Future Generation Computer Systems 76 (2017) 561–581

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Automated discovery and integration of semantic urban data streams: The ACEIS middleware

Feng Gao[a,b,c,*], Muhammad Intizar Ali[c], Edward Curry[c], Alessandra Mileo[c,d]

[a] Department of Computer Science, Wuhan University of Science and Technology, China
[b] Hubei Province Key Laboratory of Intelligent Information Processing and Real-time Industrial System, China
[c] Insight Centre for Data Analytics, National University of Ireland, Galway, Ireland
[d] Insight Centre for Data Analytics, Dublin City University, Ireland

HIGHLIGHTS

- We present the architecture of the Automated Complex Event Implementation System.
- We introduce a Complex Event Service (CES) Ontology, and demonstrate its usage.
- We define the formal semantics of the event in CES and align it with RSP semantics.
- We implement a query transformation system to create RSP queries from CES annotations.
- We show the usage of ACEIS in Smart City and optimize its capacity for concurrent users.

ARTICLE INFO

Article history:
Received 15 April 2016
Received in revised form
8 December 2016
Accepted 1 March 2017
Available online 30 March 2017

Keywords:
Semantic Web
Complex events
Service computing
RDF Stream Processing

ABSTRACT

With the growing popularity of Internet of Things (IoT) technologies and sensors deployment, more and more cities are leaning towards smart cities solutions that can leverage this rich source of streaming data to gather knowledge that can be used to solve domain-specific problems. A key challenge that needs to be faced in this respect is the ability to automatically discover and integrate heterogeneous sensor data streams on the fly for applications to use them. To provide a domain-independent platform and take full benefits from semantic technologies, in this paper we present an Automated Complex Event Implementation System (ACEIS), which serves as a middleware between sensor data streams and smart city applications. ACEIS not only automatically discovers and composes IoT streams in urban infrastructures for users' requirements expressed as complex event requests, but also automatically generates stream queries in order to detect the requested complex events, bridging the gap between high-level application users and low-level information sources. We also demonstrate the use of ACEIS in a smart travel planner scenario using real-world sensor devices and datasets.

1. Introduction

An increasing number of cities have started to embrace the idea of smart cities and are in the process of building smart city infrastructure for their citizens [1]. Such infrastructures, including sensors, open data platforms and smart city applications, can improve the day to day life for the citizens. A typical example of smart city applications is the provision of real-time tracking and timetable information for the public transport within the city.[1] The city of Aarhus provides an open data platform called ODAA,[2] which contains city related information generated by various sensors deployed within the city, e.g., traffic congestion level, air quality and trash-bin level etc. ODAA also encourages usage of their open data platform for building smart city applications. In the foreseeable future, more and more urban data will be made available. The enormous amount of data produced by sensors in our day to day life needs to be harnessed to help smart city applications taking smart decisions on-the-fly.

However, despite the increasing amount of infrastructures and datasets available, the uptake of smart city applications is

* Corresponding author at: Room 301, Unit 3, Building 9; Yang Guang Xin Yuan; JiangHan District, Wuhan City, China.
E-mail addresses: feng.gao86@wust.edu.cn (F. Gao), ali.intizar@insight-centre.org (M.I. Ali), edward.curry@insight-centre.org (E. Curry), alessandra.mileo@insight-centre.org (A. Mileo).

[1] Live bus arrivals in London: http://countdown.tfl.gov.uk/#/.
[2] Open Data Aarhus: http://odaa.dk.

http://dx.doi.org/10.1016/j.future.2017.03.002
0167-739X/© 2017 Published by Elsevier B.V.

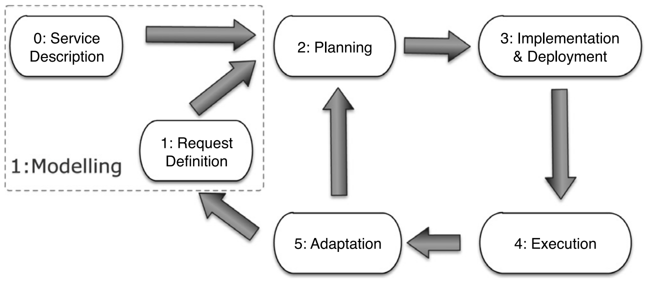


Fig. 1. Event service life-cycle.

hindered by various issues, such as the difficulty of discovering the capabilities of the available infrastructure and once discovered, integrating heterogeneous data sources and extracting up-to-date, reliable information in real-time. Complex Event Processing (CEP) [2,3] has matured from the last few decades that aggregates low-level data and provide abstracted high-level information. Recently, semantic event processing and RDF Stream Processing (RSP) [4,5] have been studied to bring semantics into CEP and deal with the data heterogeneity. Like most CEP solutions, existing RSP engines assume the streams used in the queries are identified and do not address the problem of discovering proper stream sources on-demand. Moreover, various RSP platforms have been created but a unified RSP syntax and semantics have yet to be established [6], and hence, collaborations between different RSP platforms are difficult. We plan to address this issue by providing RSP capabilities as semantically described services and aligning the formal semantics of different RSP engines.

We choose the service-oriented paradigm for enabling a collaborative, on-demand and cross-platform RSP, mainly because this way we can decouple RSP providers and consumers. Semantic Web Service (SWS) have been discussed extensively in service computing. SWS transcends conventional Web Services by applying Semantic Web techniques to realise automatic service discovery and composition [7]. However, existing SWS approaches do not cater complex event services While existing semantic service discovery and composition approaches (e.g., WSMO,[3] OWL-S[4]) show great potentials in service discovery and composition compared to syntactical service discovery [7], they are based on *Input, Output, Precondition and Effect*, e.g., in [8]. However, the functionalities of event services are determined by the semantics of the events they deliver, which is captured by the event patterns defined within an event algebra [9]. A pattern-based composition is needed for complex event services, which is not available in state-of-the-art service composition mechanisms. Apparently we are not the first that try to enable service-oriented event processing. In [10] an Enterprise Service Bus (ESB) based architecture was proposed. We essentially seek to address a similar problem as in [10], but in the context of RSP rather than conventional CEP.

In this paper, we present the Automatic Complex Event Implementation System (ACEIS), which is an automated discovery and integration system for urban data streams. We design a semantic information model to represent complex event services (as an extension of OWL-S ontology) and utilise this information model for the discovery and integration of sensor data streams. ACEIS assumes that all available sensor data streams are annotated using the Semantic Sensor Network (SSN) ontology[5] and stored in a repository. Various Quality of Service (QoS) and Quality of Information (QoI) metrics are also annotated for each sensor data stream. ACEIS receives an event service request described using our complex event service information model and automatically discovers and composes the most suitable data streams for the particular event request. ACEIS then transforms the event service composition into a stream query to be deployed and executed on a stream engine to evaluate the complex event pattern specified in the event service request. In summary, ACEIS is a middleware for managing the life cycle of event services, which includes the modelling, planning, implementation, execution and adaptation. Fig. 1 illustrates the life cycle of event services (by the analogy to Web Service life cycle). Our previous work have discussed the modelling [11], planning [12] and adaptation [13] aspects. In this paper, we present the big picture of ACEIS to show how different parts come together, with a focus on how the implementation is carried out for event services, and how the execution can be optimised. The contributions of this paper can be summarised as below:

∗ We present our Automated Complex Event Implementation System serving as a middleware between Smart City applications and sensor data streams and we provide an overview of its components and their interactions (Section 4).
∗ We describe the formal semantics of the event patterns in CES and compare it with the query semantics of semantic event processing systems to ensure a correct query transformation and evaluation (Section 6).
∗ We implement an automatic query transformation system to formulate continuous queries over semantic sensor data streams based on the alignment of event and query semantics (Section 7).
∗ We demonstrate how ACEIS is used in a Smart City Application scenario and provide evaluation and optimisation for the capacity of ACEIS, with regard to handling concurrent user queries (Sections 8, 9).

Structure of the paper: In Section 2 we introduce the background of our work (including RDF Stream Processing and Semantic Web Service) and then compare our work with the state-of-the-art. In Section 3, we present some Smart City scenarios, together with various types of sensor data streams that can be used in these scenarios as well as the challenges faced by smart city applications. We present the overall architecture of our system (ACEIS) in Section 4. A brief description of the sensor data streams discovery and integration is provided in Section 5. Section 6 lays down the formal semantics of the complex events modelled in ACEIS. Section 7 discusses our automated query transformation algorithm based on the event semantics and stream query semantics. Section 9 discusses the optimisation techniques for handling concurrent queries in ACEIS, before concluding in Section 10.

Before we move on to the next section, we provide the definitions of the terms used in this paper in Table 1.

2. Related work

In this paper, we focus on providing on-demand, cross-platform RSP using Service Oriented Architecture. In this section, we first introduce RSP and SWS as the context of our work. Publish–Subscribe Systems are also relevant for this paper, since they also discuss how different event processing results can be

[3] Web Service Modeling Ontology: http://www.wsmo.org/.
[4] OWL-S ontology: http://www.w3.org/Submission/OWL-S/.
[5] SSN ontology: http://www.w3.org/2005/Incubator/ssn/ssnx/ssn.

Table 1
Concepts and definitions relevant for event services.

Concepts	Definitions	Examples
Event	"An occurrence within a particular system or domain…"—*Event Processing in Action* [3].	Any arrival or non-arrival of new data, or information derived from those data, in an information system
Primitive event	"An event that is not viewed as summarizing, representing, or denoting a set of other events."—*EPTS*[a]	A traffic sensor observation reporting the vehicle count and average speed on a street segment.
Complex event	"An event consisting several different event instances"—*Event Processing in Action* [3]. "An event that summarises, represents, or denotes a set of other events."—*EPTS*	A traffic jam event detected from traffic sensor readings.
Event pattern	"A template containing event templates, relational operators and variables."—*EPTS*	A set of rules specifying how the traffic jam is detected from sensor readings, e.g., 80% of the sensors have reported high vehicle count and low average vehicle speed during the past 30 min repeatedly.
Service	"A service is a self-contained, logical representation of a repeatable business activity that has a specified outcome", "is a 'black box' to the consumer of the service"—*The Open Group*[b]	A data service provided via REST APIs allowing citizens to query real-time status of city infrastructures.
Event service	An asynchronous notification service that accepts subscriptions from event consumers and delivers events.	A service publishing city events to citizens based on their subscriptions.
Complex Event Service (CES)	An event service that delivers complex events detected by an underlying event engine for its consumers during the subscription, with the event pattern(s) of the complex event(s) published as part(s) of its service description.	An event service publishing traffic jam notifications.
Primitive Event Service (PES)	An event service not equipped with CEP capability or does not describe the event pattern in the service description, i.e., an event service that is not a CES.	An event service publishing directly traffic sensor readings.
Event Service Network (ESN)	A network consisting a set of interconnecting event services.	The traffic jam service, the traffic reading service, and the network allowing the former to utilise the latter.

[a] Event Processing Technical Society (EPTS): http://www.ep-ts.com/, last accessed: Dec. 2015.
[b] Open Group's definition for service: https://www.opengroup.org/soa/source-book/soa/soa.htm, last accessed: May, 2015.

shared among event consumers. Finally, we compare our work with some previous efforts on on-demand RSP/CEP.

2.1. RDF Stream Processing

RDF Stream Processing (RSP) is an emerging research area that focuses on processing semantically annotated, continuously streaming data. The vision of RSP is to perform real-time reasoning and analysis over data streams and facilitate online knowledge extraction. ETALIS [14] is one of the early attempts that realises RDF Stream Processing using Prolog as the underlying reasoning engine. ETALIS implements a set of CEP operators such as sequence, negation and logical conjunction. C-SPARQL is another RSP engine that builds upon Apache Jena libraries. Both ETALIS and C-SPARQL took a black-box approach. More recently, CQELS implements a white-box RSP approach, which provides native operator routeing mechanisms and optimisations. Despite current efforts, RSP still faces many challenges, such as coping with distributed computing environments [15] and handling complex reasoning tasks [16]. Also, some limitations regarding stability and ability to process multiple streams have been reported in CityBench [17]. Moreover, existing RSP platforms use different query languages and execution semantics [18], which hinders them from communicating and collaborating with each other.

2.2. Semantic Web Services

According to [19], WSDL concepts are familiar to software engineers thus they can easily implement and access services using WSDL. However, WSDL services are notorious for the lack of automated support for service discovery and composition [20,21], because of lacking the semantic description of service capabilities and consumers' goals as well as the reasoning ability over the capabilities and goals. Semantic Web Service (SWS) is a research area that brings together web service and Semantic Web technologies. SWS enriches web services with knowledge representations and reasoning techniques. Semantic enrichments for service descriptions, including SAWSDL,[6] WSMO and OWL-S and others, are used to facilitate automatic service discovery and composition. In SAWSDL, *modelReference* can attach to *portTypes* and message data types to indicate the category of operations and messages. Lifting and lowering schema are used to transform input and output data. In this way, composing web services based on the semantics of IO messages are made possible. However, it does not go beyond providing semantics to the service interface. In WSMO and OWL-S, the semantics of input, output, precondition and effects are captured by using ontologies and axioms. Non-functional properties (service profile) are also captured.

Service discovery and indexing based on semantic similarity between a service request and a service description can be found in [22–26]. Semantic service composition based on Artificial Intelligence (AI) planning and forward/backward chaining algorithms can be found in [27–30]. The above mentioned semantic service discovery and composition takes into account only the functional aspects of services. QoS aware service composition and optimisation is NP-hard [31]. Various techniques, e.g., [32,31,33–35], have proposed different heuristics to solve the problem efficiently.

2.3. Publish–subscribe systems

Reusing event queries/subscriptions is discussed in many publish–subscribe systems, including content-based event overlay networks [36–42] and CEP query optimisation [43,44]. In event overlay networks, event subscriptions are reused to facilitate the "downstream replication" and "upstream evaluation" principles (as described in [36]) and reduce the traffic over the network. In event query rewriting and optimisation, sub-queries can be delegated to existing event processing nodes/agents when their patterns match, in order to reduce processing burden of event engines.

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

Although the above works in event overlay networks and query rewriting share some objectives with our work in terms of improving the network and event processing efficiency, this paper is different because (1) we do not focus on routing algorithms which are central parts of event overlay network research, all nodes in the event service network can host both event producers and consumers and they are visible to all other peers and (2) we do not re-order query operators in a way such that CPU usage and latency can be minimised, which is central to query rewriting techniques. Instead, we develop means to create event service compositions based on the semantic equivalence and reusability of event patterns, and then composition plans are transformed into a set of federated stream reasoning queries, enabling a semantic complex event processing over distributed service networks.

2.4. *On-demand/unified event stream processing*

The service-oriented computing paradigm fits our need for a platform-independent, on-demand RSP due to its capability of hiding implementation details while exposing communication interfaces. Early attempts at integrating event processing into service computing can be found in [45], where an Event-Driven Service Oriented Architecture (EDSOA) is proposed. EDSOA leverages event processing to trigger Web Services but they do not address event service discovery and composition. The work in [10] provides complex event processing as regular services on an ESB and implements a greedy algorithm to choose event services with lower costs. However, [10] did not address the service satisfiability problem, that is, a pattern-based event service selection is not realised. Moreover, although it provides an event algebra, how exactly this algebra can map to existing CEP systems is not detailed.

The need for a unified event algebra (or query semantics) has been acknowledged in many recent works in semantic event processing (or RSP) [6]. Indeed, a unified event processing language is indispensable for a cross-platform event/stream processing. EVA [46] builds on and extends the Zimmer/Unland model [47] (which is also the basis of our event algebra) and provides precise event semantics. The proposed event algebra is implemented on A-mediAS [48]. Profile and result transformations from EVA to different target CEP systems are described [49] but an on-demand collaboration for these systems is not realised

There exist several on-demand CEP/RSP systems that do not rely on SOA as well. Semantic Streams [50] is inspired by SWS and uses a Prolog-based system to infer proper streams to address user interests. It supports reasoning on both functional and non-functional properties (including geospatial reasoning), but the composition relies on the stream type specification, not the exact processing pattern. In H2O [51] a hybrid processing mechanism is proposed, in which persistent queries that keeps monitoring fine-grained data (online queries) and infrequent queries over occasional events (on-demand queries) are modelled and processed on different levels. The online queries provide partial results to be used by on-demand queries. The benefit of the hybrid approach is that the on-demand queries do not have to store a lot of irrelevant events thus improves the efficiency. However, this architecture limits the expressiveness and flexibility of on-demand queries. Moreover, how to decide whether a query must be online or on-demand is not clear. In Dyknow [52], the authors leverage C-SPARQL as semantic event processing units. They also annotate streams on the meta-level (same as our approach) to facilitate on-demand stream discovery. However, Dyknow streams use proprietary stream formats, which is basically a vector of values with a timestamp and duration. Although methods are provided to transform RDF streams from/to Dyknow streams, this rigid format could limit the flexibility of stream content description. Moreover, the described stream matching mechanism only caters for simple event streams. Table 2 summarises the comparison of the on-demand/unified event stream processing.

3. Smart city applications

In this section, we first describe some sample scenarios in Smart City applications, then, we discuss the different types of sensor data streams which can be potentially utilised by smart city applications. Finally, we discuss the requirements and challenges faced by these smart city applications consuming sensor data streams.

3.1. *Sample scenarios in a smart city*

In CityPulse 101 scenarios,[7] different Smart City applications are described, including traffic management and travel planning, smart health, street lamp control, smart tourism etc. These scenarios are also ranked considering multiple dimensions, such as data availability, the need for integrating the components in CityPulse framework, usefulness for citizens and city administration. In this paper, we mainly consider the travel planning application. The essence of travel planning is to gather information that helps bring a citizen from point 'A' to 'B' in the city while considering real-time traffic condition, environmental condition, and different user preferences. We consider this use case not only because it is highly ranked on 101 scenarios, but also the fact that travel planning has an inherent level of complexity and it presents many typical challenges in Smart City applications (we will elaborate on this in Section 3.3). Moreover, it is relevant for almost all inhabitants in the city and could be a big problem in large cities.

3.2. *Smart city data streams*

Sensors are nowadays used widely in urban environments [53]. IoT technologies not only provide an infrastructure for sensor deployment but also provide a mechanism for better communication among these sensors. The continuously growing amount of data produced by these sensors opens tremendous opportunities but it is still under-explored: in order to unlock the potential hidden in this data deluge, there is a need to support more interoperable and faster development of applications that can find, capture and process it in a scalable way. Besides IoT streams, data streams from the social media can also be utilised in Smart City Applications. Urban data streams can be categorised into three different categories: physical sensors, mobile and wearable sensors, and social media data streams.

3.2.1. *Physical sensors*

Various sensors are being deployed by city administration with an aim to closely observe and monitor the city infrastructure. Traffic congestion, air quality, temperature, water pressure and trash bin level sensors are a few examples of the sensors deployed within the modern smart cities. Additionally, various sensors are being deployed in smart buildings to detect critical events happened therein, according to [54]. An example of physical sensor in travel planning is pair-wised Bluetooth sensors.[8] These sensors leverage Bluetooth connections (or detections) to the devices in vehicles to identify when a vehicle has entered (detection at first point) or left (detection at the second point) the street segment.

Sensors are inherently dynamic in nature and somehow unreliable, therefore more prone to fluctuations in quality. For example, the accuracy of a sensor might be affected by its battery level [55], air temperature, humidity [56] etc.

7 CityPulse 101 scenarios: http://www.ict-citypulse.eu/scenarios/.

8 Traffic sensor in Aarhus: https://www.aarhus.dk/da/borger/Trafik/Projekter/Regulering/Bluetooth.aspx.

Table 2
Comparison of on-demand event stream processing.

Comparable Approaches	Event Algebra		Automatic Composition			Semantic Annotation		
	Formal Semantics	Transformation	Simple Event	Complex Event	QoS	Query	Stream Meta	Stream Content
Event Driven SOA								
Laliwala et al [53]	○	○	○	○	○	○	○	○
Bo et al [10]	○	○	●	⦿	⦿	○	○	○
Unified Event Algebra								
RSP-QL [6]	●	⦿*	-	-	-	○	○	●
EVA [54,56,57]	●	●	-	-	-	○	○	○
Zimmer et al [55]	⦿	○	-	-	-	○	○	○
On-demand Stream Processing								
Semantic Streams [58]	⦿	○	●	⦿	●	●	●	○
H2O [59]	●	○	●	⦿※	○	○	○	○
Dyknow [60]	●	⦿	●	⦿	○	⦿	●	●
ACEIS (this paper)	●	⦿	●	●	●	●#	●	●

*Semantics aligned to existing systems but transformation not provided.
#Query in the form of CESO Event Request annotation.
※Only for "on-demand" queries.
●: supported ○: not supported ⦿: partially supported -: irrelevant

3.2.2. Mobile and wearable sensors

Contrary to the physical sensors deployed by city administrations and organisations, sensors attached to mobile devices provide additional information about the context of their carrier, i.e., the citizens. Nowadays, a modern smartphone, owned and carried by the majority of the citizens in the smart cities is equipped with 10–15 sensors on average, including location, temperature, light and proximity sensors. As the example introduced above, smartphones can coordinate with the Bluetooth sensors and indirectly measure traffic conditions. Also, Google has been using GPS sensors in smartphones to monitor traffic.[9] Modern cars also contain sensors to continuously monitor the performance as well as to provide assistance to drivers. Many wearable sensors are gaining popularity and many people are adopting to the use of wearable sensors with wireless connection with smartphone apps while they do physical exercises.

3.2.3. Social media data streams

Due to the increasing popularity and widespread use of social media, social media streams have become an important and valuable source of information in a smart city infrastructure [57]. For example, information about city events, including traffic, accidents and even natural disaster (e.g., in [58]), can become available much earlier on Twitter feeds than on the news. Trust, reliability and provenance are major concerns over the information arising from social media streams, but various social streams analysis methods have already been developed to overcome these concerns, as surveyed in [59]. In [60], the authors introduce the concept of *Human as Sensors* based on social media and other Web 2.0 techniques. In this paper, we borrow this concept to have a coherent model and integration for urban data streams. Hereafter, the term "sensor" in this paper refers to physical sensors, mobile and wearable sensors or human sensors.

9 https://googleblog.blogspot.com/2009/08/bright-side-of-sitting-in-traffic.html.

3.3. Requirements and challenges

Smart city applications face many challenges because of highly distributed and dynamic nature of the sensor infrastructures deployed in the smart cities. Below we discuss few of the requirements and challenges faced by smart city applications based on our experience.

- **R.1: Heterogeneous data streams federation.**
  Data Federation combines heterogeneous sets of data to provide a unified view. In the context of smart city data, data federation is a key challenge due to the dynamicity and heterogeneity of various sensor streams. Querying and accessing the data in many cases will require real-time (or near-real-time) discovery and access to the streams (and their data) and the ability to integrate different kinds of heterogeneous streaming data from various sources. Smart city frameworks should provide mechanisms to (i) seamlessly integrate real world data streams, (ii) automated search, discovery and federation of data streams, and (iii) adaptive techniques to handle failovers at run-time. In the travel planning scenario, the application developer first need to integrate user location data with physical traffic sensor data. Furthermore, weather data and air pollution data may also need integration, so that an end-user can choose the type of transportation or route based on the environmental condition.
- **R.2: Large scale data stream processing and analytics.** Smart city applications not only require efficient processing of large-scale data streams but also need efficient methods to perform data analytics in a dynamic environment by aggregating, summarising and abstracting sensor data on demand. For example, in a Smart City, there could be hundreds of thousands of commuters during peak hours. This poses a challenge for the capacity of existing RSP engines. In addition, a single user may only need to subscribe to several sensors deployed in the city, for less than an hour, and apparently different trips may need different sensors, hence finding appropriate sensors on-demand is more efficient.

- **R.3: Real-time information extraction, event detection and stream reasoning**.
  Smart city applications should be able to process event streams in real time, extract relevant information and identify values that do not follow the general trends. Beyond the identification of relevant events, extraction of high-level knowledge from heterogeneous, multimodal data streams is an important feature of Smart City. When a user is travelling, high-level decisions such as re-routing must be made online based on real-time data. Also reading changes from a single sensor should not always trigger such decision, e.g., together a twitter message stating a congestion ahead, a pollution sensor reporting increasing amount of CO_2 emission and a traffic sensor showing low vehicle speed usually indicate a major congestion, but each of the event stand-alone could be inconclusive.
- **R.4: Reliable information processing**.
  Data quality issues and provenance play an important role in smart city scenarios. For example, a traffic monitoring or travel planning application may ask for results with low latency and high accuracy. When the quality of a sensor on the road is deteriorating, it may need to be replaced by another sensor, e.g., a sensor deployed on a consecutive street segment or monitoring a different lane (if multiple sensors are used for different lanes). Smart city frameworks should provide methods and techniques (i) to evaluate the accuracy, trustworthiness, and provenance of data streams, (ii) to resolve conflicts in case of contradictory information, and (iii) continuous monitoring and testing to dynamically update QoI and trustworthiness.

Aside from the above challenges, existing research also discusses other issues, e.g., privacy control regarding sharing personal data (e.g., in [61–63]), real-time actuation based on observations (e.g., in [64]). In this paper, we mainly focus on **R.1**–**R.4**, and while we do not claim ACEIS provides complete solutions to these requirements, we will show our efforts made so far in coping with these four challenges.

4. Overview of ACEIS architecture

In order to address the challenges identified in Section 3.3, a number of solutions are developed and integrated into ACEIS. We will discuss briefly the functionalities of the components in ACEIS as well as their interactions.

Fig. 2 illustrates the architecture view of ACEIS. The architecture consists of four main components, i.e., *Knowledge Base*, *Application Interface*, *Semantic Annotation* and *ACEIS Core* component.

4.1. Knowledge base

The knowledge base stores the semantic annotations for the static description of event services as well as domain ontologies to use as background knowledge. It also stores the indexing structures for event service description to facilitate efficient event service discovery and composition. Historical observations and quality analysis results are also kept in the knowledge base.

4.2. Application interface

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

These augmented user inputs are transformed into a semantically annotated complex event service request (event request for short). The event request is consumed by ACEIS core components to discover and integrate urban streams with regard to the functional and non-functional constraints specified within the event request.

4.3. Semantic annotation

The semantic annotation component receives data streams (e.g., ODAA real-time traffic sensors data) as well as static data stores (e.g., ODAA traffic sensors metadata) as inputs. It annotates syntactical information with semantic terms defined in ontologies. The outputs of semantic annotation will be semantic data streams and static semantic datastores.

With semantic annotations of both static resource and dynamic data, ACEIS gains additional data interoperability both at design time for event service discovery/composition and at runtime for semantic event detection.

4.4. ACEIS core

The ACEIS core module serves as a middleware between low-level data streams and upper-level Smart City applications. ACEIS core is capable of discovering, composing, consuming and publishing complex event processing capabilities as reusable services. We call these services (primitive or complex) event services. An example of an event service network and the interactions between different roles in the network are shown in Fig. 3. The ACEIS core consists of three major components: resource management, data federation adaptation manager. In the following, we introduce their functionalities and interactions.

4.4.1. Resource management

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

The resource management component contains two sub-components: resource discovery component and event service composer. The resource discovery component uses conventional semantic service discovery techniques to retrieve event services delivering primitive events. It deals with the primitive event requests specified within event requests. The event service composer creates service composition plans to detect the complex events specified by event requests based on event patterns. We refer readers to [11,12] for further details of the composition algorithm used by the event service composer.

4.4.2. Data federation

The data federation component is responsible for implementing the composition plan over event service networks and process complex event logics using heterogeneous data sources. The composition plan is first used by the subscription manager which will make subscriptions to the event services involved in the composition plan. Later, the query transformer transforms the

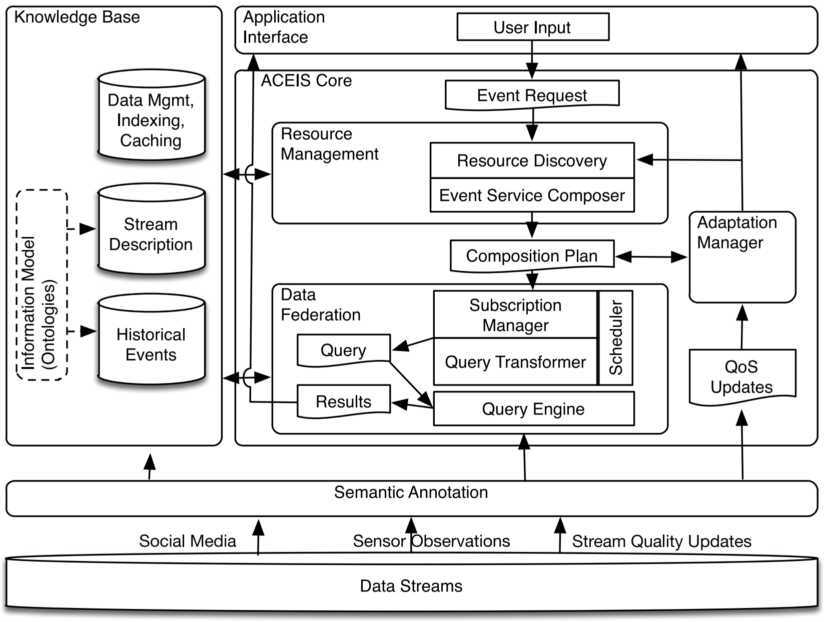


Fig. 2. ACEIS architecture overview.

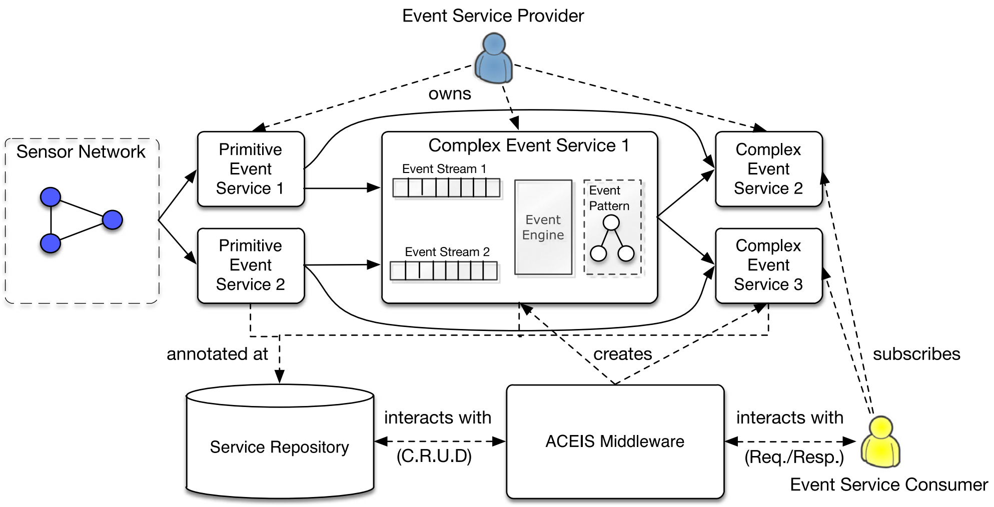


Fig. 3. Example of an event service network.

semantically annotated composition plan into a set of stream reasoning queries to be executed on a stream query engine.

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

4.4.3. Adaptation manager

The adaptation manager monitors the QoS updates for the event services and determines if the QoS properties of a deployed event service composition have violated the non-functional constraints specified in the event request. When a QoS constraint violation is detected, the adaptation manager makes an attempt to automatically find replacements for parts or whole of the deployed composition plan in order to keep the QoS performance at an acceptable level. If no possible adaptation is available, a notification is sent to the user interface, which informs the user that the QoS constraint has been violated and the attempt of automatic recovery has failed. Different adaptation strategies and their performance evaluation are discussed in [13].

5. Semantic sensor data stream discovery & integration

Sensor data streams are modelled as event services in ACEIS, and hence the discovery and integration of urban data streams are translated into event service discovery and composition problems. By providing an ontology for event services and allowing service providers to semantically annotate their service description documents, event services can obtain better data interoperability and facilitates automatic service discovery, composition and execution [65]. The complexity of semantic annotations may be hindering the adoption of Semantic Web Service (SWS) in the real-world [66]. To cope with this issue, several automatic annotation methods have been studied, e.g., by [67–69]. While facilitating automatic service annotations is out of the scope of this work, in this paper we assume the semantic annotations are provided manually by service providers or automatically by a program.

As shown in Section 1, Event service modelling and composition are steps taken before implementation and execution, and have been discussed thoroughly in our previous work. We refer interested readers to [11,12] for a detailed description on these topics. In this section, for the sake of completeness, we describe the key features of the ontology used for describing event services and event requests as well as the discovery and integration mechanism for the sensor data streams.

5.1. Complex Event Service ontology

A Complex Event Service (CES) ontology[10] had been developed to describe event services and requests. The CES ontology is an extension of OWL-S, which is an ontology to describe, discover and compose semantic web services. We choose to extend OWL-S mainly because it provides direct support for the service profile and service quality etc., so that allows us to incrementally design our ontology. We validated our ontology together with all reused ontologies using Jena 3.0[11] (RDFS reasoner) and Pellet 3.0[12] (OWL2 DL reasoner). The validity reports showed no inconsistencies. Fig. 4 illustrates the overview of the CES ontology.

An event service is described with a *Grounding* and an *EventProfile*. The concept of *Grounding* in OWL-S informs an event consumer, how to access the event service by providing information on service protocol and message formats etc. An *EventProfile* is comparable to the *ServiceProfile* in OWL-S, which describes the events transmitted by the service. The property *hasEventSource* links an event service to its event source, which could be a sensor described in the SSN ontology, or other data sources described in domain ontologies.

An *Event Profile* describes a type of event with a *Pattern* and *Non-Functional Properties* (NFP). A *Pattern* describes the correlations between a set of member events involved in the pattern. An event pattern may have other patterns or (primitive) event services as sub-components, making it a tree structure. An event profile without a *Pattern* describes a primitive event service, otherwise, it describes a complex event service. *NFP* refers to the QoI and/or QoS metrics, e.g., precision, reliability, cost and etc., which are modelled as subclasses of *ServiceParameter* in OWL-S.

We consider the temporal relationships captured by an *Event Pattern* to 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 repetition. Besides temporal relations, event pattern may also specify causal relations between patterns and sub-patterns or member event services using transitive property `hasSubPattern`, which is an important property for reasoning over the event provenance. Data constraints in event patterns can be specified with `Filters` and `Selections`. A sliding `Window` specifies the size of the event instance sequence kept in memory. Fig. 5 reveals more details on the event pattern model. Throughout this paper, we use tree structures to represent event patterns.

An *EventRequest* is an incomplete *EventService* description, without specific bindings to the set of federated event services used by the requested complex event while capturing the desired capability of the service. *Constraints* can be specified by users to declare their requirements on the event pattern and NFPs in *EventRequests*. *Preferences* can be used to specify a weight between 0 and 1 over different quality metrics representing users' preferences on QoS metrics: a higher weight indicates the user cares more on the particular QoS metric.

It is worth noticing that currently, we have not extended OWL with respect to the temporal logics implied by the event patterns, thus reasoning on event patterns is not supported. However, CESO still allows us to match event service descriptions based on taxonomical or causal relations, which we will show later.

5.2. Primitive event service discovery

In the context of Smart City applications, a sensor data stream is an atomic unit for data stream discovery and integration. It is described as a *PrimitiveEventService* (PES) in the CES ontology, which has an event source as a *Sensor* device in the SSN ontology. The CES ontology is mainly used to describe the non-functional aspects of the PES, including service quality parameters and service groundings. The SSN ontology is used to describe the functional aspects, including *ObservedProperties* and *FeatureOfInterest*.

A sensor service description s_d is defined as a tuple $s_d = (t_d, g, q_d, P_d, FoI_d, f_d)$, where t is the sensor event type, g is the service grounding, q_d is a QoS vector describing the QoS values, P_d is the set of *ObservedProperties*, FoI_d is the set of *FeatureOfInterests* and $f_d : P_d \rightarrow FoI_d$ is a function correlating observed properties with their feature-of-interests. Similarly, a sensor service request is denoted $s_r = (t_r, q_r, P_r, FoI_r, f_r, pref, C)$. Compared to s_d, s_r do not specify service groundings, q_r represents the constraints over QoS metrics, *pref* represents the QoS weight vector specifying users' preferences on QoS metrics and C is a set of functional constraints on the values of P_r. s_d is considered a match for s_r *iff* all of the following three conditions are true:

- t_r subsumes t_d,
- q_d satisfies q_r and
- $\forall p_1 \in P_r, \exists p_2 \in P_d \implies T(p_1)$ subsumes $p_2 \wedge f_r(p_1) = f_d(p_2)$, where $T(p)$ gives the most specific type of p in a property taxonomy.

Listing 1 shows a snippet of the traffic sensor (from the travel planning scenario) description in turtle syntax. The traffic sensor monitors the estimated travel time, vehicle count and average vehicle speed on a road segment. Listing 2 shows a snippet of a sensor service request matched by the traffic sensor service. When the discovery component finds all service candidates suitable for the request, a Simple-Additive-Weighting algorithm [12] is used to rank the service candidates based on q_d, q_r and *pref*. This matching and ranking process can be accelerated by using a SPARQL query as a filter. For example, leveraging the reasoning support for subsumption relation, the query in Listing 3 can find appropriate sensor types (including sub-types), and using SPARQL filters, it can find the sensors with acceptable QoS.

5.3. Complex event service discovery and composition

To discover and integrate composite sensor streams for complex event service requests, the event patterns specified in the complex event service requests/descriptions need to be considered. State-of-the-art SWS planning and composition approaches are based on the Input, Output parameters, Preconditions and Effects (IOPE). In this 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

[10] CESO published online at: http://citypulse.insight-centre.org/ontology/ces/.
[11] https://jena.apache.org/index.html.
[12] https://github.com/Complexible/pellet.

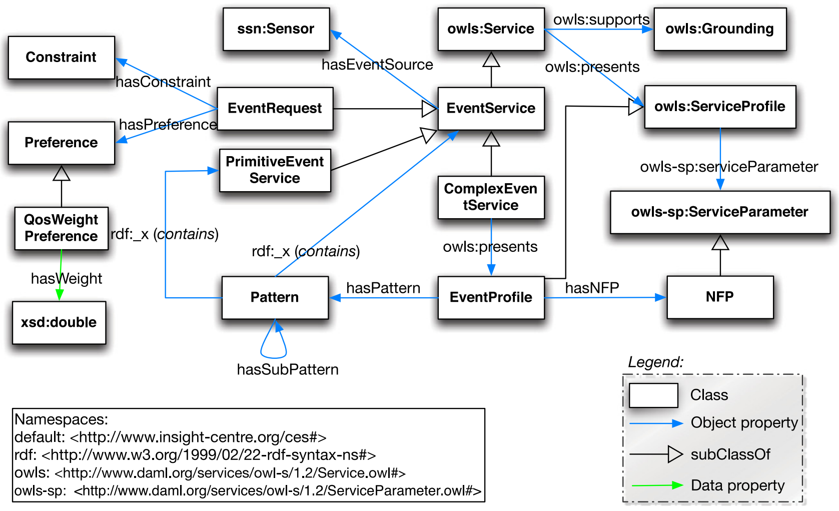


Fig. 4. Complex Event Service (CES) ontology overview.

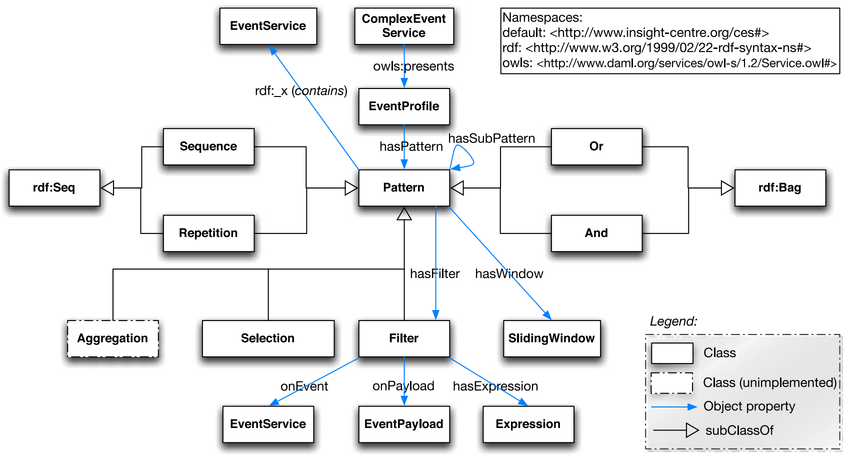


Fig. 5. Complex event pattern ontology.

```
:sampleTrafficService a ces:PrimitiveEventService;
     owls:presents :sampleProfile ;
     owls:supports :sampleGrounding;
     ces:hasEventSource :sampleTrafficSensor.

:sampleTrafficSensor a ssn:Sensor;
     ssn:observes [ a ct:AverageSpeed; ssn:isPropertyFor :FoI_1],
                 [ a ct:VehicleCount; ssn:isPropertyFor :FoI_2],
                 [ a ct:EstimatedTime; ssn:isPropertyFor :FoI_3].

:sampleProfile a ces:EventProfile ;
     owls:serviceCategory [ a ct:TrafficReportService ; owls:serviceCategoryName "traffic_report"^^xsd:string].
```

Listing 1: Traffic sensor service description

```
:sampleRequest a ces:EventRequest;
     owls:presents :requestProfile; ces:hasEventSource :requestSensor.

:requestSensor a ssn:Sensor;
     ssn:observes [ a ct:EstimatedTime; ssn:isPropertyFor :FoI_3].

:requestProfile a ces:EventProfile ;
     owls:serviceCategory [ a ct:TrafficReportService; owls:serviceCategoryName "traffic_report"^^xsd:string].
```

Listing 2: Traffic sensor service request

```
PREFIX ces: <http://www.insight-centre.org/ces#>
PREFIX ssn: <http://purl.oclc.org/NET/ssnx/ssn>
PREFIX owls: <http://www.daml.org/services/owl-s/1.2/Service.owl#>
SELECT ?eventService? WHERE { ?eventService owls:presents ?profile.
   ?profile owls:serviceCategory :TrafficReportService.
   ?eventService ces:hasEventSource ?sensor. ?sensor ssn:observes ?property.
   ?property a :AverageSpeed. ?property ssn:isPropertyOf :FoI_1.
   ?profile owls:serviceParameter ?qos. ?qos a qoi:Correctness. ?qos qoi:value ?qosV. }
FILTER (?qosV >=0.9)
```

Listing 3: SPARQL query for sensor discovery

applying intermediate effects). Typically, the reasoning procedure is carried out in a backward chaining style, i.e., starting from the target outputs and effects, find possible tasks that fulfil part of the required inputs and preconditions.

However, such IOPE-based service planning cannot be easily applied to CESs because (1) different event detection tasks may have the same types of inputs and outputs, but with different event semantics. For example, the traffic congestion events detected within different time durations or using different threshold values have different meanings, determining the data flow for event service compositions using type-based matchmakings is not feasible, and (2) it is not straightforward to define the precondition and effect of an event detection task. For example, an IOPE-based complex event service composition attempt is made by [70], in their approach, the logical correlations in event patterns (conjunctive or disjunctive relations of event types) are extracted as preconditions and handled by rule-based service middleware while the temporal correlations are left to the event engine, and the effects are simply modelled as the creation of the complex event types. We argue that both logical and temporal correlations should be processed in a coherent manner to realise planning based on event patterns, and the matchmaking of preconditions and effects in their approach are still based on event types. In ACEIS, a pattern-based and QoS-aware event service composition is facilitated using the techniques from [11,12], In the following we briefly describe the process of integrating composite sensor data streams.

In the context of integrated sensor stream discovery and composition, the definition of sensor stream description is extended to denote composite sensor stream descriptions $S_d = (ep_d, Q_d, G)$,where ep_d consists of a set of sensor stream descriptions s_d and/or a set of composite sensor stream descriptions S'_d, and a set of event operators including *Sequence*, *Repetition*, *And*, *Or*, *Selection*, *Filter* and *Window*, q_d is the aggregated QoS metrics for S_d and G is the grounding for the composite sensor stream. Similarly, a complex event service request is denoted as $S_r = (ep_r, Q_r, pref)$, where ep_r is a *canonical* event pattern consisting of a set of primitive sensor service requests s_r and a set of event operators, Q_r describes the QoS constraints for the requested complex event service and *pref* specifies the weights on QoS metrics.

An S_d is a match for S_r *iff* ep_d is *semantically equivalent* to ep_r and Q_d satisfies Q_r. When no matches are found during the discovery process for S_r, it is necessary to compose S_r with a set of S_d and/or s_d which are *reusable* to S_r. Informally, these (composite) sensor streams describe a part of the semantics of ep_r and can be reused to create a composition plan, which contains an event pattern with concrete service bindings. The composition plan can be used as a part of the event service description for the composed event service. The discovery or composition results can be ranked with regard to the QoS metrics and preferences in the same way as sensor stream discovery. Listing 4 shows a snippet of a sample complex event service request with an event pattern and some NFP constraints.

CES discovery and composition can benefit from the semantic annotations the same way as the PES discovery, i.e., using SPARQL queries as preliminary filters for the sensors involved. In addition, the *hasSubPattern* property recursively define on the *Pattern* can be used to infer causal relations between event patterns, based on the rules define in Listing 5 and a sample query in Listing 6.

6. Formal semantics of event patterns in CES

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 the CES ontology. In this section, we lay down the formal semantics. We first discuss a meta-model for complex event semantics. Then, 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 design of the semantics of event patterns in the CES ontology. Finally, we present the abstract syntax for the event patterns in CES.

6.1. Meta-model of event semantics

In [9] a meta-model is proposed for defining the formal semantics of complex events, i.e., what does a complex event pattern mean and how to detect this event pattern over an *Event Instance Sequence* (EIS). According to [9] the semantics of complex events can be defined by answering three basic questions: (1) how to use a limited set of operators, constructs and descriptors to specify various complex event types (i.e., complex event patterns) unambiguously, (2) how to determine which subset of the EIS belongs to a complex event type when there are more than one subsets satisfying the constraints specified by the complex event types and (3) whether an event instance can be used in multiple EISs mapping different complex event types. We thus distinguish between three basic dimensions for describing event semantics: *Event Type Pattern*, *Event Instance Selection* and *Event Instance Consumption*, for answering these three questions, respectively. On top of these three basic dimensions, an additional dimension is whether events are considered instantaneous or lasting for an interval. We call this dimension *Event Duration*. In the following, we elaborate on the details of each dimension.

6.1.1. Event Duration

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

6.1.2. Event Type Pattern

An *Event Type Pattern* can be categorised based on 3 dimensions: *Operators*, *Coupling* and *Context Condition*. The operators specify temporal constraints over EISs, including binary operators:

```
:SampleEventRequest a ces:EventRequest;
   owls:presents :SampleEventProfile.

:SampleEventProfile rdf:type owls:EventProfile;
   ces:hasPattern [ rdf:type ces:And, rdf:Bag;
                 rdf:_1 :locationRequest; rdf:_2 :seg1CongestionRequest; rdf:_3 :seg2CongestionRequest;
                 rdf:_4 :seg3CongestionRequest; ces:hasWindow "5"^^xsd:integer];
   ces:hasConstraint [ rdf:type ces:NFPConstraint;
                    ces:onProperty ces:Availability;
                    ces:hasExpression [ emvo:greaterThan "0.9"^^xsd:double]],
                    [ rdf:type ces:NFPConstraint;
                    ces:onProperty ces:Accuracy;
                    ces:hasExpression [ emvo:greaterThan "0.9"^^xsd:double]].
```

Listing 4: Complex event service request

```
[Rule1: (?x rdfs:member ?y ) -> (?x ces:hasSubPattern ?y )]
[Rule2: (?ep1 ces:hasSubPattern ?s ) (?s owls:presents ?p ) (?p ces:hasPattern ?ep2 )
   -> (?ep1 ces:hasSubPattern ?ep2 )]
```

Listing 5: Rules to entail sub-pattern

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

Listing 6: Tracking causal relation via SPARQL query

Sequence (;), Simultaneous (==), Conjunction ($\wedge$), Disjunction ($\vee$), unary operator *Negation ($\neg$)* and n-ary operator *Repetition*.

For two event types E_1, E_2, $;(E_1, E_2)$ indicates the timestamps of event instances of type E_1 are older than the timestamps of event instances of type E_2[13]; $==(E_1, E_2)$ indicates the timestamp(s) of the event instances are equal; $\wedge(E_1, E_2)$ and $\vee(E_1, E_2)$ indicate both and at least one of the instances of E_1 and E_2 should occur regardlessly of the temporal order, respectively.

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 practice, 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 $\wedge(;(E_1, \ldots, E_n))^r$ 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 jth 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 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 a non-continuous event pattern (non-continuous)$E_5 := ;(E_1, E_2)$ but cannot trigger (continuous)$E_6 :=; (E_1, E_2)$.

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.).

6.1.3. *Event Instance Selection*

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.

6.1.4. *Event Instance Consumption*

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.

6.2. *Comparison of existing approaches*

In this section we compare the event/query semantics in existing CEP/stream processing systems using the meta-model presented in Section 6.1 and elaborate on the semantics we use in ACEIS. In [72] 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. In Table 3 we compare the event semantics used in RDF Stream Processing (RSP) engines, including ETALIS [14], C-SPARQL [73] and CQELS [74], as well as in a conventional CEP system, i.e., BEMN [75], and in ACEIS. The event pattern definition language used in ACEIS is designed to be a user-friendly, high-level language (extending the graphical notations from BEMN) 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.

13 When considering overlaps for interval-based events the sequence operator can have more variants e.g.: meets, finishes and participates etc. see [71].

Table 3
Comparison of event semantics.

Dimensions of Event Semantics	ETALIS	C-SPARQL	CQELS	BEMN	ACEIS
Event Duration					
Instantaneous	●	●	●	●	●
Interval	●	○	○	○	○
Event Type Pattern					
Operators					
Sequence	●	◉	○	●	●
Simultaneous	●	◉	○	○	◉
Conjunction	●	●	●	●	●
Disjunction	●	●	○	●	●
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

6.2.1. *Event Duration*

All investigated approaches support using instantaneous events, i.e., annotating events and triples with a single timestamp. Only ETALIS fully supports interval-based events, since it allows triples to be annotated with a start and end timestamp. C-SPARQL partially supports intervals for complex events, i.e., events consists of multiple triples with different timestamps. To capture the interval for such complex events in 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.

6.2.2. *Event Type Pattern*

The *Sequence* operator is supported by all investigated approaches except for CQELS. The *Simultaneous* operator is directly supported by ETALIS using the *EqJoin* operator extended from SPARQL *join* and indirectly supported by C-SPARQL and ACEIS by comparing timestamps of events and triples. The *Conjunction* and *Disjunction* operators are supported by all investigated approaches except CQELS, since the "OPTIONAL" keyword is not implemented in the currently released version[14] of CQELS. *Negation* is directly supported by BEMN using *Inhibition* and indirectly supported by ETALIS and C-SPARQL using the combination of *LeftJoin* operator and *bound* filters. CQELS does support the "NOT EXISTS" filter in SPARQL 1.1, since there is no support for sequential pattern in CQELS and recall that the semantics of negation in event patterns are typically used within a duration, i.e., implying it needs to be used together with sequential pattern, we do not consider CQELS supports the negation semantics in Section 6.1.2. Currently, ACEIS 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 ACEIS 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 the same number of (e.g., n) 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. ACEIS supports both kinds of windows. All approaches support *non-continuous* coupling, i.e., irrelevant events and triples will not affect the results derived from relevant ones. All approaches support context conditions on *data* using filters.

To summarise, ETALIS can support applications that evaluate *Conjunction, Disjunction, Sequence, Simultaneous* and *Negation* patterns, which are also supported by C-SPARQL, but the *Sequence, Simultaneous* and *Negation* are indirectly supported by C-SPARQL using filter functions. CQELS only provides support for *Conjunction* pattern. This fragmentation and complementary support for different operators require to carefully analyse what expressivity and what operators are required in the application, in order to select the best system. This limitation of semantic approaches to complex event processing is well known in the semantic web community. In fact, concrete standardisation efforts to define a common model for producing, transmitting and continuously querying RDF Streams are ongoing in the W3C RSP working group,[15] and we are heavily involved in this standardisation activity.

6.2.3. *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 ACEIS 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 (see Section 4.3 in [72]) and we consider the latest events usually to be more important.

6.2.4. *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 ACEIS, 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.

6.3. *Abstract syntax of event pattern in CES ontology*

Using the CES ontology and the event semantics defined above, an event service provider can describe event services and store these service descriptions in a service repository; an event service consumer can formulate an event service query to specify his

[14] CQELS version 1.0.0 home: https://code.google.com/p/cqels/.

[15] RDF Stream Processing Community Group: https://www.w3.org/community/rsp/.

requirement on event services. In the following, we give the abstract syntax of event patterns described in the CES ontology.

An *Event Declaration* describes a (complex) event type without considering the NFPs. It is a tuple

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

where *src* is the service location where the events described by *ed* are hosted, t is the term for the domain specific event type, *ep* is the event pattern for E and D is its data payload as a set of event properties sets, e.g., timestamps, event identifier, message contents, etc. Recall we defined primitive and composite sensor service descriptions s_d and S_d in Section 5.2. Now *ed* can be seen as a generalised definition for s_d and S_d, where sensor observation properties and feature-of-interests are generalised into payloads, since a generic event may contain message payloads other than physical properties observed from the world.

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

$$ep = (w, \mathcal{E}, OP, R, S, Sel, F)$$

where

- w is a sliding window specified for *ep*, we consider w as a time duration or a number of events to be kept;
- $\mathcal{E}$ is a set of member event declarations involved in *ep*;
- for an 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), $r \in \mathcal{N}^+$ is the cardinality of repetition, $r > 1$ for repetition operators, and $r = 1$ otherwise;
- $R \subset (OP \times (OP \cup \mathcal{E}))$ is a set of asymmetric relations on operators and member events, it captures the provenance (i.e., causal) relation within *ep*, $\forall(op, n) \in R$, the execution of the operator node *op* relies on the execution result of another operator node n when $n \in OP$, or the occurrence of an event declaration node n when $n \in \mathcal{E}$;
- $S \subset (OP \cup \mathcal{E}) \times (OP \cup \mathcal{E})$ is a set of asymmetric relations on operators and member events, it gives the temporal order within *ep*, $\forall(n_1, n_2) \in S, \exists n \in OP \wedge (n, n_1), (n, n_2) \in R \wedge n.t_{op} = (Seq|Rep_o|Rep_n)$ where n_1, n_2 are two nodes in *ep*, also, the occurrence of n_1 (if $n_1 \in \mathcal{E}$) or the last member event instance that completes the execution of n_1 (if $n_1 \in OP$) should happen before the occurrence of n_2 (if $n_2 \in \mathcal{E}$) or the first member event instance that completes the execution of n_2 (if $n_2 \in OP$);
- $Sel \subseteq \bigcup_{E' \in \mathcal{E}} E'.D'$ is a set of selected properties from the payloads of member events. The selected properties are typically part of the payload D for E;
- F is a set of filters expressing 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.

It is evident that an event pattern defined according to the above semantics and syntax can be constructed by recursively appending operator and event declaration nodes as child/leaf nodes to a root operator node, thus can be organised into a tree structure, called an *Event Syntax Tree* (EST).

7. Query transformation

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 (regular and constraint validation) stream queries and registers the queries at the stream engine. In this section, the algorithms for transforming composition plans into regular semantic stream queries are discussed.

In the current ACEIS implementation, CQELS and C-SPARQL are used as the semantic stream processing engines.[16] 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 will not take too much effort to adapt to different event ontologies as long as the essential information (i.e., the 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 SSN ontology. A sample traffic sensor reading annotated as *Observation* in SSN is shown in Listing 7. 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.

7.1. Semantics alignment

To ensure the query transformation creates queries that detect the right event patterns, it is required to map the semantics of event operators to query operators. Table 4 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:

- An *Event Declaration E* in an event pattern *ep* indicate the occurrences of event instances of type E. As shown in Listing 7 the occurrences of sensor events are annotated as observations. If we use SPARQL to query the occurrences of sensor observations, a single triple pattern

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

  can suffice. Given a set of mappings Ψ, $u \in \Psi$ is a partial function from variables to values, such that $u(var(t))$ gives the mapping value (i.e., the IRI) of an occurred observation where $var(t)$ is the set of variables in t. To get only the observations produced by *ed*, we could use a *BasicGraphPattern* (BGP)

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

  where $E.src$ is the source (i.e., service id) of E specified in the composition plan. Then, $\Psi(var(P))$ gives all the IRIs of sensor observations produced by *ed*. $\Psi(P)$ 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 *ed*, denoted $T_{id}(E)$ and this pattern *event id pattern* for *ed*, 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 4 reuse and extend the query semantics of SPARQL, therefore we can use the same BGPs[17] to query the occurrence of events instances of type *ed*.
- An *And* operator indicates instances of the connected 2 sub-event types E_1, E_2 should occur, i.e., Given $E_3 := \wedge(E_1, E_2)$, $T_{id}(E_3) = T_{id}(E_1) \cup T_{id}(E_2)$, where $T_{id}(E_1) \neq \emptyset \wedge T_{id}(E_2) \neq \emptyset$. This event operator can be implemented by *join* ($\bowtie$) in SPARQL.

16 ETALIS engine is not integrated into the current prototype implementation but we do present the semantic alignments for ETALIS operators for completeness.

17 In CQELS *StreamGraphPattern* (SGP) is used as an extension of BGP.

```
: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 7: Traffic sensor stream data in SSN

Given P_1, P_2, Ψ_1, Ψ_2 such that $\Psi_1(P_1) = T_{id}(E_1)$, $\Psi_2(P_2) = T_{id}(E_2)$, it is evident that Ψ_1 *join* Ψ_2 creates a new set of mappings $\Psi_3 = \Psi_1 \bowtie \Psi_2$ such that $dom(u_3) = dom(u_1) \cup dom(u_2)$ where $u_1 \in \Psi_1, u_2 \in \Psi_2, u_3 \in \Psi_3$. Notice that u_1, u_2 are always compatible because they are disjoint. Since u_3 is also a partial function, it must provide mapping values for each variable $v \in dom(u_3)$, i.e., $\Psi_3 = \emptyset \iff \Psi_1 = \emptyset \vee \Psi_2 = \emptyset$. The *Join* operator in SPARQL is reused in the semantic stream query engines so that the *And* operator can be implemented by *join*. However, using *join* is only correct if we are operating in the *cumulative* event instance selection policy (recall Sections 6.1.3, 6.2.3), 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.

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

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

  It is evident that Ψ_4 can be implemented by the *OPTIONAL* keyword and the condition can be implemented by a set of *bound* filters.
- 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 [14] 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 $u_1 \in \Psi_1, u_2 \in \Psi_2, f_t(v_1) < f_t(v_2)$ 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 ed_1 occurred before those of type ed_2. Currently CQELS (public version 1.0.0) does not support *SeqJoin* or provide functions to access the timestamps of the stream triples, therefore *Sequence* is not supported in CQELS.
- *Repetition* is a generalisation of sequence, recall definitions in Section 6.1.2, an overlapping (i.e., Rep_o) or non-overlapping (i.e., Rep_n) repetition can be transformed into a conjunction of sequence or a sequence of sequence, respectively. Therefore, repetition can be implemented in C-SPARQL and ETALIS by combining the ways they implement $\wedge$ and ; event operators, while CQELS does not support repetition because the sequence operator is not allowed in CQELS.
- *Selection* retrieves event payloads from member event instances. If payload $p \in D$ where D is the set of payloads for event E is selected, information on p can be queried by adding triple patterns to $P_{id}(E)$:

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

  and *project* the relevant variables into the query results. Notice that for brevity we do not list all triple patterns required here.
- *Filter* and *Window* operators in event patterns is be mapped to *Filter* and *Window* operators the three engines, respectively. Notice that in ETALIS an explicit *Window* operator does not exist, the window operator is implemented by using a filter $F(getDuration() < \delta, \Psi)$ where *getDuration* is a function retrieving the duration all mappings in Ψ and δ is a time interval.
- *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 the same results produced by CQELS and C-SPARQL engines, a post-processing filter is deployed on the CQELS result handler that reports only the results when the time window slides and simulates the time driven query execution.

7.2. Transformation algorithm

Previously (see Section 5.1), we briefly described how event patterns are specified in the CES ontology and what are the semantics of event patterns (Section 6). 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 *ED*; the edges represent the provenance relation in the complex event detection: the parent node is detected based on the detection of the child nodes.

Using a top-down traversal of the event pattern tree and querying the semantics alignment table for each event operator encountered during the traversal, the event pattern in the composition plan is transformed into a CQELS query following the divide-and-conquer style. Algorithm 1 shows the pseudo code of the main parts of the query transformation algorithm. Lines 1–6 in Algorithm 1, 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

Table 4
Semantics alignment for event operators.

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

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 with a pair of brackets (Line 26). Listing 8 shows the transformation result for the event request in Listing 4. 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).

Algorithm 1 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 ← GETSELECTCLAUSE(comp.ep)
 3:     whereClause ← GETWHERECLAUSE(comp.ep)
 4:     queryStr ← 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 ← GETROOTNODE(ep), whereClause ← ∅
 9:     if root ∈ $Op_{seq} \cup Op_{rep}$ then
10:         fail and terminate
11:     else if root ∈ EventServiceDescription then
12:         whereClause ← GETSGP(ep, root)
13:     else if root ∈ $Op_{and}$ then
14:         for subPattern ← GETSUBPATTERNS(ep, root) do
15:             whereClause ← whereClause + GETWHERECLAUSE(subPattern)
16:         end for
17:     else if root ∈ $Op_{or}$ then
18:         for subPattern ← GETSUBPATTERNS(ep, root) do
19:             whereClause ← whereClause + "optional" + GETWHERECLAUSE(subPattern)
20:         end for
21:         whereClause ← whereClause + GETBOUNDFILTERS(ep)
22:     end if
23:     if filters ← GETFILTERS(ep) ≠ ∅ then
24:         whereClause ← whereClause + GETFILTERS(filters)
25:     end if
26: return "{" + whereClause + "}"
27: end procedure
```

Following the similar approach above, we developed query transformation algorithms for C-SPARQL. For the sake of brevity we do not show the details here. The major difference to CQELS query transformation is the support for sequence event operators, which can be implemented based on timestamp filters.

7.3. Event (Re-)construction from stream query results

The query solutions derived from evaluating the query in Listing 8 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 annotated complex events. While the schema/ontology used to reconstruct the complex events may vary depending on the applications, in the current ACEIS implementation, we reconstruct the results as a set of primitive events (observations) annotated with SSN.

8. Prototype implementation and discussion

In this section, we demonstrate the use of our proposed components in the context of smart city applications. We incorporated our components within the navigational service "Context-aware Multimodal Realtime Travel Planner" (a concretised Travel Planner from Section 3.1) developed for the city of the Aarhus.[18] The Travel Planner application aims at providing the ideal route for its users while taking the current context of the user into account. Unlike the state-of-the-art travel planning solutions with a choice of fastest or shortest route, it allows its users to provide multi-dimensional requirements and preferences e.g. weather conditions, air quality, traffic and people intensity, parking availability, traffic schedules etc. Travel Planner can also continuously monitor the current context of its user and relevant events (e.g. traffic accidents) on the planned routes. The user will be prompted to opt for a detour if the real-time conditions on the planned journey no longer meet the user specified requirements and preferences.

The city of Aarhus has deployed a set of traffic sensors on major roads of the city to continuously monitor the traffic conditions within the city. In addition to traffic sensors, there are also weather, parking and air pollution sensors deployed. Consider, Alice (a citizen of Aarhus) who needs to travel from her home to work (see location specifications in Fig. 6(a)). Different means of transportation are generally available to her including walking, biking, car and public transport. Transport options can be optimised to Alice's personalised functional requirements e.g. estimated travel time, weather condition and air quality. Fig. 6(b) shows the screenshot of the system[19] to gather users' functional requirements for the path optimisation.

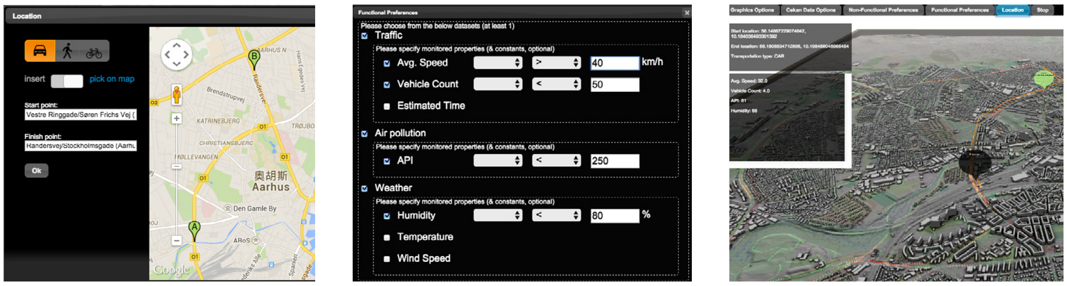
After gathering all functional and non-functional requirements from the user, ACEIS will generate an event request following the information model presented in Fig. 4. ACEIS processes the event request to automatically discover the relevant sensors and evaluates the compliance of the selected sensors with the user's requirement and preferences. ACEIS also generates a composition plan to continuously monitor the conditions of the proposed travel plan. It is worth mentioning that the patterns requested by this application use only conjunction patterns, therefore, both CQELS and C-SPARQL engines can be used to evaluation the composition plans.

[18] Real Time Travel Planner Scenario: http://www.ict-citypulse.eu/scenarios/scenario/1.

[19] We are thankful to CityPulse team http://www.ict-citypulse.eu for the interface development.

```
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



Start point: Vestre Ringgade/Søren Frichs Vej (
Finish point: Randersvej/Stockholmsgade (Aarhu

Please choose from the below datasets (at least 1)
Traffic
Please specify monitored properties (& constants, optional)
Avg. Speed > 40 km/h
Vehicle Count < 50
Estimated Time
Air pollution
Please specify monitored properties (& constants, optional)
API < 250
Weather
Please specify monitored properties (& constants, optional)
Humidity < 80 %
Temperature
Wind Speed

Start location: 56.14887229074842, 10.1840364933013 92
End location: 56.1808834712895, 10.198456048985454
Transportation type: CAR
Avg. Speed: 32.0
Vehicle Count: 4.0
API: 81
Humidity: 68

(a) Start and finish location. (b) Requirements for the travel. (c) Travel path monitoring.

Fig. 6. Travel Planner Demo.

As shown in Fig. 6(c), Alice is presented with her ideal route and will be able to select each leg of the journey based on concurrent and projected aggregated conditions. During her journey, Alice is continuously notified of the contextual conditions on her planned journey. However, if conditions change dynamically and any of the user defined constraints are violated during her travel, ACIES is capable of re-calculating the optimised path considering new conditions.

In addition, based on the experience of developing the prototype of ACEIS, the semantic alignments and query transformation can be extended efficiently for other RSP engines, because of the similarity of the query semantics and the fact that they adopt a SPARQL-like syntax (as in the SPARQL$_{stream}$ [76], ETALIS [14] and Streaming SPARQL [77]). For example, the design and implementation of the first query transformation algorithm for CQELS took about 1.5 person per month (ppm), while developing the second query transformation algorithm for C-SPARQL took only 0.25 ppm.

9. Query performance optimisation for concurrent queries

In order to investigate the feasibility of using RSP engines in large-scale applications such as the smart travel planner deployed at the city scale, performance evaluation and optimisation with regard to concurrent queries are required. In CityBench[20] [17] some initial results for handling concurrency are presented. However, only duplicates for the same query are used as concurrent queries and only 20 queries are tested at one time over a single engine instance. In the following, the performance of single CQELS and C-SPARQL engines is analysed when processing multiple different queries generated from the travel planner application. Then, the optimisation technique of using multiple engine instances in parallel and evaluate the improvement in query performance is discussed. Finally, the experiment results of the stress tests are presented in order to find out the capability of the server that hosts RSP engines using the aforementioned optimisation techniques. The experiments on concurrent queries are deployed on 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, 8–16 triple patterns.[21] The stream rate is configured to 15 triples per second per stream. Thus, for a single query, the input rate is 30–60 triples per second.

9.1. Multiple different queries over single engine instance

Figs. 7 and 8 show the performance of CQELS and C-SPARQL engines when dealing with multiple queries. In the result data series, the letter "p" denotes the number of engine instances deployed and "q" represents the number of queries deployed.

The results in Figs. 7 and 8 indicate that for both types of engines, the query latency increases when handling more queries, and CQELS is relatively more efficient when handling multiple queries. Also, when the number of concurrent queries exceeds 30, the query latency is not *stable*, i.e., does not converge to stable values and will stop producing results after a period of time.

9.2. Optimisation using multiple engine instances

One natural thought in handling many concurrent queries is to deploy multiple engine instances in parallel and distribute

20 Citybench source code and datasets available at: https://github.com/CityBench/Benchmark.

21 This setting is the typical situation in the travel planner scenario.

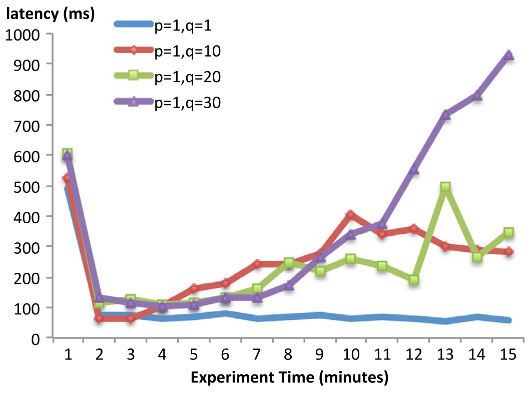


Fig. 7. Latency of multiple different queries over single CQELS engine.

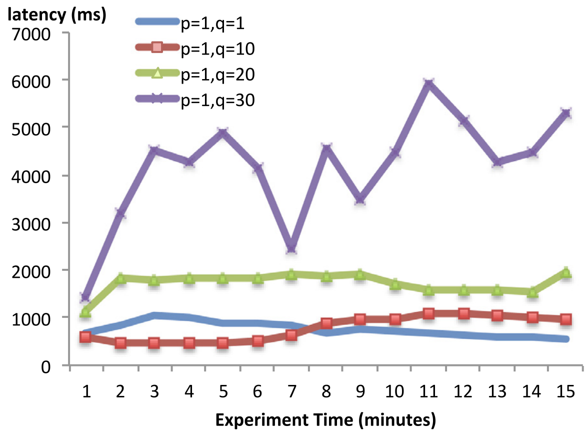


Fig. 8. Latency of multiple different queries over single C-SPARQL engine.

the workload over different engines. Thanks to the service-oriented nature of ACEIS, queries can reuse results from different engine instances and even from different types of engines. However, a load balancing strategy is needed to determine at run-time which queries are going to be deployed on which engine instances. For this purpose, an additional Scheduler module is developed, which consists of a query dispatcher and a performance monitor. The performance monitor gathers the real-time status of the query engine instances, such as query latency, number of queries deployed and overall memory consumption, etc. When a composition plan arrives at the subscription manager, the subscription manager queries the dispatcher for the current best engine instance for the composition. The dispatcher calculates the best engine instance based on the status of the engines reported by the performance monitor and send the identifier of the engine instance to the subscription manager. The subscription manager then deploys the query derived from the composition plan to the best engine instance. When necessary, the dispatcher will create new engine instances. The interactions between the scheduler and other components in ACEIS is depicted in Fig. 9.

The scheduler controls the load balancing using different strategies. The simplest strategy is to initialise a fixed amount of engine instances in the beginning and keep the same amount of queries on each engine instance. This strategy is called the Equalised Query (denoted "EQ") strategy. Another strategy is to dynamically create new instances based on the current system load. This strategy is called the Elastic (denoted "EL") strategy. Since the experiments on a single engine instance suggest that an engine instance may become unstable when dealing with more than 30 concurrent queries, in EL, a new engine instance when the current engine reaches n queries, where $n \leq 30$ (n is set to 20 in the experiments below).

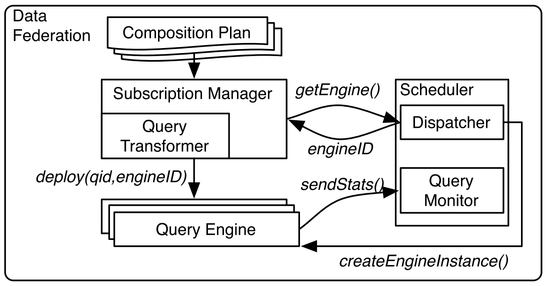


Fig. 9. Concurrent query scheduler inside the Data Federation component in ACEIS.

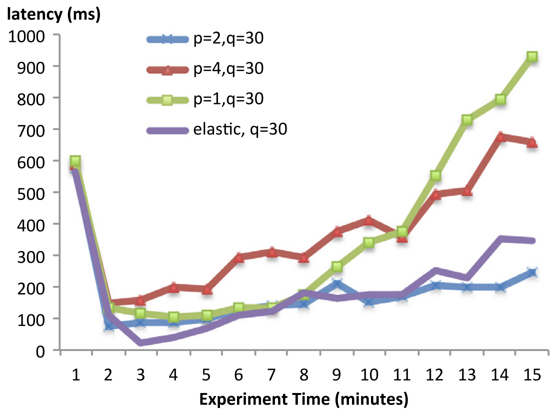


Fig. 10. Latency of CQELS engines using EQ.

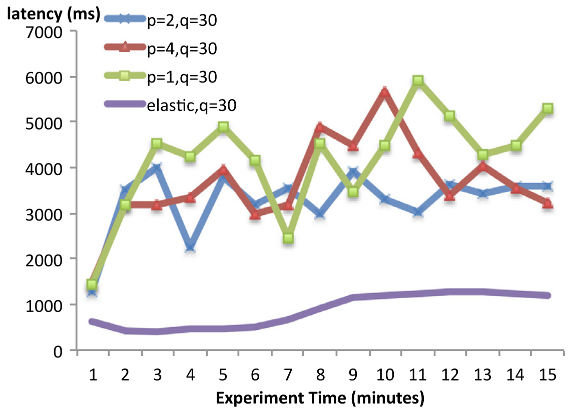


Fig. 11. Latency of C-SPARQL engines using EQ.

Figs. 10 and 11 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 equalised queries in this experiment. Indeed, using multiple engines demands more resources such as memory and initialising all engines upfront creates overhead. Figs. 12 and 13 show the memory usage for CQELS and C-SPARQL under different numbers of concurrent queries and engine instances, respectively.

From the results in Figs. 12 and 13, it is clear that the memory consumption increases as the number of concurrent queries as well as the number of engine instances increase. Also, CQELS uses less memory than C-SPARQL when dealing with fewer queries but the memory growth rate over the number of queries and engine instances are faster than C-SPARQL.

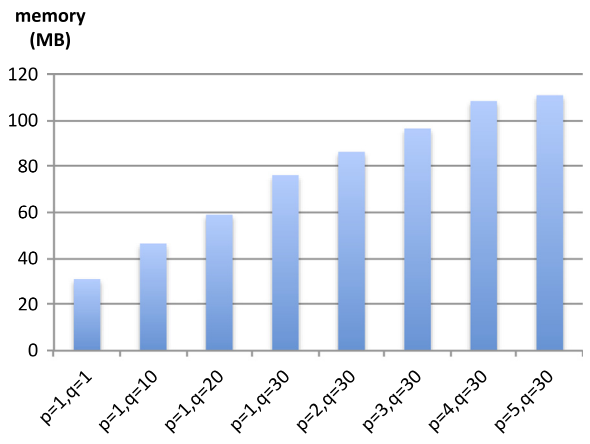


Fig. 12. Memory consumption of multiple CQELS engines.

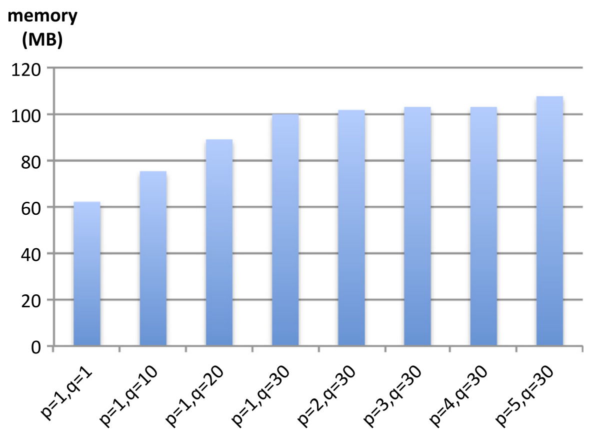


Fig. 13. Memory consumption of multiple C-SPARQL engines.

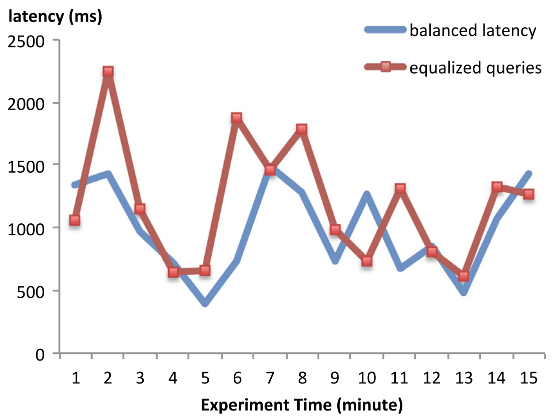


Fig. 14. Latency of CQELS engines using EQ and BL while $p = 5, q = 50$.

Since the memory availability is limited in any system, the elastic approach will have to stop creating new engine instances at some point. Then it will regress to the equalised queries approach. An alternative way is to deploy queries on the engine that has the lowest average query latency. This strategy is called the Balanced Latency (denoted "BL") strategy. The results in Figs. 14 and 15 show that the balanced latency strategy outperforms equalised query on both CQELS and C-SPARQL when dealing with 50 concurrent queries with 5 instances. In particular, C-SPARQL is unstable when using the EQ strategy but is stabilised when using BL. The results in Fig. 16 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

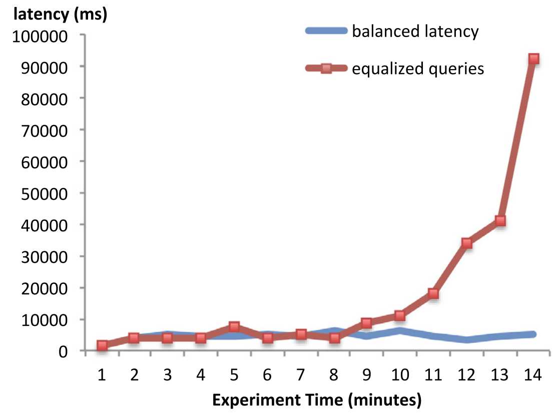


Fig. 15. Latency of C-SPARQL engines using EQ and BL while $p = 5, q = 50$.

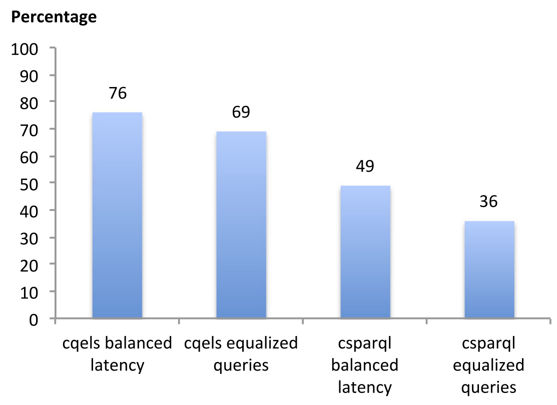


Fig. 16. Query latency distribution, $p = 5, q = 50$.

500 ms is 76% and 69% when using BL and EQ respectively. For C-SPARQL engines, the number of query results with latency less than 5000 ms is 49% and 36% when using BL and EQ respectively. The combined strategy of using the elastic approach in the beginning and switch to balanced query strategy when the memory limit has been reached is called Elastic-Balanced-Latency strategy (denoted "EBL").

9.3. Stress tests

In order to further investigate the feasibility of running federated RSP queries in large scale, i.e., with high input rate, large amount of input streams, and high volume of concurrent users, stress tests are conducted to evaluate the system with hundreds to thousands of queries (deploying a new query every 1–3 s) with the EBL load balancing strategy. The query latencies over an hour for CQELS and C-SPARQL engines are shown in Figs. 17 and 18.

The stress test results show that CQELS can handle 1000 concurrent queries with a 15–20 s delay while C-SPARQL has a much more limited capacity of processing no more than 100 queries in a stable status. It is also worth mentioning that during the experiments 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 whether this behaviour of C-SPARQL is by design or an implementation issue.

10. Conclusion and future directions

In this paper, we have identified several challenges for Smart City applications. We presented ACEIS to automatically discover

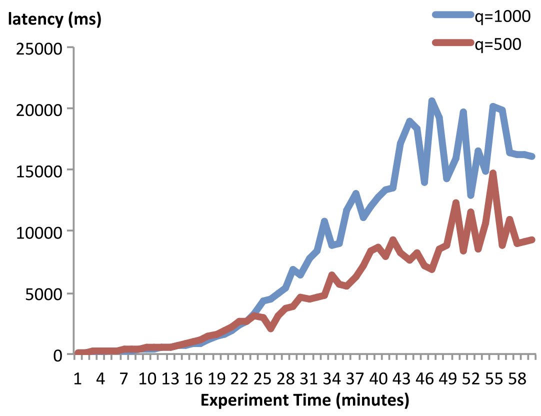


Fig. 17. Latency of CQELS engines using EBL.

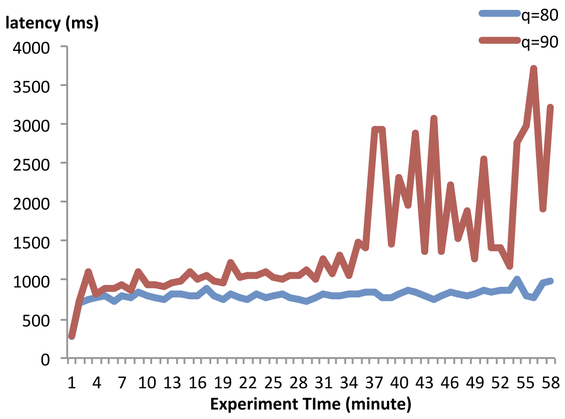


Fig. 18. Latency of C-SPARQL engines using EBL.

and integrate heterogeneous sensor data streams and thus addressing the data stream federation challenge. ACEIS receives requirements from users and applications as event request and discovers or composes the relevant data streams to address both functional and non-functional requirements specified in the event requests. The discovery and composition process in ACEIS rely on the CES ontology designed for describing complex event services as extended OWL-S services. Based on the discovery and composition results, together with alignments between event and query semantics, ACEIS automatically generates stream reasoning queries (i.e., CQELS and C-SPARQL queries) via query transformation and registers the queries to the stream engines. These queries operate on live semantic data streams produced by various (physical or human) sensors to detect complex events. To demonstrate how ACEIS can be used, we integrate it in a travel planner scenario, where users' functional and non-functional requirements for the travel planning are addressed and a live traffic monitoring feature on the planned route is offered. We implement different load balancing techniques for optimising the performance of handling concurrency in ACEIS. The experiment results show that leveraging the "elastic-balanced-latency" strategy, we increase the capacity of ACEIS when using CQELS and C-SPARQL from about 30–1000 queries and 30–90 queries, respectively.

Looking back at the requirements defined in Section 3.3, we can see how ACEIS implemented in the travel planner prototype helps to fulfil these requirements. In this prototype, ACEIS deals with heterogeneous data/event sources (e.g., traffic, air pollution, weather etc.) on the fly, hence satisfying the requirement **R.1**. Apart from functional requirements, non-functional requirements, e.g. sensor accuracy and measurement delay, are also taken into consideration while choosing the sensors, addressing partially the reliable information processing requirement **R.4**. The ability of the ACIES to detour is achieved by real-time monitoring and event detection, which satisfies the requirement **R.3**. Large scale stream processing and analysis (**R.2**) is discussed in Section 9, although we made some improvements for the system capacity, more study in this direction is needed.

In future work, we plan to study further the efficiency, coverage and extensibility of our proposed ontology. We also want to dynamically define optimal window size for live stream queries of complex events while considering individual update frequency of the all underlying data streams. Another direction for the future work is to further investigate methods to improve the capacity of RSP engines for handling concurrent queries, e.g., implementing a distributed way of evaluating federated RSP queries.

Acknowledgements

This research has been partially supported by Science Foundation Ireland (SFI) under grant No. SFI/12/RC/2289, EU FP7 CityPulse Project under grant No. 603095, and the Key Projects of National Social Science Foundation of China (11 & ZD189).

References

[1] S. Hasan, E. Curry, Thematic event processing, in: Proceedings of the 15th International Middleware Conference, Middleware'14, ACM, New York, NY, USA, 2014, pp. 109–120. http://dx.doi.org/10.1145/2663165.2663335.
[2] D. Luckham, The power of events: An introduction to complex event processing in distributed enterprise systems, in: N. Bassiliades, G. Governatori, A. Paschke (Eds.), Interchange and Reasoning on the Web Rule Representation, in: Lecture Notes in Computer Science, vol. 5321, Springer Berlin / Heidelberg, 2008, pp. 3–3. http://dx.doi.org/10.1007/978-3-540-88808-6_2.
[3] O. Etzion, P. Niblett, Event Processing in Action, Manning Publications Co., 2010.
[4] D. Le-Phuoc, M. Dao-Tran, J.X. Parreira, 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.
[5] D.F. Barbieri, D. Braga, S. Ceri, E. Della Valle, M. Grossniklaus, C-sparql: Sparql for continuous querying, in: Proc. of WWW, ACM, 2009, pp. 1061–1062.
[6] D. DellAglio, J.P. Calbimonte, E.D. Valle, O. Corcho, Towards a Unified Language for RDF Stream Query Processing, Springer International Publishing, 2015.
[7] S. McIlraith, T.C. Son, H. Zeng, Semantic web services, IEEE Intell. Syst. 16 (2) (2001) 46–53. http://dx.doi.org/10.1109/5254.920599.
[8] L. Zeng, A.H. Ngu, B. Benatallah, R. Podorozhny, H. Lei, Dynamic composition and optimization of web services, Distrib. Parallel Databases 24 (1–3) (2008) 45–72.
[9] D. Zimmer, R. Unland, On the semantics of complex events in active database management systems, in: 15th International Conference on Data Engineering, 1999. Proceedings., 1999, pp. 392–399. http://dx.doi.org/10.1109/ICDE.1999.754955.
[10] D. Bo, D. Kun, Z. Xiaoyi, A high performance enterprise service bus platform for complex event processing, in: International Conference on Grid and Cooperative Computing, 2008, pp. 577–582.
[11] F. Gao, E. Curry, S. Bhiri, Complex event service provision and composition based on event pattern matchmaking, in: Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems, ACM, Mumbai, India, 2014, http://dx.doi.org/10.1145/2611286.2611287.
[12] F. Gao, M.I. Ali, E. Curry, A. Mileo, Qos-Aware Stream Federation and Optimization Based on Service Composition, Vol. 12, IGI Global, 2016, pp. 43–67.
[13] F. Gao, M.I. Ali, E. Curry, A. Mileo, Qos-aware adaptation for complex event service, in: Proceedings of the 31st Annual ACM Symposium on Applied Computing, SAC'16, ACM, New York, NY, USA, 2016, pp. 1597–1604. http://dx.doi.org/10.1145/2851613.2851806, URL http://doi.acm.org/10.1145/2851613.2851806.
[14] D. Anicic, P. Fodor, S. Rudolph, N. Stojanovic, Ep-sparql: a unified language for event processing and stream reasoning, in: Proceedings of the 20th International Conference on World wide web, WWW'11, 2011, pp. 635–644.
[15] D. Le-Phuoc, H.N.M. Quoc, C. Le Van, M. Hauswirth, Elastic and scalable processing of linked stream data in the cloud, in: The Semantic Web–ISWC 2013, Springer, 2013, pp. 280–297.
[16] Y. Ren, J.Z. Pan, Optimising ontology stream reasoning with truth maintenance system, in: ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October, 2011, pp. 831–836.
[17] M.I. Ali, F. Gao, A. Mileo, Citybench: A configurable benchmark to evaluate rsp engines using smart city datasets, in: International Semantic Web Conference, Springer, 2015, pp. 374–389.

[18] J.P. Calbimonte, Rdf stream processing: Let's react, in: Proceedings of International Workshop on Ordering & Reasoning.

[19] M. Sabou, D. Richards, S. Van Splunter, An experience report on using daml-s, in: The Proceedings of the Twelfth International World Wide Web Conference Workshop on E-Services and the Semantic Web, ESSW'03. Budapest, 2003.

[20] D.A. D'Mello, V. Ananthanarayana, A review of dynamic web service description and discovery techniques, in: 2010 First International Conference on Integrated Intelligent Computing, (ICIIC), IEEE, 2010, pp. 246–251.

[21] N. Milanovic, M. Malek, Current solutions for web service composition, IEEE Internet Comput. 8 (6) (2004) 51–59.

[22] S. Dasgupta, S. Bhat, Y. Lee, Sgps: a semantic scheme for web service similarity, in: Proceedings of the 18th International Conference on World wide web, ACM, 2009, pp. 1125–1126.

[23] J. Hau, W. Lee, J. Darlington, A semantic similarity measure for semantic web services, in: Web Service Semantics Workshop at WWW, 2005, pp. 10–14.

[24] M. Klusch, B. Fries, K. Sycara, Automated semantic web service discovery with owls-mx, in: Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems, ACM, 2006, pp. 915–922.

[25] S. Dasgupta, S. Bhat, Y. Lee, Taxonomic clustering and query matching for efficient service discovery, in: 2011 IEEE International Conference on Web Services, (ICWS), IEEE, 2011, pp. 363–370.

[26] E. Stroulia, Y. Wang, Structural and semantic matching for assessing web-service similarity, Int. J. Coop. Inf. Syst. 14 (04) (2005) 407–437.

[27] M. Carman, L. Serafini, P. Traverso, Web service composition as planning, in: ICAPS 2003 Workshop on Planning for Web Services, 2003, pp. 1636–1642.

[28] J. Peer, A pddl based tool for automatic web service composition, in: Principles and Practice of Semantic Web Reasoning, Springer, 2004, pp. 149–163.

[29] A. Riabov, Z. Liu, Scalable planning for distributed stream processing systems., in: ICAPS, 2006, pp. 31–41.

[30] Z. Liu, A. Ranganathan, A. Riabov, A planning approach for message-oriented semantic web service composition, in: Proceedings of the National Conference on Artificial Intelligence, Vol. 22, AAAI Press, MIT Press, Menlo Park, CA, Cambridge, MA, London, 1999, p. 1389. 2007.

[31] R. Berbner, M. Spahn, N. Repp, O. Heckmann, R. Steinmetz, Heuristics for qos-aware web service composition, in: International Conference on Web Services, 2006, ICWS'06, IEEE, 2006, pp. 72–82.

[32] Z. Duan, Z.-L. Zhang, Y.T. Hou, Service overlay networks: Slas, qos, and bandwidth provisioning, IEEE/ACM Trans. Netw. 11 (6) (2003) 870–883.

[33] M. Alrifai, 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, ACM, 2009, pp. 881–890.

[34] F. Karatas, D. Kesdogan, An approach for compliance-aware service selection with genetic algorithms, in: Service-Oriented Computing, Springer, 2013, pp. 465–473.

[35] Z. Ye, A. Bouguettaya, X. Zhou, Qos-aware cloud service composition using time series, in: Service-Oriented Computing, Springer, 2013, pp. 9–22.

[36] A. Carzaniga, D.S. Rosenblum, A.L. Wolf, Design and evaluation of a wide-area event notification service, ACM Trans. Comput. Syst. 19 (3) (2001) 332–383. http://dx.doi.org/10.1145/380749.380767.

[37] E. Curry, Increasing mom flexibility with portable rule bases, IEEE Internet Comput. 10 (6) (2006) 26–32. http://dx.doi.org/10.1109/MIC.2006.128.

[38] G. Li, H.-A. Jacobsen, Composite subscriptions in content-based publish/subscribe systems, in: Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware, Middleware'05, Springer-Verlag, New York, Inc., New York, NY, USA, 2005, pp. 249–269.

[39] J. Keeney, D. Roblek, D. Jones, D. Lewis, D. O'Sullivan, Extending siena to support more expressive and flexible subscriptions, in: R. Baldoni (Ed.), DEBS, in: ACM International Conference Proceeding Series, vol. 332, ACM, 2008, pp. 35–46.

[40] Z. Long, B. Jin, F. Qi, D. Cao, Reuse strategies in distributed complex event detection, in: Quality Software, 2009. QSIC'09. 9th International Conference on, 2009, pp. 325–330. http://dx.doi.org/10.1109/QSIC.2009.49.

[41] S. Hasan, E. Curry, Approximate semantic matching of events for the Internet of things, ACM Trans. Internet Technol. 14 (1) (2014) 2:1–2:23. http://dx.doi.org/10.1145/2633684.

[42] G. Mühl, Large-scale content-based publish-subscribe systems (Ph.D. thesis), TU Darmstadt, 2002.

[43] M. Akdere, U. Çetintemel, N. Tatbul, Plan-based complex event detection across distributed sources, Proc. VLDB Endow. 1 (1) (2008) 66–77.

[44] N.P. Schultz-Møller, M. Migliavacca, P. Pietzuch, Distributed complex event processing with query rewriting, in: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems, DEBS'09, 2009, pp. 4:1–4:12. http://dx.doi.org/10.1145/1619258.1619264.

[45] Z. Laliwala, S. Chaudhary, Event-driven service-oriented architecture, in: 2008 International Conference on Service Systems and Service Management, 2008, pp. 1–6. http://dx.doi.org/10.1109/ICSSSM.2008.4598452.

[46] A. Hinze, A. Voisard, Eva: An event algebra supporting complex event specification, Inf. Syst. 48 (2015) 1–25. http://dx.doi.org/10.1016/j.is.2014.07.003, URL http://www.sciencedirect.com/science/article/pii/S0306437914001252.

[47] D. Zimmer, R. Unland, On the semantics of complex events in active database management systems, in: 15th International Conference on Data Engineering, 1999. Proceedings., IEEE, 1999, pp. 392–399.

[48] A. Hinze, A-medias: An adaptive event notification system, in: Proceedings of the 2Nd International Workshop on Distributed Event-Based Systems, DEBS'03, ACM, New York, NY, USA, 2003, pp. 1–8. http://dx.doi.org/10.1145/966618.966623.

[49] D. Jung, A. Hinze, A Meta-Service for Event Notification, Springer Berlin, Heidelberg, Berlin, Heidelberg, 2004, pp. 283–300.

[50] K. Whitehouse, F. Zhao, J. Liu, Semantic Streams: A Framework for Composable Semantic Interpretation of Sensor Data, Springer Berlin, Heidelberg, 2006.

[51] Q. Zhou, Y. Simmhan, V. Prasanna, Towards hybrid online on-demand querying of realtime data with stateful complex event processing, in: Big Data, 2013 IEEE International Conference on, 2013, pp. 199–205. http://dx.doi.org/10.1109/BigData.2013.6691575.

[52] D. De Leng, F. Heintz, Towards on-demand semantic event processing for stream reasoning, in: International Conference on Information Fusion, 2014, pp. 1–8.

[53] S. Bischof, A. Karapantelakis, C.S. Nechifor, A. Sheth, A. Mileo, P. Barnaghi, Semantic modeling of smart city data, in: Proc. of the W3C Workshop on the Web of Things: Enablers and Services for an Open Web of Devices, W3C, Berlin, Germany, 2014.

[54] D. Arditi, G. Mangano, A. De Marco, Assessing the smartness of buildings, Facilities 33 (9/10) (2015) 553–572.

[55] A. Boulis, S. Ganeriwal, M.B. Srivastava, Aggregation in sensor networks: an energy–accuracy trade-off, Ad Hoc Networks 1 (2) (2003) 317–331.

[56] J.B. Johnson, G.L. Schaefer, The influence of thermal, hydrologic, and snow deformation mechanisms on snow water equivalent pressure sensor accuracy, Hydrol. Process. 16 (18) (2002) 3529–3542.

[57] A. Mainka, S. Hartmann, W.G. Stock, I. Peters, Looking for friends and followers: a global investigation of governmental social media use, Transforming Gov.: People Process Policy 9 (2) (2015) 237–254.

[58] T. Sakaki, M. Okazaki, Y. Matsuo, Earthquake shakes twitter users: real-time event detection by social sensors, in: Proceedings of the 19th International Conference on World wide web, ACM, 2010, pp. 851–860.

[59] B. Liu, L. Zhang, A survey of opinion mining and sentiment analysis, in: Mining Text Data, Springer, 2012, pp. 415–463.

[60] M.F. Goodchild, Citizens as sensors: the world of volunteered geography, GeoJournal 69 (4) (2007) 211–221.

[61] A. Martinez-Balleste, P. Perez-martinez, A. Solanas, The pursuit of citizens' privacy: a privacy-aware smart city is possible, IEEE Commun. Mag. 51 (6) (2013) 136–141.

[62] G. Pan, G. Qi, W. Zhang, et al., Trace analysis and mining for smart cities: issues, methods, and applications, IEEE Commun. Mag. 51 (6) (2013) 120–126.

[63] A. Bartoli, J. Hernández-Serrano, M. Soriano, M. Dohler, A. Kountouris, D. Barthel, Security and privacy in your smart city, in: Proceedings of the Barcelona Smart Cities Congress, 2011.

[64] I. Stojmenovic, Machine-to-machine communications with in-network data aggregation, processing, and actuation for large-scale cyber-physical systems, IEEE Internet Things J. 1 (2) (2014) 122–128.

[65] D. Martin, M. Burstein, D. Mcdermott, S. Mcilraith, M. Paolucci, K. Sycara, D.L. Mcguinness, E. Sirin, N. Srinivasan, Bringing semantics to web services with owl-s, World Wide Web 10 (3) (2007) 243–277.

[66] K. Haniewicz, M. Kaczmarek, D. Zyskowski, Semantic web services applications–a reality check, Wirtschaftsinformatik 50 (1) (2008) 39–46.

[67] M. Bruno, G. Canfora, M. Di Penta, R. Scognamiglio, An approach to support web service classification and annotation, in: The 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service, 2005. EEE'05. Proceedings., IEEE, 2005, pp. 138–143.

[68] K. Belhajjame, S.M. Embury, N.W. Paton, R. Stevens, C.A. Goble, Automatic annotation of web services based on workflow definitions, ACM Trans. Web (TWEB) 2 (2) (2008) 11.

[69] F. Gao, S. Bhiri, Capability annotation of actions based on their textual descriptions, in: 2014 IEEE 23rd International WETICE Conference, (WETICE), IEEE, 2014, pp. 257–262.

[70] J. Schiefer, S. Rozsnyai, C. Rauscher, G. Saurer, Event-driven rules for sensing and responding to business situations, in: DEBS, Vol. 233, DEBS'07, ACM, 2007, pp. 198–205.

[71] J.F. Allen, L.F. Allen, Maintaining knowledge about temporal intervals, Commun. ACM (1983) 832–843.

[72] G. Cugola, A. Margara, Processing flows of information: From data stream to complex event processing, ACM Comput. Surv. 44 (3) (2012) 15:1–15:62. http://dx.doi.org/10.1145/2187671.2187677.

[73] D.F. Barbieri, D. Braga, S. Ceri, E.D. Valle, M. Grossniklaus, Querying rdf streams with c-sparql, SIGMOD Rec. 39 (1) (2010) 20–26. http://dx.doi.org/10.1145/1860702.1860705.

[74] D. Le-Phuoc, M. Dao-Tran, J.X. Parreira, M. Hauswirth, A native and adaptive approach for unified processing of linked streams and linked data, in: Proceedings of the 10th International Conference on The Semantic Web - Volume Part I, Springer-Verlag, 2011.

[75] G. Decker, A. Grosskopf, A. Barros, A graphical notation for modeling complex events in business processes, in: edoc, IEEE Computer Society, 2007, p. 27. http://dx.doi.org/10.1109/EDOC.2007.41.

[76] J.-P. Calbimonte, O. Corcho, A.J.G. Gray, Enabling ontology-based access to streaming data sources, in: Proceedings of the 9th International Semantic Web Conference on The Semantic Web - Volume Part I, ISWC'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 96–111.

[77] A. Bolles, M. Grawunder, J. Jacobi, Streaming SPARQL-Extending SPARQL to Process Data Streams, Springer, 2008.

Feng Gao received his B.Sc. degree in Software Engineering from Wuhan University, China, in September 2008. He received an M.Eng. degree in Telecommunication from Dublin City University with honors in 2009. Currently he is a Ph.D. student at the INSIGHT Centre for Data Analytics, National University of Ireland, Galway. His current research interests includes Semantic Web, Complex Event Processing and Service Computing. He has passed his Ph.D. viva recently on 24th March 2016 and is conferred with Ph.D. from National University if Ireland, Galway in June 2016. He is currently a Lecturer in the Department of Computer Science, Wuhan University of Science and Technology since Sept. 2016.

Muhammad Intizar Ali is an Adjunct Lecturer, Research Fellow and Project Leader at the Unit for Reasoning and Querying at Insight Centre for Data Analytics, National University of Ireland, Galway. His research interests include Semantic Web, Data Integration, Internet of Things (IoT), Linked Data, Federated Query Processing, Stream Query Processing and Optimal Query Processing over large scale distributed data sources. He is actively involved in various EU funded and industry-funded projects aimed at providing IoT enabled adaptive intelligence for smart city applications and smart enterprise communication systems. He is serving as a PC member of various journals, international conferences and workshops. He is also actively participating in W3C efforts for standardisation in RDF Stream Processing Community Group and Web of Things Interest Group. Dr. Ali obtained his Ph.D. (with distinction) from Vienna University of Technology, Austria in 2011.

Edward Curry is Vice President of the Big Data Value Association (www.BDVA.eu) a non-profit industry-led organisation with the objective of increasing the competitiveness of European Companies with data-driven innovation. Edward is a research leader at the Insight Centre for Data Analytics (www.insight-centre.org) and a funded investigator at LERO The Irish Software Research Centre (www.lero.ie). Edward has worked extensively with industry and government advising on the adoption patterns, practicalities, and benefits of new technologies. Edward has published over 120 scientific articles in journals, books, and international conferences. He has presented at numerous events and has given invited talks at Berkeley, Stanford, and MIT. In 2010, he was a guest speaker at the MIT Sloan CIO Symposium to an audience of 600+ CIOs and senior IT executives. His research projects include studies of smart cities, energy intelligence, semantic information management, event-based systems, and collaborative data management. He is a member of the scientific leadership committee of Insight, and a Lecturer in Informatics at the National University of Ireland Galway (NUIG). He has a Ph.D. from the National University of Ireland Galway.

Alessandra Mileo is a Lecturer, Senior Research Fellow in INSIGHT Centre for Data Analytics, School of Computing, Dublin City University since 2016. She was a Senior Research Fellow, Adjunct Lecturer and Unit Leader at the INSIGHT Research Centre for Data Analytics, NUI Galway, Ireland (formerly DERI). Since 2011, she has been leading the Reasoning and Querying Unit, focusing on the ability to unlock the potentials hidden in the fast growing torrent of data generated on the Internet of Things, and investigating the resulting economical and social impact on application domains including Smart Cities, Smart Transport and remote health monitoring. She is Principal Investigator of the EU FP7 CityPulse project on large-scale data analytics for smart cities, and for the primary industry collaboration within the Research Centre portfolio on Enabling the Internet of Everything: a Linked Data infrastructure for networking, managing and analysing streaming information. Her research interests include web stream reasoning, deductive systems, Internet of Things, Semantic Web and Linked Data, adaptive algorithms, inductive learning, large-scale query processing and federation, context-aware systems, and she has published more than 40 articles in international conferences and Journals. As part of the Steering Committee of the INSIGHT Centre for Data Analytics, Dr. Mileo contributed to the global strategy for H2020 which includes the proposal of a Magna Carta for Data. Dr. Alessandra Mileo has a M.Sc. (2002) and a Ph.D. (2006) in Computer Science, both obtained with distinction from the University of Milan.