IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

SPECIAL SECTION ON SMART CITIES

Received December 31, 2015, accepted January 29, 2016, date of current version March 30, 2016.

Digital Object Identifier 10.1109/ACCESS.2016.2541999

CityPulse: Large Scale Data Analytics Framework for Smart Cities

DAN PUIU[1], PAYAM BARNAGHI[2], (Senior Member, IEEE), RALF TÖNJES[3], DANIEL KÜMPER[3], MUHAMMAD INTIZAR ALI[4], ALESSANDRA MILEO[4], JOSIANE XAVIER PARREIRA[5], MARTEN FISCHER[3], SEFKI KOLOZALI[2], NAZLI FARAJIDAVAR[2], FENG GAO[4], THORBEN IGGENA[3], THU-LE PHAM[4], COSMIN-SEPTIMIU NECHIFOR[1], DANIEL PUSCHMANN[2], AND JOAO FERNANDES[6]

[1]Siemens Romania, Brasov 500007, Romania
[2]Institute for Communication Systems, University of Surrey, University of Surrey, Surrey GU2 7XH, U.K.
[3]University of Applied Sciences Osnabrück, Lingen 49809, Germany
[4]Insight Centre for Data Analytics, National University of Ireland Galway, Lower Dangan, Galway, Ireland
[5]Siemens AG Austria, Vienna 1210, Austria
[6]Alexandra Institute, Aarhus 8200, Denmark

Corresponding author: D. Puiu (dan.puiu@siemens.com)

This work was supported by the European Union Seventh Framework Programme through the CityPulse Project under Grant 603095.

ABSTRACT Our world and our lives are changing in many ways. Communication, networking, and computing technologies are among the most influential enablers that shape our lives today. Digital data and connected worlds of physical objects, people, and devices are rapidly changing the way we work, travel, socialize, and interact with our surroundings, and they have a profound impact on different domains, such as healthcare, environmental monitoring, urban systems, and control and management applications, among several other areas. Cities currently face an increasing demand for providing services that can have an impact on people's everyday lives. The CityPulse framework supports smart city service creation by means of a distributed system for semantic discovery, data analytics, and interpretation of large-scale (near-)real-time Internet of Things data and social media data streams. To goal is to break away from silo applications and enable cross-domain data integration. The CityPulse framework integrates multimodal, mixed quality, uncertain and incomplete data to create reliable, dependable information and continuously adapts data processing techniques to meet the quality of information requirements from end users. Different than existing solutions that mainly offer unified views of the data, the CityPulse framework is also equipped with powerful data analytics modules that perform intelligent data aggregation, event detection, quality assessment, contextual filtering, and decision support. This paper presents the framework, describes its components, and demonstrates how they interact to support easy development of custom-made applications for citizens. The benefits and the effectiveness of the framework are demonstrated in a use-case scenario implementation presented in this paper.

INDEX TERMS Data analytics framework, smart cities.

I. INTRODUCTION

Cities have always faced a demand from their citizens to provide services that support a quality of life and enhanced services. While cities have been evolving in terms of opportunities, these opportunities also reveal many challenges that can impact citizens' daily life [1]. Technology has always been at the centre of this evolution, and over the years it has greatly changed our world and lives. Digital data and connected worlds of physical objects, people and devices are affecting the way we work, travel, socialise and interact with our surroundings and have a profound impact on different domains such as healthcare, environmental monitoring, urban systems, and control and management applications, among several other areas.

Smart cities initiatives are exploring advancements in Internet of Things (IoT) domain to tackle common urban challenges such as reducing energy consumption, traffic congestion and environmental pollution. However, the current services are still largely limited to specific domains, thus creating disconnected silos. Despite the wealth of available

information from numerous data sources, city authorities still encounter several difficulties in implementing, sustaining, and optimizing operations and interactions among different city departments and services [2]. There still remains a need for smart city application tools which support easy development of smart applications. The technical issues include heterogeneity, velocity, mixed quality, uncertainty and incompleteness of the data collected from the smart city environments.

The CityPulse[1] project started with the idea of tackling these challenges, in order to help municipalities and developers in creating better city services. This paper presents the CityPulse framework – a distributed, large scale approach for semantic discovery, data analytics and reasoning of large-scale real-time Internet of Things and relevant social data streams for knowledge extraction in a city environment.

The CityPulse framework allows the development of applications that can provide a continuous and dynamic view of a city, thus enabling users to always know what is happening, when it is happening and how it affects citizens, tourists, companies and city administrators. Having such diverse insights on the pulse of a city is possible by the fact that the framework allows to integrate, manipulate and process a huge variety of data in a flexible and extensible way. Different than existing solutions that only offer unified views of the data, the CityPulse framework is also equipped with powerful data analytics modules. In summary, the main contributions of the CityPulse framework are:

- Data annotation and aggregation modules based on novel algorithms that adapts to the changes in the input sources in order to minimise information loss;
- An event detection module that generates higher level information, which is also semantically annotated;
- A data federation module that implements a novel algorithm to automatically find suitable data input sources at run time, according to the user specifications;
- A quality monitoring module that implements a novel method that applies machine learning to assess the quality of the data provided by input sources;
- A context filtering module that constantly monitors the user's current activity to automatically select relevant events;
- A decision support module which combines semantic technologies and Answer Set Programming (ASP) to provide an expressive and scalable decision support solution.

To demonstrate the benefits and usefulness of the framework, we have developed an application in the context of smart mobility – the adaptive Travel Planner smart phone application. The prototype dynamically integrates traffic and parking information to guide users to available parking spots, while constantly monitoring events in the city and notifying users regarding the events that occur in their driving route.

[1]http://www.ict-citypulse.eu/

The remainder of this paper is organised as follows: Section II describes related work to smart city frameworks. Section III details the components involved in CityPulse. Based on the framework, Section IV demonstrates a use case scenario, an adaptive Travel Planner that has been developed within the CityPulse project. Section V concludes the paper and provides an outlook of future work.

II. SMART CITY FRAMEWORKS

IoT ecosystems play a vital role to gather rich sources of information from smart cities. Different cities have already deployed IoT infrastructures and various sensory devices to collect continuous data from cities. For example, Intel Labs Europe in collaboration with Dublin City Council is in process of deploying citywide IoT infrastructure to monitor and detect city environmental parameters [3]. IBM collects Dublin city traffic data generated from state owned sensors deployed over major roads of the city [4]. Singapore Supertrees collect environmental data including air quality, temperature and rainfall.[2] In the streets of Singapore, a network of traffic sensors and GPS enabled devices embedded in taxicabs, tracks city traffic and predicts future traffic congestions. The city of Aarhus in Denmark has deployed traffic sensors across major roads of the city. Similar IoT infrastructures have been deployed by many smart city initiatives across the globe. These IoT infrastructures act as a major source of continuous data collection and the enormous amount of data can be harnessed by many smart city applications.

A large number of research projects and other similar initiatives mainly focus on collection and provision of IoT data generated from smart cities; e.g. ODAA platform[3] provides open data access to data collected from the City of Aarhus using IoT infrastructure deployed within the city. San Francisco Open Data[4] and City of Chicago Data Portal[5] provide a centralized collection of relevant smart city datasets, which are publicly accessible.

iCity [5] and SmartSantander [6] provide a centralised platform to access data generated from multiple heterogeneous sensors installed in different locations in several European cities. These platforms also facilitate application developers by providing access to different services and APIs for smart city application development. However, both platforms aim at providing access to low-level sensor observations and any kind of data analytics over such raw data should be provided within domain specific smart city applications, which results into potential replication of data analytics' functionalities and silo architectures of domain specific smart city applications.

Realising the importance of semantic technologies to mitigate heterogeneity of data collection from cities, several efforts have been conducted to use semantic technologies for IoT data collection such as OpenIoT [7], Spitfire [8]

[2]http://www.governing.com/topics/economic-dev/gov-singapore-smartest-city.html

[3]http://www.odaa.dk

[4]https://data.sfgov.org

[5]https://data.cityofchicago.org

TABLE 1. IoT and Smart City Frameworks Comparison (●:Yes ○: No ⊙: Partial).

IoT Smart City Platforms and their supported features	iCity	Smart Santandler	Open IoT	iCore	Spit Fire	PLAY	Star City	VITAL	CityPulse
IoT Data Collection	●	●	●	●	●	●	●	●	●
Semantic Interoperability	⊙	⊙	●	●	●	○	●	●	●
Event Detection and Data Analytics	○	○	○	○	○	●	●	○	●
Application Development Support	●	●	⊙	⊙	⊙	○	○	⊙	●

and iCore [9]. The European project OpenIoT provides a middleware for uniform access to IoT data, through the use of semantic models such as SSN. The Spitfire project uses semantic technologies to provide a uniform way to search, interpret and transform sensory data. Spitfire also provides a minimal set of services to access integrated IoT data, which act as an abstraction layer between application layer and data layer. Supported service oriented architecture facilitates easy access to data but allows a limited set of operations which can be performed over collected data. The iCore project proposes a cognitive framework for IoT and smart city applications, hiding the heterogeneity of objects and devices, providing concepts such as virtual objects and composition of virtual objects.

Real-time IoT data collected from the cities can play a major role for designing smart city data analytics frameworks, which can automatically detect important city events (e.g. traffic accident) and trigger actions to recover from such situations. PLAY [10] provides an event-driven middleware that is able to process complex event detection in large highly distributed and heterogeneous systems. PLAY architecture facilitates event-driven adaptive process management, which can automatically adapt after sensing the contextual information. Outsmart[6] is a resource-oriented middleware combined with rule-based system that allows the management of distributed heterogeneous IoT resources. IBM's Star City is a semantic traffic analytics and reasoning system, which integrates traffic related sensor data from human and sensor based data collected from city [11]. Star City supports real-time IoT data analytics for event detection pertaining to the traffic domain e.g. traffic jams, accidents or road congestions. Also for the traffic domain, Zhao *et. all*, propose a hybrid processing system [12] which can be used to perform large scale data analytics of the data coming from the traffic sensors. The system supports streaming and historical traffic sensor data processing, which combines spatio-temporal data partitioning, pipelined parallel processing, and stream computing techniques to support hybrid processing of traffic sensor data in real-time.

[6]http://www.fi-ppp-outsmart.eu

Recently some smart city projects and initiatives have contributed to the strategic and technological development of cities by providing application level support for smart city applications. The VITAL project[7] federates heterogeneous IoT platforms via semantics in a cloud-based environment with focus on smart cities. This project provides a uniform access layer for heterogeneous IoT platforms (X-GSN, Xively,[8] FIT, Hi Reply[9] and OpenIoT) to collect smart city data [13], [14]. In the Vital project, access to existing IoT platforms is realised by adapting to the provided interfaces and abstraction layers via a RESTfull platform. Developers have to strictly follow the HTTP-REST concept to build services.

The state-of-the-art for smart city frameworks has major focus on existing smart city platforms and the existing works are mainly in four key areas: (i) data acquisition (ii) semantic interoperability, (iii) real-time data analysis and event detection, and (iv) smart city application development support. While, the work conducted in CityPulse is complimentary to data acquisition and semantic interoperability, CityPulse progresses well beyond the state-of-the-art when it comes to real-time data analytics techniques and smart city application development support. Table 1 presents a comparison of smart city platforms and their supported features. As shown in the table, additional to data acquisition and semantic interoperability, the CityPulse framework provides a complete set of domain independent real-time data analytics tools such as data federation, data aggregation, event detection, quality analysis and decision support. The application development is support through a set of APIs provided by CityPulse. These APIs provide open access to the complete smart city data analytics framework and can prove to be a game changer for smart city application development. API's level support for major components of the CityPulse framework facilitates a loosely coupled architecture for smart city applications development. Application developers can either use the complete processing pipeline of the CityPulse framework or use

[7]http://vital-iot.eu
[8]https://xively.com/platform/
[9]http://www.reply.eu/en/content/hi-reply

only preferred components depending on their application requirements.

The NYC Open Data provides heterogeneous data on business, city government, education, environment, health, social services, transportation, to the general public. An annual event named BigApps [15] competition joins hundreds of developers, designers, makers and marketers in a competition to address different challenges through technology. Examples of previous challenges include "Zero Waste Challenge", "Affordable Housing Challenge" among others. Other than access to a large number of datasets there is no provision of any data analytics or data pre-processing components that the community of developers can make use of in order to develop their services and applications. The CityPulse framework has to potential to ease the development process.

Similarly Khan *et. al* [16] proposed a prototype which has been designed and developed to demonstrate the effectiveness of the cloud based analytics service for Bristol aggregated open data to identify correlations between selected urban environment indicators such as Quality of Life. The proposed system is divided in to three tires to enable the development of a unified knowledge base. The lowest layer in the architecture consists of distributed and heterogeneous repositories and various sensors that are subscribed to the system. The resource data mapping and linking layer (middle layer) finds new scenarios and supports workflows to develop relations that were not possible in the isolated data repositories. An analytic engine in top layer processes the data for application specific purposes. Modules such as decision support, contextual filtering or technical adaptation are not included in this framework.

The Amsterdam Smart City (ASC) [17] includes a series of projects ranging over several domains in order to make the city smart. The project areas range over Smart Mobility, Smart Living, Smart Society, Smart Areas, Smart Economy, Big and Open Data, and Infrastructure. On a technical level, to achieve the projects goals individual solutions have been developed and optimised to solve one problem of the city at a time.

III. THE CityPulse FRAMEWORK

The CityPulse framework integrates and processes large volumes of streaming city data in a flexible and extensible way. Service and application creation is facilitated by open APIs that are exposed by CityPulse components.

The CityPulse components are depicted in Figure 1 and can be divided in two main categories:

- Large scale data stream processing modules: representing the tools which allow the application developer to interact with the heterogeneous and unreliable data sources from the cities. The tools allow also discovering, summarizing and processing the data streams.
- Adaptive decision support modules: containing the tools which can be used for making various recommendations based on the user context and the current status of the city.

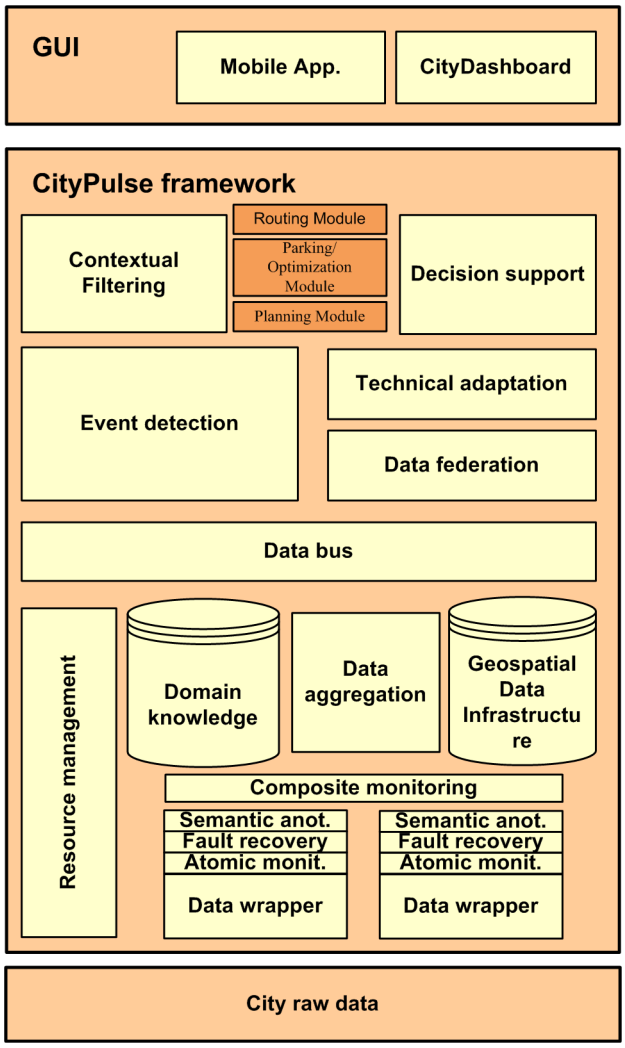

FIGURE 1. The components of the CityPulse framework with their APIs.

The tools from the first category are used to handle and process the city data streams. The CityPulse enabled applications will use Cloud-based components to run the services. In this way the components continuously monitor and process the streams. From this point, any application can interact with the components, via the exposed APIs, in order to obtain at any moment, information about the current status of the city.

In the CityPulse applications, the adaptive decision support components are triggered when a recommendation is needed or a certain context/situation needs to be monitored. As a result of that this components are not running continuously as the ones from the fist category. The remaining of this section presents the CityPulse components.

A. LARGE SCALE DATA STREAM PROCESSING MODULES

1) DATA WRAPPERS AND SEMANTIC ANNOTATION

The *Data wrapper* component offers an easy and generic way to describe the characteristics of a set of sensors using

sensory meta-data. This meta-data is called SensorDescription. The SensorDescription contains general information about the data stream, such as the source endpoint (e.g. HTTP URL), the author/operator of the stream, the update interval when new observations can be fetched, the location of the sensor, and the category of data provided (e.g. traffic data). The observations in the data stream are provided with features such as data type, minimum and maximum observed values, and configuration parameters for the aggregation method to use.

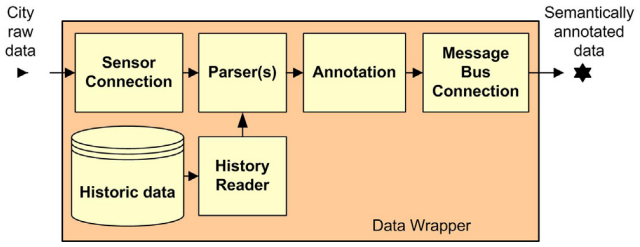

FIGURE 2. Data wrapper modules and processing chain.

Figure 2 depicts the process chain that is involved in the *Data wrapper* component. The Sensor Connection is responsible for the collection of sensor readings from the external resources. Typically, this is accomplished via a network connection (e.g. through a RESTful endpoint), but others such as a serial link (USB) or querying a database are possible as well. Once the raw data has been fetched, the received message is passed onto an instance of a Parser, which will extract the relevant information from the sensory resource. In addition, the History Reader module provides an access to the historical data for the *Resource management*'s replay mode. The historical data can be embedded directly into *Data wrappers* (as compressed archive) or provided by an external data resource. The semantic annotation module enables to annotate the parsed sensory data based on the CityPulse ontologies, such as Stream Annotation Ontology (SAO), and Quality Ontology (QO) and publish them on the message bus. The annotation module is generic, and there is no need of adjustment by the domain expert.

To semantically annotate data streams, CityPulse uses lightweight information models that are developed on top of the well-known information models, such as SSN Ontology, PROV-O,[10] and OWL-S.[11] Figure 3 shows an overview of the CityPulse information models, which consists of 4 main modules, namely Stream Annotation Ontology (SAO),[12] Quality Ontology,[13] User Profiles, and Complex Event information.[14]

SAO is used to express the temporal features (e.g. segments, window size) as well as data analysis features

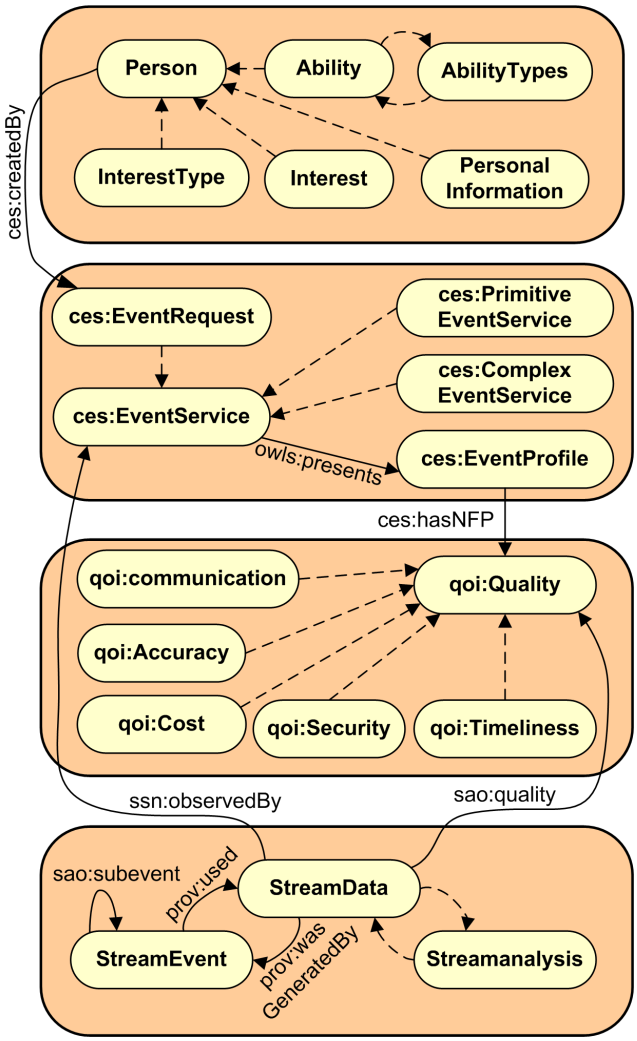

FIGURE 3. CityPulse information model.

(e.g. FFT, DFT, SAX) of a data stream. It allows publishing content-derived data about IoT streams and provides concepts such as sao:StreamData, sao:Segment, sao:StreamAnalysis on top of the TimeLine ontology and the IoTest model. We extent the ssn:Observation concept with sao:StreamData to annotate sensory observations, and provide a link to sao:Segment to describe temporal features in a granular detail for each data segment. Each data point and segment are also linked to the sao:StreamAnalysis concept, where we describe the name and parameters of the method that has been used to analyse the data stream. The Quality Ontology consists of typical quality categories, such as qoi:Accuracy, qoi:Timeliness, qoi:Communication, qoi:Cost. It uses sao:Point or sao:Segment to annotate observations with quality.

To build the provenance relationship, the StreamData class is subclassed from prov:Entity. This Entity has an origin indicated by the hasProvenance relation to the prov:Agent. Complex Event Processing Service ontology is an extension of the OWL-S ontology that allows defining a data stream as

[10]http://www.w3.org/TR/prov-o/
[11]http://www.w3.org/Submission/OWL-S/
[12]http://iot.ee.surrey.ac.uk/citypulse/ontologies/sao/sao
[13]http://purl.oclc.org/NET/UASO/qoi
[14]http://citypulse.insight-centre.org/ontology/ces/

a primitive or complex event service. It expresses the temporal relationships captured by an event pattern according to three basic types: sequence, parallel conjunction and parallel alternation.

User profiles store the description of the characteristics of people and their preferences, which are used as contextual information for the applications. Once the sensory observations are semantically annotated, they are published on the framework's message bus, which can then be consumed by other components.

2) RESOURCE MANAGEMENT

A resource in the context of the *Resource management* is a *Data wrapper*. The application developer can use the *Resource management* in order to achieve three main tasks. First it can be used to deploy all *Data wrapper* units placed in a predefined folder on start-up. Each unit consists of an archived version of the *Data wrapper*'s code together with a deployment descriptor – a JSON file stating the module – and class name of the *Data wrapper* to be instantiated.

The second task of the *Resource management* is to distribute the *Data wrapper*'s output (observations) to other framework components. For flexibility reasons the components in the proposed framework are loosely coupled over a message bus. The framework uses the open Advanced Message Queuing Protocol (AMQP) [18] to exchange messages over the bus. The *Resource management* is responsible for publishing semantically annotated observations and aggregated data coming from different parts of the framework (e.g. the *Data federation* and *Event detection* modules, or even directly from CityPulse applications). Other components can then consume the published data by subscribing to one or more topics, which are defined by the Domain Expert within the SensorDescription. The topics of the messages are structured in such a way that allows using wildcards in order to receive messages of the same kind from multiple sources. Lastly, the *Resource management* provides an API, where other framework components or external third party application developers can perform management functions or access details about the deployed *Data wrapper*. Those functions include the deployment of *Data wrappers*, access to previous observations, and retrieval of a *Data wrapper*'s SensorDescription. The API can be accessed via a HTTP interface.

The application developers can configure the *Resource management* to operate in two different modes: normal mode and replay mode. In replay mode a synthetic clock is used, capable of running faster than real-time. Furthermore, instead of live sensor observations, historical data is used. This way the framework offers the possibility to experiment with newly developed algorithms, without interfering with the live system, or to investigate historic events more closely in fast motion. Features of the *Resource management* are controlled over a series of command line parameters.

3) DATA AGGREGATION

Data aggregation component deals with large volumes of data using time series analysis and data compression techniques to reduce the size of raw sensory observations that are delivered by data wrappers. This allows reducing the communication overhead in the CityPulse framework and helps performing more advanced tasks in large scale, such as clustering, outlier detection or event detection. To effectively access and use sensory data, semantic representation of the aggregations and abstractions are crucial to provide machine-interpretable observations for higher-level interpretations of the real world context. Most of the current smart city frameworks transmit raw sensory data and do not provide energy efficient time-series data analysis as well as granular semantic representation of the temporal and spatial information for the aggregated data.

Numerous approaches have been utilized for time series analysis, including Discrete Fourier Transform (DFT) [19] Discrete, Discrete Wavelet Transform (DWT) [20], [21] Singular Value Decomposition (SVD) [22] Piecewise Aggregate, Piecewise Aggregate Approximation (PAA) [23] Contrary to the numerical approaches, symbolic representation of discretised time series data includes significant benefits of existing algorithms. This involves efficient manipulation of symbolic representations as well as the framing of time.

The CityPulse framework enables a domain expert to select the data aggregation method in the configuration phase. While it supports some of the traditional approaches, such as DFT, PAA, DWT, it uses a multi-resolution data aggregation approach, called SensorSAX, which is an extension of Symbolic Aggregate Approximation (SAX), as a default method. SensorSAX is computationally not expensive, ensures a substantial data reduction and supports the lower bounding principle.

The SensorSAX parameters need to be decided manually and remain predetermined. However, due to the fact that IoT data streams can be highly dynamic and may need to adjust to rapid changes of the observed phenomena at different frequency level, the optimal value for the window length cannot remain the same. SensorSAX overcomes this challenge by presenting 3 new parameters, namely, minimum window size, maximum window size, and sensitivity level. While first two parameters, minimum and maximum window sizes, enable to predetermine length of the window, sensitivity level enables to calculate the optimum window size for the data stream between the plausible ranges. For that it finds the maximum window size, say w^*, in $w_{min} \leq w^* \leq w_{max}$, such that for $1 \leq i \leq w^*$ where $\sigma(c_i) \leq sl$ holds. For such w^*, it computes average $\bar{c}$ of $\{c_1, \ldots, c_{w^*}\}$ by:

$$\bar{c} = \frac{1}{w^*} \sum_{i=1}^{w*} c_i$$

Then, it finds the breakpoints β_j to obtain a sax letter, $\hat{c}$, such that $\beta_{j-1} \leq \hat{c} < \beta_j$. These letters, $\hat{c}$, form a SAX

word, $\hat{C}$. SAX words have got a fix length and allow having different letters in the same word (e.g. ''aabe'' with a word length of ''4''). SensorSAX is energy-efficient and process-efficient approach that enables a remarkable data reduction for data streams.

4) DATA FEDERATION

The CityPulse framework uses the *Data federation* component to answer users' queries, e.g., what is the average vehicle speed on my current route to the destination over the past 5 minutes, over federated data streams. To do this, this component first needs to find relevant streams (or stream federations) for the user according to the functional and non-functional requirements specified in the request. Then, it translates the users' requests into RDF Stream Processing (RSP) queries and evaluates the queries over the relevant streams to obtain query results. This component integrates Semantic Web technologies (SW) [24], Service Oriented Architecture (SOA) [25] and Complex Event Processing (CEP) [26] to provide a data federation solution for smart city applications based on Event Services. This way we decouple the data stream providers and consumers, allowing the CityPulse framework to discover and compose heterogeneous data streams on-demand, regardless of systems or platforms providing the data streams. The query transformation algorithms implemented allows the CityPulse framework to use different RSP engines to answer the users requests. Meanwhile, dynamic reasoning can be supported when using some RSP engines, e.g., RDF level reasoning in CSPARQL [27], RDFS level materialisation can be realized by extending CQELS [28].

Once the data has been annotated by the *Data wrappers* and the and provided as event services, the problem of creating optimal federation of data streams is transformed into a service discovery and composition problem. However, unlike conventional goal-driven service composition approaches which rely on matchmaking of input/output message types or pre-/post-conditions in service descriptions, the event service composition identifies the reusability of event services based on the comparison of the event semantics described in event patterns [29]. In addition, a genetic algorithm is developed in the *Data federation* component to optimize the QoS for event service compositions [30].

Figure 4 illustrates the architecture and processing chain of this component. Upon receiving an event request without an event pattern, an event service discovery is performed to find matching sensor data streams based on the matchmaking of requested and provided sensor descriptions. Then, a subscription is made to the found sensor data stream and the consumer starts receiving sensor observations from the data stream. If the event request contains an event pattern, the event service composition algorithm is invoked to create a composition plan, describing how the streams are composed to answer the query. Then, the composition plan is transformed into an executable query of the target system, e.g., CQELS [28] or C-SPARQL [27] engines, and the query is evaluated while the continuous results are delivered to the consumer.

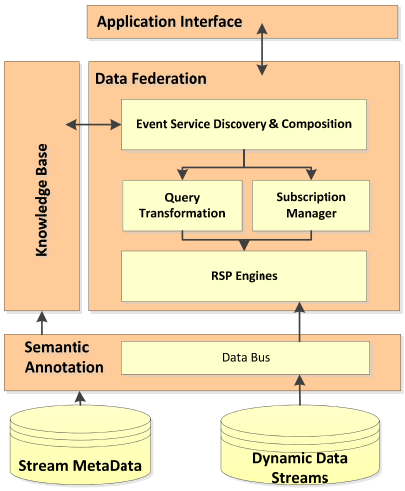

FIGURE 4. Architecture of data federation.

The *Data federation* component receives its inputs from the application interface. It queries the service metadata stored in the Domain Knowledge to perform stream discovery and composition. It then subscribes to the *Data Bus* to consume real-time data and its outputs can be delivered to the application interface or *Decision support*.

At design time, a third-party developer can configure the data endpoints for the *Data federation*. He/she can also configure the load balancing strategy used for the component, in order to specify how concurrent queries are distributed over multiple RSP engine instances. At run-time, a 3rd party developer can register requests (containing functional and non-functional requirements) via a web socket connection. The backend system will invoke the API's register method and perform necessary steps to deploy the RSP query and deliver the continuous results via the same web socket session opened by the client for registering requests. If the request is specified as an one-time query, e.g., asking only the latest observations of some sensors, the backend system will request a single query result from the API, using the snapshot values cached by the *Resource Management* component.

5) EVENT DETECTION

The CityPulse framework exposes two modules to detect the events happening in the cities. The first module uses directly the sensory data sources from the city. The second one can be used to process the data from the social media sources. In the case of CityPulse it is used for processing streams of tweets from Twitter.

The stream *Event detection* component provides the generic tools for processing the annotated as well as aggregated data streams to obtain events occurring into the city. This component has to be highly flexible in deploying new event detection mechanisms, since different smart city applications require different events to be detected from the same

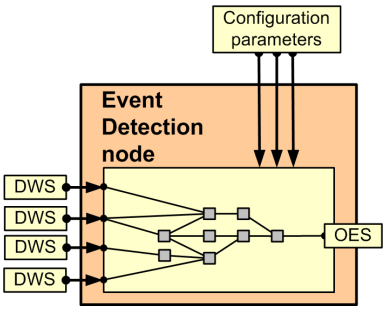

FIGURE 5. Event detection node.

data sources. The component has been developed using the Esper engine [31].

Usually, the development of an event driven application consist of two main steps: 1) real-time data acquisition, interpretation and validation; 2) execution of event detection mechanism in order to detect the patterns of events.

The first step is done automatically by the CityPulse framework. When a new specific event detection mechanism is deployed for processing the data coming from a set of streams, the *Event detection* component performs automatically (with no intervention from the application developer) the following actions:

- makes a request to the *Resource management* to identify the description of the streams (i.e., the routing keys where the requested streams are published on the *Data Bus*, and the details about how to interpret an observation);
- connects to the *Data Bus* and continuously converts the received observations from RDF format to the one requested by the Esper engine.

In order to fulfil the second step of the event detection mechanism the application developer has to develop an event detection node, under the form of a Java class, which contains the event detection pattern.

The event detection node (see Figure 5) can be seen as a black box with inputs being input streams from *Data wrappers* (DWS) and configuration parameters, and having as output the stream of detected events (OES).

In order to develop a new event detection node the application developer has to extend a dedicated Java class, which provides methods for defining the event detection pattern. Existing event detection nodes can be reused by simply changing the configuration parameters and the input streams.

The second module exposed by the *Event detection* component is used to process the Social Media streams. The module analyses and annotates large-scale real-time Twitter data streams. A dedicated *Data wrapper* is used to connect to the Twitter stream API and collect the data under the form of tweets. The *Data wrapper* uses the Google-translate API to automatically detect the source language and translate the tweets to English to facilitate the Natural Language processing step. This in fact enables the application developer to fetch data from any area (e.g. the area surrounding a certain city).

The Social Media *Event detection* component reads a sequence of words in the sentence (Tweet), and passes it to a Natural Language Processing (NLP) unit (see Figure 6).

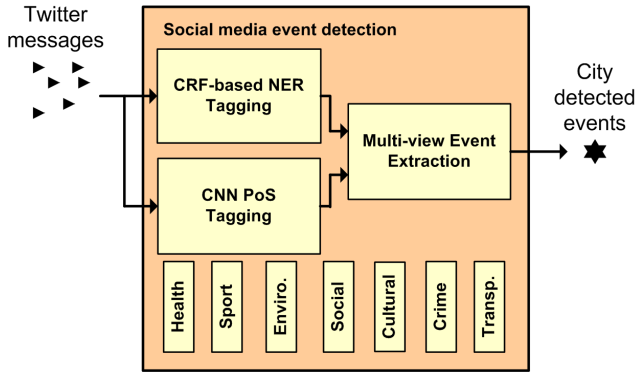

FIGURE 6. Social Media stream processing and event detection component.

The processing unit is composed of three sub-components: a Conditional Random Field Name Entity Recognition [32], [33], a Convolutional Neural Network for deep learning for Part of Speech tagging, and a multi-view event extraction.

During the design time, the internal sub-components are trained with a large corpus of Wikipedia documents and historical Twitter data to guarantee a generalisable Natural Language Processing model.

The Conditional Random Field Name Entity Recognition component assigns event tags to the words in a Tweet given an event-categories set ({*TransportationEvent, EnvironmentalEvent, CulturalEvent, SocialEvent, SportEvent, HealthEvent, CriminalEvent*}), which is tailored for city related events. The categories have been defined generic enough to cover future sub-categorical event assignments (e.g traffic and weather forecast events can be perceived as sub-categories of TransportationEvent and EnvironmentalEvent categories, respectively.) and they will be available to third party developers for adoption and utilisation for new scenarios. Additionally, their respective event vocabularies are adaptive and extendible to future events.

During the run time, the Conditional Random Field Name Entity Recognition component assigns event tags to the words in a Tweet from the event categories set. Simultaneously, the trained Convolutional Neural Network [34] component generates Part-of-Speech tags for atomic elements of the Tweet. The obtained two views of the data (Conditional Random Field Name Entity Recognition view and Convolutional Neural Network Part of Speech view) are then fed into a novel multi-view event extraction component, where the obtained tags of the two views are mutually validated and scored for a final sentence-level inference. An example of sentence level inference is in the case of tweets such as "*seeing someone being given a parking ticket*" where individual words *"parking"* and *"ticket"* can belong to Transportation and

Cultural events categories respectively while considering the sentence grammar can clear up this confusion and assign the tweet to Transportation category. The real-time extracted city events are then used by *Real-time Adaptive Urban Reasoning* component to obtain a more comprehensive interpretation of the city events. The extracted knowledge is utilised to complement the sensor stream information extraction and allows obtaining of a more detailed interpretation of the city events when complementary citizen sensory data can be extracted via social media processing.

6) QUALITY MONITORING

The CityPulse framework offers a two-layered quality calculation mechanism to annotate data streams with a Quality of Information (QoI) metric. The lower layer, called *Atomic monitoring*, is a stream based quality calculation whereas the upper layer, *Composite monitoring*, combines different data streams to include several sensor observations into QoI calculation. This enables applications utilising the CityPulse framework to select the best fitting data streams for their needs. The current implementation of the framework supports five QoI metrics: Age, Completeness, Correctness, Frequency, and Latency.

To calculate this QoI metrics within the *Atomic monitoring* the domain expert has to specify a SensorDescription within the *Data wrapper* (compare section III A). The fields, which are needed for the QoI metrics are listed below:

```
1   sensordescription.maxLatency = 2
2   sensordescription.updateInterval = 60
3   sensordescription.fields = ["v1", "v2", “v3”]
4   sensordescription.field.v1.min = 0
5   sensordescription.field.v1.max = "@v3"
6   sensordescription.field.v1.dataType = "int"
7   sensordescription.field.v2.dataType = "datetime"
8   sensordescription.field.v2.format = "%Y-%m-%dT%H:%M:%S"
9   sensordescription.field.v3.dataType = “int”
10  …
```

Listing 1. Sensor description.

The first line states the maximum latency in seconds that should be met when the data wrapper accesses new data. The following updateInterval specifies the time interval new observations can be fetched (i.e. with pull connection) or the maximum time interval observations are published (i.e. with push connection). The third line, namely ‘‘sensordescription.fields’’, contains all features contained in a single observation of the sensor stream. In this example the sensor stream consists of three different features. *v1* is a value of data type integer. The minimum value is 0. For the maximum there is a special annotation with an @ followed by another fieldname. This states that the maximum value is the current value in the same observation identified by the given fieldname. *v2* specifies a field containing a timestamp. The ‘‘format’’ parameter describes the format of the incoming data fields. *v3* is another value of type int. Other parameters for this field are omitted in this example.

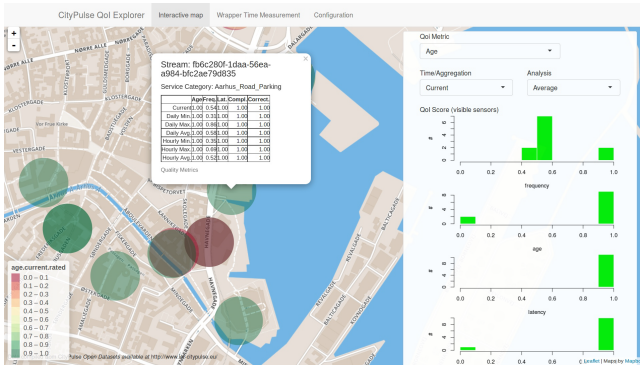

Figure 7. QoI explorer.

Within the *Atomic monitoring* the values from the sensor description are compared to the current observations delivered by the data stream. Based on the comparison and an internal rating algorithm the QoI for the data stream is calculated.

To allow the domain expert to observe the quality of data streams the CityPulse framework provides a tool called QoI explorer to monitor deployed sensors in the city. Here, a map visualises the state of each sensor (see Figure 7).

The tool shows an overview of all sensors within the city. The QoI of the streams is marked with colours. There are different options to select specific QoI metrics or detailed views for individual streams.

The *Composite monitoring* layer combines information from the *Atomic monitoring* components. Based on entity, time, and geospatial relationships the different sources are evaluated and checked for plausibility. Using the *Composite monitoring* it is possible to detect faulty information sources within a group of data sources by comparing to the other group members. *Composite monitoring* compares acquired data with similar streams to determine correlations and looks for divergences. When outliers of single data streams are detected, the *Composite monitoring* allows to search for similar patterns in related data streams. Therefore, outliers can be separated from corrupted sensor data.

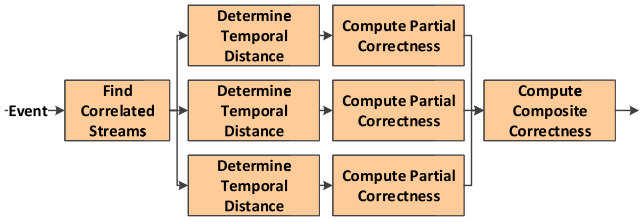

Figure 8. Composite monitoring process.

Figure 8 shows all the (four) phases of the process of the *Composite monitoring* which calculates the plausibility by utilising available overlapping information.

During the first phase spatially related streams are determined, which are able to support the information of the event. In the second phase, the temporal distance is determined and used to create a propagation model of the event. In the third

step, correctness plausibility for each individual correlated stream is calculated by comparing seasonally adjusted time series with the event. By weighting the individual values with their spatio-temporal distance a combined composite correctness value is calculated.

The *Composite monitoring* can be triggered by a set of events that will be compared against raw data streams. The Domain Expert can improve the *Composite monitoring* by providing models that determine the configuration for spatial distance model, temporal distance propagation and the mapping between individual data stream types.

7) FAULT RECOVERY

The *Fault recovery* component ensures continuous and adequate operation of the CityPulse application by generating estimated values for the data stream when the quality drops or it has temporally missing observations. When the quality of the data stream is low for a longer time period, an alternative data source has to be selected. The selection can be performed automatically by the *Technical adaptation* component. In other words, the technical adaptation process does not have to be triggered if the QoI of a stream is low only for a short period of time because the *Fault recovery* component provides estimated values.

As presented in Figure 1 the *Fault recovery* component is integrated into the data wrapper. The fault recovery mechanism is triggered to generate an estimated value when the atomic monitoring component has determined that the current observation is invalid or missing.

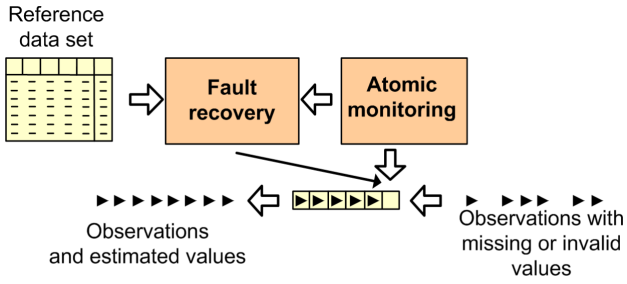

Figure 9. **Fault recovery component workflow.**

The building blocks of the *Fault recovery* component are presented in Figure 9. The component has a buffer which temporary stores the latest observations generated by the data stream, and a reference dataset which contains sequences of valid consecutive observations from the data stream. When an estimated value is requested, the k-nearest neighbour algorithm [35] is used to select a few sequences of observations from the references dataset, which are similar to the current situation. At the end the estimated value is computed from the selected sequences of observations.

During the normal operation, when the QoI of the stream is high, the fault recovery component extends the reference data set with the sequences of observations from the buffer if a similar signal pattern was not included before.

Initially, when the *Data wrapper* is deployed, the reference data set is empty and based on the normal operation (from stream quality point of view) it is extended. As a result of that, the work of the 3rd party application developer is reduced, because he does not have to collect historic data from the stream, to clean and to validate it in order to create the reference dataset. Using the API exposed by the resource management, the 3rd party application developer can turn on and off this component based on the CityPulse application requirements.

8) GEO-SPATIAL DATABASE

Reasoning in cities depends heavily on the spatial context. For example, while temperature tends to marginally vary across a neighbourhood, noise propagation depends on shielding buildings and traffic flows on road networks, ongoing construction work, traffic density etc. Hence, spatial reasoning requires appropriate distance measures. However, the integration of large amounts of data sources requires efficient methods to provide the necessary information in (real) time. A *Geospatial Data Infrastructure* is integrated to utilise infrastructure knowledge of the city. It enables calculation of different distance measures and allows enhanced information interpolation to increase reliability. Furthermore, an enhanced routing system enables multidimensional weighting on path, e.g., depending on distance, duration, pollution, events or combined metrics. Thereby, it is possible to avoid certain areas or block partial routes for specific applications (see Figure 10).

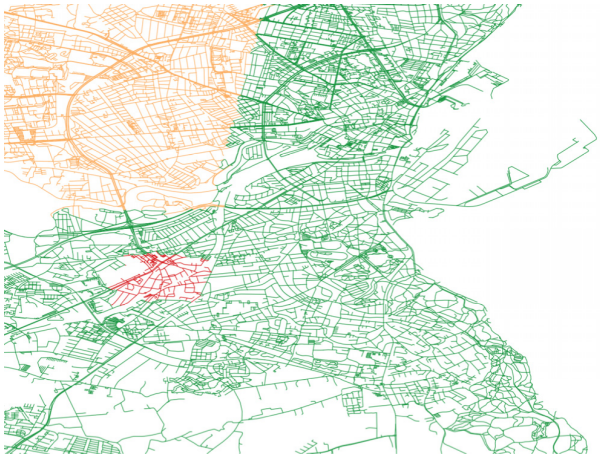

Figure 10. **Weighting edges on a street graph (Green = Neutral, Yellow = Higher Cost, Red = Infinite Cost).**

As an example, Figure 3 depicts classes of edges in the routable graph based on the weighted metric for pollution. The geo-spatial database can be used for developing CityPulse applications via a Java API, which enables the following functionalities:

- Calculate routes with:
  - avoidance of areas,

- on a weighted street graph depending on priorities, e.g. pollution level,
- multiple alternative routes, which are sorted by a cost function;

- Find sensors in an area or on a route;
- Find events in an area or on a route;
- Find objects like hospitals, waste-bins, or further public infrastructure.

9) CITY DASHBOARD

The CityPulse framework provides immediate and intuitive visual access to the results of its intelligent processing and manipulation of data and events. The ability to record and store historical (cleaned and summarised) data for post-processing makes it possible to analyse the status of the city not only on the go but also at any point in time, enabling diagnosing and ''post mortem'' analysis of any incidents or relevant situation that might have occurred. To facilitate that, a dashboard for visualising the dynamic data of the smart cities is provided on top of the CityPulse framework. Based on this dashboard, the user has the possibility to visualise a holistic and summarised view of data across multiple contexts or a detailed view of data of interest, as well as to monitor the city life as it evolves and as things happens. The investigation of past city events or incidents can be conducted from different perspectives, e.g. by observing the correlations between various streams, since the streaming data is stored in the framework for a period of time which can be configured, and it can be retrieved for visualisation and analysis at any moment. Figure 11 depicts a snapshot of the CityPulse dashboard application.

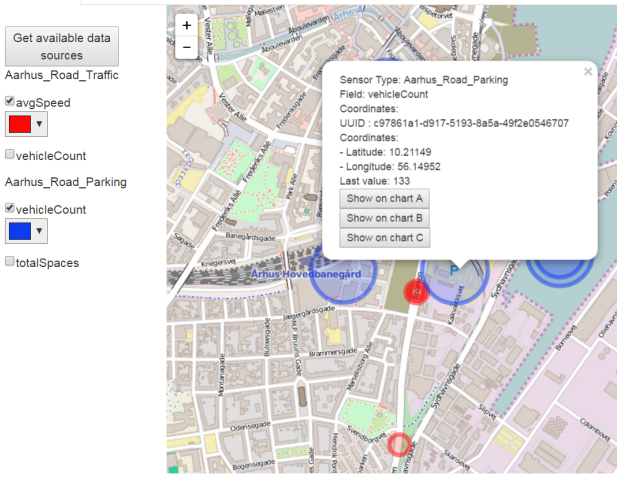

Figure 11. CityPulse dashboard application.

In order to display the status of the city, the dashboard application connects directly to the to the resource management or to the data bus for fetching the description of the available streams or the real-time/historic observations. The dashboard application can be used out of the box and there are no configuration or development steps that have to be done by the application developer.

B. REAL-TIME ADAPTIVE URBAN REASONING

Smart city applications in changing environments require to take into account user preferences and requirements, as well as dynamic contextual information represented by real-time events, in order to provide optimal decision support to the end user at any time.

The event-driven adaptation and context-driven user-centricity of the CityPulse framework are materialized by a close loop between the *Contextual Filtering* component, the user application, and the *Decision support* component as illustrated in Figure 12.

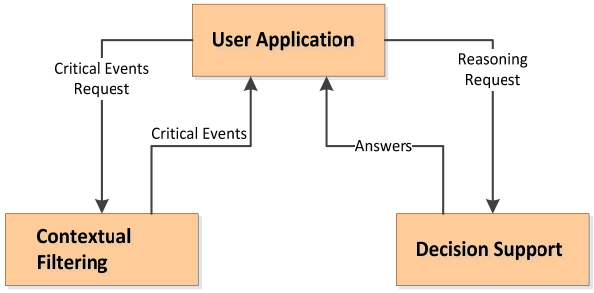

Figure 12. Event-based user-centric decision support.

The user-centric and event-driven reasoning capabilities of the framework strongly rely on the tight interaction between the user application and the *Contextual Filtering* component respectively, with the former being in charge of the bidirectional communication with the other two. The user application is also tightly connected with the *Decision support* component in requesting to compute answers to a decision problem by reflecting constraints and preferences specified by the user himself.

This user-driven loop between the *Contextual Filtering* and the *Decision support* component makes it easier for application developer to decide whether to give complete control to the end user on when and how to request adaptation after critical events have been detected, or automatically suggest new solutions.

In the remainder of this section we provide details of both the *Contextual Filtering* and the *Decision support* components for real-time adaptive urban reasoning.

1) CONTEXTUAL FILTERING

The main role of the *Contextual Filtering* component is to continuously identify and filter events that might affect the optimal result of the decision making task (performed by the *Decision support* component), and react to such changes in the real world by requesting the *Decision support* for the computation of a new solution when needed. This not only ensures that the selected solution provided to the user remains the best option when situations change, but it also empowers the CityPulse framework to automatically provide alternative decisions whenever the selected best decision is no longer the best for a particular situation.

The adaptive capability of identifying and reacting to unexpected events in a user-centric way relies on two aspects:

```
@prefix geo:http://www.w3.org/2003/01/geo/wgs84_pos# .
@prefix sao:  <http://purl.oclc.org/NET/UNIS/sao/sao#> .
@prefix tl:   <http://purl.org/NET/c4dm/timeline.owl#> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .
@prefix prov: <http://www.w3.org/ns/prov#> .
@prefix ec:   <http://purl.oclc.org/NET/UNIS/sao/ec#> .

sao:c2d69ca8-b404-4006-ad48-9317397251ab  a  ec:TrafficJam ;
      ec:hasSource    "SENSOR" ;
      sao:hasLevel    "1"^^xsd:long ;
      sao:hasLocation [ a       geo:Instant ;
                        geo:lat  "56.17091325696965"^^xsd:double ;
                        geo:lon  "10.15728564169882"^^xsd:double
                      ] ;
      sao:hasType     ec:TransportationEvent ;
      tl:time         "2015-11-26T13:27:46.079Z"^^xsd:dateTime .
```

Listing 2. An example of an annotated event.

```
(r1) related_city_event(Id) :- filtering_event(Category),    city_event(Id,
Category, Source).
(r2) 1 <= {selected_city_event(EventId) : related_city_event(EventId),
 not expired_event(EventId)} <= 1.
(r3) value(RankEleName,Value):- selected_city_event(EventId),
ranking_city_event_data(EventId, RankEleName,Value).
(r4) value_with_ranking_type(RankingElementName, M):-
value(RankingElementName,Value),
ranking_multiplier(RankingElementName,Int), M = Value*Int.
(r5) sum(C) : value_with_ranking_type("EVENT_LEVEL",Value1),
 value_with_ranking_type("DISTANCE",Value2),
 C = Value1+Value2.
(r6) criticality(C) :- C = M/100, sum(M).
(r7) critical_city_event(EventId,C):- selected_city_event(EventId),
criticality(C).
```

Listing 3. Logic rules for contextual filtering of events with ranking through linear combination.

i) a characterisation of the user implicit and explicit context provided by the user application (including user requirements, preferences, events of interests and activities), and ii) a stream of events provided by the *Event detection* component.

The filtering capability of the *Contextual Filtering* component is key not only for context-awareness, but also for scalability. In fact, the *Contextual Filtering* component only subscribes to a subset of events among those provided by the *Event detection* component. The type of events the *Contextual Filtering* subscribes to are application-specific and are in part dependent on the domain (determined at design-time), and in part contextualized, depending on the specific reasoning task or user preferences specified by the user application (determined at run-time). For example, in the Travel Planner application (see section IV), traffic and weather conditions are relevant types of events that can be characterise at design time. However, when the user application provides a solution to go from a starting point A to an ending point B, only traffic and weather conditions in areas around that specific path are potentially relevant, and they can be augmented by the user interest in other types of events on the way (such as cultural or social gathering).

The occurrence of such relevant events is notified to the *Contextual Filtering* component by the *Event detection* component, which provides additional metadata describing the event. Listing 2 illustrates an example of an annotated event about a traffic jam, as it is received by the *Contextual Filtering* component. Information about the user context can be gathered by the *Contextual Filtering* in several ways: it can be either explicitly stated in the event request or in the user application (e.g. specifying events of interest), or it can be explicitly or implicitly acquired by identifying user's current activity (e.g. using the speed to detect that the user is in a car, or having the user specifying in the application what type of transportation he/she is using). With this information, the *Contextual Filtering* component is able to: i) select, among the list of detected filtered events, the ones that are contextually relevant, and ii) continuously rank their level of criticality to decide when an action is to be triggered.

The level of criticality of events is dynamically assessed by the *Contextual Filtering* component based on metrics such as the distance between a detected event and user's current location, or an explicit measure of how severe the event is (referred to as Event Level). The current implementation of the *Contextual Filtering* component uses a linear combination of these metrics. An application developer can configure such metrics and users can modify them in order to satisfy their own requirements. In terms of implementation, the *Contextual Filtering* component is encoded as logical rules. The underlying logic used to implement the *Contextual Filtering* component is based on the Stable Model Semantics of Answer Set Programming (ASP) [36] and relies on the efficient implementation of such semantics in the Clingo engine [37]. The ability of ASP to support common-sense and default reasoning makes it possible to activate and deactivate certain rules when a combination of unexpected changes in the real world affect each other.

Listing 3 is an extract of the logic rules used in the *Contextual Filtering* component. Rule 1 filters out unrelated events. Rule 2 generates sets of solutions containing one critical event each per solution, provided that the event is not[15] expired. Rules 3-7 compute the criticality of an event. Based on this level of criticality, the *Contextual Filtering* refers back to the user application that an action is required, which can be either generating a new request for the *Decision support* component to automatically provide an alternative (better) solution, or informing the user about the critical event let the user decide whether a new solution is needed. The fully declarative nature of ASP facilitates automatic generation of such rules for application developers, starting from high-level specification of what events are relevant for the application and how the ranking metrics are formally defined. This is a crucial advantage for customizing the *Contextual Filtering* component to application-dependent behaviour.

Context-awareness has been long studied in the domain of service provision and mobile computing, as the ability of an application to "automatically adapt to the discovered context (environmental situations or conditions) by changing

[15]Note that we use default negation here, a powerful way of reasoning by default that is typical of non-monotonic approaches such as those based on ASP.

the application behavior according to the latest context"[38], but scalable solutions to do that in dynamic settings and going beyond complex event detection towards non-monotonic reasoning is still under investigation [39], [40].

The main novelty of this approach used by the *Contextual Filtering* component for adaptive and context-aware reasoning in dynamic environments is the ability to use such complex reasoning capabilities to efficiently and dynamically select only relevant information to tackle the challenge of converting data into knowledge in dynamic environments in a scalable way.

2) EVENT-BASED USER-CENTRIC DECISION SUPPORT

The *Decision support* component of the CityPulse framework represents *higher-level intelligence*, and the main role of this component is to enable reactive decision support functionalities to be easily deployed, providing the most suitable answers at the right time.

The reasoning capabilities needed to support users in making better decisions require handling incomplete, diverse and unreliable input, as well as constraints and preferences in the deduction process. This expressivity in the *Decision support* component is achieved by using a declarative non-monotonic logic reasoning approach based on Answer Set Programming. Semantic technologies for handling data streams, in fact, cannot exhibit complex reasoning capabilities such as the ability of managing defaults, common-sense, preferences, recursion, and non-determinism. Conversely, state-of-the-art logic-based non-monotonic reasoners can perform such tasks but are only suitable for data that changes in low volumes at low frequency. Therefore, the main challenge addressed by the *Decision support* component is to enable expressive reasoning for decision support in a scalable way. In our approach we combine the advantages of semantic query processing and non-monotonic reasoning based on the general idea behind StreamRule [41]. We have identified features that can affect scalability of such a combined approach and we have exploited the user-centric features and adaptive filtering of relevant events to reduce the input size and therefore the search space for the decision support task, thus increasing the potential for better scalability. As mentioned earlier in this paper, the user-centric and event-driven features of the *Decision support* component strongly rely on the tight interaction with the user application and the *Contextual Filtering* component respectively, with the former being in charge of the bidirectional communication with the other two, as represented in Figure 12.

In what follows, we detail the input and output used by the *Decision support* component, and also introduce the main reasoning modules that are currently implemented as part of the CityPulse framework. The input for the *Decision support* component is illustrated in Figure 13.

A Reasoning request consists of:

- User Reference: uniquely identifies the user that made the request. Such reference is related to user credentials that will be used in the final integration activities in order to manage user logins and instances of the CityPulse framework in different cities.
- Type: indicates the reasoning task required by the application. This is used directly by the *Decision support* component to select which set of rules to apply, and needs to be identified among a set of available options at design time by the application developer.
- Functional Details: represent the qualitative criteria used in the reasoning task to produce a solution that best fits the user needs.

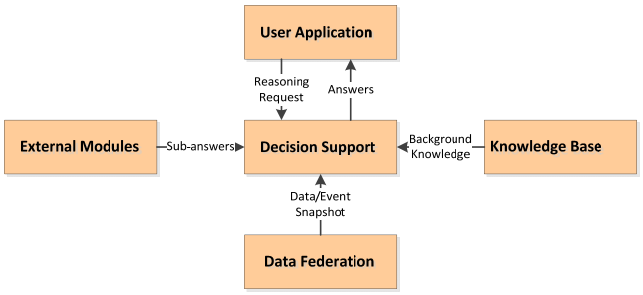

Figure 13. Decision Support I/O.

The Functional details are composed by:

- Functional Parameters, defining mandatory information for the Reasoning Request (such as start and end location in a travel planner scenario);
- Functional Constraints, defining a numerical threshold for specific functional aspects of the Reasoning Request (such as cost of a trip, distance or travel time in a travel planner scenario). These restrictions are evaluated as hard constraints, which needs to be fulfilled by each of the alternative solutions offered to the user;
- Functional Preferences, which encode two types of soft constraints: a qualitative optimisation statement defined on the same functional aspects used in the Functional Constraint (such as minimisation of the travel time); or a qualitative partial order over such optimization statements (such as preference on the minimisation of the distance over minimization of the travel time). Preferences are used by the Decision support component to provide to the user the optimal solution among those verifying the functional constraints.

Given to the fully declarative approach of ASP, we are able to provide a specification of all aspects of a Reasoning Request, which can be automatically mapped into logic rules. As a result, we achieve a high degree of flexibility to adapt to new scenarios and requirements, which makes it easier for developers to build new applications. In the reasoning process performed by the *Decision support* component, declarative rules derived from the Reasoning Request are combined in a single logic ASP program with the following input:

- Sub-Answers from external modules. These are facts computed by external models as subtasks of the decision support problem. Calls to such external modules can be used to improve scalability by reducing the

solution space, or for privacy reasons. For example, in a travel planner scenario, a reduced list of all possible routes to go from A to B within a geographical area is provided by the *Geospatial Data Infrastructure* component. This does not imply any contextual reasoning, qualitative optimization, or event-driven adaptation, which is instead provided by the *Decision support* component.

- Background Knowledge. This is static information about a particular domain (such as the location of parking areas).
- Events Snapshot. This information comes from the *Data federation* component, and consists of the latest values of related events in the city (such as traffic levels in particular areas). Events snapshots are used the first time a solution is computed and ensures this solution is based on the most updated values. Dynamic changes to these values are then continuously detected and used by the *Decision support* component via the adaptive filtering mechanism of the *Contextual Filtering* component.

The *Decision support* component produces a set of answers to the Reasoning Request that satisfy all user's requirements and preferences in the best possible way. These solutions are computed by applying sets of rules deployed as scenario-driven decision support modules. We currently support three different types of decision support modules, covering a broad range of application scenarios:

- *Routing Module*. It provides the best solution(s) for a routing task, which is continuously updated based on incoming events and their criticality; the travel planner scenario relies on this module, and so do other scenarios where finding the optimal route is the main task, but the selection criteria (including constraints and preferences) might vary. Examples include the ability to schedule optimal pick-up for health services, green bike tours and alike.
- *Optimal Selection Module*. It provides the best selection among a set of alternative based on optimisation criteria, constraints and preferences; the parking scenario is part of this category.
- *Planning Module*. It provides optimal solutions to a planning problem, and continuously updates the options when the selected one is no longer feasible (based on Functional Constraints) or is no longer optimal (based on Functional Preferences); scenarios that are part of the cultural sector (such as planning activities in the city based on user interests or schedule) or the energy sector (such as planning household usage based on user-defined constraints and cost) are candidate scenarios for using this module.

These modules are available as API to be used by application developers in a broad range of applications. Since decision support tasks are strongly domain-dependent, different type of reasoning tasks would require additional Decision Support modules to be developed. The CityPulse framework provides a set of guidelines for developing new decision support modules, which requires knowledge of Answer Set Programming to develop the proper application logic.

3) TECHNICAL ADAPTATION

The CityPulse framework leverages the *Technical adaptation* component to automatically detect critical quality updates for the federated data streams used in the *Data federation* component and make adjustments. IoT streams are inherently dynamic in nature and often unreliable, hence more prone of getting fluctuations in the quality metrics. Therefore, it is utmost necessary to have a quality-aware adaptation mechanism for IoT streams. Using the *Technical adaptation* component, the quality of user queries deployed by the *Data federation* component can be maintained automatically by dynamically replacing data sources, unlike existing adaptive CEP systems which leverage query-rewriting and re-ordering of the query operators.

The CityPulse framework facilitates adaptability in quality-aware federation of IoT streams for smart city applications. In order to provide technical adaptation to increase robustness of smart city applications, following three steps are involved:

1. Monitoring Quality Updates: to monitor any updates in the quality metrics of the IoT streams involved in stream federation;
2. Evaluate Criticality: to determine whether any particular quality update is critical and if there is any adaptation action that should be carried out based on the composition plan and non-functional requirements; and
3. Adaptation Handling: when adaptation is triggered, determine the adaptation scope (i.e., part of the composition plan (produced by the *Data federation* component) that needs to be changed), make an adaptation request to the data federation and recompose the adaptation scope and redeploy the newly derived composition plan.

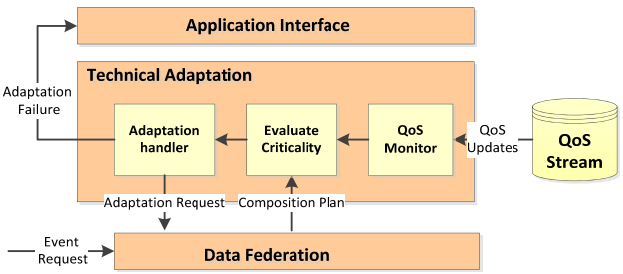

Figure 14. Adaptability in stream federation.

Figure 14 illustrates the process and the components used to achieve adaptability in stream federation following the above mentioned three steps.

During the event streams discovery and composition process, numerical quality vectors can be used to specify constraints over the quality metrics for data streams. For exam-

ple, if we consider some typical quality attributes, including latency, price, energy consumption, bandwidth consumption, availability, completeness, accuracy and security, a numerical quality vector:

$$Q = \langle L, P, E, B, Ava, C, Acc, S \rangle$$

can be used to specify the quality of a data stream w.r.t. these dimensions along with the static data stream description. Similarly, a quality constraint vector Q' can be used to specify the quality constraints as thresholds for the quality metrics in the stream discovery or composition requests. If none of the quality values in Q breaks the threshold in Q', the data stream is considered to be a candidate for the discovery and composition. A weight vector:

$$W = \langle Lw, Pw, Ew, Bw, Avaw, Cw, Accw, Sw \rangle$$

contains 0 to 1 weights for each quality metrics within the requests, representing the preferences over quality metrics. The weight vector is used to normalize the quality vectors for all candidates based on simple additive weighting, the results are ranked and the top candidate is chosen. During runtime, the quality vectors will be re-computed based on most recent quality value updates, and the constraints are re-evaluated to determine if the quality of the data stream still satisfy the constraints. If optimal candidates must be used for the application domain at all times, updated quality vectors are also re-ranked, and the new top candidate is chosen to be the adaptation result. It is worth mentioning that such preferences and requirements are specified as a default configuration within a specific application, and they can be overwritten by user specific settings.

The *Technical adaptation* component is closely integrated with the *Data federation* component. It receives quality updates disseminated in the *Data Bus*. The quality updates are originally produced by the Quality Monitoring component. The *Technical adaptation* component is transparent to the end user as long as the adaptations are successful. Otherwise it may produce a failure notification to the application interface. The adaptation strategies of the *Technical adaptation* component can be configured by a 3rd party developer by changing the *AdaptationMode* parameter in the API offered by the *Data federation* component. Then, an initialization API is invoked at the backend system to create an *AdaptationManager* instance for the specific request.

C. COMPONENTS INTERDEPENDENCIES

The CityPulse components are highly flexible which allow multiple configurations of exploitation. In other words, the application developer can deploy only a subset of the components based on the requirements of the application which have to be developed.

There are interdependencies among the components of the framework and it is possible that the application developer will have to deploy also other CityPulse components in order to have a certain feature running.

Considering the out of the box installation, when the application developer simply deploys and configures a component, he has to deploy also the CityPulse components which are bellow the considered component in the architecture (see Figure 1). For other types of installations the application developer can replace one or several components with its own custom made modules. This is possible because the components are using REST and AMQP protocols to communicate.

IV. CONTEXT-AWARE REAL TIME TRAVEL PLANNER

In order to demonstrate how the CityPulse framework can be used to develop applications for smart cities and citizens, we have implemented a context-aware real time *Travel Planner* using the live data from the city of Aarhus, Denmark. The scenario was defined in collaboration with the IT departments of the Aarhus municipality and the following criteria have been considered: the impact of the application for the citizens and the data availability. The selected scenario belongs to the traffic domain, but as it was presented earlier in this paper the CityPulse framework is generic and it can be applied in any domain.

This scenario aims to provide travel-planning solutions, which go beyond the state of the art solutions by allowing users to provide multi dimensional requirements and preferences such as air quality, traffic conditions and parking availability. In this way the users receive parking and route recommendations based on the current context of the city. In addition to this, *Travel Planner* continuously monitors the user context and events detected on the planned route. User will be prompted to opt for a detour if the real time conditions on the planned journey do not meet the user specified criteria anymore.

In this case, the application developer has performed the following activities:

- Configured the CityPulse framework components;
- Deployed the components into a back end server;
- Developed a smart phone application using the APIs exposed by the framework in order to respond to the user requests. This application does not perform any data processing.

For this application, and very probably for most of the CityPulse enabled applications, the framework components can be divided in two different categories. First of all, the large scale data stream processing modules are configured and deployed in order to permanently monitor the status of the Aarhus city. In this way, when a new user logs into the mobile application and generates a request for a recommendation, the back-end application can provide the answer based on the current traffic or pollution situation from the city.

The second category of components, which perform real-time adaptive urban reasoning, are triggered when the user requests the routing or parking recommendation.

The following subsections present the specific workflows for the two categories of components mentioned above. For each particular workflow the activities, which have to be per-

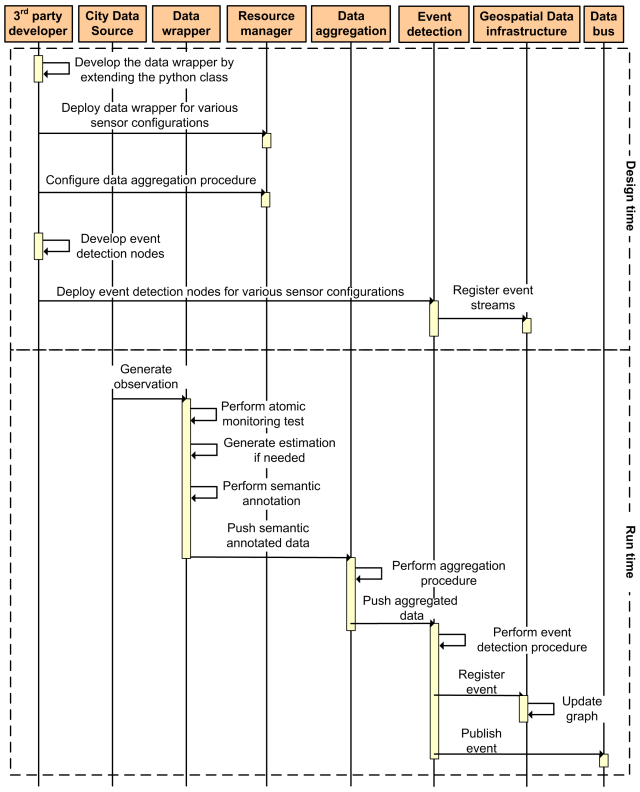

Figure 15. **Large scale data stream processing workflow.**

form by the application developer at design time, are marked at the beginning.

A. LARGE SCALE DATA STREAM PROCESSING WORKFLOW

For the considered scenario the large scale data stream processing modules are configured to process in real time the parking and traffic data coming from the city sensors with the scope of detecting relevant events for the users traveling in the city. Figure 15 depicts the workflow and this subsection is dedicated to explain the activities.

For the realization of the envisaged Travel Planner application the required data streams must be made available within the framework at first in the design phase. For this, corresponding *Data wrappers* for both data streams needed to be developed. The Aarhus Traffic wrapper was realised implementing a HTTP Pull Connection (i.e. HTTP client) extending the abstract Sensor Connection. It fetches new sensor readings from the ODAA portal. The response messages are JSON encoded documents. Consequently a JSON Parser was implemented to extract the relevant information out of these messages, namely the number of vehicles passing the two measurement points (''vehicleCount'') and their average speed (''avgSpeed'').

The second *Data wrapper* implemented fetches the parking data of parking garages in Aarhus. Similar to the traffic data stream the Aarhus Parking data is provided by the ODAA platform and encoded as JSON message. Therefore the same HTTP Pull Connection but a different JSON Parser is used. The stream provides information about the total number of

```
insert into ParkingGarageStatusStream
select * from ParkingGarageStream.win:time(parkingMonitoringInterval sec)
having (max(ParkingGarageStream.
numberOfCars)-min(ParkingGarageStream.numberOfCars)/
ParkingGarageStream.parkingCapacity) > ocupancyChangeRateTreshold)
```

Listing 4. **Detect parking occupancy rate events.**

parking spaces in the garage (''totalSpace'') and the number of occupied spaces/vehicles in the garage (''vehicleCount'').

Both *Data Wrappers* were deployed in the *Resource management*. During run time new observations are fetched in a five minute interval for the Aarhus Traffic stream and in a one minute interval for the Aarhus Parking stream respectively.

The next step was to configure the *Data aggregation* component to use the SensorSAX algorithm as aggregation method for the traffic and parking observations. Due to the fact that it is a multi-resolution approach, it is triggered based on the variation in data stream, and can be configured by sensitivity level to instantly report any change to the system for further processing, such as *Event detection*.

The last step considered during the design time was to develop the Event detection nodes, which are used to process the Aarhus traffic and parking aggregated data streams with the scope of identifying relevant events from the end user perspective. In that sense, Event detection nodes have been developed and deployed in order to detect the traffic jams and the parking status changes. In the following paragraphs we will present the Event detection node used for parking places fast vacancy modification (in other words when a lot of vehicles enter the parking garage in a very short period of time).

The input of the event detection adapter is represented by one parking data stream and the configuration parameters are:

- *ocupancyChangeRateTreshold*: the rate of car entering into the garage in order to generate the event;
- *parkingMonitoringInterval*: the length of the time interval for which the occupancy change rate is computed.

The statement from Listing 4 represents the detection logic which has to be included into the parking Event Detection node in order to achieve the goal. First, it computes the minimum and the maximum number of cars which have been in the garage in the last *parkingMonitoringInterval* seconds. Then, the statement determines with what percentage the parking occupancy has been modified by dividing the difference between the maximum and the minimum values to the total capacity of the garage. If the percentage is bigger than the *ocupancyChangeRateTreshold*, then a parking place fast vacancy modification event is generated.

For the implementation of the Travel Planner application the domain experts do not need to make changes to the *Atomic monitoring* nor *Composite monitoring*. As the quality calculation follows an application independent approach, there is no need for any changes. The adaptations the Domain Expert has to do are limited to providing a description for newly deployed sensors (SensorDescription, see Listing 1)

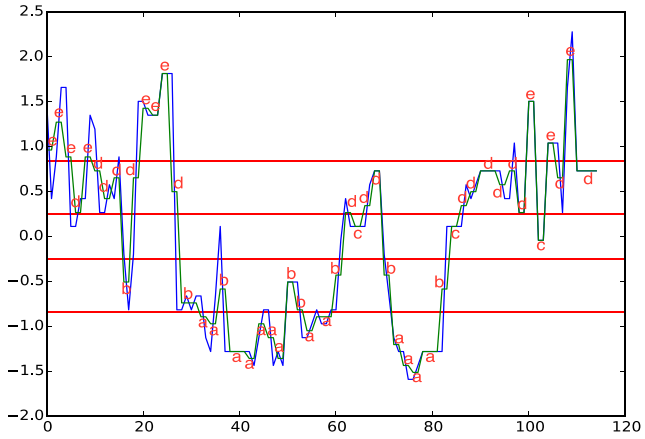

Figure 16. Data aggregation using SensorSAX with minimum window size is 1 and sensitivity level is 0.3 for average speed observation of Aarhus traffic data stream.

used by the application. In the case of the Travel Planner application SensorDescriptions for the traffic and the parking stream were provided. The introduced QoI Explorer enables the application developer to check the quality of sensors along a selected route and to check the results for plausibility.

At run time, after the CityPulse components have been deployed, the *Data wrappers* start to fetch observations from the Aarhus city data streams. When an observation is received the atomic monitoring is performed and the QoI description of the stream is updated. If the observation is missing or the quality of the stream is low, the *Fault recovery* component is triggered to generate an estimation.

The observations along with the QoI determined in the *Atomic monitoring* is annotated semantically afterwards. The annotation process is generic and uses the details provided in the SensorDescription, where required information such as the general domain of the stream; the nature/concept of an observation; and the unit of measurement for the observation are specified.

Next, the numeric values of the observation are aggregated according using the selected algorithm. Figure 16 depicts the data captured for average speed via the corresponding sensor points and illustrate SensorSAX patterns created from the raw data.

At the end the aggregated streams of observations are processed by the *Event detection* component in order to extract the parking and traffic events. Once an event is detected it is published on the *Data Bus*, to be further used by the *Decision support* and *Contextual Filtering* components and a notification is sent to the *Geospatial Data Infrastructure*. In this way, the process of computing the routes (see the Geospatial Data Infrastructure APIs from Section 3) is influences by the current city context.

B. REAL-TIME ADAPTIVE URBAN REASONING FOR TRAVEL PLANNER

For the considered scenario, the real time adaptive urban reasoning components are used to provide answers, when a user generates routes and parking recommendation requests.

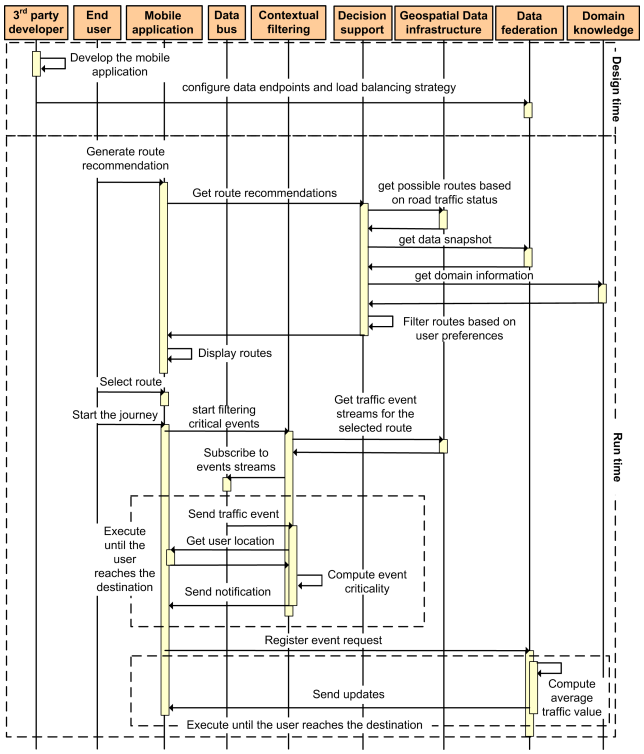

Figure 17. Real-time Adaptive Urban Reasoning workflow for Travel Planner.

Figure 17 depicts the workflow executed by the components for providing an answer when route a recommendation is requested.

As mentioned above, all the CityPulse framework components are deployed on a back-end server and are accessible via a set of APIs. As a result of that the application developer has only to develop a user-friendly front-end application, which calls the framework APIs. In our case we have developed an Android application.

Figure 18 depicts the user interfaces used by the end user to set the travel preferences and the destination point.

After the user has filled in the details and made the request using the user interface, the mobile application generates the appropriate request for the *Decision support* component which has the following main fields:

- Type: indicating what decision support module is to be used for this application (“TRAVEL-PLANNER” in this case);
- Functional details: specifying possible values of user’s requirements, including:
  - Functional parameters: mandatory information that the user provides such as starting and ending locations, starting date and time, and transportation type (car, bicycle, or walk).
  - Functional constraints: numerical thresholds for cost of a trip, distance, or travel time.
  - Functional preferences: the user can specify his preferences along selected routes, which hold the