can be executed as parallel sub-tasks and achieve better efficiency. Different methods can be used to determine the division of composition tasks, for example, learning methods could be used to determine the division based on analysing historical composition results. The reusability index developed in this thesis may also be decentralised to allow distributed service query, such as in [248].

  Another possibility is to leverage logic programming approaches, e.g., Constraint Programming [249], to address the event service composition problem. This would require encoding the event patterns as logical rules processable by the logic systems and use the reasoning capability to create optimal compositions.

- **Improving the QoS-aware event service adaptation.** The adaptability of ACEIS can be improved in several aspects. Firstly, the criticality of QoS updates are evaluated based on precise QoS values, i.e., using hard constraints, in the current ACEIS implementation. In highly fluctuated environments, this may result in frequent and unnecessary adaptation actions. Alternatively, fuzzy logics can be used to determine the QoS update criticality as well as adaptation actions, e.g, in [250, 251]. Results from [250] show that using fuzzy inferencing systems can reduce the number of adaptation required, applying similar techniques over ACEIS could help reducing the cost of adaptation. This implies redesigning the QoS rules, and taking into consideration the adaptation cost while making choices for adaptation.

  Secondly, constraint delegation and negotiation techniques can be used to divide global QoS constraints into local constraints, and transform the scale of the QoS optimisation problem from global to local to reduce the complexity, e.g., in [201]. Integer Programming techniques can be used to determine how to delegate the constraints, however, means for handling dynamic service attributes is needed, in order to cope with IoT devices and Smart City applications.

  Thirdly, the message loss in adaptation needs to be minimised. As discussed in Chapter 8, existing adaptation mechanisms may incur more than 30% message loss rate. The main reason of message loss is that existing event engines cannot may dynamic changes to queries deployed, while keeping the continuity of the query evaluation. To enable this feature, the key issue is to maintain the event window as well as the query execution plan at run-time to cope with the changes in the query.

- **Optimising RSP engine performance.** RDF stream processing is still in its early stages and further investigation and research is required to provide scalable and mature solutions for integrating RSP engines in smart city applications. The usage and optimisation of RSP technologies in this thesis can serve as a good baseline for future optimisation and improvement in RSP engines. For example, in Chapter 9, the limitation of CQELS engine with regard to multiple stream joins are observed from the benchmarking results. To further investigate this issue, evaluations on different types of joins, e.g., "star-" or "chain-" shaped joins over different streams needs to be studied. Also, for improving the performance for concurrent queries, it could be beneficial to reuse intermediate query results among multiple queries within the same engine instance, which could be challenging because dependencies are introduced into a set of dynamic queries.

- **Automating event service annotation.** The current ACEIS implementation assumes event service annotations are provided by the service providers. In reality, this assumption may not hold, because providing the annotations could be a challenging task [**?**]. Automatic service annotations are studied extensively, e.g., in [252–255]. However, existing approaches focus on providing annotations for service tasks and requests. Providing automatic semantic annotations for event services and event patterns remains largely unexplored.

10.5 Lessons Learned

Looking back to the progress made during the years spent completing this thesis, I have learned the following lessons:

- *Literature reviews should be updated frequently:* as the research communities develop, some statements can easily get out-dated and continuous efforts are required to keep the literature up-to-date.

- *The scope of the Ph.D. research should be chosen wisely:* the topic cannot be too generic, one could end up doing everything and nothing at all: in the early stage of this research it was set out to explore both complex events and tasks, later I realised it is too ambitious and had to remove the part on the tasks; it also should not be too specific, which may limit the use of the outcomes.

- *Problem formalisation before solutions:* having only an "intuition" for the problem is insufficient and could be misleading. Formalising the problem always offer more clarity on the issue and its solutions - the formalisation of the event semantics introduced in this thesis is actually developed at later stages, which resulted in revisions of a lot of algorithms and evaluations.

- *Planning is more important than implementation:* rushing into experiment implementation could cost you time and effort, a well-thought experiment design is crucial. The experiments for Chapter 7 was revised because of the overlooked issues regarding scalability, query complexity and GA parameter configurations etc.
- *"Bad" results are not failures:* they often reveal much more interesting insights to the problem than "expected results", e.g., the limitations of the RSP capacity were not expected until the evaluation results are obtained, however, this opened new research opportunity for optimisation.

10.6 Conclusion

In this thesis, data streams and continuous query results over the data streams are modelled as primitive and complex event services. The data items and stream metadata are annotated using SSN and CESO, respectively, in order to improve data (and metadata) interoperability. This way, the problem of federating heterogeneous data streams can be transformed into the problem of modelling, composing and implementing Semantics Event Services (SESs). Basic principles and notions in conventional Web Service composition can thus be borrowed and extended to address the issues of managing the life-cycle of SESs. For example, the notion of service index is extended to create a reusability index for SESs, the Genetic Algorithms for optimising Web Service compositions are revised to cope with SES compositions. Evaluations of the composition algorithms show that they are scalable and effective for large service repositories and complex event requests. Leveraging the formal semantics defined for the event patterns considered in this thesis and its alignment with the query semantics of existing RSP engines, query transformation algorithms are developed to create executable RSP queries for different types of RSP engines, allowing a cross-platform collaboration of RSP engines. Adaptation strategies are then developed and tested to improve the adaptability of SES compositions. Run-time adjustments are thus made possible for RSP engines in order to keep the user-defined, multi-modal QoS constraints satisfied.

The aforementioned techniques are bundled into a stream federation middleware named Automatic Complex Event Implementation System (ACEIS). ACEIS facilitates easy-to-use, on-demand and scalable data/event stream processing and it has been deployed in prototypes of Smart City applications to evaluate the feasibility of using SESs. In the evaluation, particularly during the benchmarking of the RSP performance, the limitations of RSP capacity with regard to handling concurrent queries are revealed and efforts are made to improve the RSP capacity by applying load balancing over parallel RSP engine instances.

Appendix A

Examples of CESO

```
@prefix : <http://www.insight-centre.org/dataset/SampleEventService#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix ssn: <http://purl.oclc.org/NET/ssnx/ssn#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix owls: <http://www.daml.org/services/owl-s/1.2/Service.owl#> .
@prefix owlssc: <http://www.daml.org/services/owl-s/1.2/ServiceCategory.owl#> .
@prefix qoi: <http://ict-citypulse.eu/ontologies/StreamQoI/> .
@prefix ces: <http://www.insight-centre.org/ces#> .

:trafficSensor-234      a       ssn:Sensor , ces:PrimitiveEventService ;
      ssn:observes :Property-avgSpeed-234 , :Property-vehicleCount-234 ;
      owls:presents :234-Profile ;
      owls:supports
              [ a       ces:HttpGrounding ;
                ces:httpService
              ] .

:234-Profile    a       ces:EventProfile ;
      owlssc:serviceCategory
              [ a       owlssc:ServiceCategory ;
                owlssc:serviceCategoryName
                        "traffic_report"^^xsd:string
              ] ;
      ces:hasNFP
              [ a qoi:Correctness ;
                qoi:value "0.99"^^xsd:float
              ] .

:Property-avgSpeed      a       ct:AvgSpeed;
      ssn:isPropertyOf :FoI-1 .

:Property-vehicleCount  a       ct:VehicleCount ;
      ssn:isPropertyOf :FoI-2 .
```

Listing A.1: Example of a PES description in CESO

```
@prefix : <http://www.insight-centre.org/dataset/SampleEventService#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix ssn: <http://purl.oclc.org/NET/ssnx/ssn#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix owls: <http://www.daml.org/services/owl-s/1.2/Service.owl#> .
@prefix ces: <http://www.insight-centre.org/ces#> .

:ComplexEventService-1  a       ces:ComplexEventService ;
      owls:presents :ComplexEventService-1-Profile ;
      owls:supports
              [ a       ces:HttpGrounding ;
                ces:httpService
              ] .

:ComplexEventService-1-Profile  a       ces:EventProfile ;
      ces:hasPattern
              [ a       rdf:Bag , ces:And ;
                rdf:_1  [ a       ces:ServiceNode ;
                          ces:hasNodeId "node-1"^^xsd:string ;
                          ces:hasService :trafficSensor-230
                        ] ;
                rdf:_2  [ a       ces:ServiceNode ;
                          ces:hasNodeId "node-2"^^xsd:string ;
                          ces:hasService :trafficSensor-234
                        ] ;
      ces:hasSelection
                        [ a       ces:Selection ;
                          ces:hasNodeId "node-1" ;
                          ces:selectedProperty :Property-avgSpeed-230
                        ] ;
      ces:hasSelection
                        [ a       ces:Selection ;
                          ces:hasNodeId "node-2" ;
                          ces:selectedProperty :Property-avgSpeed-234
                        ] .
```

Listing A.2: Example of a PES description in CESO

```
@prefix : <http://www.insight-centre.org/dataset/SampleEventService#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix ssn: <http://purl.oclc.org/NET/ssnx/ssn#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix owls: <http://www.daml.org/services/owl-s/1.2/Service.owl#> .
@prefix ces: <http://www.insight-centre.org/ces#> .

:ComplexEventService-1  a       ces:ComplexEventService ;
      owls:presents :ComplexEventService-1-Profile ;
      owls:supports
              [ a       ces:HttpGrounding ;
                ces:httpService
              ] .

:ComplexEventService-1-Profile  a       ces:EventProfile ;
      ces:hasPattern
              [ a       rdf:Bag , ces:And ;
                rdf:_1  [ a       ces:ServiceNode ;
                          ces:hasNodeId "node-1"^^xsd:string ;
                          ces:hasService :trafficSensor-230
                        ] ;
                rdf:_2  [ a       ces:ServiceNode ;
                          ces:hasNodeId "node-2"^^xsd:string ;
                          ces:hasService :trafficSensor-234
                        ] ;
      ces:hasSelection
                        [ a       ces:Selection ;
                          ces:hasNodeId "node-1" ;
                          ces:selectedProperty :Property-avgSpeed-230
                        ] ;
      ces:hasSelection
                        [ a       ces:Selection ;
                          ces:hasNodeId "node-2" ;
                          ces:selectedProperty :Property-avgSpeed-234
                        ] .
```

Listing A.3: Example of an Event Request in CESO

Appendix B

XML serialization of BEMN$^+$

```
<?xml version=”1.0” encoding=”ISO−8859−1” ?>
<xs:schema xmlns:xs=”http://www.w3.org/2001/XMLSchema”>

<!−− definition of simple elements −−>
<xs:element name=”varID” type=”xs:IDREF”/>
<xs:element name=”aggID” type=”xs:IDREF”/>

<!−− definition of attributes −−>
<xs:attribute name=”headType”>
  <xs:simpleType>
    <xs:restriction base=”xs:string”>
      <xs:pattern value=”select|ask”/>
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>

<xs:attribute name=”nodeType”>
  <xs:simpleType>
    <xs:restriction base=”xs:string”>
      <xs:pattern value=”start|and|or|event|end”/>
    </xs:restriction>
  </xs:simpleType>
</xs:attribute> −−>

<xs:attribute name=”op”>
<xs:simpleType>
    <xs:restriction base=”xs:string”>
      <xs:pattern value=”gt|lt|eq”/>
    </xs:restriction>
  </xs:simpleType>
```

```
</xs:attribute>

<xs:attribute name=”aggregateOp”>
<xs:simpleType>
    <xs:restriction base=”xs:string”>
      <xs:pattern value=”sum|all|count|avg”/>
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>

<xs:attribute name=”varType”>
<xs:simpleType>
    <xs:restriction base=”xs:string”>
      <xs:pattern value=”loc|time|id|payload”/>
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>


<!–– definition of complex elements ––>
<xs:element name=”CEP” maxOccurs=”1”>
  <xs:complexType>
    <xs:sequence>
      <xs:element ref=”head” minOccurs=”1” maxOccurs=”1”/>
      <xs:element ref=”node”maxOccurs=”unbounded”/>
      <xs:element ref=”arc”maxOccurs=”unbounded”/>
      <xs:element ref=”filter”maxOccurs=”unbounded”/>
    </xs:sequence>
  <xs:attribute name=”patternName” type=”xs:string” use=”required”/>
  </xs:complexType>
</xs:element>

<xs:element name=”head”>
  <xs:complexType>
    <xs:sequence>
      <xs:element ref=”varID” minOccurs=”0” maxOccurs=”unbounded”/>
      <xs:element ref=”aggregate” minOccurs=”0” maxOccurs=”unbounded”/>
    </xs:sequence>
  </xs:complexType>
  <xs:attribute ref=”headType” use=”required”/>
</xs:element>

<xs:element name=”aggregate”>
  <xs:complexType>
```

```
    <xs:attribute name=”id” type=”xs:ID” use=”required”/>
    <xs:attribute name=”varID” type=”xs:IDREF” use=”required”/>
    <xs:attribute ref=”aggregateOp” use=”required”/>
    <xs:attribute name=”label” type=”xs:string” use=”optional” />
  </xs:complexType>
</xs:element>

<xs:element name=”node”>
  <xs:complexType>
    <xs:sequence>
      <xs:element ref=”event” minOccurs=”0” maxOccurs=”1”/>
    </xs:sequence>
    <xs:attribute name=”id” type=”xs:ID” use=”required”/>
    <xs:attribute ref=”nodeType” use=”required”/>
    <xs:attribute name=”label” type=”xs:string” use=”optional” />
  </xs:complexType>
</xs:element>

<xs:element name=”event”>
  <xs:complexType>
    <xs:sequence>
      <xs:element name= sample type=”xs:string”>
      <xs:element ref=”var” minOccurs=”0” maxOccurs=”unbounded”/>
    </xs:sequence>
    <xs:attribute name=”eventType” type=”xs:QName” use=”required”/>
    <xs:attribute name=”observedProperty” type=”xs:QName” use=”required”/>
    <xs:attribute name=”featureOfInterest” type=”xs:QName” use=”required”/>
    <xs:attribute name=”eventSource” type=”xs:anyURL” use=”optional”/>
    <xs:attribute name=”timeWindow” type=”xs:string” use=”optional”/>
  </xs:complexType>
</xs:element>

<xs:element name=”var”>
  <xs:complexType>
    <xs:sequence>
       <xs:element name=”varPattern” type=”xs:string”
        minOccurs=”0” maxOccurs=”1”>
    </xs:sequence>
    <xs:attribute name=”id” type=”xs:ID” use=”required”/>
    <xs:attribute ref=”varType” use=”required”/>
  </xs:complexType>
</xs:element>

<xs:element name=”arc”>
```

```
 <xs:complexType>
  <xs:attribute name=”from” type=”xs:IDREF” use=”required”/>
  <xs:attribute ref=”to” type=”xs:IDREF” use=”required”/>
 </xs:complexType>
</xs:element>

<xs:element name=”filter”>
 <xs:complexType>
  <xs:sequence>
   <xs:element ref=”left”minOccurs=”1” maxOccurs=”1”/>
   <xs:element ref=”right”minOccurs=”1” maxOccurs=”1”/>
  </xs:sequence>
 </xs:complexType>
 <xs:attribute name=”id” ype=”xs:ID” use=”required”/>
 <!−− <xs:attribute ref=”filterType” use=”required”/> −−>
 <xs:attribute ref=”op” use=”required”/>
</xs:element>

<xs:element name=”left”>
 <xs:complexType>
  <xs:sequence>
   <xs:choice>
    <xs:element ref=”varID” />
    <xs:element ref=”aggID” />
   </xs:choice>
  </xs:sequence>
 </xs:complexType>
</xs:element>

<xs:element name=”right”>
 <xs:complexType>
  <xs:choice>
   <xs:element ref=”varID”/>
   <xs:element name=”value” type=”xs:decimal”/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
</xs:schema>
```

Listing B.1: XML Schema for Extended BEMN

References

[1] Opher Etzion and Peter Niblett. *Event processing in action*. Manning Publications Co., 2010.

[2] V. Tsiatsis, P. Anatharam, P. Barnaghi, M. Fischer, F. Ganz, M. I. Ali, S. Kolozali, D. Kümper, A. Mileo, C.-S. Nechifor, D. Puiu, R. Tönjes, and T. Iggena. Smartcity framework. Project Deliverable 2.2, 9 2014.

[3] Gero Decker, Alexander Grosskopf, and Alistair Barros. A graphical notation for modeling complex events in business processes. In *edoc*, page 27. IEEE Computer Society, October 2007. ISBN 0-7695-2891-0. doi: 10.1109/EDOC.2007.41.

[4] P. Barnaghi, R. Tonjes, J. Holler, M. Hauswirth, A. Sheth, and P. Anantharam. Citypulse: Real-time iot stream processing and large-scale data analytics for smart city applications. In *ESWC*, 2014.

[5] Dave Evans. The internet of things: How the next evolution of the internet is changing everything. *CISCO white paper*, 1, 2011.

[6] Adam Jacobs. The pathologies of big data. *Commun. ACM*, 52(8):36–44, August 2009. ISSN 0001-0782. doi: 10.1145/1536616.1536632. URL http://doi.acm.org/10.1145/1536616.1536632.

[7] David Luckham. The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. In Nick Bassiliades, Guido Governatori, and Adrian Paschke, editors, *Interchange and Reasoning on the Web Rule Representation*, volume 5321 of *Lecture Notes in Computer Science*, pages 3–3. Springer Berlin / Heidelberg, 2008. URL $http://dx.doi.org/10.1007/978-3-540-88808-6_2$. $10.1007/978-3-540-88808-6_2$.

[8] Nenad Stojanovic, Ljiljana Stojanovic, Yongchun Xu, and Boban Stajic. Mobile cep in real-time big data processing: Challenges and opportunities. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 256–265. ACM, 2014. ISBN 978-1-4503-2737-4. doi: 10.1145/2611286.2611311. URL http://doi.acm.org/10.1145/2611286.2611311.

[9] Roland Stühmer and Nenad Stojanovic. Large-scale, situation-driven and quality-aware event marketplace: the concept, challenges and opportunities. In *Proceedings of the 5th ACM international conference on Distributed event-based system*, pages 403–404. ACM, 2011.

[10] Yiannis Verginadis, Ioannis Patiniotakis, Nikos Papageorgiou, and Roland Stuehmer. Service adaptation recommender in the event marketplace: conceptual view. In *The Semantic Web: ESWC 2011 Workshops*, pages 194–201. Springer, 2012.

[11] Mike P. Papazoglou. Service oriented computing: Concepts, characteristics and directions. In *Proceedings of the Fourth International Conference on Web Information Systems*

Engineering, WISE '03, pages 3–12, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1999-7. URL http://dl.acm.org/citation.cfm?id=960322.960404.

[12] Olga Levina and Vladimir Stantchev. Realizing event-driven soa. In *2009 Fourth International Conference on Internet and Web Applications and Services*, pages 37–42. IEEE, 2009.

[13] Jean-Louis Maréchaux. Combining service-oriented architecture and event-driven architecture using an enterprise service bus. *IBM Developer Works*, pages 1269–1275, 2006.

[14] Brenda M Michelson. Event-driven architecture overview. *Patricia Seybold Group*, 2, 2006.

[15] Stefan Appel, Sebastian Frischbier, Tobias Freudenreich, and Alejandro Buchmann. Eventlets: Components for the integration of event streams with soa. In *Service-Oriented Computing and Applications (SOCA), 2012 5th IEEE International Conference on*, pages 1–9. IEEE, 2012.

[16] Stefan Bischof, Athanasios Karapantelakis, Cosmin Septimiu Nechifor, Amit Sheth, Alessandra Mileo, and Payam Barnaghi. Semantic modeling of smart city data. In *Proc. of the W3C Workshop on the Web of Things: Enablers and services for an open Web of Devices*, Berlin, Germany, June 2014. W3C.

[17] Sylva Girtelschmid, Matthias Steinbauer, Vikash Kumar, Anna Fensel, and Gabriele Kotsis. Big data in large scale intelligent smart city installations. In *Proceedings of International Conference on Information Integration and Web-based Applications &#38; Services*, IIWAS '13, pages 428:428–428:432, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2113-6. doi: 10.1145/2539150.2539224. URL http://doi.acm.org/10.1145/2539150.2539224.

[18] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked Data – The Story So Far. *International Journal on Semantic Web and Information Systems*, (3), 2009.

[19] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. Ep-sparql: a unified language for event processing and stream reasoning. In *Proceedings of the 20th international conference on World wide web*, WWW '11, pages 635–644, 2011.

[20] Marc Schaaf, Stella Gatziu Grivas, Dennie Ackermann, Arne Diekmann, Arne Koschel, and Irina Astrova. Semantic complex event processing. *Recent Researches in Applied Information Science*, pages 38–43, 2012.

[21] Kia Teymourian, Malte Rohde, and Adrian Paschke. Fusion of background knowledge and streams of events. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, pages 302–313. ACM, 2012.

[22] Adrian Paschke and Alexander Kozlenkov. Rule-based event processing and reaction rules. In *Rule Interchange and Applications*, pages 53–66. Springer, 2009.

[23] Kerry Taylor and Lucas Leidinger. Ontology-driven complex event processing in heterogeneous sensor networks. In *The Semanic Web: Research and Applications*, pages 285–299. Springer, 2011.

[24] Zang Li, Chao-Hsien Chu, Wen Yao, and Richard A Behr. Ontology-driven event detection and indexing in smart spaces. In *Semantic Computing (ICSC), 2010 IEEE Fourth International Conference on*, pages 285–292. IEEE, 2010.

[25] Sheila A. McIlraith, Tran Cao Son, and Honglei Zeng. Semantic web services. *IEEE Intelligent Systems*, 16(2):46–53, 2001. ISSN 1541-1672. doi: http://doi.ieeecomputersociety.org/10.1109/5254.920599.

[26] Matthias Klusch, Benedikt Fries, and Katia Sycara. Automated semantic web service discovery with owls-mx. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 915–922. ACM, 2006.

[27] Jinghai Rao and Xiaomeng Su. A survey of automated web service composition methods. In Jorge Cardoso and Amit Sheth, editors, *Semantic Web Services and Web Process Composition*, volume 3387 of *Lecture Notes in Computer Science*, pages 43–54. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-24328-1. doi: 10.1007/978-3-540-30581-1_5. URL http://dx.doi.org/10.1007/978-3-540-30581-1_5.

[28] Qian Ma, Hao Wang, Ying Li, Guotong Xie, and Feng Liu. A semantic qos-aware discovery framework for web services. In *Web Services, 2008. ICWS'08. IEEE International Conference on*, pages 129–136. IEEE, 2008.

[29] Collins english dictionary - complete & unabridged 10th edition. Dec 2015.

[30] Jian Yang and Mike P Papazoglou. Service components for managing the life-cycle of service compositions. *Information Systems*, 29(2):97–125, 2004.

[31] Qing Gu and Patricia Lago. A stakeholder-driven service life cycle model for soa. In *2nd international workshop on Service oriented software engineering: in conjunction with the 6th ESEC/FSE joint meeting*, pages 1–7. ACM, 2007.

[32] Shruti P. Mahambre, Madhu Kumar S.D., and Umesh Bellur. A taxonomy of qos-aware, adaptive event-dissemination middleware. *IEEE Internet Computing*, 11(4):35–44, 2007. ISSN 1089-7801. doi: http://doi.ieeecomputersociety.org/10.1109/MIC.2007.77.

[33] Pythagoras Karampiperis, Giannis Mouchakis, Georgios Paliouras, and Vangelis Karkaletsis. Er designer toolkit: a graphical event definition authoring tool. In *PETRA'11*, pages 34–34, 2011.

[34] N. Milanovic and M. Malek. Current solutions for web service composition. *Internet Computing, IEEE*, 8(6):51 – 59, nov.-dec. 2004. ISSN 1089-7801. doi: 10.1109/MIC.2004.58.

[35] Sonia Ben Mokhtar, Anupam Kaul, Nikolaos Georgantas, and Valérie Issarny. Efficient semantic service discovery in pervasive computing environments. In *Proceedings of*

the ACM/IFIP/USENIX 2006 International Conference on Middleware, pages 240–259. Springer-Verlag New York, Inc., 2006.

[36] Elisabetta Di Nitto, Carlo Ghezzi, Andreas Metzger, Mike Papazoglou, and Klaus Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering*, 15(3-4):313–341, 2008.

[37] Danh Le-Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *The Semantic Web–ISWC 2011*, pages 370–388. Springer, 2011.

[38] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. C-sparql: Sparql for continuous querying. In *Proceedings of the 18th international conference on World wide web*, pages 1061–1062. ACM, 2009.

[39] Danh Le-Phuoc, Minh Dao-Tran, Minh-Duc Pham, Peter Boncz, Thomas Eiter, and Michael Fink. Linked stream data processing engines: Facts and figures. In *Proc. of ISWC 2012*, pages 300–312. Springer, 2012.

[40] Ying Zhang, Minh-Duc Pham, Óscar Corcho, and Jean-Paul Calbimonte. Srbench: A streaming rdf/sparql benchmark. In *Proc. of ISWC 2012*, pages 641–657, 2012.

[41] Marc Koerner and Odej Kao. Multiple service load-balancing with openflow. In *High Performance Switching and Routing (HPSR), 2012 IEEE 13th International Conference on*, pages 210–214. IEEE, 2012.

[42] George Porter and Randy H Katz. Effective web service load balancing through statistical monitoring. *Communications of the ACM*, 49(3):48–54, 2006.

[43] Bhaskaran Raman and Randy H Katz. Load balancing and stability issues in algorithms for service composition. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, volume 2, pages 1477–1487. IEEE, 2003.

[44] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, August 2001. ISSN 0734-2071. doi: 10.1145/380749.380767. URL http://doi.acm.org/10.1145/380749.380767.

[45] Andreas Ulbrich, Gero Mühl, Torben Weis, and Kurt Geihs. Programming abstractions for content-based publish/subscribe in object-oriented languages. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE*, volume 3291 of *Lecture Notes in Computer Science*, pages 1538–1557. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-23662-7. doi: 10.1007/978-3-540-30469-2_44. URL http://dx.doi.org/10.1007/978-3-540-30469-2_44.

[46] Guoli Li and Hans-Arno Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware*, pages 249–269. Springer-Verlag New York, Inc., 2005.

[47] Sinan Sen, Nenad Stojanovic, and Ruofeng Lin. A graphical editor for complex event pattern generation. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 41:1–41:2, 2009. ISBN 978-1-60558-665-6.

[48] Paolo Bellavista, Antonio Corradi, and Andrea Reale. Quality of service in wide scale publish—subscribe systems. *Communications Surveys & Tutorials, IEEE*, 16(3):1591–1616, 2014.

[49] Thomas Fischer, Andreas M Wahl, and Richard Lenz. Automated quality-of-service-aware configuration of publish-subscribe systems at design-time. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, pages 118–129. ACM, 2014.

[50] David Ingram. Reconfigurable middleware for high availability sensor systems. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 20:1–20:11. ACM, 2009. ISBN 978-1-60558-665-6. doi: 10.1145/1619258.1619285. URL http://doi.acm.org/10.1145/1619258.1619285.

[51] Raphaël Barazzutti, Pascal Felber, Hugues Mercier, Emanuel Onica, and Etienne Rivière. Thrifty privacy: Efficient support for privacy-preserving publish/subscribe. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 225–236. ACM, 2012. ISBN 978-1-4503-1315-5. doi: 10.1145/2335484.2335509. URL http://doi.acm.org/10.1145/2335484.2335509.

[52] Raphaël Barazzutti, Pascal Felber, Christof Fetzer, Emanuel Onica, Jean-François Pineau, Marcelo Pasin, Etienne Rivière, and Stefan Weigert. Streamhub: A massively parallel architecture for high-performance content-based publish/subscribe. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 63–74. ACM, 2013. ISBN 978-1-4503-1758-0. doi: 10.1145/2488222.2488260. URL http://doi.acm.org/10.1145/2488222.2488260.

[53] Shruti P. Mahambre and Umesh Bellur. An adaptive approach for ensuring reliability in event based middleware. In *Proceedings of the Second International Conference on Distributed Event-based Systems*, DEBS '08, pages 157–168. ACM, 2008. ISBN 978-1-60558-090-6. doi: 10.1145/1385989.1386010. URL http://doi.acm.org/10.1145/1385989.1386010.

[54] Christian Esposito, Domenico Cotroneo, and Aniruddha Gokhale. Reliable publish/subscribe middleware for time-sensitive internet-scale applications. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 16:1–16:12. ACM, 2009. ISBN 978-1-60558-665-6. doi: 10.1145/1619258.1619280. URL http://doi.acm.org/10.1145/1619258.1619280.

[55] Arnd Schröter, Gero Mühl, Jan Richling, and Helge Parzyjegla. Adaptive routing in publish/subscribe systems using hybrid routing algorithms. In *Proceedings of the 7th Workshop on Reflective and Adaptive Middleware*, ARM '08, pages 51–52, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-367-9. doi: 10.1145/1462716.1462726. URL http://doi.acm.org/10.1145/1462716.1462726.

[56] Christian Kuka and Daniela Nicklas. Quality matters: supporting quality-aware pervasive applications by probabilistic data stream management. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, pages 1–12. ACM, 2014.

[57] Bugra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. Elastic scaling for data stream processing. *Parallel and Distributed Systems, IEEE Transactions on*, 25(6):1447–1463, 2014.

[58] Boris Koldehofe, Frank Dürr, and Muhammad Adnan Tariq. Tutorial: Event-based systems meet software-defined networking. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 271–280. ACM, 2013. ISBN 978-1-4503-1758-0. doi: 10.1145/2488222.2488270. URL http://doi.acm.org/10.1145/2488222.2488270.

[59] Muhammad Adnan Tariq, Boris Koldehofe, Sukanya Bhowmik, and Kurt Rothermel. Pleroma: A sdn-based high performance publish/subscribe middleware. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 217–228, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2785-5. doi: 10.1145/2663165.2663338. URL http://doi.acm.org/10.1145/2663165.2663338.

[60] Josef Schiefer, Szabolcs Rozsnyai, Christian Rauscher, and Gerd Saurer. Event-driven rules for sensing and responding to business situations. In *DEBS*, volume 233 of *ACM International Conference Proceeding Series*, pages 198–205. ACM, 2007. ISBN 978-1-59593-665-3.

[61] Liangzhao Zeng, Anne HH Ngu, Boualem Benatallah, Rodion Podorozhny, and Hui Lei. Dynamic composition and optimization of web services. *Distributed and Parallel Databases*, 24(1-3):45–72, 2008.

[62] Detlef Zimmer and Rainer Unland. On the semantics of complex events in active database management systems. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, pages 392–399. IEEE, 1999.

[63] Songyun Duan, Anastasios Kementsietsidis, Kavitha Srinivas, and Octavian Udrea. Apples and oranges: a comparison of rdf benchmarks and real rdf datasets. In *Proc. of ACM SIGMOD 2011*, pages 145–156. ACM, 2011.

[64] Mohammed A Saifullah and MA Mohammed. Scalable load balancing using enhanced server health monitoring and adimission control. In *Engineering and Technology (ICETECH), 2015 IEEE International Conference on*, pages 1–4. IEEE, 2015.

[65] Varsha Thakur and Sanjay Kumar. A comparison of select load balancing algorithms in cloud computing. *IUP Journal of Computer Sciences*, 9(1), 2015.

[66] Qin Zhou, Dariush Shirmohammadi, and WH Edwin Liu. Distribution feeder reconfiguration for service restoration and load balancing. *Power Systems, IEEE Transactions on*, 12(2):724–729, 1997.

[67] Hiroya Matsuba, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. Airfoil: A topology aware distributed load balancing service. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 325–332. IEEE, 2015.

[68] S. Misbah Deen and Mohammed Al-Qasem. A query subsumption technique. In *Proceedings of the 10th International Conference on Database and Expert Systems Applications*, DEXA '99, pages 362–371, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-66448-3. URL http://dl.acm.org/citation.cfm?id=648312.755199.

[69] Adegboyega Ojo, Edward Curry, and Tomasz Janowski. Designing next generation smart city initiatives-harnessing findings and lessons from a study of ten smart city programs. 2014.

[70] Muhammad Intizar Ali, Feng Gao, and Alessandra Mileo. Citybench: A configurable benchmark to evaluate rsp engines using smart city datasets. In *The Semantic Web-ISWC 2015*, pages 374–389. Springer, 2015.

[71] S. Kolozali, D. Puschmann, A. Karapantelakis, D. Kümper H. Liang, T. Iggena, M. I. Ali, and F. Gao. Semantic data stream annotation for automated processing. Project Deliverable 3.1, 9 2014.

[72] Martin Strohbach, Holger Ziekow, Vangelis Gazis, and Navot Akiva. Towards a big data analytics framework for iot and smart city applications. In *Modeling and Processing for Next-Generation Big-Data Technologies*, pages 257–282. Springer, 2015.

[73] Payam Barnaghi, Amit Sheth, and Cory Henson. From data to actionable knowledge: Big data challenges in the web of things [guest editors' introduction]. *Intelligent Systems, IEEE*, 28(6):6–11, 2013.

[74] Andrea D'Ambrogio. A model-driven wsdl extension for describing the qos ofweb services. In *Web Services, 2006. ICWS'06. International Conference on*, pages 789–796. IEEE, 2006.

[75] Jonathon Kopecky, Tomas Vitvar, Carine Bournez, and Joel Farrell. Sawsdl: Semantic annotations for wsdl and xml schema. *Internet Computing, IEEE*, 11(6):60–67, 2007.

[76] Cory Henson, Josh K Pschorr, Amit P Sheth, Krishnaprasad Thirunarayan, et al. Semsos: Semantic sensor observation service. In *Collaborative Technologies and Systems, 2009. CTS'09. International Symposium on*, pages 44–53. IEEE, 2009.

[77] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using rapide. *Software Engineering, IEEE Transactions on*, 21(4):336–354, 1995.

[78] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the borealis stream processing engine. In *CIDR*, volume 5, pages 277–289, 2005.

[79] Marco Seiriö and Mikael Berndtsson. Design and implementation of an eca rule markup language. In *Rules and rule markup languages for the semantic web*, pages 98–112. Springer, 2005.

[80] Yanlei Diao, Neil Immerman, and Daniel Gyllstrom. Sase+: An agile language for kleene closure over event streams, 2007.

[81] Lars Brenna, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker White. Cayuga: a high-performance event processing engine. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1100–1102. ACM, 2007.

[82] Wei Liu, Zongtian Liu, Jianfeng Fu, Rong Hu, and Zhaomang Zhong. Extending owl for modeling event-oriented ontology. In *Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International Conference on*, pages 581–586. IEEE, 2010.

[83] Mohammad Alrifai and Thomas Risse. Combining global optimization with local selection for efficient qos-aware service composition. In *Proceedings of the 18th international conference on World wide web*, pages 881–890. ACM, 2009.

[84] M.C. Jaeger, G. Rojec-Goldmann, and G. Muhl. Qos aggregation for web service composition using workflow patterns. In *Proceedings of the Eighth IEEE InternationalEnterprise Distributed Object Computing Conference. EDOC 04.*, pages 149–159, 2004. doi: 10.1109/EDOC.2004.1342512.

[85] Quanwang Wu, Qingsheng Zhu, and Xing Jian. Qos-aware multi-granularity service composition based on generalized component services. In Samik Basu, Cesare Pautasso, Liang Zhang, and Xiang Fu, editors, *Service-Oriented Computing*, volume 8274 of *Lecture Notes in Computer Science*, pages 446–455. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-45004-4. doi: 10.1007/978-3-642-45005-1_33. URL http://dx.doi.org/10.1007/978-3-642-45005-1_33.

[86] Milan Zeleny and James L Cochrane. *Multiple criteria decision making*, volume 25. McGraw-Hill New York, 1982.

[87] Liangzhao Zeng, B. Benatallah, A. H H Ngu, M. Dumas, J. Kalagnanam, and H. Chang. Qos-aware middleware for web services composition. *Software Engineering, IEEE Transactions on*, 30(5):311–327, 2004. ISSN 0098-5589. doi: 10.1109/TSE.2004.11.

[88] Rainer Berbner, Michael Spahn, Nicolas Repp, Oliver Heckmann, and Ralf Steinmetz. Heuristics for qos-aware web service composition. In *Web Services, 2006. ICWS'06. International Conference on*, pages 72–82. IEEE, 2006.

[89] Liang-Jie Zhang and Bing Li. Requirements driven dynamic services composition for web services and grid solutions. *Journal of Grid Computing*, 2(2):121–140, 2004. ISSN 1570-7873. doi: 10.1007/s10723-004-4202-1. URL http://dx.doi.org/10.1007/s10723-004-4202-1.

[90] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. A lightweight approach for qos-aware service composition. In *Proceedings of 2nd international conference on service oriented computing (ICSOC 04)*, 2004.

[91] Chengwen Zhang, Sen Su, and Junliang Chen. A novel genetic algorithm for qos-aware web services selection. In *Proceedings of the Second international conference on Data Engineering Issues in E-Commerce and Services*, DEECS'06, pages 224–235, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-35440-9, 978-3-540-35440-6. doi: 10.1007/11780397_18. URL http://dx.doi.org/10.1007/11780397_18.

[92] Chunming Gao, Meiling Cai, and Huowang Chen. Qos-aware service composition based on tree-coded genetic algorithm. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 01*, COMPSAC '07, pages 361–367, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2870-8. doi: 10.1109/COMPSAC.2007.174. URL http://dx.doi.org/10.1109/COMPSAC.2007.174.

[93] W. M. P. van der Aalst and A. H. M. ter Hofstede. Yawl: Yet another workflow language. *Inf. Syst.*, 30(4):245–275, June 2005. ISSN 0306-4379. doi: 10.1016/j.is.2004.02.002. URL http://dx.doi.org/10.1016/j.is.2004.02.002.

[94] Xitong Li, Yushun Fan, and Feng Jiang. A classification of service composition mismatches to support service mediation. In *Grid and Cooperative Computing, 2007. GCC 2007. Sixth International Conference on*, pages 315–321. IEEE, 2007.

[95] Jörg Nitzsche, Tammo Van Lessen, Dimka Karastoyanova, and Frank Leymann. Bpel for semantic web services (bpel4sws). In *On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops*, pages 179–188. Springer, 2007.

[96] J. Kopecky, T. Vitvar, C. Bournez, and J. Farrell. Sawsdl: Semantic annotations for wsdl and xml schema. *Internet Computing, IEEE*, 11(6):60–67, 2007.

[97] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 4:1–4:12. ACM, 2009. ISBN 978-1-60558-665-6. doi: 10.1145/1619258.1619264. URL http://doi.acm.org/10.1145/1619258.1619264.

[98] Christopher Dabrowski and Kevin Mills. Understanding self-healing in service-discovery systems. In *Proceedings of the first workshop on Self-healing systems*, pages 15–20. ACM, 2002.

[99] Haitao Qu, Meina Song, Rihua Wang, Wu Qu, Xiaoxiang Luo, Peng Zhao, and Junde Song. An adaptive service composition performance guarantee mechanism in peer-to-peer network. In *Computer Network and Multimedia Technology, 2009. CNMT 2009. International Symposium on*, pages 1–4. IEEE, 2009.

[100] Andreas Klein, Fuyuki Ishikawa, and Shinichi Honiden. Sanga: A self-adaptive network-aware approach to service composition. *Services Computing, IEEE Transactions on*, 7(3): 452–464, 2014.

[101] Lei Yu, Wang Zhili, Meng Lingli, Wang Jiang, Luoming Meng, and Qiu Xue-song. Adaptive web services composition using q-learning in cloud. In *Services (SERVICES), 2013 IEEE Ninth World Congress on*, pages 393–396. IEEE, 2013.

[102] Jim Gray. *Benchmark handbook: for database and transaction processing systems.* Morgan Kaufmann Publishers Inc., 1992.

[103] Thomas Scharrenbach, Jacopo Urbani, Alessandro Margara, Emanuele Della Valle, and Abraham Bernstein. Seven commandments for benchmarking semantic flow processing systems. In *The Semantic Web: Semantics and Big Data*, pages 305–319. Springer, 2013.

[104] Berners T. Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, May 2001. URL http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21.

[105] David J Russomanno, Cartik Kothari, and Omoju Thomas. Sensor ontologies: from shallow to deep models. In *System Theory, 2005. SSST'05. Proceedings of the Thirty-Seventh Southeastern Symposium on*, pages 107–112. IEEE, 2005.

[106] Payam Barnaghi, Stefan Meissner, Mirko Presser, and Klaus Moessner. Sense and sens' ability: Semantic data modelling for sensor networks. In *Conference Proceedings of ICT Mobile Summit 2009*, 2009.

[107] Mike Botts, George Percivall, Carl Reed, and John Davidson. Ogc® sensor web enablement: Overview and high level architecture. In *GeoSensor networks*, pages 175–190. Springer, 2008.

[108] Randall Perrey and Mark Lycett. Service-oriented architecture. In *Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on*, pages 116–119. IEEE, 2003.

[109] Wil MP Van der Aalst. The application of petri nets to workflow management. *Journal of circuits, systems, and computers*, 8(01):21–66, 1998.

[110] Wil MP Van der Aalst. Formalization and verification of event-driven process chains. *Information and Software technology*, 41(10):639–650, 1999.

[111] WilM.P. van der Aalst, ArthurH.M. ter Hofstede, and Mathias Weske. Business process management: A survey. In WilM.P. van der Aalst and Mathias Weske, editors, *Business Process Management*, volume 2678 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-40318-0. doi: 10.1007/3-540-44895-0_1. URL http://dx.doi.org/10.1007/3-540-44895-0_1.

[112] Alistair Barros, Marlon Dumas, and Phillipa Oaks. A critical overview of the web services choreography description language. *BPTrends Newsletter*, 3:1–24, 2005.

[113] P. Wohed, W.M.P. van der Aalst, M. Dumas, A.H.M. ter Hofstede, and N. Russell. On the suitability of bpmn for business process modelling. In Schahram Dustdar, JoséLuiz Fiadeiro, and AmitP. Sheth, editors, *Business Process Management*, volume 4102 of *Lecture Notes in Computer Science*, pages 161–176. Springer Berlin Heidelberg, 2006. ISBN

978-3-540-38901-9. doi: 10.1007/11841760_12. URL http://dx.doi.org/10.1007/11841760_12.

[114] Marta Sabou, Debbie Richards, and Sander Van Splunter. An experience report on using daml-s. In *The Proceedings of the Twelfth International World Wide Web Conference Workshop on E-Services and the Semantic Web (ESSW'03). Budapest*, 2003.

[115] Demian Antony D'Mello and VS Ananthanarayana. A review of dynamic web service description and discovery techniques. In *Integrated Intelligent Computing (ICIIC), 2010 First International Conference on*, pages 246–251. IEEE, 2010.

[116] Nikola Milanovic and Miroslaw Malek. Current solutions for web service composition. *IEEE Internet Computing*, 8(6):51–59, 2004.

[117] David Martin, Mark Burstein, Drew Mcdermott, Sheila Mcilraith, Massimo Paolucci, Katia Sycara, Deborah L. Mcguinness, Evren Sirin, and Naveen Srinivasan. Bringing semantics to web services with owl-s. *World Wide Web*, 10(3):243–277, September 2007. ISSN 1386-145X.

[118] Sourish Dasgupta, Satish Bhat, and Yugyung Lee. Sgps: a semantic scheme for web service similarity. In *Proceedings of the 18th international conference on World wide web*, pages 1125–1126. ACM, 2009.

[119] Jeffrey Hau, William Lee, and John Darlington. A semantic similarity measure for semantic web services. In *Web Service Semantics Workshop at WWW*, pages 10–14, 2005.

[120] Sourish Dasgupta, Satish Bhat, and Yugyung Lee. Taxonomic clustering and query matching for efficient service discovery. In *Web Services (ICWS), 2011 IEEE International Conference on*, pages 363–370. IEEE, 2011.

[121] Eleni Stroulia and Yiqiao Wang. Structural and semantic matching for assessing web-service similarity. *International Journal of Cooperative Information Systems*, 14(04):407–437, 2005.

[122] Mark Carman, Luciano Serafini, and Paolo Traverso. Web service composition as planning. In *ICAPS 2003 workshop on planning for web services*, pages 1636–1642, 2003.

[123] Joachim Peer. A pddl based tool for automatic web service composition. In *Principles and practice of semantic web reasoning*, pages 149–163. Springer, 2004.

[124] Anton Riabov and Zhen Liu. Scalable planning for distributed stream processing systems. In *ICAPS*, pages 31–41, 2006.

[125] Zhen Liu, Anand Ranganathan, and Anton Riabov. A planning approach for message-oriented semantic web service composition. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, volume 22, page 1389. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007.

[126] Brahim Medjahed, Athman Bouguettaya, and Ahmed K Elmagarmid. Composing web services on the semantic web. *The VLDB journal*, 12(4):333–351, 2003.

[127] Hai Huang, W-T Tsai, and Raymond Paul. Automated model checking and testing for composite web services. In *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*, pages 300–307. IEEE, 2005.

[128] Zhenhai Duan, Zhi-Li Zhang, and Yiwei Thomas Hou. Service overlay networks: Slas, qos, and bandwidth provisioning. *IEEE/ACM Transactions on Networking (TON)*, 11(6): 870–883, 2003.

[129] Fatih Karatas and Dogan Kesdogan. An approach for compliance-aware service selection with genetic algorithms. In *Service-Oriented Computing*, pages 465–473. Springer, 2013.

[130] Zhen Ye, Athman Bouguettaya, and Xiaofang Zhou. Qos-aware cloud service composition using time series. In *Service-Oriented Computing*, pages 9–22. Springer, 2013.

[131] Richard M Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33 (10):30–53, 1990.

[132] Norman W. Paton and Oscar Díaz. Active database systems. *ACM Comput. Surv.*, 31 (1):63–103, March 1999. ISSN 0360-0300. doi: 10.1145/311531.311623. URL http://doi.acm.org/10.1145/311531.311623.

[133] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):15, 2012.

[134] David Luckham et al. Complex event processing in financial services. 2008.

[135] Chuanzhen Zang and Yushun Fan. Complex event processing in enterprise information systems based on rfid. *Enterprise Information Systems*, 1(1):3–23, 2007.

[136] Wen Yao, Chao-Hsien Chu, and Zang Li. Leveraging complex event processing for smart hospitals using {RFID}. *Journal of Network and Computer Applications*, 34(3):799 – 810, 2011. ISSN 1084-8045. doi: http://dx.doi.org/10.1016/j.jnca.2010.04.020. URL http://www.sciencedirect.com/science/article/pii/S1084804510000949. {RFID} Technology, Systems, and Applications.

[137] Luca Filipponi, Andrea Vitaletti, Giada Landi, Vincenzo Memeo, Giorgio Laura, and Paolo Pucci. Smart city: An event driven architecture for monitoring public spaces with heterogeneous sensors. In *Sensor Technologies and Applications (SENSORCOMM), 2010 Fourth International Conference on*, pages 281–286. IEEE, 2010.

[138] Adrian Paschke, Paul Vincent, Alex Alves, and Catherine Moxey. Tutorial on advanced design patterns in event processing. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 324–334. ACM, 2012. ISBN 978-1-4503-1315-5. doi: 10.1145/2335484.2335519. URL http://doi.acm.org/10.1145/2335484.2335519.

[139] K. Mani Chandy, Michel. Charpentier, and Agostino Capponi. Towards a theory of events. In *Proceedings of the 2007 Inaugural International Conference on Distributed Event-based Systems*, DEBS '07, pages 180–187. ACM, 2007. ISBN 978-1-59593-665-3. doi: 10.1145/1266894.1266929. URL http://doi.acm.org/10.1145/1266894.1266929.

[140] S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data Knowl. Eng.*, 14(1):1–26, November 1994. ISSN 0169-023X. doi: 10.1016/0169-023X(94)90006-X. URL http://dx.doi.org/10.1016/0169-023X(94)90006-X.

[141] Ling Liu and M. Tamer Zsu. *Encyclopedia of Database Systems*. Springer Publishing Company, Incorporated, 1st edition, 2009. ISBN 0387355448, 9780387355443.

[142] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31(4):161–172, August 2001. ISSN 0146-4833. doi: 10.1145/964723.383072. URL http://doi.acm.org/10.1145/964723.383072.

[143] Fengyun Cao and Jaswinder Pal Singh. Medym: Match-early with dynamic multicast for content-based publish-subscribe networks. In *Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware*, Middleware '05, pages 292–313, New York, NY, USA, 2005. Springer-Verlag New York, Inc. URL http://dl.acm.org/citation.cfm?id=1515890.1515905.

[144] Fengyun Cao and Jaswinder Pal Singh. Efficient event routing in content-based publish-subscribe service networks. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 2, pages 929–940. IEEE, 2004.

[145] Greg Hamerly and Charles Elkan. Alternatives to the k-means algorithm that find better clusterings. In *Proceedings of the eleventh international conference on Information and knowledge management*, pages 600–607. ACM, 2002.

[146] Ki-Yeol Ryu and Jung-Tae Lee. An enhancement of siena event routing algorithms. In *Revised Papers from the International Conference on Information Networking, Wireless Communications Technologies and Network Applications-Part II*, ICOIN '02, pages 623–633, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-44255-3. URL http://dl.acm.org/citation.cfm?id=646286.688450.

[147] Robert Strom, Guruduth Banavar, Tushar Chandra, Marc Kaplan, Kevan Miller, Bodhi Mukherjee, Daniel Sturman, and Michael Ward. Gryphon: An information flow based approach to message brokering. *arXiv preprint cs/9810019*, 1998.

[148] Ludger Fiege, Gero Mühl, and Felix C Gärtner. Modular event-based systems. *The Knowledge Engineering Review*, 17(04):359–388, 2002.

[149] Yi-Min Wang, Lili Qiu, Dimitris Achlioptas, Gautam Das, Paul Larson, and Helen J Wang. Subscription partitioning and routing in content-based publish/subscribe systems. In *16th International Symposium on DIStributed Computing (DISCÕ02)*. Citeseer, 2002.

[150] Peter Robert Pietzuch. *Hermes: A scalable event-based middleware*. PhD thesis, University of Cambridge Cambridge, UK, 2004.

[151] Roberto Baldoni, Carlo Marchetti, Antonino Virgillito, and Roman Vitenberg. Content-based publish-subscribe over structured overlay networks. In *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on*, pages 437–446. IEEE, 2005.

[152] Abhishek Gupta, Ozgur D. Sahin, Divyakant Agrawal, and Amr El Abbadi. Meghdoot: Content-based publish/subscribe over p2p networks. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '04, pages 254–273, New York, NY, USA, 2004. Springer-Verlag New York, Inc. ISBN 3-540-23428-4. URL http://dl.acm.org/citation.cfm?id=1045658.1045677.

[153] Roberto Baldoni and Antonino Virgillito. Distributed event routing in publish/subscribe communication systems: a survey. *DIS, Universita di Roma La Sapienza, Tech. Rep*, page 5, 2005.

[154] Adrian Paschke. A semantic design pattern language for complex event processing. In *AAAI Spring Symposium: Intelligent Event Processing*, pages 54–60, 2009.

[155] Robin Keskisärkkä and Eva Blomqvist. Semantic complex event processing for social media monitoring-a survey. In *Proceedings of Social Media and Linked Data for Emergency Response (SMILE) Co-located with the 10th Extended Semantic Web Conference, Montpellier, France. CEUR workshop proceedings (May 2013)*, 2013.

[156] Kia Teymourian, Malte Rohde, and Adrian Paschke. Knowledge-based processing of complex stock market events. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 594–597. ACM, 2012.

[157] Lars Brenna, Johannes Gehrke, Mingsheng Hong, and Dag Johansen. Distributed event stream processing with non-deterministic finite automata. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 3:1–3:12. ACM, 2009. ISBN 978-1-60558-665-6. doi: 10.1145/1619258.1619263. URL http://doi.acm.org/10.1145/1619258.1619263.

[158] Emanuele Valle, Stefano Ceri, Davide Francesco Barbieri, Daniele Braga, and Alessandro Campi. Future internet — fis 2008. chapter A First Step Towards Stream Reasoning, pages 72–81. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-00984-6. doi: 10.1007/978-3-642-00985-3_6. URL http://dx.doi.org/10.1007/978-3-642-00985-3_6.

[159] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM, 2002.

[160] Mikko Rinne, Eva Blomqvist, Robin Keskisärkkä, and Esko Nuutila. Event processing in rdf. *Proceedings of WOP*, page 13, 2013.

[161] Alessandro Margara, Jacopo Urbani, Frank van Harmelen, and Henri Bal. Streaming the web: Reasoning over dynamic data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 25:24–44, 2014.

[162] Jean-Paul Calbimonte, Oscar Corcho, and Alasdair JG Gray. Enabling ontology-based access to streaming data sources. In *The Semantic Web–ISWC 2010*, pages 96–111. Springer, 2010.

[163] Ixent Galpin, Christian YA Brenninkmeijer, Alasdair JG Gray, Farhana Jabeen, Alvaro AA Fernandes, and Norman W Paton. Snee: a query processor for wireless sensor networks. *Distributed and Parallel Databases*, 29(1-2):31–85, 2011.

[164] William Clocksin and Christopher S Mellish. *Programming in PROLOG*. Springer Science & Business Media, 2003.

[165] Syed Gillani, Gauthier Picard, Frédérique Laforest, and Antoine Zimmermann. Towards efficient semantically enriched complex event processing and pattern matching. In *OrdRing 2014-3rd International Workshop on Ordering and Reasoning*, page 8p, 2014.

[166] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub) graph isomorphism algorithm for matching large graphs. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(10):1367–1372, 2004.

[167] Feng Gao, Edward Curry, and Sami Bhiri. Complex Event Service Provision and Composition based on Event Pattern Matchmaking. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, Mumbai, India, 2014. ACM. doi: 10.1145/2611286.2611287. URL http://dx.doi.org/10.1145/2611286.2611287.

[168] Feng Gao and Sami Bhiri. User-centric complex event modeling and implementation based on ubiquitous data service. In *Proceedings of the International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, pages 67–70, 2012.

[169] Anupriya Ankolekar, Mark Burstein, Jerry R Hobbs, Ora Lassila, David Martin, Drew McDermott, Sheila A McIlraith, Srini Narayanan, Massimo Paolucci, Terry Payne, et al. Daml-s: Web service description for the semantic web. In *The Semantic WebÑISWC 2002*, pages 348–363. Springer, 2002.

[170] Michael P Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, (11):38–45, 2007.

[171] Kyriakos Kritikos and Dimitris Plexousakis. Requirements for qos-based web service description and discovery. *Services Computing, IEEE Transactions on*, 2(4):320–337, 2009.

[172] Jorge Cardoso, Alistair Barros, Norman May, and Uwe Kylau. Towards a unified service description language for the internet of services: Requirements and first developments. In *Services Computing (SCC), 2010 IEEE International Conference on*, pages 602–609. IEEE, 2010.

[173] Rubén Lara, Holger Lausen, Sinuhé Arroyo, Jos De Bruijn, and Dieter Fensel. Semantic web services: description requirements and current technologies. In *International Workshop on Electronic Commerce, Agents, and Semantic Web Services*. Citeseer, 2003.

[174] Rainer von Ammon, Thomas Ertlmaier, Opher Etzion, Alexander Kofman, and Thomas Paulus. Integrating complex events for collaborating and dynamically changing business processes. In *Proceedings of the international conference on Service-oriented computing*, ICSOC/ServiceWave'09, pages 370–384, 2009. ISBN 3-642-16131-6, 978-3-642-16131-5.

[175] Alistair Barros, Gero Decker, and Alexander Grosskopf. Complex events in business processes. In *Business Information Systems*, pages 29–40. Springer, 2007.

[176] Lily Li and Kerry Taylor. A framework for semantic sensor network services. volume 5364 of *Lecture Notes in Computer Science*, pages 347–361. Springer Berlin / Heidelberg, 2008.

[177] Nils Glombitza, Dennis Pfisterer, and Stefan Fischer. Integrating wireless sensor networks into web service-based business processes. In *Proceedings of the 4th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks*, MidSens '09, pages 25–30, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-851-3.

[178] P. Rosales, Kyuhyup Oh, Kyuri Kim, and Jae-Yoon Jung. Leveraging business process management through complex event processing for rfid and sensor networks. In *40th ICCIE*, pages 1 –6, 2010.

[179] N. Glombitza, M. Lipphardt, C. Werner, and S. Fischer. Using graphical process modeling for realizing soa programming paradigms in sensor networks. In *Wireless On-Demand Network Systems and Services, 2009. WONS 2009. Sixth International Conference on*, pages 61 –70, feb. 2009.

[180] V. Stirbu. Towards a restful plug and play experience in the web of things. In *ICSC*, pages 512 –517, 2008.

[181] Jeong-Hee Kim, Hoon Kwon, Do-Hyeun Kim, Ho-Young Kwak, and Sang-Joon Lee. Building a service-oriented ontology for wireless sensor networks. In *Computer and Information Science, 2008. ICIS 08. Seventh IEEE/ACIS International Conference on*, pages 649 –654, may 2008.

[182] Phillipa Oaks, Arthur ter Hofstede, and David Edmond. Capabilities: Describing what services can do. In Maria Orlowska, Sanjiva Weerawarana, Michael Papazoglou, and Jian Yang, editors, *Service-Oriented Computing - ICSOC 2003*, volume 2910 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin / Heidelberg, 2003.

[183] Samir Tartir, I Budak Arpinar, and Amit P Sheth. Ontological evaluation and validation. In *Theory and applications of ontology: Computer applications*, pages 115–130. Springer, 2010.

[184] James F. Allen and Lames F. Allen. Maintaining knowledge about temporal intervals. *Communication of ACM*, pages 832–843, 1983.

[185] Thomas Moser, Heinz Roth, Szabolcs Rozsnyai, Richard Mordinyi, and Stefan Biffl. Semantic event correlation using ontologies. In *On the Move to Meaningful Internet Systems: OTM 2009*, pages 1087–1094. Springer, 2009.

[186] A Sasa and O Vasilecas. Ontology-based support for complex events. *Electronics and Electrical Engineering*, 113(7):83–88, 2011.

[187] Ingo Wegener. *Complexity theory: exploring the limits of efficient algorithms*. Springer Science & Business Media, 2005.

[188] Timos K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, March 1988. ISSN 0362-5915. doi: 10.1145/42201.42203. URL http://doi.acm.org/10.1145/42201.42203.

[189] Yanlei Diao and Michael J. Franklin. High-performance xml filtering: An overview of yfilter. *IEEE Data Eng. Bull.*, pages 41–48, 2003.

[190] E. Curry. Increasing mom flexibility with portable rule bases. *Internet Computing, IEEE*, 10(6):26–32, Nov 2006. ISSN 1089-7801. doi: 10.1109/MIC.2006.128.

[191] Guoli Li and Hans-Arno Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware*, Middleware '05, pages 249–269, New York, NY, USA, 2005. Springer-Verlag New York, Inc. URL http://dl.acm.org/citation.cfm?id=1515890.1515903.

[192] John Keeney, Dominik Roblek, Dominic Jones, David Lewis, and Declan O'Sullivan. Extending siena to support more expressive and flexible subscriptions. In *DEBS*, pages 35–46, 2008.

[193] Zhenyue Long, Beihong Jin, Fengliang Qi, and Donglei Cao. Reuse strategies in distributed complex event detection. In *Quality Software, 2009. QSIC '09. 9th International Conference on*, pages 325–330, 2009. doi: 10.1109/QSIC.2009.49.

[194] Souleiman Hasan, Sean O'Riain, and Edward Curry. Approximate semantic matching of heterogeneous events. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 252–263. ACM, 2012. ISBN 978-1-4503-1315-5. doi: 10.1145/2335484.2335512. URL http://doi.acm.org/10.1145/2335484.2335512.

[195] Gero Mühl. *Large-scale content-based publish-subscribe systems*. PhD thesis, TU Darmstadt, 2002.

[196] Mert Akdere, Uğur Çetintemel, and Nesime Tatbul. Plan-based complex event detection across distributed sources. *Proc. VLDB Endow.*, 1(1):66–77, August 2008. ISSN 2150-8097. URL http://dl.acm.org/citation.cfm?id=1453856.1453869.

[197] Mo Liu, Elke Rundensteiner, Kara Greenfield, Chetan Gupta, Song Wang, Ismail Ari, and Abhay Mehta. E-cube: Multi-dimensional event sequence analysis using hierarchical

pattern query sharing. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 889–900, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0661-4. doi: 10.1145/1989323.1989416. URL http://doi.acm.org/10.1145/1989323.1989416.

[198] Feng Gao, Edward Curry, Muhammad Ali, Sami Bhiri, and Alessandra Mileo. Qos-aware complex event service composition and optimization using genetic algorithms. In *Proceedings of the 12th International Conference on Service Oriented Computing*, Paris, France, 2014. Springer.

[199] Sefki Kolozali, Maria Bermudez-Edo, Daniel Puschmann, Frieder Ganz, and Payam Barnaghi. A knowledge-based approach for real-time iot data stream annotation and processing. In *Internet of Things (iThings), 2014 IEEE International Conference on, and Green Computing and Communications (GreenCom), IEEE and Cyber, Physical and Social Computing (CPSCom), IEEE*, pages 215–222. IEEE, 2014.

[200] D. Kuemper, T. Iggena, M. Bermudez-Edo, D. Puiu, M. Fischer, and F. Gao. Measures and methods for reliable information processing. Project Deliverable 4.1, 2015.

[201] Mohammad Alrifai and Thomas Risse. Combining global optimization with local selection for efficient qos-aware service composition. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 881–890. ACM, 2009. ISBN 978-1-60558-487-4. doi: 10.1145/1526709.1526828. URL http://doi.acm.org/10.1145/1526709.1526828.

[202] N. Carvalho, F. Araujo, and L. Rodrigues. Scalable qos-based event routing in publish-subscribe systems. In *Proceedings of the Fourth IEEE International Symposium on Network Computing and Applications*, pages 101–108, 2005. doi: 10.1109/NCA.2005.45.

[203] Stefan Behnel, Ludger Fiege, and Gero Muhl. On quality-of-service and publish-subscribe. In *Proceedings of the 26th IEEE International ConferenceWorkshops on Distributed Computing Systems*, ICDCSW '06, pages 20–, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2541-5. doi: 10.1109/ICDCSW.2006.77. URL http://dx.doi.org/10.1109/ICDCSW.2006.77.

[204] Andreas Berl, Erol Gelenbe, Marco Di Girolamo, Giovanni Giuliani, Hermann De Meer, Minh Quan Dang, and Kostas Pentikousis. Energy-efficient cloud computing. *The computer journal*, 53(7):1045–1051, 2010.

[205] Ramesh Karri and Piyush Mishra. Minimizing energy consumption of secure wireless session with qos constraints. In *Communications, 2002. ICC 2002. IEEE International Conference on*, volume 4, pages 2053–2057. IEEE, 2002.

[206] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Trans. Softw. Eng.*, 27(9):827–850, September 2001. ISSN 0098-5589. doi: 10.1109/32.950318. URL http://dx.doi.org/10.1109/32.950318.

[207] Sumeer Bhola, Robert E. Strom, Saurabh Bagchi, Yuanyuan Zhao, and Joshua S. Auerbach. Exactly-once delivery in a content-based publish-subscribe system. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 7–16, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1597-5. URL http://dl.acm.org/citation.cfm?id=647883.738567.

[208] Kristo Mela, Teemu Tiainen, and Markku Heinisuo. Comparative study of multiple criteria decision making methods for building design. *Advanced Engineering Informatics*, 26(4):716 – 726, 2012. ISSN 1474-0346. doi: http://dx.doi.org/10.1016/j.aei.2012.03.001. URL http://www.sciencedirect.com/science/article/pii/S1474034612000201. EG-ICE 2011 + SI: Modern Concurrent Engineering.

[209] Michael D. Rowe and Barbara L. Pierce. Sensitivity of the weighting summation decision method to incorrect application. *Socio-Economic Planning Sciences*, 16(4):173 – 177, 1982. ISSN 0038-0121. doi: http://dx.doi.org/10.1016/0038-0121(82)90036-2. URL http://www.sciencedirect.com/science/article/pii/0038012182900362.

[210] Rainer Berbner, Michael Spahn, Nicolas Repp, Oliver Heckmann, and Ralf Steinmetz. Heuristics for qos-aware web service composition. In *Proceedings of the IEEE International Conference on Web Services*, ICWS '06, pages 72–82, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2669-1. doi: 10.1109/ICWS.2006.69. URL http://dx.doi.org/10.1109/ICWS.2006.69.

[211] Feng Gao, Muhammad Intizar Ali, and Alessandra Mileo. Semantic Discovery and Integration of Urban Data Streams. In *Proceedings of the 13th International Semantic Web Conference (ISWC'14), Workshop on Semantics for Smarter Cities*, 2014.

[212] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

[213] P.R. Pietzuch and J.M. Bacon. Hermes: a distributed event-based middleware architecture. In *Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops, 2002.*, pages 611–618, 2002. doi: 10.1109/ICDCSW.2002.1030837.

[214] Fengyun Cao and JaswinderPal Singh. Medym: Match-early with dynamic multicast for content-based publish-subscribe networks. In Gustavo Alonso, editor, *Middleware 2005*, volume 3790 of *Lecture Notes in Computer Science*, pages 292–313. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-30323-7. doi: 10.1007/11587552_15. URL http://dx.doi.org/10.1007/11587552_15.

[215] Anja Strunk. Qos-aware service composition: A survey. In *Web Services (ECOWS), 2010 IEEE 8th European Conference on*, pages 67–74. IEEE, 2010.

[216] Florian Rosenberg, Max Benjamin Müller, Philipp Leitner, Anton Michlmayr, Athman Bouguettaya, and Schahram Dustdar. Metaheuristic optimization of large-scale qos-aware service compositions. In *Services Computing (SCC), 2010 IEEE International Conference on*, pages 97–104. IEEE, 2010.

[217] Sen Su, Chengwen Zhang, and Junliang Chen. An improved genetic algorithm for web services selection. In *Distributed Applications and Interoperable Systems*, pages 284–295. Springer, 2007.

[218] Fatih Karatas and Dogan Kesdogan. An approach for compliance-aware service selection with genetic algorithms. In Samik Basu, Cesare Pautasso, Liang Zhang, and Xiang Fu, editors, *Service-Oriented Computing*, volume 8274 of *Lecture Notes in Computer Science*, pages 465–473. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-45004-4. doi: 10.1007/978-3-642-45005-1_35. URL http://dx.doi.org/10.1007/978-3-642-45005-1_35.

[219] Anna Bouch, Allan Kuchinsky, and Nina Bhatti. Quality is in the eye of the beholder: meeting users' requirements for internet quality of service. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 297–304. ACM, 2000.

[220] Heinz Mühlenbein and Dirk Schlierkamp-Voosen. Predictive models for the breeder genetic algorithm i. continuous parameter optimization. *Evolutionary computation*, 1(1):25–49, 1993.

[221] Farhana H Zulkernine and Patrick Martin. An adaptive and intelligent sla negotiation system for web services. *Services Computing, IEEE Transactions on*, 4(1):31–43, 2011.

[222] Feng Gao, Muhammad Intizar Ali, Edward Curry, and Alessandra Mileo. Qos-aware adaptation for complex event service. In *Proceedings of the 31st ACM Symposium On Applied Computing (to appear)*, 2016.

[223] R. Iyer and L. Kleinrock. Qos control for sensor networks. In *Communications, 2003. ICC '03. IEEE International Conference on*, volume 1, pages 517–521 vol.1, May 2003. doi: 10.1109/ICC.2003.1204230.

[224] Athanassios Boulis, Saurabh Ganeriwal, and Mani B Srivastava. Aggregation in sensor networks: an energy–accuracy trade-off. *Ad hoc networks*, 1(2):317–331, 2003.

[225] Jerome B Johnson and Garry L Schaefer. The influence of thermal, hydrologic, and snow deformation mechanisms on snow water equivalent pressure sensor accuracy. *Hydrological Processes*, 16(18):3529–3542, 2002.

[226] A. Bucchiarone, A. Marconi, M. Pistore, P. Traverso, P. Bertoli, and R. Kazhamiakin. Domain objects for continuous context-aware adaptation of service-based systems. In *Web Services (ICWS), 2013 IEEE 20th International Conference on*, pages 571–578, June 2013. doi: 10.1109/ICWS.2013.82.

[227] Robin Parker. *Missing Data Problems in Machine Learning*. VDM Verlag, 2010.

[228] Muhammad Adnan Tariq, Boris Koldehofe, and Kurt Rothermel. Efficient content-based routing with network topology inference. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 51–62. ACM, 2013. ISBN 978-1-4503-1758-0. doi: 10.1145/2488222.2488262. URL http://doi.acm.org/10.1145/2488222.2488262.

[229] Thomas Fischer, Andreas M. Wahl, and Richard Lenz. Automated quality-of-service-aware configuration of publish-subscribe systems at design-time. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 118–129. ACM, 2014. ISBN 978-1-4503-2737-4. doi: 10.1145/2611286.2611293. URL http://doi.acm.org/10.1145/2611286.2611293.

[230] Souleiman Hasan and Edward Curry. Approximate semantic matching of events for the internet of things. *ACM Trans. Internet Technol.*, 14(1):2:1–2:23, August 2014. ISSN 1533-5399. doi: 10.1145/2633684. URL http://doi.acm.org/10.1145/2633684.

[231] Souleiman Hasan and Edward Curry. Thematic event processing. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 109–120, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2785-5. doi: 10.1145/2663165.2663335. URL http://doi.acm.org/10.1145/2663165.2663335.

[232] Segev Wasserkrug, Avigdor Gal, Opher Etzion, and Yulia Turchin. Complex event processing over uncertain data. In *Proceedings of the second international conference on Distributed event-based systems*, pages 253–264. ACM, 2008.

[233] Ella Rabinovich, Opher Etzion, and Avigdor Gal. Pattern rewriting framework for event processing optimization. In *Proceedings of the 5th ACM international conference on Distributed event-based system*, pages 101–112. ACM, 2011.

[234] Rohit Wagle, Henrique Andrade, Kirsten Hildrum, Chitra Venkatramani, and Michael Spicer. Distributed middleware reliability and fault tolerance support in system s. In *Proceedings of the 5th ACM international conference on Distributed event-based system*, pages 335–346. ACM, 2011.

[235] Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, and Chitra Venkatramani. Spc: A distributed, scalable platform for data mining. In *Proceedings of the 4th international workshop on Data mining standards, services and platforms*, pages 27–37. ACM, 2006.

[236] Jonas Buys, Vincenzo De Florio, and Chris Blondia. Towards context-aware adaptive fault tolerance in soa applications. In *Proceedings of the 5th ACM international conference on Distributed event-based system*, pages 63–74. ACM, 2011.

[237] Lijun Mei, W.K. Chan, and T.H. Tse. An adaptive service selection approach to service composition. In *Web Services, 2008. ICWS '08. IEEE International Conference on*, pages 70–77, Sept 2008. doi: 10.1109/ICWS.2008.22.

[238] K.P. Joshi, Y. Yesha, and T. Finin. Automating cloud services life cycle through semantic technologies. *Services Computing, IEEE Transactions on*, 7(1):109–122, Jan 2014. ISSN 1939-1374. doi: 10.1109/TSC.2012.41.

[239] Lijun Wang, Lanlan Rui, Xuesong Qiu, Wenjing Li, and Kangming Jiang. A self-adaptive recovery strategy for service composition in ubiquitous stub environments. In *Computers and Communications (ISCC), 2013 IEEE Symposium on*, pages 000892–000897, July 2013. doi: 10.1109/ISCC.2013.6755062.

[240] Christian Bizer and Andreas Schultz. Benchmarking the performance of storage systems that expose sparql endpoints. *World Wide Web Internet And Web Information Systems*, 2008.

[241] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3 (2):158–182, 2005.

[242] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. Spˆ 2bench: a sparql performance benchmark. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 222–233. IEEE, 2009.

[243] Danh Le-Phuoc, Hoan Nguyen Mau Quoc, Chan Le Van, and Manfred Hauswirth. Elastic and scalable processing of linked stream data in the cloud. In *The Semantic Web–ISWC 2013*, pages 280–297. Springer, 2013.

[244] Alejandro Rodrıguez, Robert McGrath, Yong Liu, James Myers, and I Urbana-Champaign. Semantic management of streaming data. *Proc. Semantic Sensor Networks*, 80, 2009.

[245] Fabio Grandi. T-sparql: A tsql2-like temporal query language for rdf. In *ADBIS (Local Proceedings)*, pages 21–30. Citeseer, 2010.

[246] Andre Bolles, Marco Grawunder, and Jonas Jacobi. Streaming sparql extending sparql to process data streams. In *Proc. of ESWC 2008*, pages 448–462, Berlin, Heidelberg, 2008. Springer-Verlag.

[247] Jacques Ferber. *Multi-agent systems: an introduction to distributed artificial intelligence*, volume 1. Addison-Wesley Reading, 1999.

[248] M Nedim Alpdemir, Arijit Mukherjee, Norman W Paton, Paul Watson, Alvaro AA Fernandes, Anastasios Gounaris, and Jim Smith. Service-based distributed querying on the grid. In *Service-Oriented Computing-ICSOC 2003*, pages 467–482. Springer, 2003.

[249] Vijay A Saraswat and Martin Rinard. Concurrent constraint programming. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245. ACM, 1989.

[250] Barbara Pernici and S Hossein Siadat. Selection of service adaptation strategies based on fuzzy logic. In *Services (SERVICES), 2011 IEEE World Congress on*, pages 99–106. IEEE, 2011.

[251] Barbara Pernici, S Hossein Siadat, Salima Benbernou, and Mourad Ouziri. A penalty-based approach for qos dissatisfaction using fuzzy rules. In *Service-Oriented Computing*, pages 574–581. Springer, 2011.

[252] Davide Tosi and Sandro Morasca. Supporting the semi-automatic semantic annotation of web services: A systematic literature review. *Information and Software Technology*, 61: 16–32, 2015.

[253] Li Yuan-jie and Cao Jian. Web service classification based on automatic semantic annotation and ensemble learning. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 2274–2279. IEEE, 2012.

[254] Khalid Belhajjame, Suzanne M Embury, Norman W Paton, Robert Stevens, and Carole A Goble. Automatic annotation of web services based on workflow definitions. *ACM Transactions on the Web (TWEB)*, 2(2):11, 2008.

[255] Feng Gao and S. Bhiri. Capability annotation of actions based on their textual descriptions. In *WETICE Conference (WETICE), 2014 IEEE 23rd International*, pages 257–262, June 2014. doi: 10.1109/WETICE.2014.68.