GRANT AGREEMENT No 609035
FP7-SMARTCITIES-2013

Real-Time IoT Stream Processing and Large-scale Data Analytics for Smart City Applications

Collaborative Project

Data Federation and Aggregation in Large-Scale Urban Data Streams

Document Ref.	D3.2
Document Type	Report
Workpackage	WP3
Lead Contractor	NUIG
Author(s)	Feng Gao (NUIG), Muhammad Intizar Ali (NUIG), Alessandra Mileo (NUIG), Dan Puiu (SIE), Ioana Stefan (SIE)
Contributing Partners	NUIG, SIE, UNIS
Planned Delivery Date	M24
Actual Delivery Date	31.08.2015
Dissemination Level	Public
Status	Completed
Version	1.3
Reviewed by	Josiane Xavier Parreira, João Fernandes

Executive Summary

CityPulse aims at providing a smart city framework, which can integrate heterogeneous information from a multitude of information sources including Internet of Things and Internet of People. The Data Federation component within the smart city framework provides a form of data virtualisation in which heterogeneous data streams are made accessible to data consumers as one integrated data stream. Existing approaches for stream discovery and integration are mainly reliant over static artefacts and any change in the underlying resources requires manual changes in the provided integration solution. In smart city infrastructures consisting of autonomous heterogeneous IoT streams, dynamicity is a major issue for data integrated solution and demands adaptive approaches for on-demand stream discovery and integration while catering for individual users' requirements and preferences.

This report describes an information model for automated IoT stream discovery and federation called event service ontology. Event service ontology is used for semantic description of IoT data streams and an event profile is generated for all IoT streams registered within the smart city framework. An event request containing smart city application user's requirements and preferences is also generated using the provided information model. A pattern based event service composition algorithm is presented which processes the event request from the application's user and discovers the most relevant IoT streams for that particular user, after automatic matchmaking between event request and IoT stream event profile. An event service composition plan to automatically integrate the discovered data streams is thus created. The event service composition plan also makes sure that all the requirements and preferences described in the event request are satisfied during stream federation. Different approaches to optimize stream federation and constraint satisfaction are discussed and evaluations of the proposed algorithms are presented.

Contents

1. Introduction 8
2. Requirements 10
3. State of the Art 11
3.1 User-centric Request Definition 12
3.2 Event Service Planning based on Functional Aspects 12
3.3 Event Service Planning based on Non-functional Aspects 12
3.3.1 QoS aggregation 13
3.3.2 Composition plan creation and optimisation 14
3.4 Automatic Event Service Implementation 14
4. Information Model for Stream Federation: Event Service Ontology and Event Pattern Semantics 15
4.1 Complex Event Service Ontology 15
4.1.1 Overview 16
4.1.2 Event Profile 17
4.1.3 Event Pattern 18
4.1.4 Event Request 19
4.2 Syntax and Semantics for Event Patterns 20
4.2.1 Abstract Syntax of Event Patterns in BEMN+ 20
4.2.2 Formal Semantics of Event Pattern 21
4.3 Summary 25
5. Scalable Federation Plan Creation: Pattern-based Event Service Composition 25
5.1 Canonical Event Pattern 26
5.1.1 Definitions of Event Syntax Tree 26
5.1.2 Complete Event Pattern 26
5.1.3 Irreducible Event Pattern 27
5.1.4 Syntax Tree Reduction Algorithm 29
5.2 Event Pattern Discovery and Composition 30
5.2.1 Network Optimization based on Traffic Estimation 32
5.2.2 Event Pattern Composition based on Substitution 33
5.2.3 Event Pattern Composition based on Re-usability Index 34
5.3 Experiment Evaluation 37
5.3.1 General Experiment Settings 37

5.3.2 Performance of Event Query Reduction 38
5.3.3 Performance of Event Reusability Forest Construction 38
5.3.4 Performance of Event Composition 39
5.4 Summary 41
6. Scalable Federation Plan Optimization: Constraint-aware Event Service Composition 41
6.1 QoS Model and Aggregation Schema 42
6.1.1 QoS Aggregation 42
6.1.2 Event QoS Utility Function 44
6.2 Genetic Algorithm for QoS-Aware Event Service Composition Optimization 45
6.2.1 Population Initialization 45
6.2.2 Genetic Encodings for Event Syntax Trees 46
6.2.3 Crossover and Mutation Operations 46
6.3 Evaluation 48
6.3.1 Experiment Scenario: Smart Travel Planner 48
6.3.2 Part 1: Performance of the Genetic Algorithm 49
6.3.3 Part 2: Validation of QoS Aggregation Rules 54
6.4 QoS Aggregation and Stream Selection using Answer Set Programming 56
6.5 Summary 56
7. Enactment of Federation: Automatic Event Service Implementation 57
7.1 Semantics Alignment 58
7.1.1 Event Declaration 58
7.1.2 AND Operator 58
7.1.3 OR Operator 59
7.1.4 SEQUENCE Operator 59
7.1.5 REPETITION Operator 59
7.1.6 Selection 59
7.1.7 Filter and Window 59
7.1.8 Data or Time Driven Query Execution 60
7.2 Transformation Algorithm 60
7.2.1 CQELS Query Transformation 60
7.2.2 C-SPARQL Query Transformation 62
7.2.3 Event (Re-) Construction from Stream Query Results 63
7.3 Summary 64
8. Optimisation Techniques for Concurrent Query Processing 64

8.1 Single Engine Performance 65
8.2 Performance Optimization with Multiple Engine Instances 66
8.3 Stress Tests on Single Server Node 68
8.4 Summary 69
9. Conclusions 70
9.1 Outcomes 70
9.2 On-going Works 71
9.3 Future Steps 71

Figures

Figure 1: Data Federation- ACEIS Architecture 9
Figure 2: Life Cycle of Event Services 10
Figure 3: Service Workflow and Event Pattern 13
Figure 4: Overview of Citypulse Information Model 16
Figure 5: Overview of the CES Ontology 17
Figure 6: Event Profile in CESO 18
Figure 7: Event Pattern in CESO 19
Figure 8: Event Request in CESO 20
Figure 11: Examples of Event Pattern Reduction 28
Figure 12: Example of Event Service Composition Plan 31
Figure 13: Example of Creating DST Combinations 34
Figure 14: Example of Event Pattern Reusability 35
Figure 15: Execution Time of Query Reduction 38
Figure 16: Execution Time of ERF Construction 39
Figure 17: Execution Time of Composition (Indexed vs. Un-indexed) 40
Figure 18: Impact of Reuse Probability on Indexed Composition 40
Figure 19: marking the reusable nodes 46
Figure 20: example of genetic encodings and crossover 47
Figure 22: Traffic planning event request for Alice (denoted Q_a) 50
Figure 23: a variant of Bob's request (denoted Q_b) 50
Figure 24: QoS utilities derived by BF, GA and random pick 51
Figure 25: Composition time required by BF and GA for Qa 51
Figure 26: GA scalability over EP size and ERF size 52
Figure 27: CE-score over mutation rate and population size 53
Figure 28: CE-score using different crossover rate 54
Figure 29: average utility using flexible ("p=x") and fixed ("pf=x") population size 54
Figure 30: composition plans for Qa under different weight vectors 55
Figure 31: Query Latency over Different Number of Streams 65
Figure 32: Query Latency over Single C-SPARQL Engine 66

Figure 33: Query Latency over Single CQELS Engine 66
Figure 34: Query Latency over Multiple CQELS Engines 66
Figure 35: Query Latency over Multiple C-SPARQL Engines 66
Figure 36: Memory Usage by CQELS engines 67
Figure 37: Memory Usage by C-SPARQL Engines 67
Figure 38: Average Latency of CQELS, p=5, q=50 68
Figure 39: Average Latency of C-SPARQL, p=5, q=50 68
Figure 40: Query Latency Distribution: p=5, q=50 68
Figure 41: Average Latency of CQELS Using EBL Strategy 69
Figure 42: Average Latency of C-SPARQL Using EBL Strategy 69

Tables

Table 1: Comparison of Event Semantics in Existing Approaches 23
Table 2: Overall QoS Calculation 43
Table 3: QoS Aggregation Rules based on Composition Patterns 43
Table 4: Simulated Sensor Repositories 50
Table 5: Queries Used in Experiments 50
Table 6: Validation for QoS Aggregation and Estimation 56
Table 7: Semantics Alignment for Event Operators 58

Listings

Listing 1: algorithm for creating a complete event pattern 27
Listing 2: algorithm for creating an irreducible event pattern 30
Listing 3: algorithm for event composition based on top-down substitution 33
Listing 4: algorithm for inserting a canonical pattern into an ERF 36
Listing 5: algorithm for event composition based on ERF 37
Listing 6: traffic sensor data in SSN 57
Listing 7: CQELS query transformation algorithm 61
Listing 8: CQELS query example 62
Listing 9: C-SPARQL query transformation algorithm 63
Listing 10: C-SPARQL query example 64

Abbreviations

A

Abstract Composition Plan (ACP) 45

B

Brute-Force (BF) 49

C

Complex Event Processing (CEP) 9
Complex Event Service (CES) 9
Concrete Composition Plan (CCP) 45

D

Data Stream Management Systems (DSMS) 13
Direct Sub-Tree (DST) 34

E

Estimated (network) Traffic Demand (ETD) 26
Event Broker Networks (EBN) 13
Event Instance Sequence (EIS) 22
Event Reusability Forest (ERF) 35
Event Reusability Hierarchy (ERH) 35
Event Service Network (ESN) 10
Event Syntax Tree (EST) 26

F

Functional Properties (FP) 12

G

Genetic Algorithm (GA) 15

I

Information Flow Processing (IFP) 23
Integer Programming (IP) 15
Internet of Things (IoT) 9

M

Multi-Criteria Decision Making (MCDM) 43

N

Non-Functional Properties (NFP) 12

O

Open Data Aarhus (ODAA) 49

P

Primitive Event Service (PES) 10

Q

Quality-of-Service (QoS) 12

R

RDF Stream Processing (RSP) 64

S

Semantic Web (SW) 9
Semantic Web Service (SWS) 13
Service Oriented Computing (SOC) 9
Simple Additive Weighting (SAW) 14
Software Defined Network (SDN) 13
Stream Annotation Ontology (SAO) 16

1. Introduction

A smart city needs to provide both hardware to collect information on the events happening within the urban environment as well as software to help making smart decisions in urban life. The main goals of smart city applications are to enhance quality of urban services, to reduce costs, and to engage the citizens in a more effective way. The application domain for smart city applications spans from government services, public transport and crisis management, to individual health care, smart home and travel planning. Furthermore, a smart city application could be an integration of programs from different application domains, as well as an engagement of different city departments, city-contracted entrepreneurs and individual enterprises providing services. Hence, the data generated from urban infrastructure for smart city applications may arrive in different formats, e.g., traffic information, parking spaces, bus timetables, environment sensors for pollution or weather warnings etc., as well as from different interfaces, e.g., APIs, websites, web services etc. Due to the data and interface heterogeneity, an aggregation of information from various sources is typically done using static artefacts and is prone to get stale in dynamic urban infrastructure.

Noticeably, many of these application domains require an infrastructure, which can observe the environment to detect real-time events occurring in the city. Smart city applications need to process, integrate and analyse streaming data generated from Internet of Things (IoT) with minimal delay and trigger actions to adapt to the dynamic urban infrastructure. Contrary to the traditional data integration approaches, IoT stream federation cannot rely over static artefacts mainly due to the dynamicity of the IoT streams. Furthermore, in smart cities infrastructure it is more likely that multiple data streams can serve the same purpose e.g. multiple air quality sensors within the same vicinity or multiple traffic sensors over the same road. In such scenarios, discovery and selection of the most optimal data streams, which can cater for individual requirements and preferences, is a challenging task. Moreover, a real-time data analysis is needed for urban data in huge volumes and these analysis results should be context-aware and knowledge-based to reflect the true picture of the users' context and its surroundings.

Data federation is a core component for the smart city framework, as it is responsible to process application request for IoT streams and automatically discover the most relevant data streams after catering individual requirements and preferences for a particular user request. Data federation is also responsible for automatically integrate heterogeneous data streams and perform complex event processing over the integrated data streams. In order to address the aforementioned tasks for the data federation component, we propose to integrate Semantic Web (SW), Service Oriented Computing (SOC) and Complex Event Processing (CEP) and provide a data federation solution for smart city applications based on *Event Services*[1]. We refer to an event service providing complex events which describes the complex event pattern in its service description as a *Complex Event Service* (CES), otherwise if the event service provides primitive events or do not describe event

[1] In Smart City Framework, we consider IoT data streams as event services, where primitive event services refer to IoT data streams generated from a single sensor based observations and complex event services refer to integrated IoT streams having complex event pattern.

patterns in its service description we call it a *Primitive Event Service* (PES). A service network consisting of CESs and PESs is called an *Event Service Network* (ESN). We refer to an event service that delivers semantically annotated events and describes the service metadata with semantic annotations as a *Semantic Event Service* (SES).

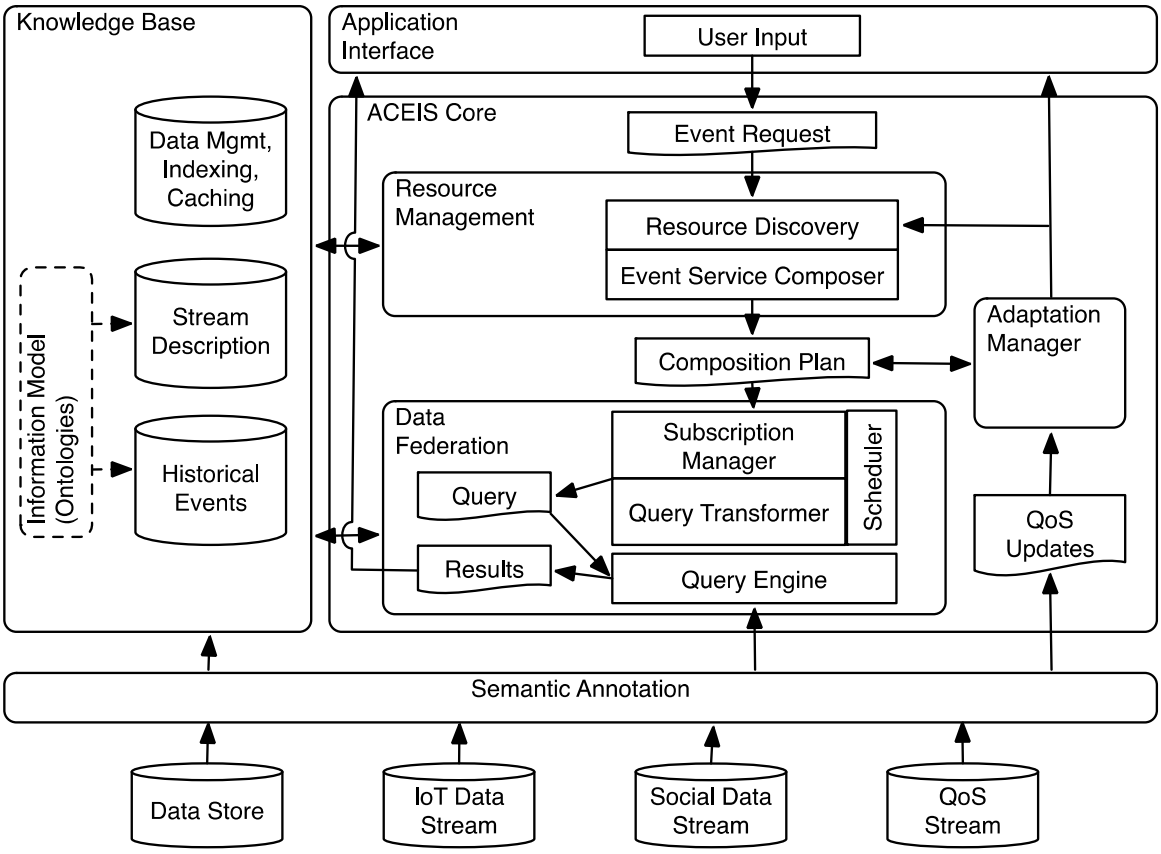


Figure 1: Data Federation- ACEIS Architecture

Figure 1 illustrates the architecture of the Automated Complex Event Implementation System (ACEIS) and its interactions with the Application Interface, the Knowledge Base and the Semantic Annotation Component. The core architecture of ACEIS consists of three main components, namely, (i) *Resource Management*, (ii) *Data Federation,* and (iii) *Adaptation Manager*. 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. The resource management component is responsible for discovering and composing event services based on static service descriptions stored in the knowledge base. It receives event requests generated by the application interface containing users' functional/non-functional requirements and preferences, and 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 data federation component is responsible for implementing the composition plan event service networks and process

complex event logics using heterogeneous data sources. The composition plan is firstly used by the subscription manager, which will make subscriptions to the event services involved in composition plan. Later, the query transformer transforms the semantically annotated composition plan into a set of stream reasoning queries to be executed on a stream query engine. The adaptation manager in the ACEIS architecture is part of *Technical Adaptation* component of CityPulse Framework and is presented in Deliverable 5.1.

The remainder of this report is organized as follows: Section 2 introduces the key requirements and activities in the life cycle of SES; Section 3 analyses the state-of-the-art with regard to the requirements; Section 4 describes the information model used for describing (integrated) sensor data streams as SESs together with the semantics of event patterns for stream federation; Section 5 describes the means for SES composition based on pattern identification and reuse; Section 6 describes the means for optimizing SES composition with regard to non-functional requirements and preferences; Section 7 describes the means for an automatic implementation of event service compositions and RDF stream query transformation algorithms. In Section 8, we present various optimisation techniques and empirical analysis of large-scale RDF stream query processing before concluding in Section 9.

2. Requirements

In order to reveal the problems of realising the aforementioned ESN, we analyse the different activities related to event services from their creation to termination. We identify the following 5 key activities in the life cycle of semantic event services, as depicted in Figure 2:

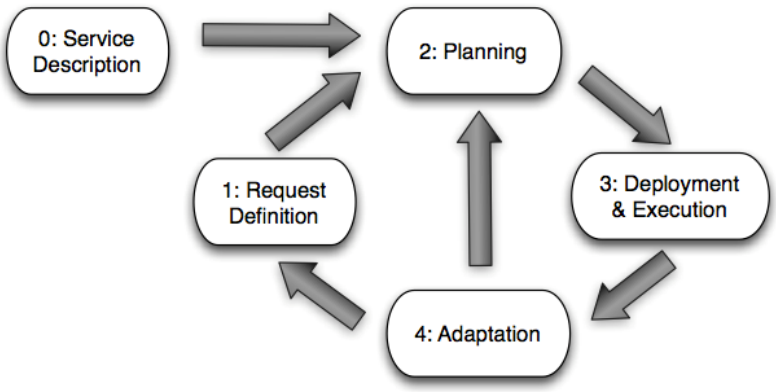


Figure 2: Life Cycle of Event Services

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 specifies those requirements in an event service request.

2. **Planning:** an agent receives a consumer's request and matches 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.

We consider an efficient and effective management of the event service life cycle has the following three basic requirements:

- **User-centric Event Request Definition**: the event requests should reflect each individual user's requirements or constraints on both *Functional Properties* (FP) and *Non-Functional Properties* (NFP) of complex events. Users should be able to specify different events they are interested in by specifying FP e.g. event type and pattern. Additional to FP, it is very likely that different users may have different set of preferences for the NFPs: some may ask for accurate results while others may ask for more timely notifications, etc. The implemented event services should be capable to tackle these requirements and constraints.
- **Automatic Event Service Planning**: the service planning activity should be able to automatically discover and compose CESs according to users' functional and non-functional requirements. Planning based on the functional aspects requires comparing the semantic equivalence of event patterns, while planning based on the non-functional aspects requires calculating and comparing the composition plans with regard to the Quality-of-Service (QoS) parameters. To fully benefit from automatic implementation and enable an on-demand event service implementation, the automatic planning should be efficient to be carried out at run-time.
- **Automatic Event Service Implementation**: the deployment of the composition plans should also be automatic to facilitate automatic execution. 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-compose and re-deploy composition plans, to adapt to changes. The adaptation process should be efficient to minimise information loss and maximise the performance of the event services over time.

3. State of the Art

We argue that current event based systems cannot fully satisfy the requirements listed above. In the following we discuss these limitations with regard to these requirements with more details.

3.1 User-centric Request Definition

In event processing, requested complex events are usually defined by event patterns. An event pattern describes how a set of member events is correlated and contributes to the detection of a complex event, which is the functional specification of a complex event. Existing event processing engines (e.g., Rapide[2], EPL[3]) and CEP engines (e.g., StreamDrill[4], Esper[5] and StreamBase[6]) do not specify the constraints on NFPs. To fully support customisation, the ability of describing constraints on NFPs for both complex and primitive events is needed. Supporting NFP has recently gain some research interests in *Data Stream Management Systems* (DSMS) and *Event Broker Networks* (EBN), and some initial results have been presented, for example, discussion over usage *Software Defined Network* (SDN) concepts to improve the performance of EBNs is presented in [1], [2]. 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 have the ability to describe complex events with patterns.

3.2 Event Service Planning based on Functional Aspects

Facilitating automatic service planning is an important contribution of introducing semantics to service descriptions [6]. State-of-the-art *Semantic Web Service* (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. In IOPE based 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.

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. The precondition of a complex event may be described by incorporating temporal logics and specifying comprehensive rules for temporal reasoning, as in [7]. 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 we declare the creation of the complex event types as effects, it is not practical to use the type information for matching effects and preconditions because the type information alone cannot identify the semantics of a complex event. Therefore, a pattern-based event service discovery and composition is necessary for automatic event service planning.

3.3 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 report, we consider

[2] RAPIDE: http://www.complexevents.com/rapide/, last retrieved: Jan, 2015.
[3] Event Processing Language: https://docs.oracle.com/cd/E1315701/wlevs/docs30/eplguide/overview.html, last retrieved: Jan, 2015.
[4] StreamDrill: https://streamdrill.com/, last retrieved: Jan, 2015.
[5] Esper engine: http://esper.codehaus.org/, last retrieved: Jan, 2015.
[6] StreamBase CEP engine: http://www.streambase.com/wp-content/, last retrieved: Jan, 2015.

only non-functional requirements regarding the QoS parameters. 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-optimized composition plans efficiently.

3.3.1 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 [8],[9],[10]. Existing works have covered a broad range of QoS parameters and many have used a utility function based on *Simple Additive Weighting* (SAW) [11] 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 have a different QoS aggregation schema. For example, 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 are different to those based on workflow patterns (as in [9]).

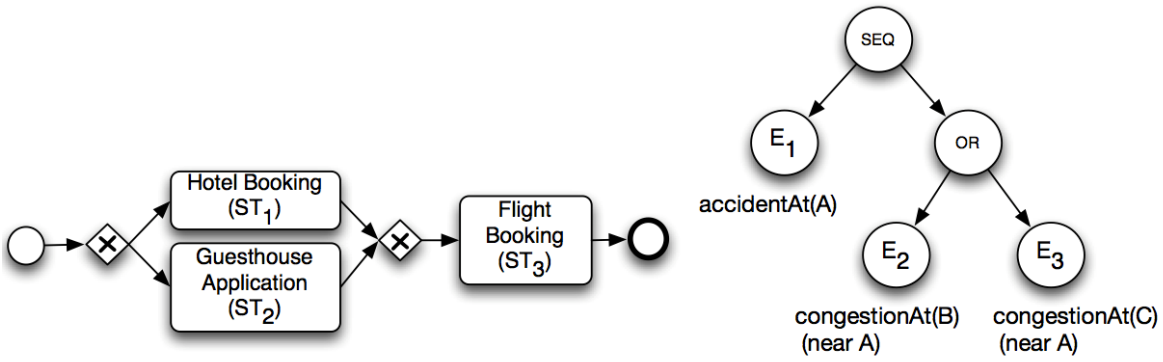


Figure 3: Service Workflow and Event Pattern

Consider a simple service workflow for travel planning as shown in Figure 3 (left) and a severe traffic accident event pattern in Figure 3 (right). 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 by C_2) is given by $C_2 = cost(E_1)+cost(E_2)$, because the energy consumption of event detection tasks starts when subscriptions are made, and the cost exists even when no events are detected, same applies for other computational resources. Also, the latency of the work flow (denoted by 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 our knowledge currently there are no suitable QoS aggregation schema for CESs.

Another approach for QoS-aggregation and constraints resolution is to use *Answer Set Programming* (ASP) systems. Existing ASP solutions are divided into two categories: grounding systems (which are generating program's ground instantiation) as DLV, GRINGO, lparse, XASP, and solving systems (claspD, DLV, GnT, ASSAT, CMODELS, Smodels, SUP)[7]. An example of combining ASP belonging to both categories to produce better results has been presented in ASP systems competition[8], where Potascco team[9] achieved better results by using GRINGO-4 for grounding and CLAPS as back-end. Similarly, ASP based QoS-aggregation and stream selection solutions can be implemented using both categories of ASP systems, e.g. using Clingo-4 system, which combines GRINGO and CLAPS into a single ASP based solution.

3.3.2 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., [12],[8], [13]) and Genetic Algorithm (GA) (e.g., [14],[15],[16],[17]) based solutions. In [12] the limitation of local optimisation and the necessity of global planning are elaborated. The authors also explain why the brutal 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 a SAW based utility function to determine the desirability of an execution plan. This approach is extended in [13] with more heuristics to promote the efficiency. In [8] 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.

GA based solutions are therefore proposed to address the issues, e.g., [14],[15],[16],[17]. However, the above GA based approaches can only cater for IOPE based service compositions. CES composition is pattern-based, as described earlier. Therefore, novel algorithms based on genetic evolution, including suitable genetic encoding for event patterns, crossover and mutation operations are of utmost necessity.

3.4 Automatic Event Service Implementation

Conventional web service composition can be implemented by various workflow/process engines, e.g., YAWL [18], BPEL[10], BPMN 2.0[11] 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

[7] https://en.wikipedia.org/wiki/Answer_set_programming#Comparison_of_implementations
[8] Fifth Answer Set Programming Competition (ASPCOMP 2014)
https://www.mat.unical.it/aspcomp2014/#Competition_Results_and_Data
[9] http://potassco.sourceforge.net
[10] WS-BPEL, version 2.0: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, last retrieved: March, 2015.
[11] Business Process Modeling Notation Specification, version 2.0: http://www.omg.org/spec/BPMN/2.0/

problematic, because there can be semantic or syntactical mismatches between service inputs and outputs. In such cases, service mediation might be needed [19]. Semantic web service composition also provides solutions for mismatches in the messages used in service compositions, e.g., in BPEL4SWS [20], the message *lifting* and *lowering* mechanisms supported in SAWSDL [21] are used to transform XML data representations to RDF graphs and vice versa.

Implementing CES composition is different from conventional service composition. Composition plans for CESs are descriptive event/stream queries rather than imperative workflows, and the actual workflow of query operators, i.e., query execution plans, are determined at run-time by event/stream processing engines. 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 set of operators, query semantics and syntax, e.g., Etalis [22], CQELS [23] and C-SPARQL [24]. Currently there is no standardised event/stream query languages, and because of the differences of the supported operators, language syntax and semantics, an automatic CES implementation requires a query transformation technique to deploy a federated CES queries over different event/stream engines.

4. Information Model for Stream Federation: Event Service Ontology and Event Pattern Semantics

The functionality of a CES is determined by the semantics of the complex events delivered by the service. In this section, a semantic information model (ontology) for describing (complex) event services and requests is presented and the abstract syntax and semantics of event patterns are presented.

4.1 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. We consider atomic events like sensor observations delivered in event streams as primitive events, 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. Current approaches have discussed extensively on PES modelling using traditional service description frameworks, e.g., sensor services discussed in [25],[26],[27],[28],[29],[30]. However, CES description, discovery and composition remain largely unexplored. We developed a *Complex Event Service Ontology* (CESO) 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),

CESO is an extension of OWL-S, which is a standard semantic web service ontology; it imports concepts from SSN to describe sensor capabilities for PESs when those PESs are provided by sensors.

CESO is used in combination with the Stream Annotation Ontology[12] (SAO) and the Stream Quality Ontology[13] (SQO). SAO is used for semantic annotation of stream data and QoS/QoI information about the streams is annotated using SQO. The relations between CESO, SAO and SQO are depicted in Figure 4. In the following we introduce the basic concepts in CESO.

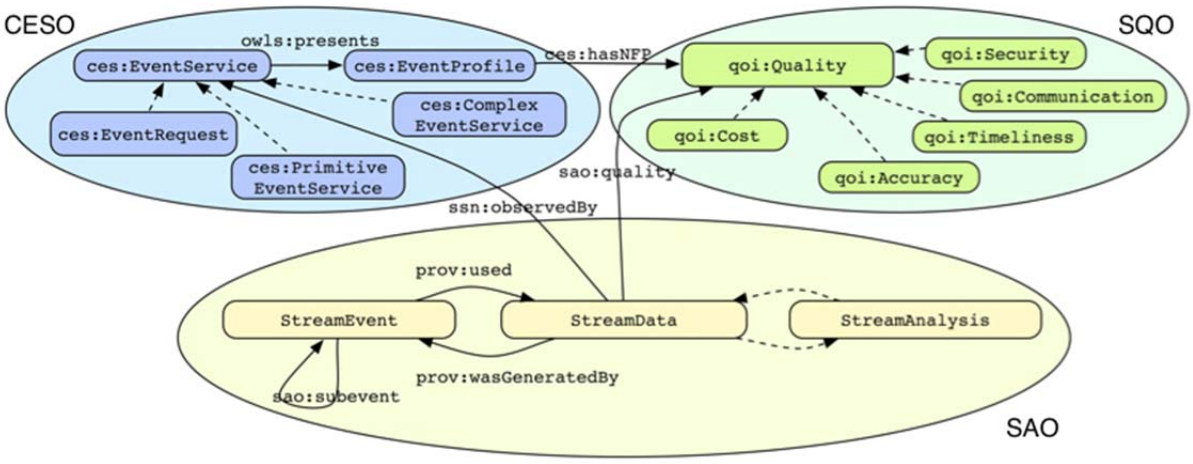


Figure 4: Overview of Citypulse Information Model

4.1.1 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, message formats etc. 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. Figure 5 illustrates the overview of the CESO.

[12] Stream Annotation Ontology: http://iot.ee.surrey.ac.uk/citypulse/ontologies/sao/sao, last accessed: March, 2015.
[13] Stream Quality Ontology: https://mobcom.ecs.hs-osnabrueck.de/cpquality/, last accessed: March,2015.

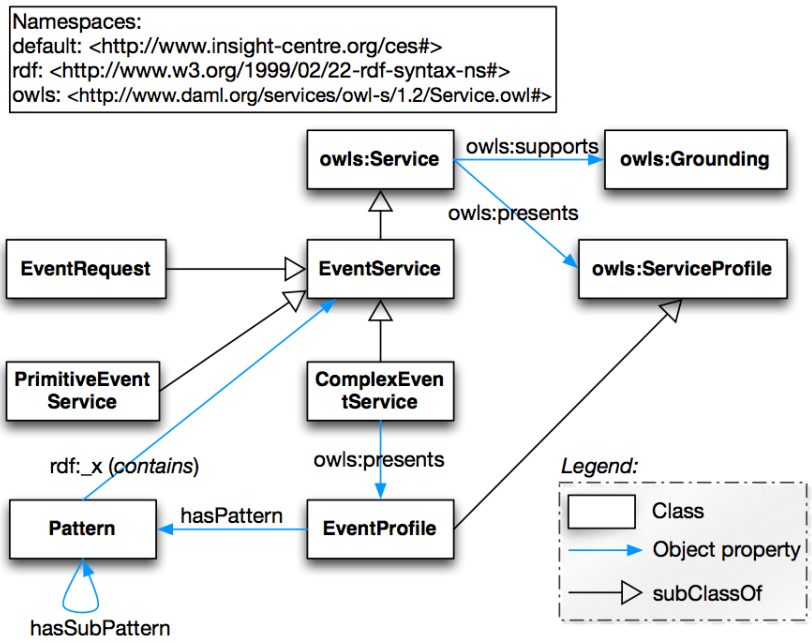


Figure 5: Overview of the CES Ontology

4.1.2 Event Profile

An *EventProfile* describes a type of event with a *Pattern* and *Non-Functional Properties* (NFP). A *Pattern* describes 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 a 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 6 shows the ontology for describing event profiles.

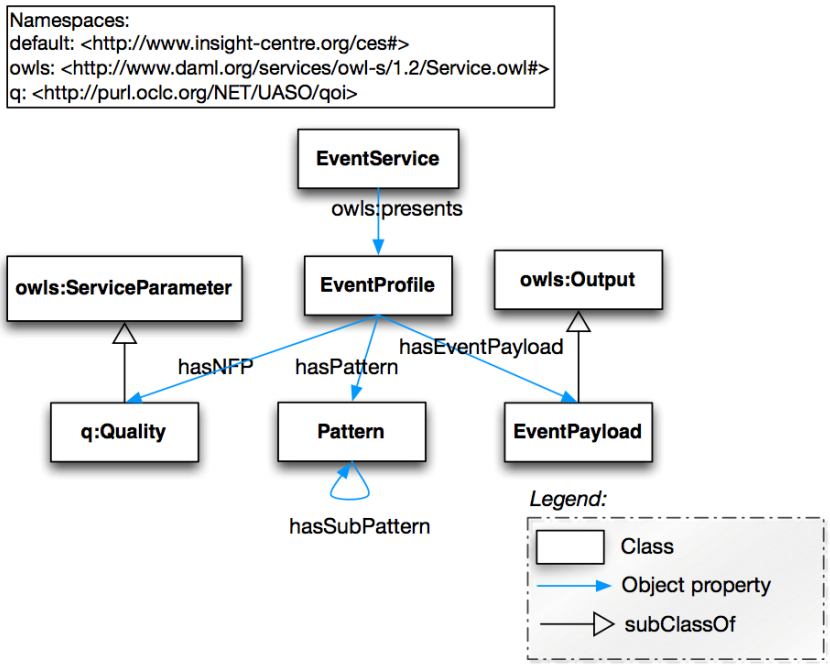


Figure 6: Event Profile in CESO

4.1.3 Event Pattern

We consider that the temporal relationships captured by an *EventPattern* have three basic types: sequence, parallel conjunction and parallel alternation. 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 we define three types of patterns 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 repetitions. A repetition can be an overlapping or non-overlapping, 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. It is useful for analysing the causal relations between events delivered in ESN. Figure 7 reveals further details of the event pattern model.

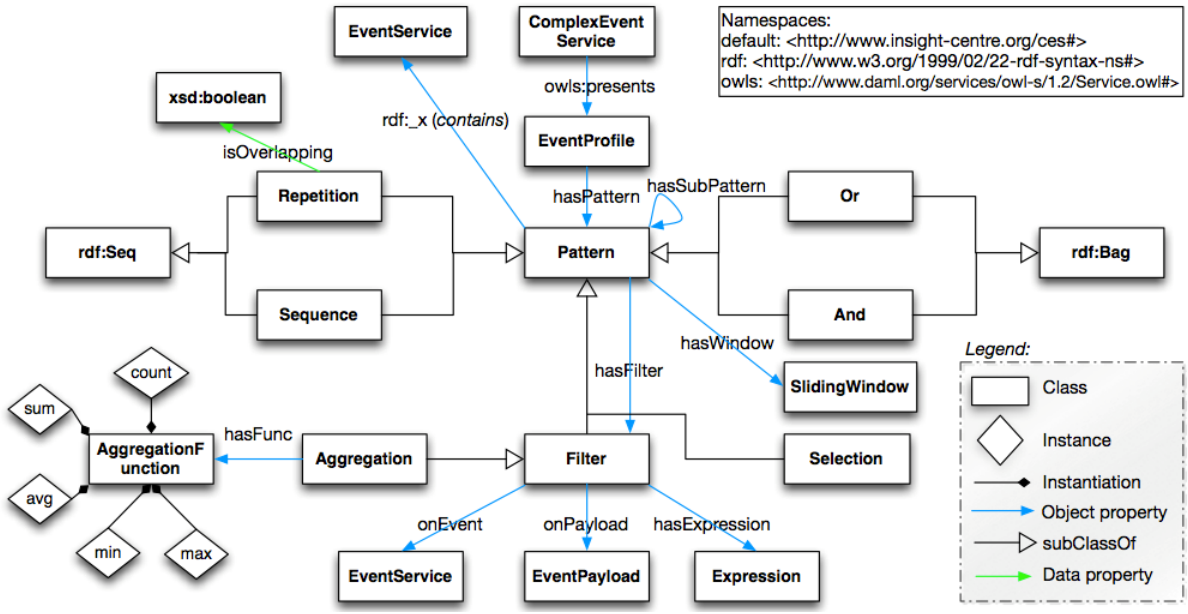


Figure 7: Event Pattern in CESO

The pattern types modelled in CESO cover the entire basic event pattern types except for exclusive-disjunction and negation, spatial and spatiotemporal patterns, and aggregations with attribute-based patterns. Indeed, the focus of this report is not providing and developing yet another comprehensive and highly expressive event pattern language, instead we aim to prove 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.

4.1.4 Event Request

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 users' preferences on QoS metrics: higher weight indicates the user has higher preference of that particular QoS metric. Figure 8 shows the ontology for describing event requests.

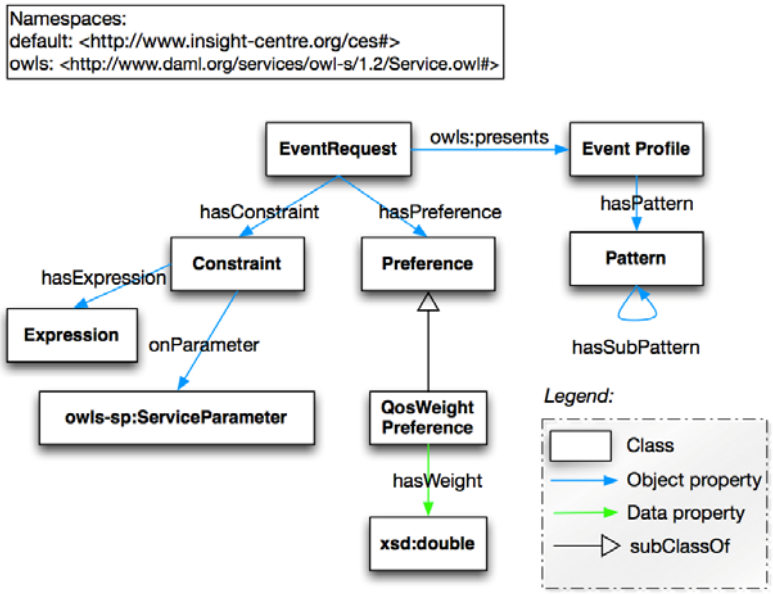


Figure 8: Event Request in CESO

4.2 Syntax and Semantics for Event Patterns

We extend the Business Event Modelling Notations (BEMN) [5] to describe the syntax of event patterns used in CESO. The formal semantics of the event patterns considered in CESO is also described. We call the event pattern modelling language with the revised syntax and semantics BEMN+.

4.2.1 Abstract Syntax of Event Patterns in BEMN+

Using the CESO, an event service provider can describe event services and store these service descriptions in a service repository; an event service consumer can in turn, formulate a event service query to specify his requirement on event services. In the following the abstract syntax of event patterns described in the CESO 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 *E* are hosted, *type* is the domain specific event service type, *ep* is the event pattern of the events delivered by *E,* and *D* is the data payload as a set of event properties sets, e.g., timestamps, event identifier, message contents,etc.

An *Event Pattern* describes the detailed semantics of a complex event. It is a tuple *ep=(ℰ, OP, Gr, R, Pr, Sel, F, Pol, W)*, where:

- $\mathcal{E}$ is a set of member event declarations, i.e., member event service descriptions, involved in ep;
- for a member event declaration $E' \in \mathcal{E}$ we denote D' as 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), $r \in \mathbb{N}^+$ 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, $gt: Gr \longrightarrow \{n, r_o, r_n, agg\}$ 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, and 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*, and also, the occurrence of n_1 should happen before the occurrence of n_2;
- $Sel : \bigcup_{E' \in \mathcal{E}} E'.D' \longrightarrow D$ 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., $: \bigcup_{E' \in \mathcal{E}} E'.D'$) . 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*. $F_{gr}: F \longrightarrow Gr$ 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 a 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) = (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. The provenance relation, operators and event declarations can be used in a 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 representations of ESTs omit many elements in BEMN+, such as groupings, time window, event payloads and selections. Because of the brevity of ESTs, throughout the report we use ESTs when illustrating event patterns. We postpone the detailed definition and use for ESTs in Section 5.

4.2.2 Formal Semantics of Event Pattern

In order to ensure correctness in complex event stream integration and execution, we need to define the formal semantics of the event patterns specified in CESO. In this section we lay down the formal semantics. We first present a meta-model for complex event semantics in [39]. In this document, we use this meta-model to compare 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+.

Meta-model of Event Semantics

An **Event Duration** can be categorised into instantaneous or interval-based. The fundamental difference between instantaneous and interval-based events is whether one or two (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 Event Instance Sequences (EISs), including binary operators: {*Sequence* (;), *Simultaneous* (==), *Conjunction* (∧), *Disjunction* (∨)}, unary operator *Negation* (¬)} and n-ary operator *Repetition*.

For two event types E_1, E_2, $;(E_1, E_2)$ indicates that the timestamps of event instances of type E_1 are older than the timestamps of event instances of type E_2[14]; $==(E_1, E_2)$ 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 regardless 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 $E_1, \ldots, E_n$, $(;(E_1,\ldots,E_n))^r$ indicates that the sequence of instances of $E_1,\ldots,E_n$ must repeat for r times. Repetitions have two modes: *overlapping* and *non-overlapping*, denoted by $\wedge (;(E_1,\ldots,E_n))^r$ and $;(;(E_1,\ldots,E_n))^r$, 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 j th instance of event type E_i). It is evident that 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_3^1, e_2^1) 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.

[14] When considering overlaps for interval-based events the sequence operator can have more variants e.g.: meets, finishes and participates etc. see [38]

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_i^j is used to trigger E_a, all e_i^k in the EIS before the terminator (i.e., last event instance in EIS triggering E_a) are removed.

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 are derived from the complex event meta-model presented earlier. In [34] a thorough survey has been conducted on existing Information Flow Processing (IFP) systems, however it does not describe the features of recent semantic stream processing systems. A survey on stream reasoning engines can be found in [35]. In Table 1 we compare the event semantics used in existing semantic CEP systems or DSMSs, including ETALIS, C-SPARQL, CQELS, BEMN and BEMN+. Recall that the event pattern definition language used in BEMN+ is designed to be a user-friendly, high-level language while sufficiently expressive to capture most of the event/query semantics in the existing IFP systems. In the following we elaborate on the semantics supported by these systems in each dimension.

Table 1: Comparison of Event Semantics in Existing Approaches

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

- **Event Duration** All investigated approaches support usage of 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 and end timestamp. C-SPARQL partially supports intervals for complex events, i.e., events consist of multiple triples with different timestamps. To capture the interval for such complex events in C-SPARQL one must use the *f:timestamp* function provided by C-SPARQL 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 in-directly supported by C-SPARQL and BEMN+ by comparing timestamps of events and triples. The *Conjunction* and *Disjunction* operators are supported by all investigated approaches. *Negation* is directly supported by BEMN using *Inhibition* and in-directly supported by ETALIS, C-SPARQL and CQELS using the combination of *LeftJoin* operator and *bound* filters. Currently BEMN+ do not support negations as it will introduce complexity in complex event 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 C-SPARQL and CQELS since they allow triple-size-based windows. However, one must assume 1) events in a triple stream consist of a same number of triples and 2) all triples are synchronised in the stream and never lost in communication to use triple-based windows of size $n \times m$ to keep m event instances in the window. 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 the relevant ones. All approaches support context conditions on *data* using filters.
- **Event Instance Selection** ETALIS, C-SPARQL 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. In BEMN+ we support both *cumulative* and *last* selection, since we want to be compatible with existing stream reasoning engines which extend SPARQL semantics and we do not want to neglect the fact that in some traditional CEP systems, a minimum event instance selection policy is desired due to performance concerns (details in [34]) and we consider the latest events usually to be more important.
- **Event Instance Consumption** Existing semantic IFP engines like ETALIS, C-SPARQL 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 mode by configuring the event type definitions and subscription scopes. In BEMN+, we designed a decentralised system in which queries are evaluated by different event engines on distributed servers and the messages are delivered via publish-subscribe systems, therefore we only support a *shared* event instance consumption.

4.3 Summary

In this section, the information model used to define and describe complex events and CESs are elaborated. A Complex Event Service Ontology (CESO) is introduced to describe CESs. CESO is an extension of the standardized 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 is annotated as *EventPattern* in CESO. The semantics is aligned to an extended version of the Business Event Modelling Notation, called BEMN+. The syntax and semantics of BEMN+ are introduced. The comparisons between the BEMN+ semantics and other RDF stream processing engines are elaborated. Part of the research in this section is published at [43,44,45]

5. Scalable Federation Plan Creation: Pattern-based Event Service Composition

CEP is a suitable technique to detect high-level business events expressed with event patterns. Despite the extensive research on CEP, providing CEP applications as reusable services that allow the composition of CESs 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.

Providing complex events as services accelerates the CEP system implementation, because it allows the event consumers and providers to be loosely coupled in the system. 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.

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 realized?
2. How to determine if two event patterns are functionally equivalent (i.e., produce the same complex event notifications), provided that different event patterns might have identical meanings?
3. How to choose optimal event service composition plans that consumes 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?

Section 3 has already answered the first question and in this section, we try to address the remaining questions. The remainder of the section is organized as follows; Section 5.1 answers the second question by presenting the operations and algorithms to reduce event patterns into canonical forms so that they can be compared. Section 5.2 answers the third and fourth questions, it first provides the definition of *Network Optimized* event service compositions based on Estimated (network) Traffic Demand (ETD) of compositions, and provide a way to calculate the ETD. Then it presents two composition algorithms creating structurally optimized event service compositions: a slow algorithm based on event substitution and a fast algorithm based on the reusability index of event patterns. Section 5.3 demonstrates the performance of the proposed algorithms with prototype experiments.

5.1 Canonical Event Pattern

In section 4 the syntactical and semantic definition of an event pattern used in this report are presented. Recall that an event pattern is defined as a tuple consists 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. 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 *E1 :=∧(E2,E3,E4)* is semantically equivalent to another event type *pattern E5 :=∧(E6,E4)* where *E6 :=∧(E2,E3)*. 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.

5.1.1 Definitions of Event Syntax Tree

An event syntax tree describes an event pattern with a tree. More formally, given an event pattern *ep=(ℰ, OP, Gr, R, Pr, Sel, F,Pol,W)*, a syntax tree *T(v)=(V,R,F)* is generated from *ep*, where *V= (OP ∪ ℰ)* is the set of vertices representing operators and member events, *v∈V* is the root node, i.e., *∄(v',v) ∈ R*, typically the root of an EST is an operator, and *R* is the set of directed edges representing the provenance relation. If *(v1,v2) ∈ R*, *v2* is called a child node of *v1, if (v1,v2) ∈ R**, *v2* is called a descendant of *v1*, 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 *(v1,v2) ∈Pr*, then *v1* is placed to the left of *v2*. Each node in *V* is labelled 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 labelled by the payload (*F(v)=∅* 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 ⟶ 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.

5.1.2 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. By checking recursively the event pattern definitions of the leaves, it is trivial to build complete event patterns. A complete event pattern provides a complete information on the logical rules specified for a complex event. The function deriving the complete event pattern is denoted $f_{complete,}$ implemented as Listing 1. The pattern completion function does not alter the semantics of event patterns.

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
    $p_c \leftarrow (\mathcal{E}'', OP'', Gr'', R'', Pr'', Sel'', F'', Pol'')$
    $\mathcal{E}'' \leftarrow (\mathcal{E} \cup \mathcal{E}')\backslash v$
    $OP'' \leftarrow OP \cup OP''$
    $Gr'' \leftarrow$ REPLACEELEMENT$(v, (\mathcal{E}' \cup OP'), Gr) \cup Gr'$
    $R'' \leftarrow$ REPLACEELEMENT$(v, (root(est(p'))), R) \cup R'$
    $Pr'' \leftarrow$ REPLACEELEMENT$(v, (root(est(p'))), Pr) \cup Pr'$
    $Sel'' \leftarrow (Sel \cup \bigcup_{sel' \in Sel'} sel' \mid range(sel') \in dom(Sel(v)))\backslash Sel(v)$
    $F'' \leftarrow$ REPLACEELEMENT$(v, (root(est(p'))), F) \cup F'$
    $Pol'' \leftarrow (Pol \cup Pol')\backslash Pol(v)$
    $W'' \leftarrow (W \cup W')\backslash W(v)$
return p_c
  end if
end procedure
Require: Node to be replaced v, replacement(s) V, modifying elements O_m in p.
Ensure: Modified elements O_m.
procedure REPLACEELEMENT(v, V, O_m)
  for $o \in O_m$ **do**
    if o is a set, relation or function that involves v **then**
      replace the element v in o with the element(s) V
    end if
  end for
return O_m
end procedure

Listing 1: Algorithm for Creating a Complete Event Pattern

5.1.3 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.

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. Notice that the sequential and repetition merge exemplified in Figure 11 does 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. Different rules apply when performing lift and merge operations on different types of nodes. Examples of lifting and merging operations are shown in Figure 11. 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.

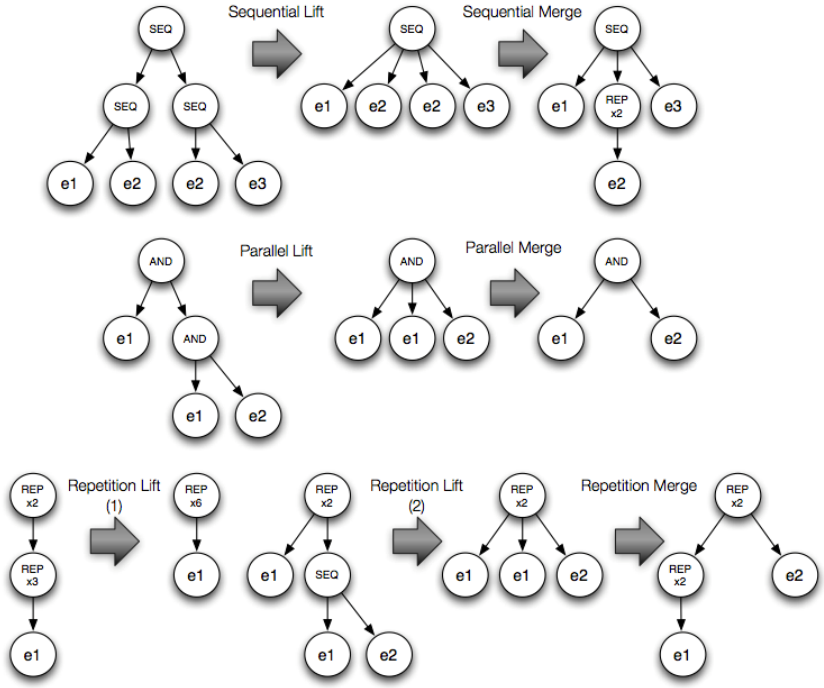


Figure 9: Examples of Event Pattern Reduction

- **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 $lift_s: (V_{seq}, V_{seq}, P) \longrightarrow P$ where P is the set of event patterns and V_{seq} is the set of sequence operators. Given $p=(\mathcal{E}, OP, Gr, R, Pr, Sel, F, Pol, W)$, $p'=(\mathcal{E}, OP', Gr, R', Pr', Sel, F', Pol, W)$,

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

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 Direct Sub-Tree[15] (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 differences of two child nodes n_1 ,n_2 are the filters attached, and each filter in v_1 is covered[16] 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.

5.1.4 Syntax Tree Reduction Algorithm

The algorithm to create irreducible syntax trees is shown in Listing 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 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).

[15] A DST of a tree *T* is the sub-tree whose root is a child of the root of *T*

[16] f1 covers $f_2 \Leftrightarrow P(f1) \supseteq P(f2)$, where P(f1),P(f2) are the notifications produced by filters f1,f2, respectively.

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 ← getRoot(est)
7:      LIFTTREE(root, est)
8:      for height − 1 → rootDepth → 0 do
9:          nodesToMerge ← getNodesByDepth(est, rootDepth)
10:         for node ∈ 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
```

Listing 2: Algorithm to Create an Irreducible Event Pattern

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 direct sub-trees of a certain node. The *liftTree* method in Line 7 and 13 carries out the lifting operations on a sub-tree. We denote the reduction function implemented in Algorithm 2 f_{reduce} and the canonical event pattern is therefore given by $f_{canonical}=f_{complete}\bullet f_{reduce}$. Two event patterns are semantically equivalent if and only if their canonical patterns are isomorphic, i.e.,

$$p \doteq p' \iff isomorphic(f_{canonical}(p), f_{canonical}(p'))$$

5.2 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 that 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 CES as well.

When a CES is reused, an appropriate portion (sub-tree or part of sub-tree) of the query EST is replaced with the event declaration of the CES, which transforms the portion into an event declaration node (a leaf node) with a complex event type and a service location. When all the leaf 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. An example of a composition plan created with a

query and a set of event services in shown in Figure 12. 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.

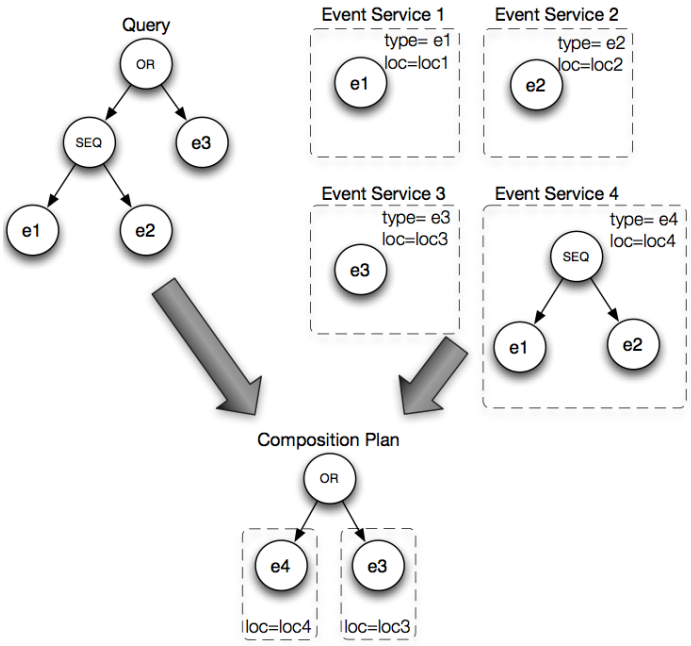


Figure 10: Example of Event Service Composition Plan

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;
2. all events delivered by event services are error-free, synchronized and complete;
3. all events have similar payload size;
4. in general, complex events are less frequently detected than their member events.[17]

The first two assumptions draw the scope for our 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. Also, data quality or quality-of-service issues are not considered in this section. The third and fourth assumptions allow us to propose a heuristic for achieving the goal of reusing event patterns: to minimize traffic, a CES composition should contain as few as possible the member event services, meanwhile, it should choose more coarse-grained member events. An event composition plan is said to be *network optimized* when it demands the least amount of data traffic over the network. Also, consuming least amount of events will reduce computation resource required for a query.

In the following, the method for determining whether a composition plan is network optimized is given. Then, a slow algorithm, which derives optimized event compositions, is presented. It traverses

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

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* is developed.

5.2.1 Network Optimization based on Traffic Estimation

Intuitively, to determine whether a composition plan is network optimized, 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 demands for 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 CESs, 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 $T_c(v) = (V,E) = est(f_{complete}(ep))$ where $v \in V$ is the root node, $v(n)$ denotes the frequency estimation of the member event represented by the sub tree $T_c(n) \subseteq T_c(v)$. Obviously, $v(v)$ is the frequency estimation of event described by E. The traffic demand of ep is denoted $Traffic(ep) = \sum v(n)$ 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 $v(n)$ and $v(m)$ is given by

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

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 allows us to calculate the maximum estimated frequencies for a set of member event services *(max{v(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 optimized composition plans can be selected. However there is a limitation of the given equation: 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.

5.2.2 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 following: 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.

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

Algorithm 3 Creates optimal composition plans.

Require: Query Tree: ST, Candidate Trees: $cand$ Query Root: $root$.

```
1:  procedure COMPOSE(ST, cand, root)
2:      matchingED ← getSubstitutes(ST, cand)
3:      if matchingED ≠ ∅ then
4:          replacePattern(ST, matchingED, root)
5:      else if root.type = REPETITION then
6:          hasReplacement ← false
7:          for f ∈ getFactors(root.r) ∪ 1, r > f >= 1 do
8:              newRoot ← createRepetition(root.r/f)
9:              root.setCardinality(f)
10:             matching ← getSubstitute(ST, cand)
11:             if matching ≠ ∅ then
12:                 ST ← ST.addRoot(newRoot)
13:                 replacePattern(ST, matching, root)
14:                 hasReplacement ← true
15:                 break
16:             end if
17:         end for
18:         if hasReplacement = false then
19:             for dst ∈ getDSTs(root, ST) do
20:                 COMPOSE(ST, cand, dst.getRoot())
21:             end for
22:         end if
23:     else
24:         hasMatchedPartition ← ANALYZEPARTITION(ST, cand, root)
25:         if hasMatchedPartition = false then
26:             for dst ∈ getDSTs(root, ST) do
27:                 COMPOSE(ST, cand, dst.getRoot())
28:             end for
29:         end if
30:     end if
31: return results ← getOptimal(results)
32: end procedure
33:
```

Listing 3: Algorithm for Event Composition based on Top-down Substitution

The *getSubstitutes* method in line 2 is a key operation in Listing 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 different non-overlapping partitions (using the *getDSTPs* method in line 34, example of the operation is illustrated in Figure 13) 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 41) and corresponding replacements are made. If no partitions have complete substitutions, the composition algorithm is invoked on each DSTs of the root node.

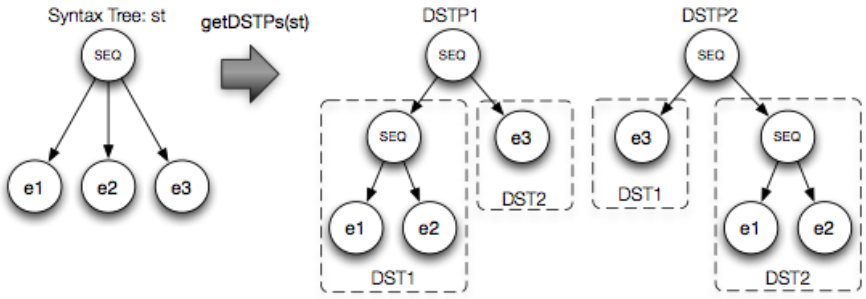


Figure 11: Example of Creating DST Combinations

The algorithm in Listing 3 guarantees the creation of network optimized composition plans because all sub-trees and 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 candidate. The *getSubstitutes* operation needs to be executed for every sub-tree and sub-tree partition for the query, comparing with every existing candidate, i.e., the time complexity of the composition algorithm w.r.t. *getSubstitutes* is exponential. Clearly, the algorithm cannot scale and a much faster way to compose CESs is needed.

5.2.3 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.

Reusability of event patterns

Informally, An event pattern is *reusable* to another (denoted $R(ep_1,ep_2)$), 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 *ep1* is directly reusable to *ep2*, denoted $R_d(ep_1,ep_2)$, *iff* ep_1 is a direct sub-event-pattern of ep_2.

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; $F_{multiply}$: $T\times \mathbb{N}^+ \longrightarrow T$ 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.

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 T is a set of nodes (canonical event patterns) and $R\subset P\times P$ is a set of edges (reusable relations) connecting nodes. 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 this definition, if an ERH is built for the three event patterns in Figure 14, the edge at the top-right is ignored. The nodes do not reuse any other nodes are called roots in the ERH, the nodes cannot be reused by other nodes are leaves.

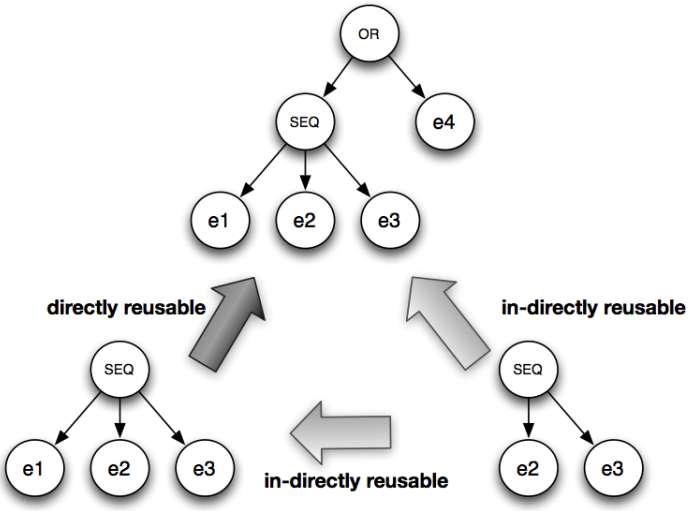


Figure 12: Example of Event Pattern Reusability

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 Listing 4.

Algorithm 4 Insert an canonical pattern *ep* to reusability forest *erf*.

Require: Canonical Pattern *ep*, ERF *erf*.

```
1:  procedure INSERT(ep, erf)
2:      roots ← getRoots(erf), leaves ← getLeaves(erf)
3:      erf.addNode(ep)
4:      parents ← getReusable(roots, ep)
5:      drawEdges(parentse, p)
6:      childNodes ← getChildNodes(parents, erf)
7:      parents ∪ 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
```

Listing 4: Algorithm for Inserting a Canonical Pattern into an ERF

The above algorithm takes the canonical event pattern *ep* and an event reusability forest as inputs.

As the first step, it finds all p∈ 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 $p' \trianglelefteq ep$ is found, the algorithm terminates. This step is omitted in Listing 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 the algorithm in Listing 4 to prune the irrelevant parts of the hierarchy and reduce the number of comparisons required.

Event Composition Algorithm with ERF

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 generalization 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 accomplishes this task is given in Listing 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 ← event declarations of the isomorphic node return getOptimal(EDs)
5:      end if
6:      parents ← getParents(Q, erf)
7:      for p ∈ 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
```

Listing 5: algorithm for event composition based on ERF

The *directlyReplace* operation in line 9 replaces 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 the definition on the in-directly reusable relations, 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 $O(n (2^d)^h)$ graph isomorphism checks to find proper substitutes, The algorithm in Listing 5 only needs $O(m)$ sub-graph isomorphism checkings to find reusable components, while $m \leq n$ if all input parameters are the same. The efficiency of event composition is greatly improved by the algorithm in Listing 5.

5.3 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 2), event reusability forest construction (Algorithm 4) and event compositions (Algorithm 3 and 5). In the following, the general experiment settings are described, then, the detailed settings for each experiment its results are elaborated.

5.3.1 General Experiment Settings

All the experiments are carried out on a Macbook Pro with a 2.53 GHz duo core cpu and 4 GB 1067 MHz memory. Algorithms are developed using Java. The Java Virtual Machine is configured with a minimal heap size of 64 MB and a maximal heap size of 256 MB. In order to have more accurate results, all results are averaged from 5 to 10 test iterations for each test setting. To ensure that the test results are unbiased, an Event Pattern Generator (EPG) is developed to create random event patterns/queries. The EPG will choose 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. To ensure the random event pattern creation stops at some point, also to have some control on the size of patterns (number of nodes in the query tree) created, EPG receives two parameters: height and degree, specifying the maximal tree height and degree of the output. EPG is used to create simulated datasets in our experiments.

5.3.2 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 groups the execution time by the size of patterns. In this way the minimal, maximum and average reduction time are derived for event patterns of different sizes. Since we observed that the pattern reduction time might vary even for patterns with the same size, groups with small sample spaces, i.e., containing less than 10 patterns, are excluded from the results. Figure 15 shows the results of the experiment.

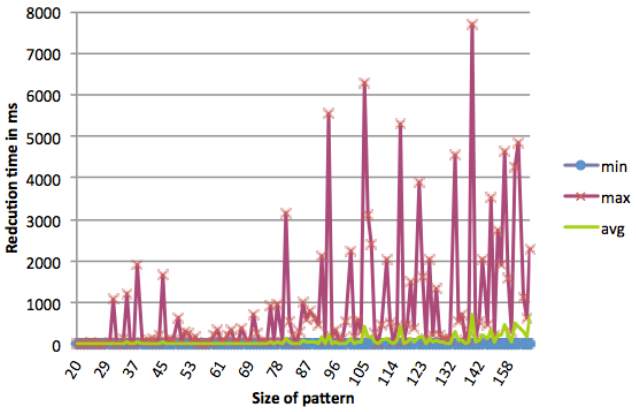


Figure 13: Execution Time of Query Reduction

The results in Figure 15 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% event patterns took more than one second to be reduced, and in extreme cases it goes up to 8 seconds. After investigating the data set 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 transformed into sequence operators during merge operations, they may significantly increase the total size of the pattern. As a result the 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.

5.3.3 Performance of Event Reusability Forest Construction

The feasibility and efficiency of ERF construction is evaluated by measuring the construction 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 and we observe the time needed for the construction. Figure 16 shows the results of the experiment.

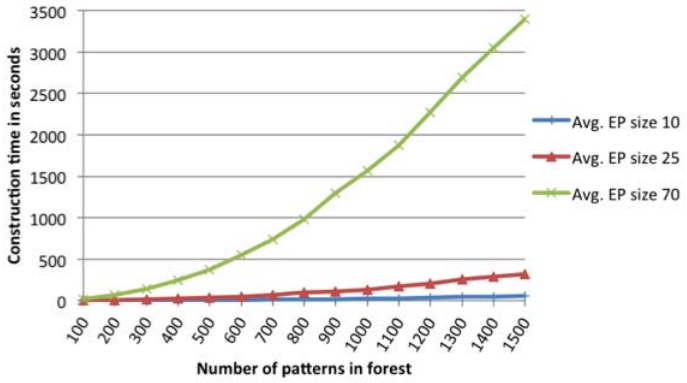


Figure 14: Execution Time of ERF Construction

In the results from Figure 16, 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 1500 event patterns, it took 58 seconds to complete the construction. Similarly, the red line in middle represents the set of event patterns with 25 nodes in average and completes the construction in 323 seconds. Finally, the green line represents the set of event patterns with 70 nodes in average and took 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. 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.

5.3.4 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 un-indexed algorithms and their results are compared. More specifically, the EPG is used to create 500 and 1000 event patterns with an average pattern size of 25 nodes as candidates. Then, the EPG is used again to create 3 sets of event patterns as queries. There are 100 event patterns in each query set and their average pattern sizes are 10, 14 and 25 nodes. Figure 17 shows the time needed for each query set against each candidate set using indexed or un-indexed composition algorithms.

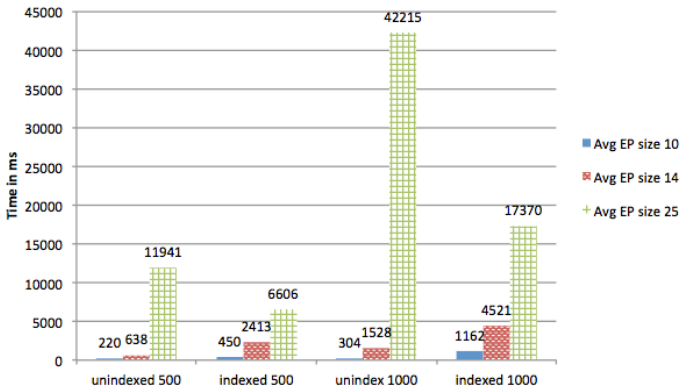


Figure 15: Execution Time of Composition (Indexed vs. Un-indexed)

The results indicate that for small event patterns, the un-indexed approach outperforms the indexed one, but for large event patterns the indexed algorithm is much faster. This is aligned with our discussions in Section 5: reusability checking is more complicated than graph isomorphism, but the number of sub-graphs 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 un-indexed algorithm terminates before finish due to insufficient memory.

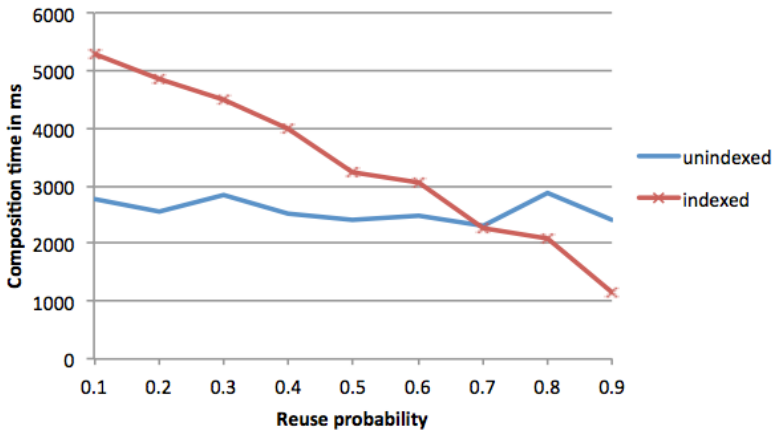


Figure 16: Impact of Reuse Probability on Indexed Composition

Another factor causing the slow 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, and the EPG reuses existing patterns with the assigned rate. Figure 18 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 un-indexed one when the probability of reuse is above 70%.

5.4 Summary

In this chapter, the discovery and composition methods are discussed for CESs described with the event pattern semantics and formats from Section 4. A canonical form of event patterns can be derived by first creating a complete pattern and then remove 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., if they 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 the service repository is large. It is also observable that when more reusable relations can be found in the service repository, the performance of the indexed composition improves significantly. Part of the research in this section is published at [44]

6. Scalable Federation Plan Optimization: Constraint-aware Event Service Composition

In Section 5 we have provided CEP applications as reusable services, where the reusability of those event services is determined by examining complex event patterns and primitive event types. Event service composition based on functional properties was therefore addressed. However, it is still difficult to determine which event service candidates (or service compositions) are the best suit for the users' and applications' non-functional (i.e., quality-of-service) requirements. A sub-optimal service composition may lead to inaccurate event detection, lack of system robustness etc.

In this section, we aim to enable a QoS-aware event service composition and optimization. In order to facilitate this, two issues should be considered: QoS aggregation and composition efficiency. The QoS aggregation for a CES 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. This section addresses both issues by: 1) creating QoS aggregation rules and utility functions to estimate and assess QoS for event service compositions, and 2) enabling efficient event service compositions and optimization with regard to QoS constraints and preferences based on Genetic Algorithms.

6.1 QoS Model and Aggregation Schema

The event QoS attributes describe the non-functional performance of event services (and service compositions). In this section, the following QoS attributes are investigated:

- *Latency (L)* describes the delay in time for 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),
- *Price (P)* describes the monetary cost for an event service,
- *Energy Consumption (Eng)* describes the energy cost for an event service,
- *Bandwidth Consumption (B)* describes the usage of network bandwidth for an event service,
- *Availability (Ava)* describes the possibility of an event service being accessible, it can be numerically represented in percentages, denoted Ava,
- *Completeness (C)* describes the completeness of events delivered by an event service, it can be numerically represented as recall rates in percentages,
- *Accuracy (Acc)* describes the correctness of events delivered by an event service, it can be numerically represented as precision in percentages, and
- *Security (S)* describes the security protocol used by event services numerically represented as integer security levels (higher numerical value indicates higher security levels).

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.

6.1.1 QoS 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 local event patterns evaluated by the event engine and the set of member event services directly involved. Indeed, the QoS performance varies on which services are used to produce the member events and how event operators logically correlate them. For 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. Table 2 summarizes how different QoS parameters of an event service composition are calculated based on the three factors. Note that for a simple 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*.

Table 2: Overall QoS 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	n/a	$C = C_i \times C_c$
Correctness	Acc_i	Acc_c	Acc_e	$Acc = Acc_i \times Acc_c \times Acc_e$
Encryption	S_i	S_c	n/a	$S = \min(S_i, S_c)$

Table 3: QoS Aggregation Rules based on Composition Patterns

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

The *Composition Pattern* is a key factor in aggregating QoS properties for event service compositions. Event patterns are represented as event syntax trees. A step-wise aggregation over ESTs is adopted to aggregate QoS properties. More specifically, we apply aggregation rules iteratively from leaves to roots on ESTs. Aggregation rules for different QoS dimensions can be event operator dependent or independent, i.e., the aggregated QoS on a parent node in an EST may or may not depend on the event operator type of the parent node. Table 3 shows the detailed rules for each quality dimension. In the following we explain the rationale for each rule.

1. **Price**, **Energy Consumption** are operator independent properties. They can be specified in different manners, e.g., price can be charged over subscription time or volume, similar for energy consumption. For simplicity we assume they are specified over time. The overall price or energy cost of $\mathcal{E}$ is the sum of the price or energy cost of the immediately composed event services (denoted $\mathcal{E}_{ice}$).
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 product of the completeness and the frequencies of the services in $\mathcal{E}_{ice}$ (denoted $f(e)$, $e \in \mathcal{E}_{ice}$). We refer readers to Section 5.2 for detailed description on estimating frequencies of event services.
3. **Availability**, **Accuracy** and **Security** are operator independent properties. The availability and accuracy of $\mathcal{E}$ is the product of event service availability and accuracy in $\mathcal{E}_{ice}$. The rationale of using the product of the accuracies as the aggregated accuracy is that we consider a result is incorrect if one of the inputs used to calculate the result is incorrect. By the same logic we aggregated the availability with multiplication. We consider the security level is determined by the most vulnerable event service in $\mathcal{E}_{ice}$.
4. **Latency** of event $\mathcal{E}$ is an operator dependent property. It is determined by the last event completing the event pattern of $\mathcal{E}$. Therefore, if the root operator of $\mathcal{E}$ is sequence or repetition, the latency of $\mathcal{E}$ is same as the last event in the direct sub-events of $\mathcal{E}$ (denoted $\mathcal{E}_{dst}$). Since it is

hard to predict when the last direct sub-event occurs under parallel operators (i.e., "And" and "Or" operator), we make an approximation with the average of the latencies of the event services in $\mathcal{E}_{dst}$.

5. **Completeness** is an operator dependent property. The completeness of $\mathcal{E}$ can be estimated based on its direct sub-event frequencies (denoted $f(e),\ e \in \mathcal{E}_{dst}$), and direct sub-event reliabilities. The idea is to derive the estimated receiving frequency of $\mathcal{E}$ by investigating its direct sub-event sending frequencies and reliability, and then divide the estimated receiving frequency by the estimated logical frequency (the theoretical value of how often should $\mathcal{E}$ occur). For *Sequence*, *Repetiton*[18] or *And* operators, the estimated receiving frequency of $\mathcal{E}$ is the minimum of the products of the sending frequency and reliability of the event services in $\mathcal{E}_{dst}$. Conversely, for *Or* operators the maximum is used as the estimated receiving frequency.

6.1.2 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 [36]). While defining a sophisticated utility function is not the focus of this report, we propose to use the Simple Additive Weighting (SAW [11]) technique to define the service QoS utility. SAW is widely used in QoS-aware service composition optimization, e.g., in [8],[13],[15],[10],[12]. It is worth noting that by applying SAW we have the following three assumptions [37]:

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 for us because we do not take into account the probability of errors in the QoS aggregation. The second assumption also holds because a total order can be applied to all eight QoS dimensions discussed in Section 6.1, 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 the current QoS model, we do not consider the correlation between quality attributes.

Given a quality vector of an event service composition $Q=<L,P,E,B,Ava,C,Acc,S>$ representing the service QoS capability, we denote q as one of the eight quality dimensions in the vector, $O(q)$ as the theoretical optimum value in the quality dimension of q, $C(q)$ as the user-defined value specifying the hard constraints (i.e., worst acceptable value) on the dimension, and $0 \leq W(q) \leq 1$ as the weighting function of the quality metric, representing users' preferences. Further, we distinguish

[18] The function card($\mathcal{E}$) gives the cardinality of the repetition, other operators have a default cardinality of 1.

between QoS properties with 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 $Q_-=\{L,P,E,B\}$ is the set of properties with the negative tendency (smaller values the better). The QoS utility U is derived by:

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

where $q_i \in Q_+$, $q_j \in Q_-$. According to the above equation 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 $\overline{U} = U+|Q_-|)/(|Q_+|+|Q_-|)$.

6.2 Genetic Algorithm for QoS-Aware Event Service Composition Optimization

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, then the total number of concrete composition plans is estimated to be $n \times l^m$. In large-scale scenarios, it is highly inefficient to enumerate all possible compositions of event services and evaluate their overall performance. In this section, we propose a heuristic method based on *Genetic Algorithms* (GA) to derive global optimizations 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 section follows these steps. The ``fitness'' of each solution can be evaluated by the QoS utility function in Section 6.1.2. Compared to traditional GA based optimizations 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 Section 5, where hierarchies (i.e., ERHs, introduced in Section 5.2.3) of reusable event services are maintained.

6.2.1 Population Initialization

Given an event service composition request represented by a canonical event pattern ep, the initialization 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; next we explain how ACPs are derived based on ERF.

When an event pattern ep is inserted into the ERF, we can mark its *reusable nodes* 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 19. 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, we can derive the ACPs for ep.

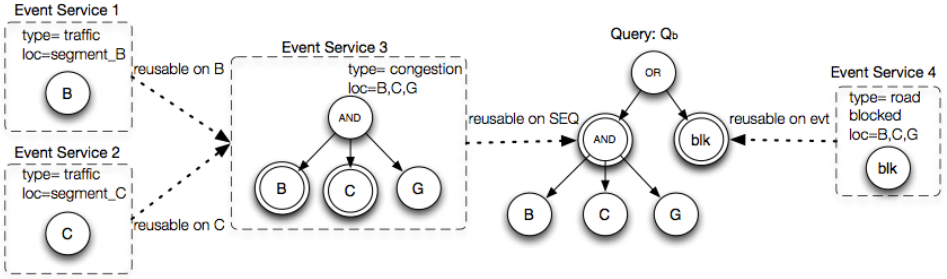


Figure 17: Marking the Re-usable Nodes

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.

6.2.2 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 which service tasks they relate to. Similarly, we encode the event detection tasks (leaf nodes) in a CCP with values to indicate the service bindings used. However, the positions of the values (arranged in any tree traversal orders) cannot represent which parts of the event detection task do the reused event services contribute in, since the CCPs are unordered 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 counter-parts. 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 service binding for the leaf, as shown in Figure 20. For example, a gene for the leaf node "n13" in P2 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 P2.

6.2.3 Crossover and Mutation Operations

After the population initialization and encoding, the preparation tasks for GA based optimization are completed. The algorithm iterates the cycle of select, crossover and mutation to find optimal solutions. The selection is trivial; individuals with better finesses (i.e., QoS utility) are more likely to

be chosen to reproduce. In the following we explain the details on crossover and mutation operations for event service composition optimization.

Crossover

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 event patterns. An example of crossover is illustrated in Figure 20. 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:

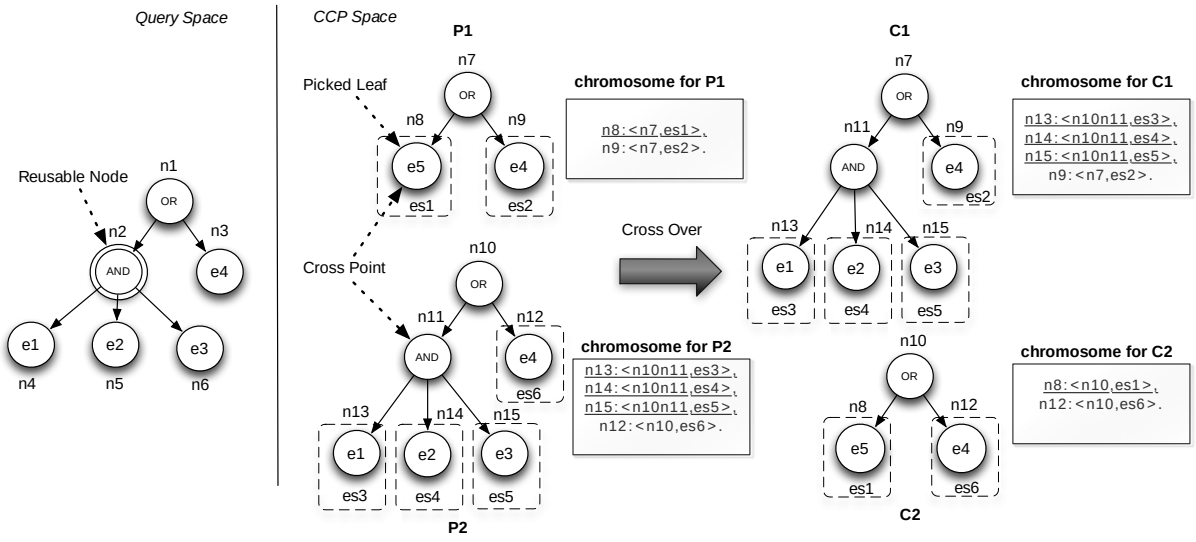


Figure 18: Example of Genetic Encodings and Crossover

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, i.e., $code_1$, as follows: replace each node id in the prefix of $code_1$ with the operator type,
2. For each leaf l_1 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 5.2.3), 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.

Mutation

The mutation operation changes the composition plan for a leaf node in a CCP. To do that we 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 we use the same random CCP creation process specified in the population initialization (see Section 6.2.1) to alter its implementation.

Elitism

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

6.3 Evaluation

In this section, we present the evaluation results of the proposed approaches. We put our experiments in the context of the travel planning scenario introduced in Section 6.3.1 by using both real and synthetic sensor datasets for the city of Aarhus. The evaluation has two parts: in the first part, we analyse in detail the performance of the genetic algorithm. In the second part, we demonstrate the usefulness of the QoS aggregation rules. We also present the datasets used and then show the evaluation results. All experiments are carried out on a Macbook Pro with a 2.53 GHz duo core cpu and 4 GB 1067 MHz memory. Prototypes are developed in Java (JDK 1.7). Experiment results are average of 30 iterations.

6.3.1 Experiment Scenario: Smart Travel Planner

Figure 19: Traffic Sensors in Aarhus City

The city of Aarhus has deployed a set of traffic sensors on the streets. These 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 21 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 21 to her work place (point F in Figure 21) within the time frame. Alice decides not to pick a route randomly since it is rush hour and there's good chance that she may hit traffic. Instead, Alice uses a travel planer application on her smart phone 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 21, 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 same segments (B-C-G and E-F) on the route that Alice chooses and they 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 optimization needs to be efficient to enable real-time re-planning and adapt to the fast changing service environment, simply enumerating all possible composition plans will not scale.

6.3.2 Part 1: Performance of the Genetic Algorithm

In this part of the evaluation, we compare the QoS utility derived by Brute-Force (BF) enumeration and GA developed and test the scalability of the GA. Moreover we analyse the impact of different GA parameters and provide guidelines to identify optimal GA settings.

Datasets

Open Data Aarhus (ODAA) is a public platform that publishes sensor data and metadata about the city of Aarhus. Currently there are 449 pairs of traffic sensors in ODAA, each pair is deployed on one street segment for one direction and reports the traffic conditions on the street segment. These traffic sensors are used in the experiments to answer requests on travel planning. We also include some other sensors in our dataset that 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 (i.e., Q_a in Figure 22 that calculates the sum of the estimated travel time from A to F on the map in Figure 21 when a traffic update from all segments are received, meanwhile, keep track of Alice's latest location) or Bob's (i.e., Q_b in Figure 23 that notifies the user when there are traffic congestions (labelled ``cng'') on route B to G, or if a road construction has blocked route (labelled ``blk'')), i.e., they are noise to queries like Q_a and Q_b (but could used in other travel related queries). In total we use 900 real sensors from ODAA, in which about half of them are noise. We denote this dataset sensor repository R0.

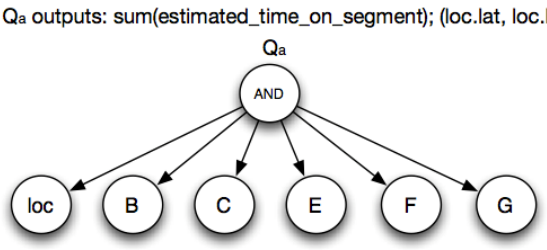


Figure 20: Traffic Planning Event Request for Alice (denoted by Q_a).

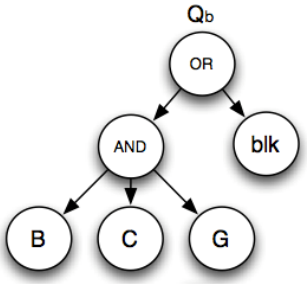


Figure 21: Road Congestion Event Request for Bob (denoted by Q_b).

Each sensor in R0 is annotated with a simulated random quality vector *<L,Acc,C,S>* where *L∈[0ms, 300ms], Acc, C ∈[50%,100%], S∈[1,5]* and frequency *f∈[0.2Hz,1Hz]*. We do not model price or energy consumption in the experiments because their aggregation rules are similar to bandwidth consumption. For similar reasons we also do not model availability. To test the algorithms on a larger scale, we further increase the size of the sensor repository by adding N functionally equivalent sensors to each sensor in R0 with a random quality vector, resulting in the 9 different repositories as shown in Table 4. In the experiments we use a loose constraint[19] to enlarge the search space and we set all QoS weights set to 1.0[20]. The queries used in the experiments are summarised in Table 5.

Table 4: 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

Table 5: Queries Used in Experiments

Query	Description	Nodes
Q_a	Alice's query on estimated travel time on route, depicted in Figure 22.	1 AND operator, 6 streams.
Q_b	Bob's query on whether a route has been congested or blocked, depicted in Figure 23.	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

[19] Constraint used in the evaluation: (L>= 3000ms, Acc>= 0, C >= 0, S >= 1, B <= 50)

[20] In this report, we consider the weights representing users' personal preferences and do not differentiate between "good" or "bad" weight settings.

		operator, 10 streams

QoS Utility Results and Scalability

In this set of experiments we first demonstrate the usefulness of the GA by comparing it to a BF algorithm. Figure 24 and 25 show the experiment results of both BF and GA results for Q_a over R3 to R9 (R1 and R2 are not tested here because their solution spaces are too small for GA), 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 24 shows the utilities of the composition plans derived from BF, GA and a random picking approach for Q_a, Q_a'. Figure 25 shows the time taken for the BF and GA to find the best plan for Q_a.

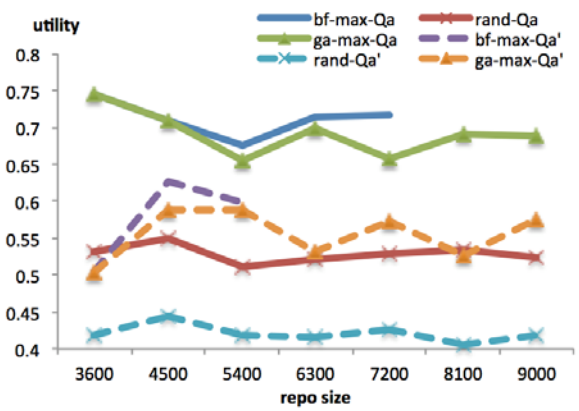


Figure 22: QoS Utilities, BF, GA and Random Pick

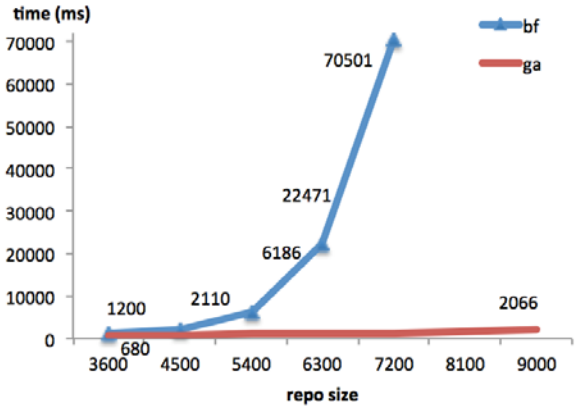


Figure 23: Composition Time Required by BF and GA for Qa

The best utility obtained by the GA is the highest utility of the individual in the last generation before the GA stops. In the current implementation, the GA is stopped when the current population size is less than 5 or the difference between the best and the average utility in the generation is less than 0.01, i.e., the evolution has converged. Given the best utility from BF $\overline{U}_{bf}$, best utility from GA $\overline{U}_{ga}$ and the random utility of the dataset $\overline{U}_{rand}$}, we calculate the degree of optimization as d=($\overline{U}_{ga}$ - $\overline{U}_{rand}$)/($\overline{U}_{bf}$ - $\overline{U}_{rand}$). From the results in Figure 24 we can see 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 R8 and R9, because of memory limits (heap size set to 1024 MB). We can see that for smaller repositories, d is bigger. This is because under the same GA settings[21], 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 6.2.3 makes sure that, if found, the global optimum "survives" till the end of evolution, e.g., in the GA results for Qa over R3,R4 in Figure 24.

[21] In the experiments in Section 6.3.2, the GA has the following parameter settings: the initial population size is set to 200, crossover rate is 95%, mutation rate is 3%.

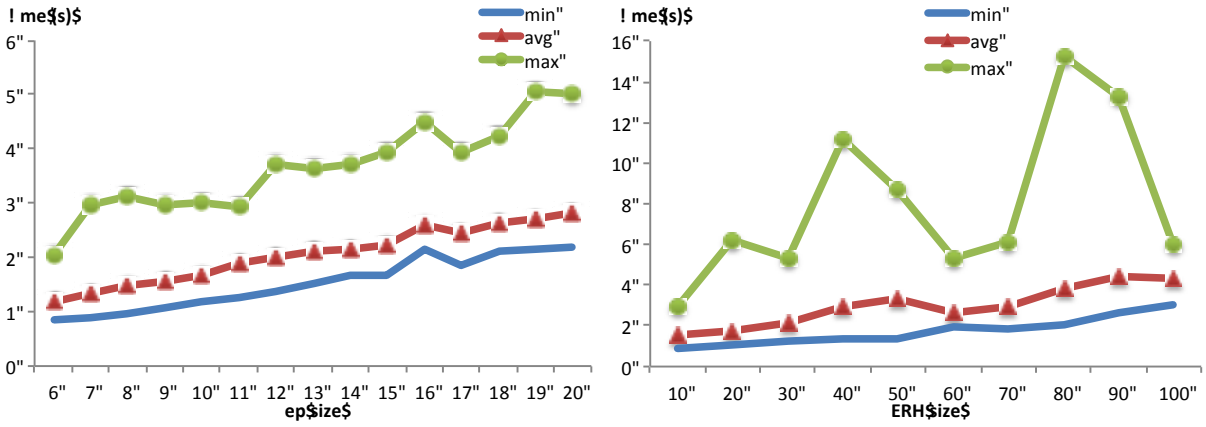


Figure 24: GA Scalability over EP Size and ERF Size

It is evident that a BF approach for QoS optimization is not scalable because of the NP-hard nature of the problem. From the results in Figure 25 we can see that the composition time of Q_a grows exponentially for the BF algorithm when the size of the repository increases, while the time take for the GA to converge grows linearly.

Figure 25 shows the scalability of the the GA using a fixed query (i.e., Q_a) over different repository sizes with no CESs deployed in those repositories. Now we analyse further the scalability of the GA using queries with different sizes, i.e., different total number of event operator nodes and event service nodes, as well as different number of CESs in the ERF. To test the GA performance with different event pattern sizes using different operators, we use the EST of Q_b as a base and replace its leaf nodes with randomly created sub-trees (invalid ESTs excluded). Then we test the GA convergence time of these queries over R5. Results from Figure 26 (left) indicate that the GA execution time increases linearly with regard to the query size.

In order to test the scalability over different number of CESs in the ERF (called ERF size), we deploy 10 to 100 random CESs to R5, resulting in 10 new repositories. We test the GA on a query created in the previous step (denoted Q_b') with the size of 12 nodes (2 operators, 10 sensor services) and record the execution time in Figure 26 (right). To ensure each CES could be used in the composition plan, all CESs added are sub-patterns of Q_b'. From the results we can see 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 of those cases, we found that most (over 90%) of the time is spent on population initialisation, and this is caused by the complexity of the ERF, i.e., number of edges considered during ACP creation.

Fine-tuning the Parameters

In the experiments above, a fixed set of settings is used as the GA parameters, including crossover rate, mutation rate and population size. In order to find good settings of the GA in our given problem domain, we fine tune the mutation rate, population size and crossover rate based on the default setting used above, i.e., we change one parameter value at a time while keeping other parameters unchanged.

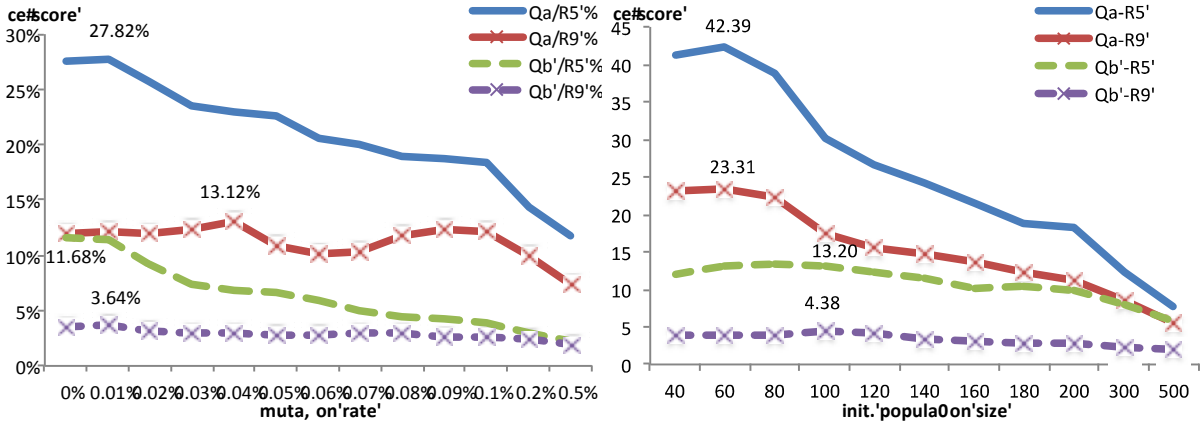


Figure 25: CE-score over mutation rate and population size

In order to determine the effect of the parameter tuning, we define a Cost-Effectiveness score (i.e., CE-score) as follows: given the random pick utility of a dataset $\overline{U}_{rand}$, we have the final utility derived by GA $\overline{U}_{ga}$ and the number of milliseconds taken for the GA to converge t_{ga}, CE-score = $(\overline{U}_{ga}-\overline{U}_{rand})*10^5/t_{ga}$. We test two queries Q_a, Q_b' over two new repositories R5', R9', which are R5 and R9 with 50 and 100 additional CESs, respectively. Hence we have 4 test combinations on both simple and complex queries and repositories. The results for fine-tuning the mutation rate, population size and crossover rate are shown in Figure 27 and 28.

From the results in Figure 27 (left) we can see that the optimal mutation rate is quite small for all tests, i.e., from 0% to 0.4%. Results in Figure 27 (right) indicate that for smaller solutions spaces such as Q_a over R5' and R9', 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 28 indicate that for Q_a over R5', 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., from 90% to 100%, is desired. It is worth noticing that in the results from Figure 28 and 29, the changes of the score for Q_b' over R9' is not significant. This is due to the fact that the GA spends much more time trying to initiate the population, making the cost-effectiveness score small and the differences moderate.

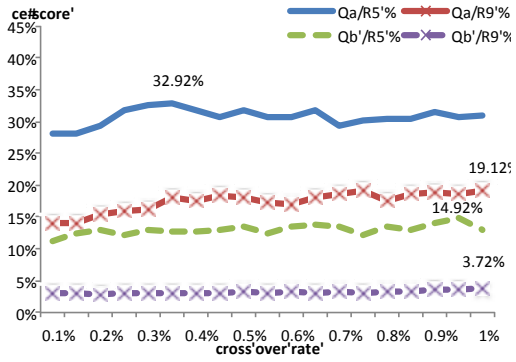


Figure 26: CE-score Using Different Crossover Rate

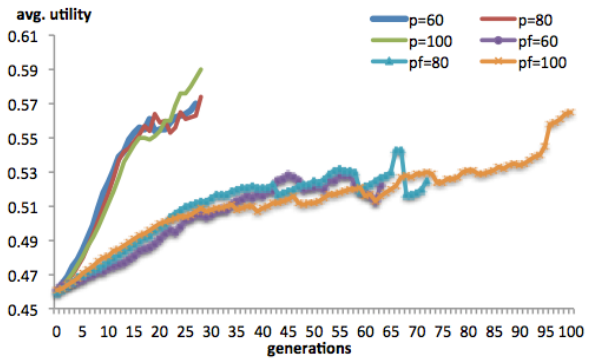


Figure 27: Average Utility Using Flexible ("p=x") and Fixed ("pf=x") Population Size

In all previous experiments we use the selection policy such that 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 in our case because the algorithm is executed 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 flexible population size, we show the average utility (of Q_b' over R9') over the generations in Figure 29. The results show that the number of generations for flexible population sizes is similar while larger sizes achieve higher utilities. In addition, the duration of generations in fixed population sizes is 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 flexible populations. In summary, we can confirm that using a flexible population size is better than a fixed population size for the GA described in this section.

6.3.3 Part 2: Validation of QoS Aggregation Rules

In this part of evaluation we show how the QoS aggregation works in a simulated environment.

Datasets and Experiment Settings

In order to demonstrate the effect of QoS aggregation and optimisation, we generate two composition plans with the GA for Qa over R9' using same constraint as in Section 6.3.2: CP_1 is optimized for latency, with the weight of latency set to 1.0 and other QoS weights set to 0.1; while CP_2 is optimized for bandwidth consumption, with the weight of bandwidth consumption set to 1.0 and others 0.1. The reason we generate two plans for best latency and bandwidth consumption is that the result plans are the most different in structure, as shown in Figure 30.

When the two composition plans are generated, we transform the composition plans into stream reasoning queries (C-SPARQL query) using the query transformation algorithm defined in Section 7.2.1. We evaluate the queries over 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, we simulate the event streams on a local test machine, i.e., we create artificial delays, wrong and lost messages according to the QoS values in the quality vector, and 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, 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 6.1.1). We observe the results and the query performance over these simulated streams and compare it with the QoS estimation using the rules in Table 2 and 3, to see the differences between the practical quality of composed event streams and the theoretical quality as per our estimation.

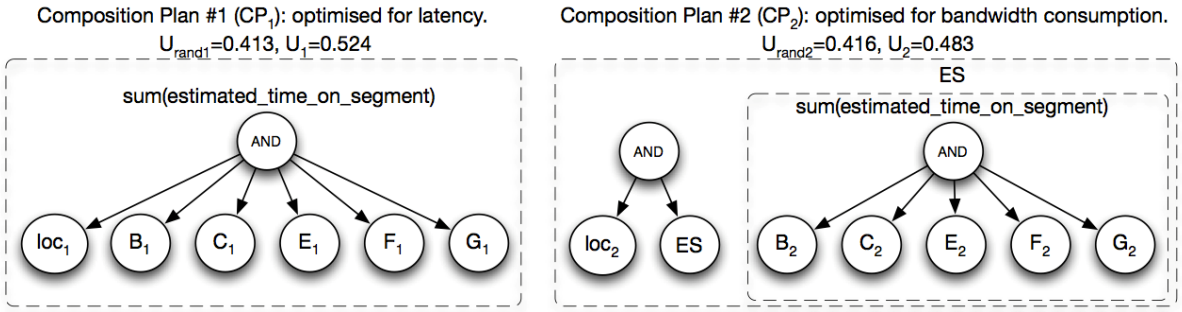


Figure 28: Composition Plans for Qa Under Different Weight Vectors

Simulation Results

The results of the comparison between theoretical and simulated quality of the event service composition is shown in Table 6. 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 3. These rules take into account the *Composition Pattern* of the query as well as the *Service Infrastructure* quality of the composed services. We denote this quality 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 we deploy the queries with optimal *Service Infrastructure* quality, i.e., no artificial delay, mistake or loss, and we record the quality of query executions in the third column. We denote this engine quality QoS_{ee}. The simulated end-to-end quality is recorded in the fourth column, denoted QoS_s. We calculate the theoretical end-to-end quality based on QoS_{cp} and QoS_{ee} using Table 2. Notice that the *Service Infrastructure* qualities of the queries themselves are not considered, since we do not measure the results provided to external service consumers, rather, the quality measurement stops at the point when query results are generated. We denote this theoretical end-to-end quality QoS_t and calculate the deviation $d=(QoS_s/QoS_t) - 1$, which is recorded in the last column.

From the results we can see that the GA is very effective in optimizing 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.

We can also see 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 C-SPARQL 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.

Table 6: 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%

Another interesting observation from Table 6 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 C-SPARQL 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.

6.4 QoS Aggregation and Stream Selection using Answer Set Programming

An alternate to the GA can be to use Answer Set Programming (ASP) to evaluate QoS-Aggregation and constraint satisfaction for QoS and QoI preferences. ASP based approach will be further evaluated in the Activity 4.2 “Conflict Resolution” of the City Project and will be documented in Deliverable 4.2. Goal of the provision of ASP based QoS aggregation is to provide a comparison between ASP based results with genetic algorithm based implementation described earlier.

6.5 Summary

In this section, we have addressed the issue of enabling QoS-aware event stream federation and optimisation using services in an IoT context. A QoS aggregation schema is proposed to calculate the overall QoS vector for a federated IoT stream (an event service composition). Based on a user-defined constraint and weight vector, a QoS utility function is defined to calculate the degree of optimisation for event composition. A genetic algorithm is developed and evaluated to efficiently create optimal event service compositions. We have evaluated the proposed approach over a travel planning scenario with both real and synthetic datasets. The experimental results show that the genetic algorithm is scalable, and can give near-optimal (about 89% optimal) results efficiently. We

have provided analysis 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. The experiments on the validation of QoS aggregation show that our estimation model does not deviate too far from the practical results. Also, this experiment shows that to improve the query quality, a distributed way of federating stream queries is needed. As future work, we plan to optimise the GA approach with regard to the population initialisation, as this process takes the majority of the computation time. We also plan to investigate how interdependent quality metrics can be modelled and optimised. Moreover, investigating the QoS aggregation over different RDF stream processing engines is also on the agenda. Part of the research in this section is published/submitted at [41,43]

7. Enactment of Federation: Automatic Event Service Implementation

In order to implement a composition plan, the subscription manager needs to make 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. In this section, the algorithms for transforming composition plans into semantic stream 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 6: Traffic Sensor Data using SSN

In the current ACEIS implementation, CQELS and C-SPARQL are used as the semantic stream processing engines [22]. These engines consume semantically annotated events. The query transformation algorithm in ACEIS depends on the schema of annotated events, i.e., the ontologies used. However, it can be adapted to different event ontologies with little effort as long as the essential information (i.e., source of event and event payload) is provided. Without loss of generality, we assume the primitive events in the smart city context are annotated as sensor observations in the SSN ontology. A sample traffic sensor reading annotated as an *Observation* in SSN is shown in Listing 6. In the following we first discuss how the operator semantics in ACEIS can be implemented by the operators in existing semantic CEP systems, then we present the detailed query transformation algorithm for generating CQELS and C-SPARQL queries according to the semantic alignments.

[22] ETALIS engine is not integrated in the current prototype implementation but we do present the semantic alignments for ETALIS operators for completeness.

7.1 Semantics Alignment

The semantics of event operators to query operators need to be aligned in order to ensure the query transformation creates queries that detect the right event patterns. Table 7 summarises how event operators in CES can be implemented by query operators in CQELS, C-SPARQL and ETALIS. In the following, we elaborate the details.

Table 7: Semantics Alignment for Event Operators

ACEIS	E	And	Or	Seq	Rep_o	Rep_n	Sel	Filter	Window
CQELS	SGP	$\bowtie$	$\bowtie$	-	-	-	BGP+*proj*	Filter	Window
C-SPARQL	BGP	$\bowtie$	$\bowtie$	f_t	$\bowtie + f_t$	f_t^+	BGP+*proj*	Filter	Window
ETALIS	BGP	$\bowtie$	$\bowtie$	SeqJoin	$\bowtie$ +SeqJoin	SeqJoin$^+$	BGP+*proj*	Filter	getDuration()

7.1.1 Event Declaration

An Event Declaration E in an event pattern ep indicates the occurrences of event instances of type E. As shown in Listing 6 the occurrences of sensor events are annotated as observations. If we use SPARQL to query the occurrences of sensor observations, a single triple pattern t = (?id, rdf:type, ssn:Observation) can suffice. Given a set of mappings Ψ, u ∈ Ψ is a partial function from variables to values, such that u(var(t)) gives the mapping value (i.e., the IRI) of an occurred observation where var(t) is the set of variables in t. To get only the observations produced by E, we could use a *BasicGraphPattern* (BGP) P=(tU (?id, ssn:observedBy,ed.src)) where E.src is the source (i.e., service id) of E specified in the composition plan. Then, Ψ (var(P)) gives all the IRIs of sensor observations produced by E. Ψ (P) gives the set of triples by replacing the variables in t with corresponding values from Ψ. We refer to this set of triples *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 observation, which are the member event instances in the EIS triggering E. The engines in Table 7 reuse and extend the query semantics of SPARQL, therefore we can use the same BGPs[23] to query the occurrence of events instances of type E.

7.1.2 AND Operator

An *And* operator indicates instances of the connected 2 sub-event types E_1, E_2 should occur, i.e., given $E_3 := \Lambda(E_1,E_2)$, $T_{id}(E_3) = T_{id}(E_1) \cup T_{id}(E_2)$, where $T_{id}(E_1) \neq \emptyset \wedge T_{id}(E_2) \neq \emptyset$. This event operator can be implemented by *join* (⋈) 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., $\Psi_3 = \emptyset \Leftrightarrow \Psi_1 = \emptyset \vee \Psi_2 = \emptyset$. 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 if we are operating in the *cumulative* event instance selection policy (recall Section 4.2.3), 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.

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

7.1.3 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* ($\ltimes$) operator with *bound* filters in SPARQL. To do that we create the new set of mappings: $\Psi_4 = \ltimes \Psi_1 \ltimes \Psi_2$, where Ψ_4 satisfies the condition: $\forall\, u_4 \in \Psi_4, \exists\, v_4 \in dom(u_4) \longrightarrow bound(v_4) = true$. 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.

7.1.4 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 we need to join event id triples based on their timestamps. In ETALIS a *SeqJoin* operator is defined as an extension of SPARQL *join*. For brevity we refer readers to [22] for detailed definition. In C-SPARQL such an extension does not exist. However, C-SPARQL 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 we can create a set of mappings $\Psi_5 = \Psi_1 \bowtie \Psi_2$ such that: $\forall\, u_5 \in \Psi_5$, $u_5 = u_1 \bowtie u_2$ where $u_1 \in \Psi_1, u_2 \in \Psi_2$, $f_t(v_1) \leq f_t(v_2)$ 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[24]) does not support *SeqJoin* or provide functions to access the timestamps of the stream triples, therefore *Sequence* is not supported in CQELS.

7.1.5 REPETITION Operator

Repetition is a generalization of sequence, recall definitions in Section 4.2.3, an overlapping (i.e., Rep_o) or non-overlapping (i.e., Rep_n) repetition can be transformed into a conjunction of sequences or a sequence of sequences, respectively. Therefore, repetition can be implemented in C-SPARQL and ETALIS by combining the ways they implement *And* and *Sequence* event operators, while CQELS does not support repetition because sequence is not supported in CQELS.

7.1.6 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 *projecting* the relevant variables into the query results. Notice that for brevity we do not list all triple patterns required here.

7.1.7 Filter and Window

Filter and *Window* operators in event patterns are mapped to *Filter* and *Window* operators in the three engines, respectively. Notice that in ETALIS an explicit *Window* operator does not exist, and 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.

[24] https://code.google.com/p/cqels/

7.1.8 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, C-SPARQL uses a time driven approach, in which a query is executed periodically, whenever the window slides. In order to have same results produced by CQELS and C-SPARQL engines, we deploy a post-processing component on CQELS result handler that accumulates the results periodically and simulate the time driven query execution.

7.2 Transformation Algorithm

Previously (see Section 4.1), we briefly described how event patterns are specified in the CES ontology and what are the semantics of event patterns (Section 4.2). An event pattern can be recursively defined with sub event patterns and event service descriptions, thus formulating an event syntax tree. In this section we elaborate algorithms for parsing event syntax trees and creating semantic stream queries (i.e., CQELS and C-SPARQL queries) based on the semantics alignments presented in Section 7.1. Recall that in an event syntax tree, 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.

7.2.1 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. Listing 7 shows the pseudo code of the main parts of the query transformation algorithm.

Algorithm 6 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**

Listing 7: CQELS Query Transformation Algorithm

Lines 1 to 6 in Listing 7, construct the CQELS query with three parts: a pre-defined query prefix, a select clause derived from the *getSelectClause*() function and a where 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 pattern, a *getFilters*() function is invoked to add the filter clauses to the where clause (Lines 23-25). Finally, the *Where* clause is returned (Line 26).

```
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: CQELS Query Example

Listing 8 shows the transformation result for an event request. 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).

7.2.2 C-SPARQL Query Transformation

Now we present the transformation algorithm for C-SPARQL queries as shown in Listing 9. It has the same structure as the algorithm in Listing 7. The main method (i.e., *transform*()) takes 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 C-SPARQL these from clauses are retrieved all at once by the *getFromClause*() because of the difference in syntax. The main difference in C-SPARQL 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 6, 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 require much additional effort.

An example of the transformation result is shown in Listing 10. Notice that in this C-SPARQL query, a sequence is specified for the user location update event and traffic report event, just for demonstrating the use of timestamp filters in C-SPARQL.

Algorithm 7 Transform event patterns into C-SPARQL queries.

Require: Composition Plan: *comp*, Query Prefix String *prefix*
Ensure: C-SPARQL Query String: *queryStr*
procedure TRANSFORM(*comp*, *prefixStr*)
  sClause ← GETSELECTCLAUSE(*comp.ep*)
  fClause ← GETFROMCLAUSE(*comp.ep*)
  wClause ← GETWHERECLAUSE(*comp.ep*)
  queryStr ← *prefix* + *sClause* + *fClause* + "*WHERE*" + *wClause*
return *queryStr*
end procedure
Require: Event Pattern: *ep*
Ensure: Where Clause String: *wClause*
procedure GETWHERECLAUSE(*ep*)
  root ← GETROOTNODE(*ep*), *wClause* ← ∅
  if *root* ∈ EventDeclaration **then**
    wClause ← GETBGP(*ep*, *root*)
  else if *root* ∈ Op_{and} **then**
    for *subPattern* ← GETSUBPATTERNS(*ep*, *root*) **do**
      wClause ← *wClause* + GETWHERECLAUSE(*subPattern*)
    end for
  else if *root* ∈ Op_{or} **then**
    for *subPattern* ← GETSUBPATTERNS(*ep*, *root*) **do**
      wClause ← *wClause* + "optional" + GETWHERECLAUSE(*subPattern*)
    end for
    wClause ← *wClause* + GETBOUNDFILTERS(*ep*)
  else if *root* ∈ Op_{seq} **then**
    for *subPattern* ← GETSUBPATTERNS(*ep*, *root*) **do**
      wClause ← *wClause* + GETWHERECLAUSE(*subPattern*)
    end for
    wClause ← *wClause* + GETTIMESTAMPFILTERS(*subPattern*)
  else if *root* ∈ Op_{rep} **then**
    subPattern ← EXPANDREPETITION(*comp.ep*, *root*)
    wClause ← *wClause* + GETWHERECLAUSE(*subPattern*)
  end if
  if *filters* ← GETFILTERS(*ep*) ≠ ∅ **then**
    wClause ← *wClause* + GETFILTERS(*filters*)
  end if
return "{" + *wClause* + "}"
end procedure

Listing 9: C-SPARQL Query Transformation Algorithm

7.2.3 Event (Re-) Construction from Stream Query Results

The query solutions derived from evaluating the queries in Listing 8 and 10 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 the complex events may vary depending on the applications, we 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.

```
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 10: C-SPARQL Query Example

7.3 Summary

In this section we have described the mechanism for transforming event service composition plans into RDF stream processing queries, in order to achieve automatic semantic (complex) event service deployment and execution. The semantics alignment of event operators and stream query operators are presented and analysed. Two algorithms for transforming the composition plans into CQELS and C-SPARQL queries are developed, and examples of the query transformation results are presented. Automatic event service implementation is a stepping-stone for enabling automatic service adaptation, which is elaborated in deliverable D5.1 of the CityPulse Project. Part of the research in this section is published/submitted at [40,45].

8. Optimisation Techniques for Concurrent Query Processing

As described in Section 7, we use RDF Stream Processing (RSP) engines to implement semantic event services. However, existing RSP engine implementations are still in their early stages (research prototypes only). In order to investigate the feasibility of using them in large-scale applications, performance evaluation and optimization are required. In the following, we first analyse the performance of single CQELS and C-SPARQL engines when processing multiple different queries. Then we discuss the optimization technique of using multiple engine instances in parallel and evaluate the improvement in query performance. Finally, we demonstrate the experiment results of the stress tests conducted in order to find out the capability of the server that hosts RSP engines using the aforementioned optimization techniques. We deployed our data federation server and all experiments 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 (except for the experiment in Figure 31, where more streams are used in a query), 8 - 16 triple patterns. The stream rate is configured to 15 triples per second per stream. Thus, for a single query, the input rate is 30 to 60 triples per second. These experiment settings are typical parameters we encountered in the prototypes of Citypulse scenarios.

8.1 Single Engine Performance

The query performance of a single RSP engine mainly depends on the complexity of the queries executed and the number of concurrent queries deployed. The number of federated streams in the queries affects the complexity of queries. Intuitively, if there are more streams and more joins between different streams, the query performance declines, i.e., the processing delay or query latency increases. Figure 31 shows the query latencies for CQELS and C-SPARQL while handling queries with 2, 5 and 8 federated streams. The query *Q10* in Figure 31 is one of the 12 queries evaluated in CityBench [42], which is a benchmarking tool we developed as a basis for query performance optimisation. From the result, it is evident that C-SPARQL out-performs CQELS when processing queries with multiple streams. However, it is worth mentioning that neither engine can efficiently process queries with more than 10 streams. For large queries using more than 10 streams, query rewriting that divides them into smaller sub-queries are necessary.

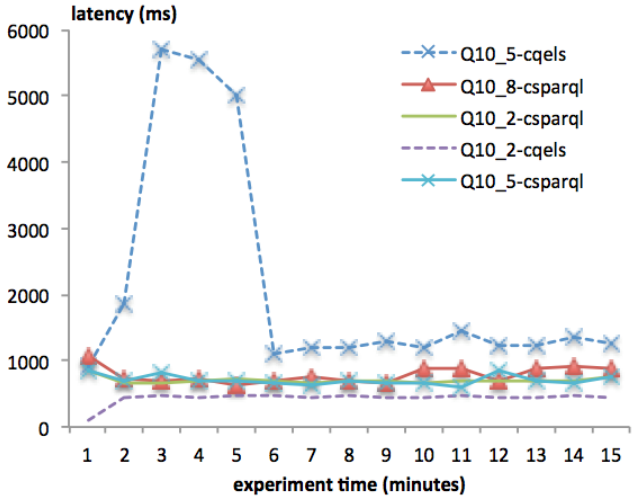


Figure 29: Query Latency over Different Number of Streams (from [42])

When the queries deployed on engines are not very complex, the main factor of the query performance is the number of concurrent queries. Figure 32 and 33 show the performance of CQELS and C-SPARQL engines when dealing with multiple queries. In the result data series, the letter "p" denotes number of engine instances deployed (explained in details in Section 8.2) and "q" represents the number of queries deployed.

The results in Figure 32 and 33 indicate that for both engines, the query latency increases when handling more queries, and CQELS is more efficient when handling multiple queries. Also, when the number of concurrent queries exceeds 30, the query is not ***stable***, i.e., the query latencies do not converge to stable values and the engine will stop producing results after a period of time.

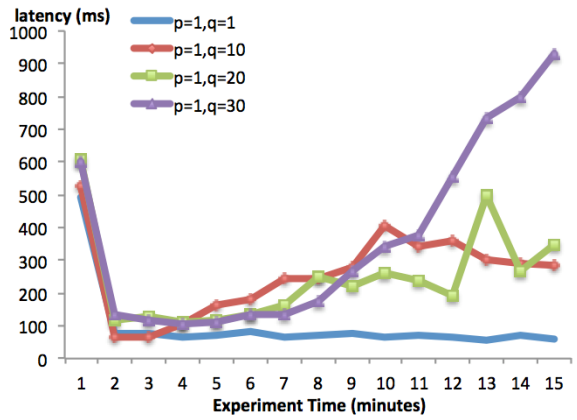


Figure 30: Query Latency over Single CQELS Engine

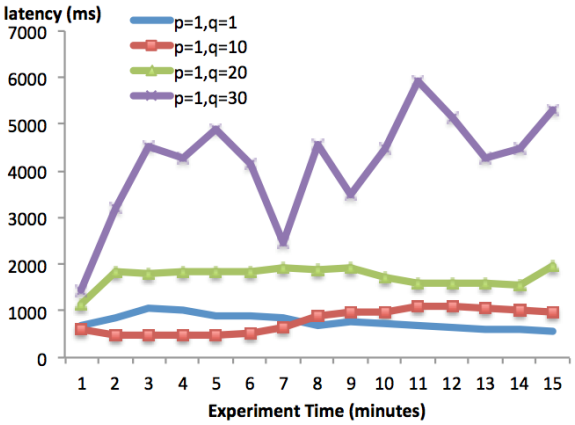


Figure 31: Query Latency over Single C-SPARQL Engine

8.2 Performance Optimization with Multiple Engine Instances

One natural thought in handling large amount of concurrent queries is to deploy multiple engine instances in parallel and distribute the workload over different engines. Thanks to the service-oriented nature of our approach, 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 we developed an additional **Scheduler** module, which consists of a query dispatcher and a performance monitor in our system that controls the load balancing using different strategies.

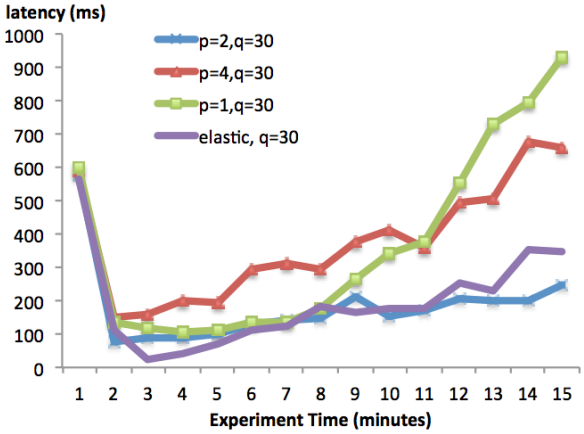


Figure 32: Query Latency over Multiple CQELS Engines

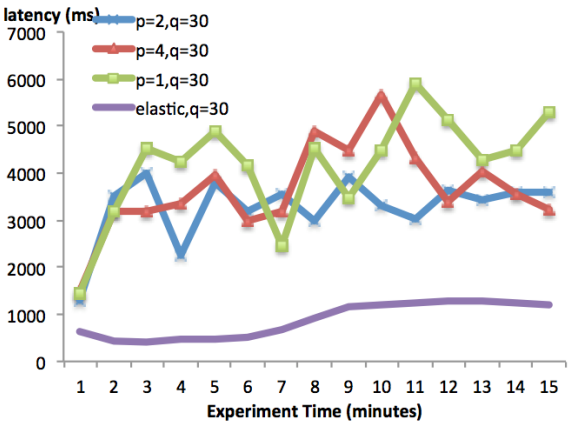


Figure 33: Query Latency over Multiple C-SPARQL Engines

The simplest strategy is to initialize a fixed amount of engine instances in the beginning and keep the same amount of queries on each engine instance. We call this strategy the *Equalized Query* (denoted "EQ") strategy. Another strategy is to dynamically create new instances based on current system load. We call this strategy *Elastic* (denoted "EL" in result figures). Since the experiments on single engine instance suggest that an engine instance may become unstable when dealing with more than 30 concurrent queries, we make the decision of creating a new engine instance when the current engine reaches 20 queries.

Figure 34 and 35 show the average query latency of multiple CQELS and C-SPARQL engines, respectively. The results show that using two engine instances reduces the query latency for both CQELS and C-SPARQL compared with single engine instance. However, more engines deployed 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 equalized queries in this experiment. Indeed, using multiple engines demands more resources such as memory and initializing all engines upfront creates overhead. Figure 36 and 37 show the memory usage for CQELS and C-SPARQL under different number of concurrent queries and engine instances, respectively.

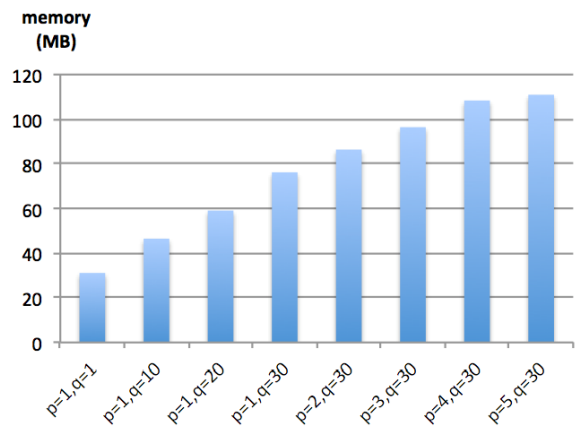


Figure 34: Memory Usage by CQELS engines

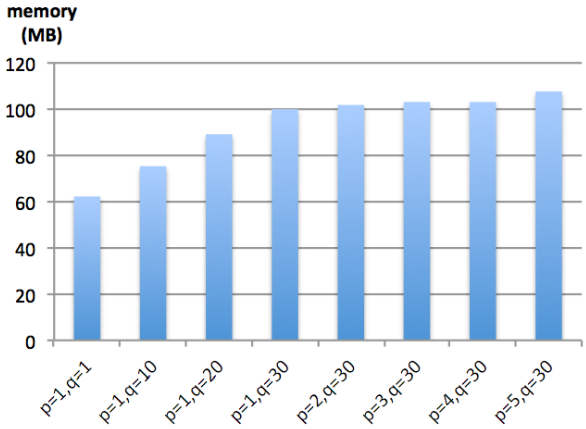


Figure 35: Memory Usage by C-SPARQL Engines

From the results in Figure 36 and 37, it is clear that the memory consumption increases with the number of concurrent queries and the number of engine instances. Also, CQELS uses less memory than C-SPARQL when dealing with less queries but the memory growth rate over the number of queries and engine instances is faster than C-SPARQL.

Since the memory availability is limited in any system, the EL approach will have to stop creating new engine instances at some point. Then it will regress to the EQ approach. An alternative way is to deploy queries on the engine that has the lowest average query latency. We call this strategy *Balanced Latency* (denoted "BL"). The results in Figure 38 and 39 show that the BL strategy outperforms EQ on both CQELS and C-SPARQL when dealing with 50 concurrent queries and 5 instances. In particular, C-SPARQL is unstable when using the EQ strategy but is stabilized when using BL.

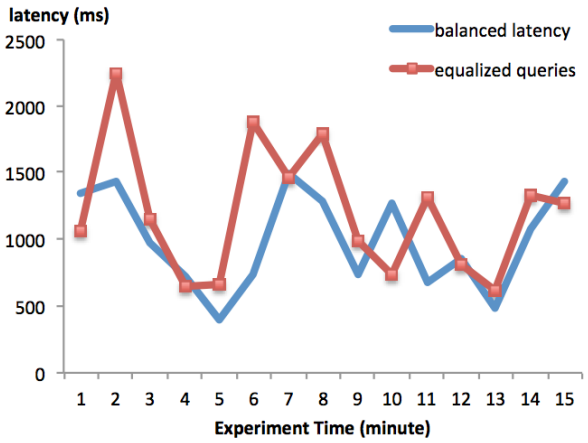


Figure 36: Average Latency of CQELS, p=5, q=50

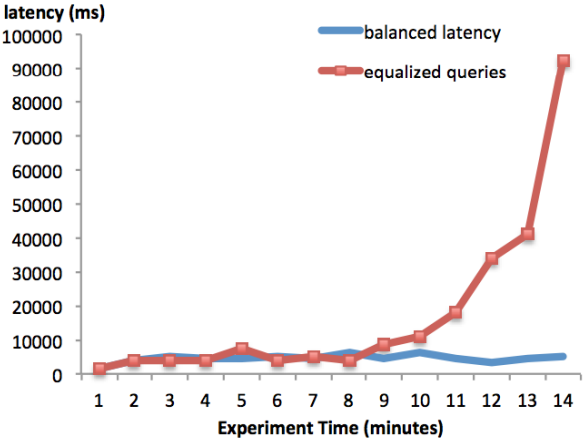


Figure 37: Average Latency of C-SPARQL, p=5, q=50

The results in Figure 40 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 C-SPARQL 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 elastic approach in the beginning and swtich to balanced query strategy when the memory limit has been reached is called Elastic-Balanced-Latency strategy (EBL).

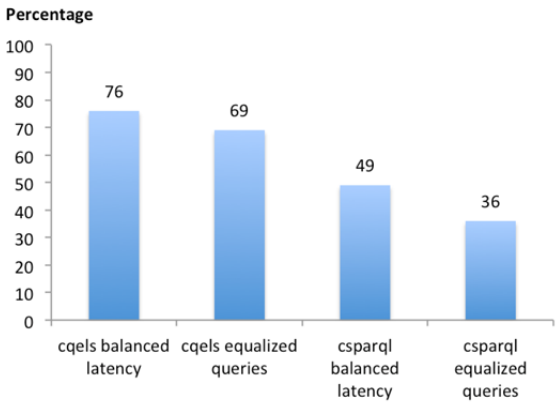
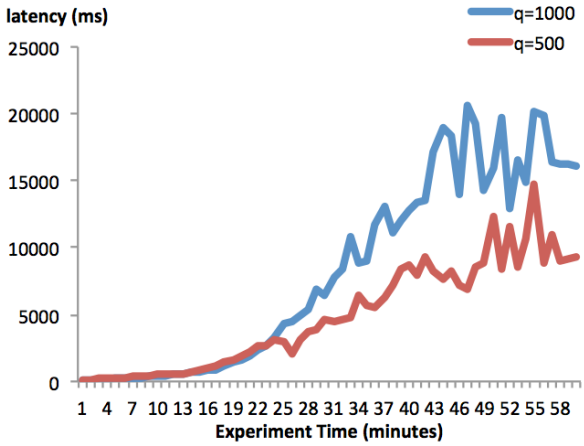


Figure 38: Query Latency Distribution: p=5, q=50

8.3 Stress Tests on Single Server Node

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, we have conducted stress tests to evaluate the systems with hundreds to thousands of queries (deployed a new query every 1-3 seconds) over the experiment duration of one hour with the EBL load balancing strategy. The query latencies for CQELS and C-SPARQL engines are shown in Figure 41 and 42.



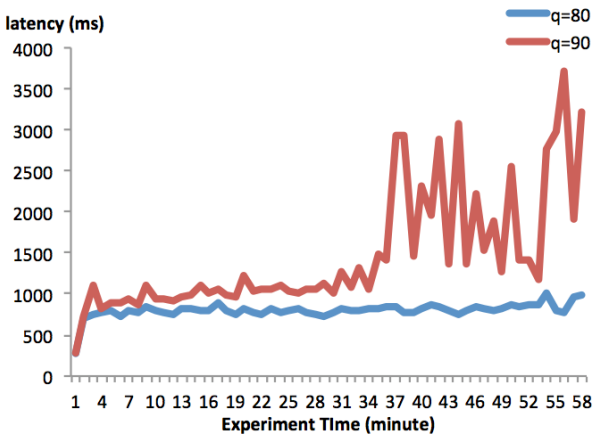


Figure 39: Average Latency of CQELS Using EBL Strategy Figure 40: Average Latency of C-SPARQL Using EBL Strategy

The stress test results show that CQELS can handle around 1000 concurrent queries with a 15-20 second delay, while C-SPARQL has a much more limited capacity of processing no more than 100 queries in a stable status (recall that "stable" means converging query latency). It is also worth mentioning that during the experiments we found that CQELS tends to use all CPU time when the workload is heavy, but C-SPARQL does not use more than 30% of the CPU time even the concurrency and query delays are high. It is not clear to us whether this behaviour of C-SPARQL is by design or an implementation issue.

8.4 Summary

In this section, the practical problem of processing large amount of queries concurrently using existing RSP engines (i.e., CQELS and C-SPARQL) was analysed. Although current RSP engines are still in their early stages and have plenty of room for performance optimization, means have been provided in this section to improve their performance in handling concurrent queries without revising the internal data processing algorithms of CQELS and C-SPARQL engines. In particular, we leverage the service-oriented nature of the data federation component and deploy the RSP queries 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, we have developed different load balancing techniques, including the equalized query, balanced latency and elastic strategy to determine which query should be delegated to which engine at run-time. Experiment results show that combing the Elastic and Balanced Latency strategy can achieve the best performance. Stress tests were carried out on CQELS and C-SPARQL 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 C-SPARQL cannot deal with more than 100 concurrent queries. However, our experiment results also suggest that C-SPARQL has a better performance in more complicated queries with more input streams. As future work we plan to further analyse the performance on distributed server clusters and using cloud services. Part of the results in this section is published in [42].

9. Conclusions

In this document, we discussed large-scale federation of urban data streams in the Smart City Framework. We lay foundations by analysing the requirements and challenges of urban data streams. We advocate the benefits of providing urban data streams as Semantic Event Services (SESs), a tackled the problem of data stream federation as the management of different activities in the semantic event service life cycle. We provided detailed insights of modelling, planning, deployment and execution of semantic event services. We bundled the relevant techniques into a single data federation component and named it as an Automatic Complex Event Implementation System (ACEIS). ACEIS acts as a data federation server within the Smart City Framework. ACEIS facilitates easy-to-use, on-demand and scalable data/event stream processing for Smart City applications while catering for individual users' requirements and preferences. In this section, we give the concluding remarks, including the outcomes, on-going works and future steps.

9.1 Outcomes

The outcome of this deliverable includes the following contributions in SES modelling, planning, deployment and execution:

- **SES Modelling**. A semantic event service model (i.e., CESO) is proposed. CESO captures event patterns and attributes to annotate complex event services and service requests. CESO allows discovery and composition of event services based on both functional and non-functional service properties. The abstract syntax and semantics of event patterns in CESO are defined in an extended version of Business Event Modelling Notation, i.e., BEMN+.
- **SES Planning.** Efficient algorithms are developed to CESs based on existing ones through the matchmaking of event patterns. The pattern based composition algorithm can produce optimised event service compositions with the least traffic demand over the service network. Algorithms are developed to propagate non-functional attributes from lower level event services to upper level, in order to estimate QoS attributes for complex event service compositions. Based on the event service QoS aggregation and estimation, heuristic algorithms are created to derive optimal event service compositions with regard to user-specified non-functional constraints and preferences based on event patterns and QoS attribute estimations.
- **SES Deployment.** The semantics of event operators and RSP query operators are compared and aligned. Based on the semantics alignments, algorithms are provided to transform CES composition plans into federated CQELS or C-SPARQL queries. Thus, an automatic deployment of SES is realized.
- **SES Execution and Optimisation.** A RSP benchmarking tool (CityBench) is developed to evaluate the query execution performance of RSP engines. Based on the evaluation results from the benchmark, optimisation techniques for handling concurrent queries with multiple engine instances are developed and tested.

9.2 On-going Works

An evaluation of data federation components for its performance and scalability in the smart city framework has been already conducted. We are currently in process of testing data federation component by prototyping smart city usecase scenario using real time data. Another on-going activity is the provision of data federation component as an APIs, which will facilitate end-users for easy deployment and integration of smart city services. We are also working on providing API specification with comprehensive documentation and publishing API as an open source release.

9.3 Future Steps

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 research and development described in this document contributes towards usage and optimisation of RSP technologies and can serve a good baseline for future optimisation and improvement in these technologies, particularly RSP engines.

ACEIS component is a pioneer distributed RSP engine, which can potentially execute distributed query over heterogeneous RSP engines. In future, we plan to provide ACEIS as a standalone distributed RSP engine, which can serve as a federation components in a variety of domains beside smart city framework. We also intend to further analyse the performance of our components over distributed server clusters and cloud services.

References

[1] B. Koldehofe, F. Dürr, and M. A. Tariq, “Tutorial: Event-based Systems Meet Software-defined Networking,” in *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, 2013, pp. 271–280.

[2] M. A. Tariq, B. Koldehofe, S. Bhowmik, and K. Rothermel, “PLEROMA: A SDN-based High Performance Publish/Subscribe Middleware,” in *Proceedings of the 15th International Middleware Conference*, 2014, pp. 217–228.

[3] P. Karampiperis, G. Mouchakis, G. Paliouras, and V. Karkaletsis, “ER designer toolkit: A graphical event definition authoring tool,” *Univers. Access Inf. Soc.*, vol. 13, no. 1, pp. 115–123, 2014.

[4] J. Schiefer, S. Rozsnyai, C. Rauscher, and G. Saurer, “Event-driven rules for sensing and responding to business situations,” in *DEBS*, 2007, vol. 233, pp. 198–205.

[5] G. Decker, A. Grosskopf, and A. Barros, “A graphical notation for modeling complex events in business processes,” in *Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOC*, 2007, pp. 27–36.

[6] S. A. McIlraith, T. C. Son, and H. Zeng, “Semantic Web Services,” *IEEE Intell. Syst.*, vol. 16, no. 2, pp. 46–53, 2001.

[7] W. Liu, Z. Liu, J. Fu, R. Hu, and Z. Zhong, “Extending OWL for modeling event-oriented ontology,” in *Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International Conference on*, 2010, pp. 581–586.

[8] M. Alrifai and T. Risse, “Combining global optimization with local selection for efficient QoS-aware service composition,” in *Proceedings of the 18th international conference on World wide web*, 2009, pp. 881–890.

[9] 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.*, 2004, pp. 149–159.

[10] Q. Wu, Q. Zhu, and X. Jian, “QoS-Aware Multi-granularity Service Composition Based on Generalized Component Services,” in *Service-Oriented Computing*, vol. 8274, S. Basu, C. Pautasso, L. Zhang, and X. Fu, Eds. Springer Berlin Heidelberg, 2013, pp. 446–455.

[11] M. Zeleny and J. L. Cochrane, *Multiple criteria decision making*, vol. 25. McGraw-Hill New York, 1982.

[12] L. Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, “QoS-aware middleware for Web services composition,” *Softw. Eng. IEEE Trans.*, vol. 30, no. 5, pp. 311–327, 2004.

[13] R. Berbner, M. Spahn, N. Repp, O. Heckmann, and R. Steinmetz, “Heuristics for qos-aware web service composition,” in *Web Services, 2006. ICWS’06. International Conference on*, 2006, pp. 72–82.

[14] L.-J. Zhang and B. Li, “Requirements Driven Dynamic Services Composition for Web Services and Grid Solutions,” *J. Grid Comput.*, vol. 2, no. 2, pp. 121–140, 2004.

[15] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani, “A lightweight approach for QoS-aware service composition,” in *Proceedings of 2nd international conference on service oriented computing (ICSOC 04)*, 2004.

[16] C. Zhang, S. Su, and J. Chen, “A novel genetic algorithm for qos-aware web services selection,” in *Data Engineering Issues in E-Commerce and Services*, Springer, 2006, pp. 224–235.

[17] C. Gao, M. Cai, and H. 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*, 2007, pp. 361–367.

[18] W. M. P. van der Aalst and A. H. M. ter Hofstede, “YAWL: Yet Another Workflow Language,” *Inf. Syst.*, vol. 30, no. 4, pp. 245–275, 2005.

[19] X. Li, Y. Fan, and F. Jiang, “A classification of service composition mismatches to support service mediation,” in *Grid and Cooperative Computing, 2007. GCC 2007. Sixth International Conference on*, 2007, pp. 315–321.

[20] J. Nitzsche, T. Van Lessen, D. Karastoyanova, and F. Leymann, “BPEL for semantic web services (BPEL4SWS),” in *On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops*, 2007, pp. 179–188.

[21] J. Kopecky, T. Vitvar, C. Bournez, and J. Farrell, “SAWSDL: Semantic Annotations for WSDL and XML Schema,” *Internet Comput. IEEE*, vol. 11, no. 6, pp. 60–67, 2007.

[22] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic, “EP-SPARQL: a unified language for event processing and stream reasoning,” in *Proceedings of the 20th international conference on World wide web*, 2011, pp. 635–644.

[23] D. Le-Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth, “A native and adaptive approach for unified processing of linked streams and linked data,” in *The Semantic Web--ISWC 2011*, Springer, 2011, pp. 370–388.

[24] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus, “C-SPARQL: SPARQL for continuous querying,” in *Proceedings of the 18th international conference on World wide web*, 2009, pp. 1061–1062.

[25] E. Bouillet, M. Feblowitz, Z. Liu, A. Ranganathan, A. Riabov, and F. Ye, “A Semantics-Based Middleware for Utilizing Heterogeneous Sensor Networks,” in *Distributed Computing in Sensor Systems*, vol. 4549, J. Aspnes, C. Scheideler, A. Arora, and S. Madden, Eds. Springer Berlin / Heidelberg, 2007, pp. 174–188.

[26] N. Glombitza, D. Pfisterer, and S. 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*, 2009, pp. 25–30.

[27] P. Rosales, K. Oh, K. Kim, and J.-Y. Jung, “Leveraging business process management through complex event processing for RFID and sensor networks,” in *40th ICCIE*, 2010, pp. 1–6.

[28] 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*, 2009, pp. 61–70.

[29] V. Stirbu, “Towards a RESTful Plug and Play Experience in the Web of Things,” in *ICSC*, 2008, pp. 512–517.

[30] J.-H. Kim, H. Kwon, D.-H. Kim, H.-Y. Kwak, and S.-J. Lee, “Building a Service-Oriented Ontology for Wireless Sensor Networks,” in *Computer and Information Science, 2008. ICIS 08. Seventh IEEE/ACIS International Conference on*, 2008, pp. 649–654.

[31] G. Decker, A. Grosskopf, and A. Barros, “A graphical notation for modeling complex events in business processes,” in *edoc*, 2007, p. 27.

[32] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann, “Specification and analysis of system architecture using Rapide,” *Softw. Eng. IEEE Trans.*, vol. 21, no. 4, pp. 336–354, 1995.

[33] A. Barros, G. Decker, and A. Grosskopf, “Complex events in business processes,” in *Business Information Systems*, 2007, pp. 29–40.

[34] G. Cugola and A. Margara, “Processing flows of information: From data stream to complex event processing,” *ACM Comput. Surv.*, vol. 44, no. 3, p. 15, 2012.

[35] A. Margara, J. Urbani, F. van Harmelen, and H. Bal, “Streaming the web: Reasoning over dynamic data,” *Web Semant. Sci. Serv. Agents World Wide Web*, vol. 25, pp. 24–44, 2014.

[36] K. Mela, T. Tiainen, and M. Heinisuo, “Comparative study of multiple criteria decision making methods for building design,” *Adv. Eng. Informatics*, vol. 26, no. 4, pp. 716–726, 2012.

[37] M. D. Rowe and B. L. Pierce, “Sensitivity of the weighting summation decision method to incorrect application,” *Socioecon. Plann. Sci.*, vol. 16, no. 4, pp. 173–177, 1982.

[38] J. F. Allen and L. F. Allen, “Maintaining knowledge about temporal intervals,” *Commun. ACM*, pp. 832–843, 1983.

[39] Detlef Zimmer and Rainer Unland. "On the semantics of complex events in active database management systems". In *Proceedings of15th International Conference on Data Engineering*, pages 392–399. IEEE, 1999.

[40] Automatic Discovery and Integration of Urban Data Streams: The ACEIS Middleware, Feng Gao, Muhammad Intizar Ali, Edward Curry, Alessandra Mileo, *Journal on Data Semantics (JoDS)*, 2015, (submitted Feb., 2015, revision requested)

[41] QoS-aware Stream Federation and Optimization based on Service Composition, Feng Gao, Muhammad Intizar Ali, Edward Curry, Alessandra Mileo, *ACM Transactions on Internet Technology (TOIT)*, 2015, (major revision submitted May., 2015, under 2nd review)

[42] 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.

[43] QoS-aware Complex Event Service Composition and Optimization using Genetic Algorithms, Feng Gao, Edward Curry, Muhammad Intizar Ali, Sami Bhiri, Alessan- dra Mileo *ICSOC , Proceedings of the 13th International Conference on Service Oriented Computing (ICSOC 14’)*, France, 2014

[44] Complex Event Service Provision and Composition based on Event Pattern Match- making, Feng Gao, Edward Curry, Sami Bhiri, *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS 14’)* , India, 2014

[45] 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), Italy, 2014