GRANT AGREEMENT No 609035
FP7-SMARTCITIES-2013

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

Collaborative Project

User-Centric Decision Support in Dynamic Environments

Document Ref.	D5.2
Document Type	Report
Workpackage	WP5
Lead Contractor	NUIG
Author(s)	Alessandra Mileo, Stefano Germano, Thu-Le Pham, Dan Puiu, Daniel Kuemper, Muhammad Intizar Ali
Contributing Partners	NUIG, SIE, UASO
Planned Delivery Date	M24
Actual Delivery Date	31st August 2015
Dissemination Level	Public
Status	Completed
Version	V1.0-final
Reviewed by	AI (Joao Fernandes), SAGO (Josiane Xavier Parreira)

Executive Summary

Smart city applications require not only techniques dedicated to dynamicity handling, but also the ability to take into account contextual information, user preferences and requirements, and real-time events to provide optimal solutions and configuration of smart city applications for the end user. The goal of the user-centric is to combine all these information sources to support reactive smart city applications that can provide a decision support solution that considers a number of user-centric factors in providing optimal matches. To this aim, user activities and contexts as well as city events are modeled and semantically enriched using Linked Data and ontologies that provide an interoperable and well-established foundation to support reasoning and decision support. Since the provided user-centric solution may become outdated due to adverse events occurring in the real world, the Decision Support component strongly interacts with the Contextual Filtering component that is in charge of detecting and reacting to such adverse events.

This deliverable describes the concepts around User-Centric Decision Support, presents extensions and combination of LOD vocabularies to semantically enrich and interlink user context and city events, and details how scalable complex decision tasks are specified and implemented as different reasoning modules. Interaction between the components and with the user and application is also detailed, and some insights are provided on the potential of combining the event filtering capability of the CityPulse framework with user preferences, user constraints and computationally complex reasoning for finding optimal solutions in an efficient way.

Table of contents

List of Figures

Abbreviations List

APIs	Application Programmable Interfaces
ASP	Answer Set Programming
CEP	Complex Event Processing
CF	Contextual Filtering
CQELS	Continuous Query Execution over Linked Streams
C-SPARQL	Continuous SPARQL
DS	Decision Support
GDI	Geospatial Data Infrastructure
GeoDB	Geospatial Data Base
GIST	Generic Index Structure
LOD	Linked Open Data
OSM	Open Street Map
OWL	Ontology Web Language
RDF	Resource Description Framework
RSP	RDF Stream Processing
UML	Unified Modeling Language
W3C	World Wide Web Consortium

1. Introduction

This deliverable describes the User-Centric Decision Support capability of the CityPulse Framework, which is strongly connected to user preferences, constraints and context coming from the application, and with the real-time adaptive urban reasoning capability realized by the Contextual Filtering component.

In this document we provide the semantic model for representing user-centric and event-driven aspects of the decision support process, and we present the algorithms and methods developed to ensure a scalable and flexible interplay between user-centric factors, dynamic aspects of the changing environment within the city, and reasoning capabilities required for decision support.

The remainder of this deliverable is organized as follows: Section 2 describes requirements and state-of-the-art on complex reasoning over data streams. Section 3 provides details about our information model for user-centric and for event-driven aspects of the CityPulse Decision Support capability. Section 4 illustrates in detail how the user-centricity aspect is formalized and automatically translated into a declarative specification. Section 5 focuses on the Decision Support functionality and its modules, while Section 6 provides details regarding the implementation of the components and related technologies that have been used to materialize the Decision Support capability within the CityPulse framework. We summarize the outcome of this activity as well as on-going work and future steps in Section 7.

2. Complex Reasoning over Data Streams: Requirements and State-of-the-art

Smart city applications require IoT discovery and matchmaking techniques dedicated to dynamicity handling. Information taken into account during the matchmaking process originates from diverse data sources including: data streams, city services, the user's social context, situational awareness (e.g., user location), preferences and application configurations. Additionally, results of the decision making process can be reflected in the real world by using actuators. For example, in smart city applications for traffic management traffic lights can be dynamically reconfigured by using actuators, according to the learned usage patterns and determined traffic intensity.

Factors such as user interests and reputation requirements can be considered in the decision support process. The exploitation of user profiles has a great potential for providing more individualized decision support to the user, and users might want to use it when interacting with smart city applications. Although elicitation and usage of user profiles is optional, we considered their potential and included explicit aspects of user profiles in the decision support process. These aspects include not only user location but also contextual activities and their dependencies with city events for a particular user in specific application settings. Insights on this user-centricity is better detailed later in this section, and further details are provided later in this document when we discuss how user-centricity is exploited (Section 4), and its role in User-Centric Decision Support (Section 5).

Smart city applications that provide decision making support mechanisms to users can help in determining optimal solutions (e.g., the best transport options depending on users' location, time and day of the week, the healthiest cycling paths based on user preferences, the best path based on

traffic conditions parking availability or cultural events and so on) using a wide spectrum of data sources.

In the remainder of this section we provide some insights on how User-Centric Decision Support exploit contextual information, use preferences and requirements to provide optimal configurations of smart city applications. Analysis of the users' context, explicit and implicit requirements and preferences will enable the decision-making component to determine city services and data sources according to context-dependent attributes such as user location and time. The main objective of enabling this user-centric approach is to provide a solution that considers a number of user-centric factors in providing optimal matches, taking into consideration the individual requirements of users of smart city applications.

2.1 Requirements

The concept of stream reasoning is considered as the application of complex reasoning techniques to data streams.

Web Stream Reasoning has emerged as a research field that explores advances in Semantic Web technologies for representing and processing data streams on one hand, and emerging approaches to perform complex rule-based inference in changing environments on the other hand. Enabling such complex reasoning on Web Stream still presents some key challenges and is therefore becoming an active area of research. From the analysis of several application scenarios, the authors in [1] extracted the key challenges for stream reasoning systems. These challenges also include the capability of reasoning for decision support, which is a key functionality for the CityPulse Decision Support component.

Such challenges include:

- **Data Integration**. In most of the scenarios collected within the project, data comes from multiple sources with various data types. This raises issues of representing and combining heterogeneous data for processing. Moreover, stream reasoning systems also use domain knowledge for performing reasoning tasks, and combine such knowledge with dynamic data streams. This background knowledge is mainly static and time-independent. This integration is challenging because retrieving and analysing large volumes of dynamic data and static knowledge can be particularly inefficient in terms of processing time and also expensive in terms of costs and amount of human intervention required.
- **Scalability.** The scalability is typically evaluated on two aspects: computational complexity (i.e., the ability to perform more complex tasks) and input size (i.e., the ability to process a larger input). It is essential that the reasoning process is scalable in both aspects.
- **Expressivity.** All scenarios aim at deriving high level knowledge from large volumes of low level knowledge. Expressivity of the Decision Support and reasoning component is known to be inversely related to its performance - the more expressive the reasoner is, the longer it takes to perform reasoning. Finding the right trade-off between expressivity and scalability is a key issue.

There are various existing approaches aiming to perform reasoning over data streams [2]. In stream processing, the existing solutions are divided into two categories: (1) Data Stream Management Systems and (2) Complex Event Processing (CEP) [1]. The former approach has some well-known engines such as CQELS and C-SPARQL that have the ability to continuously process low-level data streams at high rate. The latter approach considers observable raw data as primitive events and expresses composite events by some specific operators. These approaches do not manage incomplete information and do not perform complex reasoning tasks that are required in decision support processes, and therefore they fall into the category of stream query processing (mostly based on pattern matching) rather than stream reasoning as we intend in this document.

2.2 State-Of-The-Art

In the Semantic Web and Linked Data realm, technologies such as RDF, OWL, SPARQL have been extended to provide mechanisms for processing semantic data streams [3][4][5]. However, the variety of real-world applications in the IoT space require reasoning capabilities that can handle incomplete, diverse and unreliable input and extract knowledge from it to support users in decision tasks. While semantic technologies for handling data streams cannot exhibit complex reasoning capabilities, logic-based non-monotonic reasoning approaches can be quite costly in terms of efficiency.

To reach the goal of combining the advantages of both query processing for semantic web streams and non-monotonic reasoning, in the last few years some approaches have been proposed in the area of Semantic Web and knowledge representation. The authors in [6] focus on distributed methods for non-monotonic rule-based reasoning that tries to achieve better scalability using the MapReduce framework. Their current works perform parallel defeasible reasoning under the assumption of stratification, which imposed a severe limitation considering the range of allowed rule set, limitations that are not restrictive in more general ASP-based approaches to non-monotonic reasoning. Other attempts focus on extending the well-established declarative complex reasoning framework of ASP with dynamic data [7]. M. Gebser et al [8] proposed modelling approaches for continuous stream reasoning based on reactive ASP, utilizing time-decaying logic programs to capture sliding window data in a natural way. This is a first step towards gearing ASP to continuous reasoning tasks. However, these approaches still mainly process on low changing data and relatively smaller data sizes. Do et al. also use ASP in their stream reasoning system and the approach is based on the DLV system [9], which does not deal with continuous and window-based reasoning over data stream within the reasoner.

A similar approach is proposed in [10], where the authors present the StreamRule framework, which combines a stream processing engine and a non-monotonic reasoner. Despite some preliminary investigations, no detailed evaluation is currently available to assess the performance of StreamRule, although this work provides a baseline for exploring the applicability of complex reasoning on Semantic Web Streams.

In CityPulse, we investigate how adaptive and user-centric event filtering can be combined with the Application Logic and User Context for scalable User-Centric Decision Support. Based on the general idea behind StreamRule, we identified features that can affect scalability of such a combined system

and we exploited the exploited user-centric features and adaptive filtering of relevant events to reduce the input size and therefore the search space of the decision support task, thus increasing the potential for better scalability. Despite further investigation is required to identify optimal configuration of the combined system, the feasibility of our approach as demonstrated in the implemented scenarios is promising [11]. We are also heavily involved with the standardization activities within W3C around RDF Stream Processing[1], where more complex forms of reasoning for RDF streams are being considered both in terms of their formal specification and implementation in concrete systems.

3. Event-based and User-Centric Decision Support

In the dynamic environment such as smart cities, events based decision support is a critical aspect for smart city applications. Applications need to provide support for real time decision support whenever a new event occurs. In this section, we discuss: (i) higher-level architecture for event based and user-centric decision support, and (ii) information model to represent events and their effect over users' context.

3.1 Architecture and Data Flow

A graphical representation of the data flow is provided in Figure 1. An event-based decision support system is a combination of the Contextual Filtering and Decision Support components. All the communication between both components is controlled by the application. Having such architecture provides complete control to the end user for making any decision or selection related to the proposed solutions and it will be onus of the user to accept or decline alternative solutions provided by the decision support after the occurrence of any critical event.

It is also worth mentioning that an alternative approach of direct communication between CF and DS components can possibly lead to the automatic decisions on the behalf of user without any prior approval. As a tradeoff between rich user experience (taking automatic smart decisions on user behalf without annoying notifications) and user control (application requires permission from the user before making any event based decision on user behalf), we propose a user defined configuration at the application level, which allows users to define more granular controls to select the most suitable option for them.

[1] https://www.w3.org/community/rsp/
[2] http://citypulse.insight-centre.org/uco
[3] prefix event: http://purl.org/NET/c4dm/event.owl#
[4] prefix prov: http://www.w3.org/ns/prov#
[5] https://www.mat.unical.it/calimeri/projects/embasp/

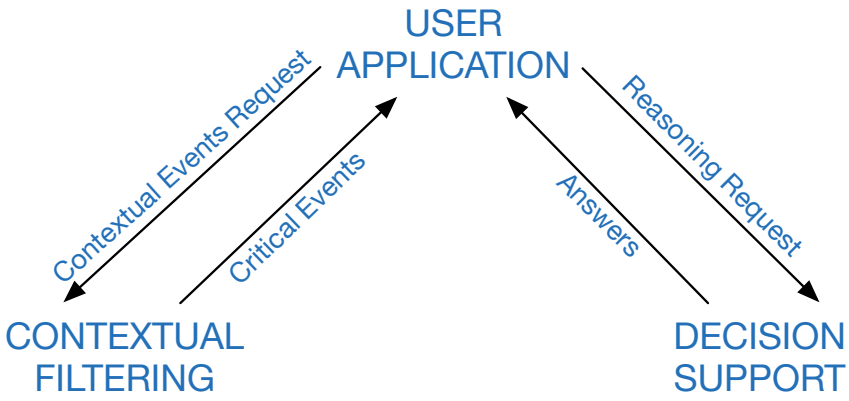

Figure 1 – Event-based User-Centric Decision Support

We consider the following three steps for event based User-Centric Decision Support component after the occurrence of any critical event:

(a) Identify a set of candidate solutions while keeping user context in the view;
(b) Identify any further events that can possibly relate to the newly identified candidate solutions and measure the level of effect of any events on the quality of the selected best option;
(c) Continuously monitor events to ensure the selected solution remains the best option throughout the life span of the application, automatically searching for alternative solutions whenever the selected best option is no longer the best.

3.2 User Context Ontology

In this section, we discuss the information model (user context ontology[2]) used to represent events and their effect over user context.

Figure 2 shows a graphical representation of the information model for city events, users' context and effect of city events over users' context. We used the Event Ontology [3] and extended it to represent city events. We categorize events into different types based on different application domains and effect of events over that domain. All city events belong to an EventCategory depending on the application domain e.g. cultural event, transportation event, and environmental event. Event categorization is not limited to the 3 categories shown in Figure 3 and it can be extended to cover more categories and therefore support a broader range of application domains.

We also introduce the concept of PredictedEvent within the user context ontology, which refers to the predicted events assessed by the smart city data analysis components. For example, based on the traffic congestion level and number of vehicles counts approaching the congested area, the CityPulse smart city framework can predict a traffic jam level in advance. We also associated the concept of *level* with each event, which is a numerical representation of a range of sub-events within a single city event. For example a transportation type event *Traffic Jam* can be represented with a

[2] http://citypulse.insight-centre.org/uco
[3] prefix event: http://purl.org/NET/c4dm/event.owl#

range of levels from 1 to 3 indicating slow traffic jam (level 1), minor traffic jam (level 2), and severe traffic jam (level 3) respectively.

We used the PROV Ontology[4] to represent the current (ongoing) activity (prov:Activity) of the user. The Activity concept represents the current context of the user and determines what actions are associated with the user at a given time and place. We link prov:Activity with the event:Event concept using the uco:affects property, which determines whether the specific city event has potential of affecting the ongoing activity of the user (user context).

We also introduce the concept of uco:UserProfile, which can contain additional information about the user and can be used to collect and semantically represent additional information while determining the current context of the user or effect of city event over the activity associated with that user.

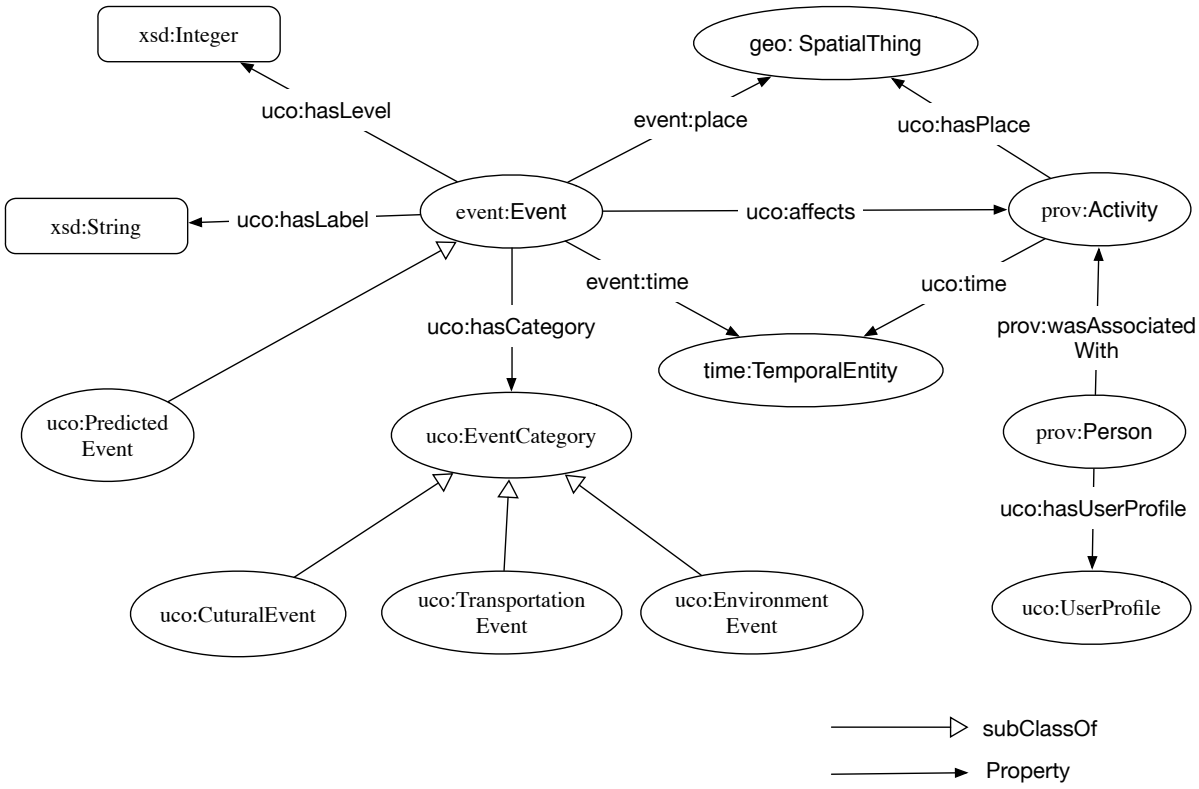

Figure 2 - City Event - User Context Ontology

[4] prefix prov: http://www.w3.org/ns/prov#

Table 1 shows a sample city event belonging to the transportation event category. Any city event can be described using five basic properties, namely *hasCategory*, *hasLabel*, *hasLevel*, *time*, and *place*.

Table 1 - A sample transportation event

hasCategory	TransportationEvent
hasLabel	TrafficJam
hasLevel	3
time	2015-10-15T12:00:00
place	(10.1321411, 56.1929527)

4. User-centricity: Parameters, Constraints and Preferences

User requirements and preferences are modelled using Linked Data and open vocabularies in order to provide a lightweight, interoperable and well-established foundation for decision support. User requirements and preferences can be specified explicitly and mapped into a representation that is independent from the specific application.

Preference-driven and constraint-based reasoning provides a powerful mechanism where dynamicity is a key issue, and enables to find an optimal match between the needs/preferences of citizens, and available data streams and city services. This section describes how user preferences, constraints and profiles are represented and mapped automatically into deduction rules to be used in the decision support process by the Decision Support Component developed in Activity 5.2.

4.1 Reasoning Request Representation and Mapping

The *Reasoning Request* is the request that an application/user should send to the CityPulse Framework in order to perform a task that involves the Decision Support component.

As illustrated in Figure 3, the Reasoning Request consists of:

- User Reference
- Type (T)
- Functional Details
    - Functional Parameters (Π)
    - Functional Constraints (Γ)
    - Functional Preferences (Ω)

In what follows we will detail each element of the Reasoning Request and define the automatic mapping or translation into deduction rules used by the Decision Support component. Such translation is defined in a general way so that independently of the Functional Details defined as strings and values, an automatic declarative rule-based specification can be obtained, which is seamlessly combined with the rules in the Decision Support Module used by a specific application.

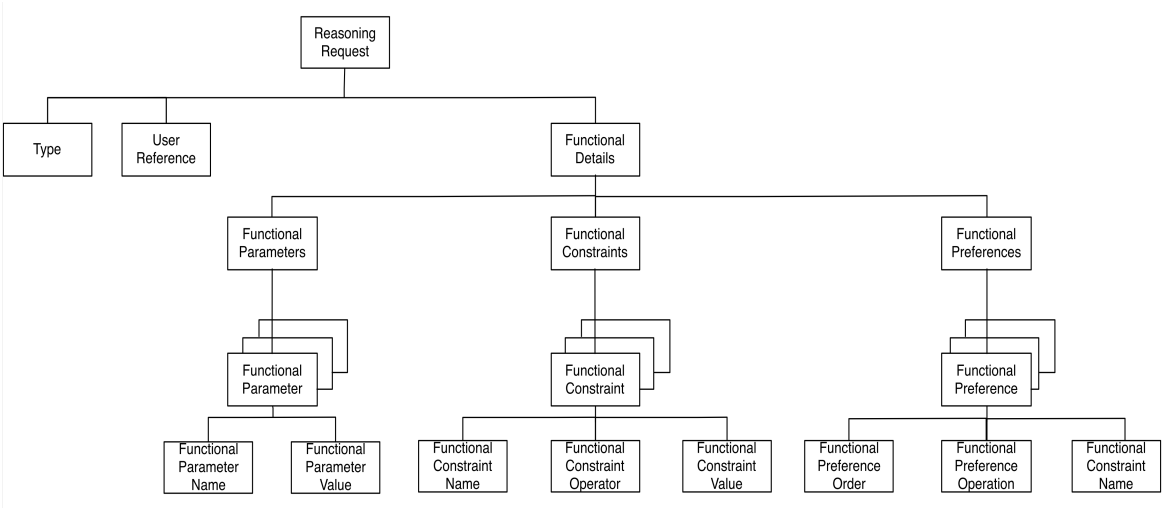

Figure 3 - Reasoning Request

The User Reference is simply an identifier of the user that made the request. Such reference is meant to be a unique identifier related to user credentials that will be used in the final integration activities (part of WP6) in order to manage user logins and instances of the CityPulse framework in different cities.

The Type determines the reasoning task required by the application. This is used directly by the Decision Support Component to perform the correct task using the reasoning engine, and needs to be identified among a set of available options at design time by the application developer. Such options have been defined for the implemented scenarios, and customization and extension of available types will be possible via the APIs as part of the outcome of Activity 5.3 due for completion at month M30.

The Functional Details represents the actual criteria for the reasoning task required by the user. In the remainder of this section we provide a precise specification of each aspect composing the functional details, and illustrate how they are automatically mapped and translated into logic rules.

4.1.1 Functional Parameters (Π)

A Functional Parameter defines a mandatory information for the Reasoning Request (for instance the "ending point" in a travel planner scenario).

The set of Functional Parameters (Π) is composed by a finite set (of cardinality n_Π) of individual Functional Parameter ($\pi^i;\ \pi^i \in \Pi\ \forall i \in [1; n_\Pi]$).

Each Functional Parameter (π^i) is composed of:

- Functional Parameter Name (N_{π^i})
- Functional Parameter Value (V_{π^i})

i.e. $\pi^i = \langle N_{\pi^i}, V_{\pi^i} \rangle$.

The Functional Parameter Name (N_{π^i}) is a *string* taken from a fixed set of strings ($\Theta_{T,N_{\pi^i}}$) and the Functional Parameter Value (V_{π^i}) is specific for each scenario.

The set of Functional Parameters (Π) is translated as the concatenation of the translations of each Functional Parameter (π^i) that composes it.

Each Functional Parameter ($\pi^i = \langle N_{\pi^i}, V_{\pi^i} \rangle$) is translated as:

$$parameter\left(N_{\pi^i}, V_{\pi^i}\right).$$

The Functional Parameter Value can be a single value or a set of possible values. When the Functional Parameter Value (V_{π^i}) is a set (e.g. expressed as enumeration of possible values), it is translated in several of the above facts, one for each item in the set.

4.1.2 Functional Constraints (Γ)

A Functional Constraint defines a numerical restriction about a specific aspect of the Reasoning Request. This restriction is "strict" and needs to be fulfilled by each of the answers (otherwise referred to as solutions) offered to the user.

The set of Functional Constraints (Γ) is composed by a finite set (of cardinality n_Γ) of individual Functional Constraint (γ^i; $\gamma^i \in \Gamma \; \forall i \in [1; n_\Gamma]$).

Each Functional Constraint (γ^i) is composed of:

- Functional Constraint Name (N_{γ^i})
- Functional Constraint Operator (O_{γ^i})
- Functional Constraint Value (V_γ)

i.e. $\Gamma^i = \langle N_{\gamma^i}, O_{\gamma^i}, V_{\gamma^i} \rangle$.

The Functional Constraint Name (N_{γ^i}) is a *string* taken from a fixed set of strings ($\Theta_{T,N_{\gamma^i}}$), and the Functional Constraint Operator (O_{γ^i}) is an *arithmetic operator* taken from a fixed set ($\Theta_{O_{\gamma^i}} = \{=, \neq, >, <, \geq, \leq\}$). For each Functional Constraint Operator (O_{γ^i}) we denote with $\bar{O}_{\gamma^i}$ its *complementary operator* ($= \mapsto \neq, \neq \mapsto =, > \mapsto \leq, < \mapsto \geq, \geq \mapsto <, \leq \mapsto >$).

The Functional Constraint Value (V_{γ^i}) is an *integer number*.

The Functional Constraints (Γ) is translated as the concatenation of the translations of each Functional Constraint (γ^i) that composes it.

Each Functional Constraint (γ^i) is translated as:

- The "real" constraint

$$\leftarrow violatedConstraint\left(N_{\gamma^i}\right).$$

- A rule to derive if it is violated:

$$violatedConstraint\left(N_{\gamma^i}\right) \leftarrow valueOf\left(N_{\gamma^i}, AV\right), AV \; \bar{O}_{\gamma^i} \; V_{\gamma^i}.$$

4.1.3 Functional Preferences (Ω)

A Functional Preference defines a "soft" constraint or priority among the verification of specific aspect of the Reasoning Request. This restriction is "weak" and should be optimized by the Decision Support component in order to provide the optimal or most preferred answers to the user.

The set of Functional Preferences (Ω) is composed by a finite set (of cardinality n_Ω) of individual Functional Preference (ω^i; $\omega^i \in \Omega \; \forall i \in [1; n_\Omega]$).

Each Functional Preference (ω^i) is composed of:

- Functional Preference Order (O_{ω^i})
- Functional Preference Operation (Opt_{ω^i})
- Functional Constraint Name (N_{ω^i})

i.e. $\omega^i = \langle O_{\omega^i}, Opt_{\omega^i}, N_{\omega^i} \rangle$.

The Functional Preference Order (O_{ω^i}) is an *integer* $\in [1; n_\Omega]$, and the Functional Preference Operation (Opt_{ω^i}) is an *optimization operator* taken from a fixed set ($\Theta_{Opt_{\omega^i}} = \{minimize, maximize\}$).

The Functional Preferences (Ω) is translated as the concatenation of the translations of each Functional Preference (ω^i) that composes it.

Each Functional Preference (ω^i) is translated as:

$$\#Opt_{\omega^i}\{AV@O_{\omega^i} : valueOf(N_{\omega^i}, AV)\}.$$

To allow more flexibility in the logic program each Functional Preference (ω^i) could be also translated as (in addition to the previous translation):

$$preference(O_{\omega^i}, Opt_{\omega^i}, N_{\omega^i}).$$

Table 2,

Table 3 and Table 4 illustrate a concrete example of the possible values of Functional Details for the Travel Planner scenario. Note that for functional parameters and constraints, we specify value type and actual value to provide a clearer idea of how the actual values are formatted and therefore parsed. Functional preferences are expressed over functional constraints, therefore the VALUE refers to the NAMO of the respective Functional Constraint on which the preference is expressed.

Table 2 - Example of functional parameters for the Travel Planner scenario

Functional Parameters	NAME	VALUE Type	Value
Starting Point	STARTING_POINT	Coordinate (Lat Long)	56.17888121694039 10.153993614949286
Ending Point	ENDING_POINT	Coordinate (Lat Long)	56.15183187883248 10.154508599080145
Starting Time/Date	STARTING_DATETIME	Date	2012-04-23T18:25:43.511Z
Transportation Type	TRANSPORTATION_TYPE	Enum	{CAR, WALK, BICYCLE}

Table 3 - Example of functional constraints for the Travel Planner scenario

Functional Constraints	NAME	OPERATOR	VALUE Type	VALUE
Travel time less than X	TRAVEL_TIME	LESS_THAN	Duration	15
Distance less than X	DISTANCE	LESS_THAN	Number	1000
Pollution amount less than X	POLLUTION	LESS_THAN	Number	13,5

Table 4 - Example of functional preferences for the Travel Planner scenario

Functional preferences	OPERATION	VALUE
Travel time	MINIMIZE	TRAVEL_TIME
Distance	MINIMIZE	DISTANCE
Air quality	MINIMIZE	POLLUTION

Note that in the example of Functional Constraints in Table 3, travel time is intended in minutes, distance in meter and pollution in CO2 emission. The application interface accommodates different units of measurements for time and distance that are understood and interpreted accordingly.

4.2 User Application and Decision Support Capabilities: Interactions

The CityPulse framework is designed to expose a set of APIs, which can be used for developing smart real-time applications that can leverage the processing power of the framework. Depending on the application requirements, it is possible that a CityPulse application needs to interact with the Decision Support Component to provide support for decision-making and recommendations to the user. In order to do that, the user interface of the application has to contain several controls that allow the user to configure/select the functional constraints, the functional parameters, and functional preferences in a seamless and intuitive way. As detailed earlier in this document, the functional parameters are used to define mandatory information required to compute solutions to the decision task, while functional constraints and preferences allow the user to specify hard and soft constrains to be considered when suggesting what solutions are considered best matches. The answer to such a composite reasoning request contains a list of possible solutions to the decision task requested by the user of the application.

In principle, the Reasoning Request is designed to allow a high degree of flexibility when defining the requirements and the functional preferences to adapt to new scenarios and applications. During the design time of the CityPulse application, the designer/software architect can decide how much of this flexibility has to be exposed to the user. As a result, based on the type of the application, it is

possible that for some functional preferences some default values are to be selected and hardcoded, while for some other applications a palette of options can be exposed to the user via the GUI.

In order to demonstrate the viability of the framework, the consortium has developed an Android application combining the following two scenarios from the CityPulse scenario portfolio:

- Context-aware multimodal real time travel planner, which provides the ideal route for its users while taking the current context of the user into account.
- Public Parking Space Availability Prediction, which handles the problem of finding a public parking space within the city.

Figure 4 contains a few snapshots of the android application.

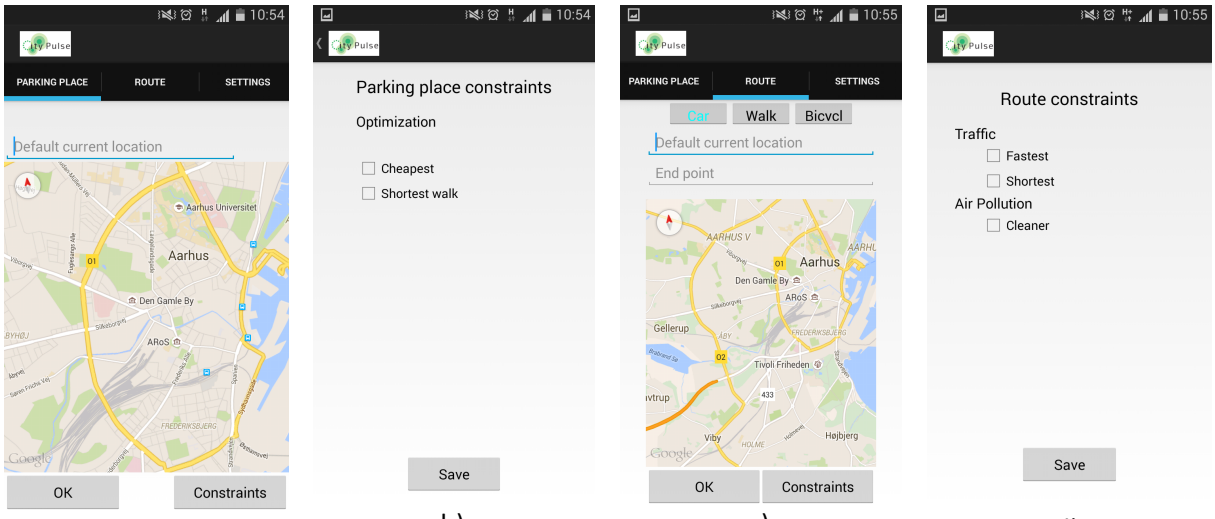

Figure 4 - Snapshots of the CityPulse enabled application: a) Parking place selection menu; b) Parking place constraints menu; c) Route selection menu; d) Route constraints menu

Figure 4 a) depicts the menu used to specify the point of interest for which the user requests a parking place in the nearby (it represents the requirement). The menu from Figure 4 b) is used to specify the preferences that have to be considered. In this case the user can opt for the cheapest parking place or the closest one to the point of interest. In a similar way, Figure 4 c) depicts the interface used to select a route from the parking place to the point of interest and Figure 4 d) is used to specify preferences for the route scenario.

Figure 5 depicts the payload message of a Reasoning Request for a parking place recommendation. In the figure are highlighted the functional constraints, the functional parameters, and functional preferences. The response containing the recommended parking places to be visualized in the application interface is listed in Figure 6.

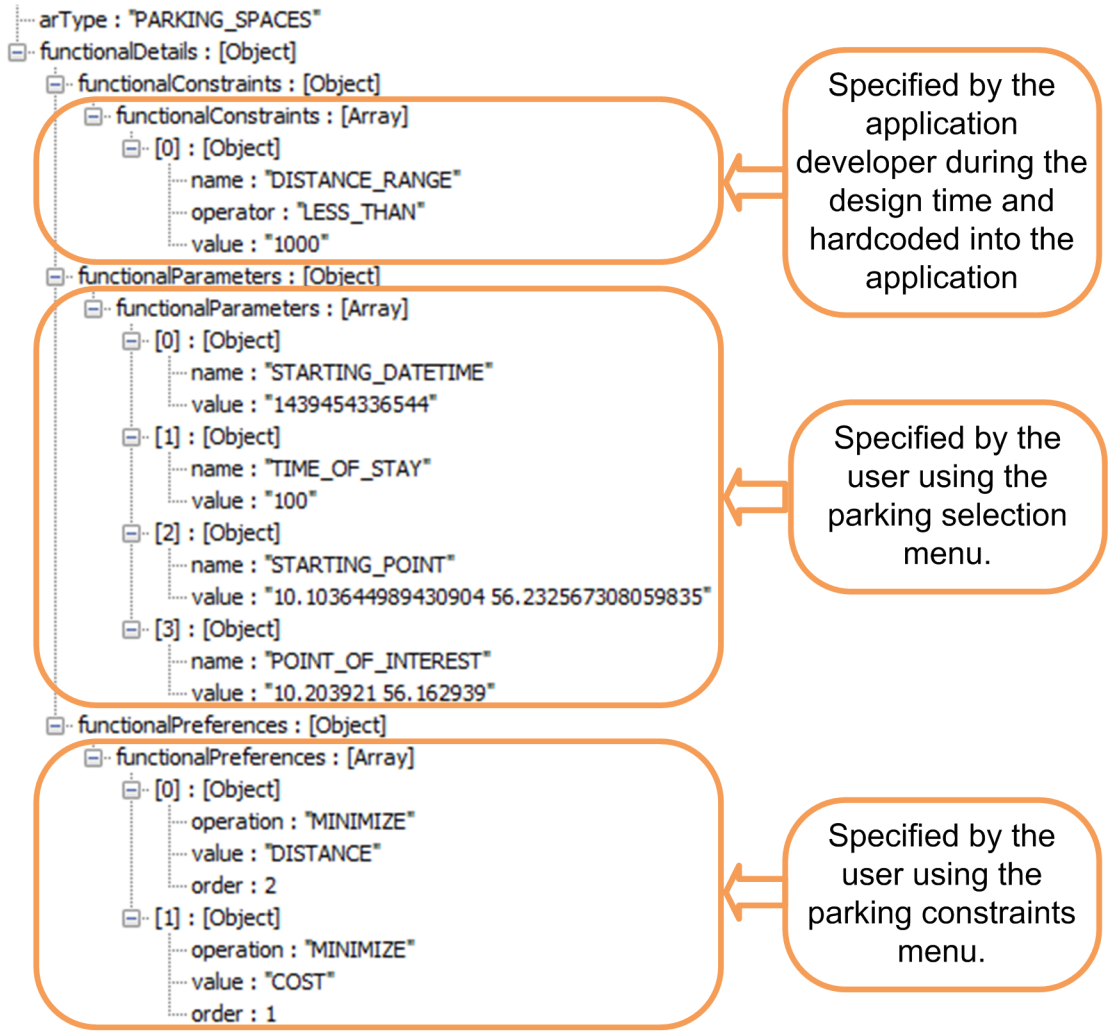

Figure 5 - The payload of a parking request for decision support

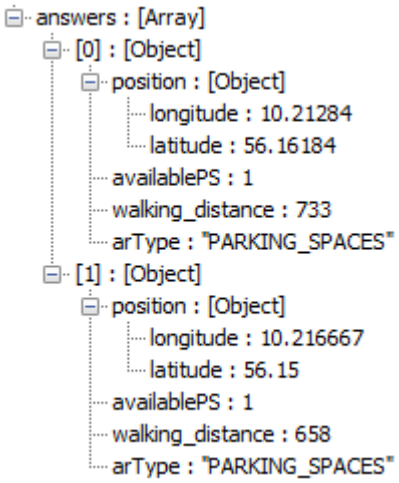

Figure 6 - A sample response of the parking decision support request

5. User-Centric Decision Support: From Event Streams to Decisions

The goal of the CityPulse User-Centric Decision Support is to utilize contextual information available as background knowledge, user patterns and preferences from the application as well as real-time events to provide optimal configurations of smart city applications and enable reactive smart city applications to be deployed.

The conceptual interactions between the Decision Support component and the other functional components in other work packages is illustrated in Figure 7.

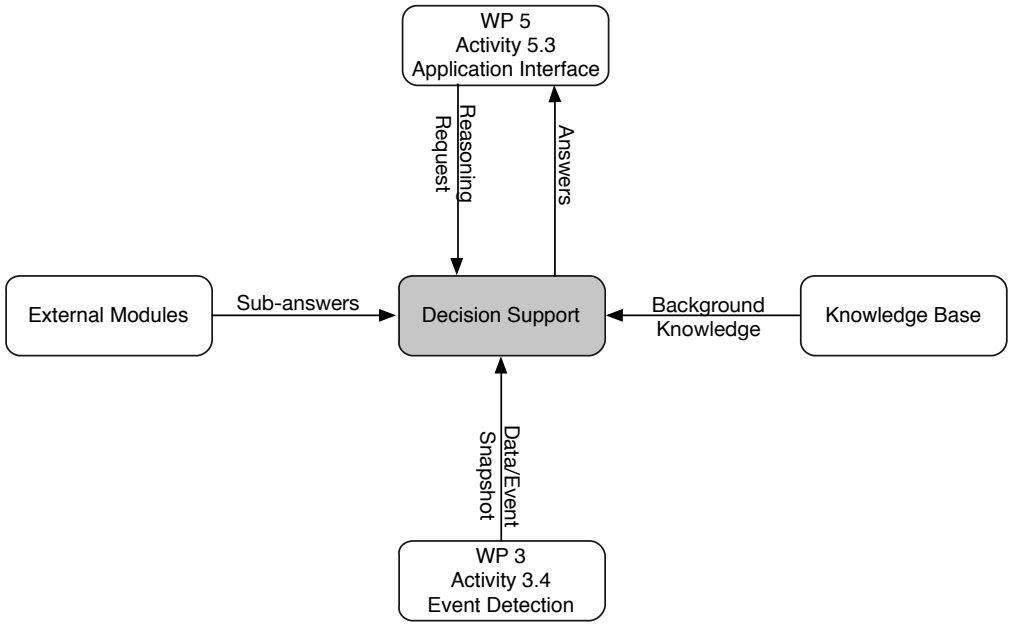

Figure 7 - Decision Support I/O

Figure 8 illustrates the process and the components used to provide adaptive and user-centric decision support based on our requirements.

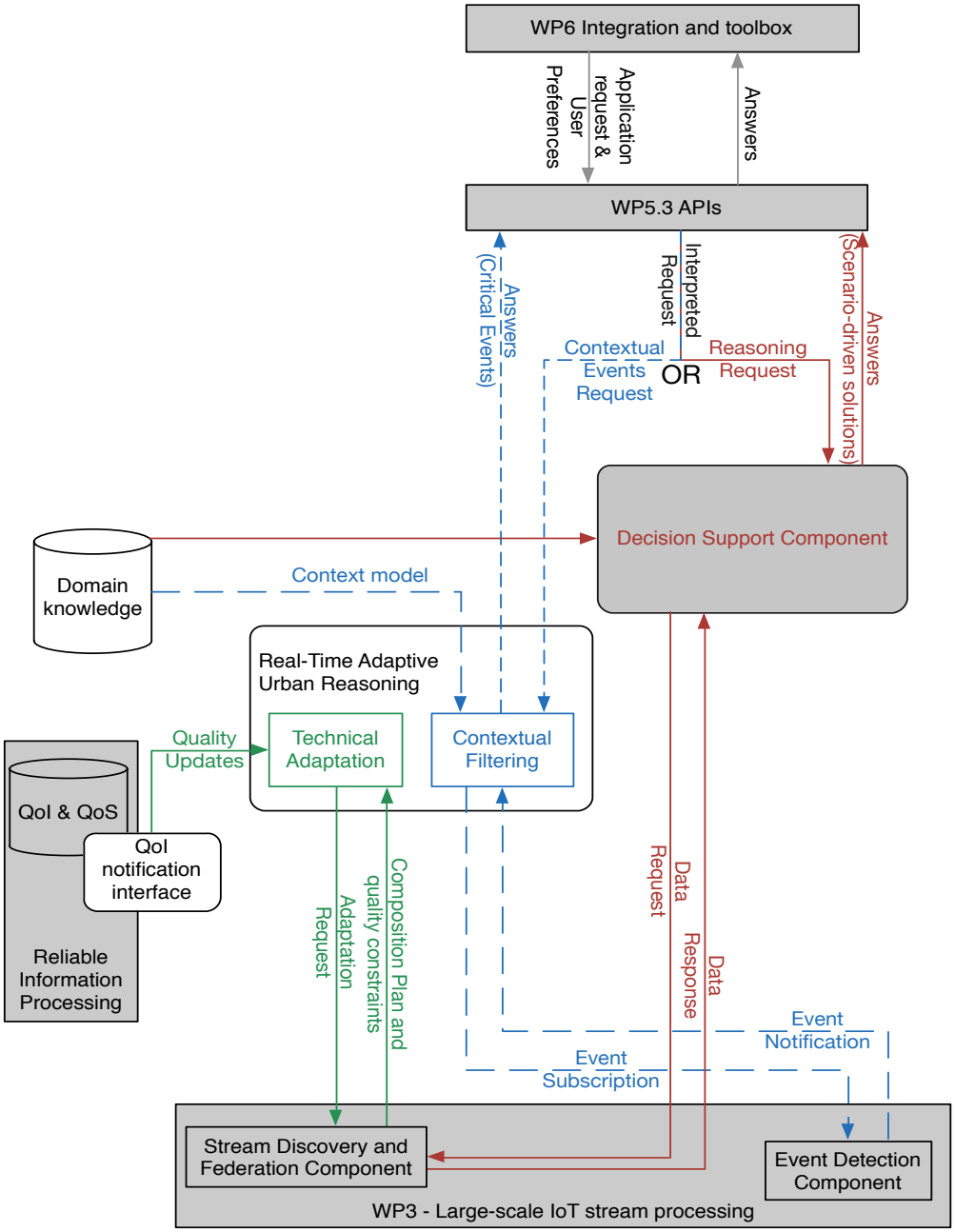

Figure 8 – User-centric and Adaptive Decision Support

The Decision Support component gets as input i) a reasoning request from the application interface which includes the parameters, constraints, and preferences of the user, ii) the background knowledge which is dependent on the scenario, iii) the annotated data or event snapshot about the scenario from WP3, and iv) the sub-answers from External Modules when they are used in the application development at design time.

The output provided by the Decision Support component consists of a list of scenario-driven solutions, which are optimal and satisfy all user's requirements and preferences in the best possible way.

User-centric solutions provided by the Decision Support component may become out of date or no longer optimal due to adverse events occurring in the environment. To deal with this situation, the Decision Support component in A5.2 interacts with the Contextual Filtering component in A5.1 through the application to detect and react to such adverse events, via notification and suggestions of possible adaptations in finding a new set of solutions for the user.

We support three different types of modules for the Decision Support functionality:

- **Routing**: provides the best solution(s) for a routing task, continuously updated based on incoming events and their criticality; the travel planner scenario is part of this category and so are other scenarios involving routing and using other selection criteria (such as in the healthcare domain for pick-up and nursing services, environmental scenarios related to green paths for bikes and alike).
- **Optimal selection among alternatives**: provides a selection of one specific item based on a set of optimization constraints and preferences; the parking scenario is part of this category.
- **Planning**: provides alternative solutions to a planning problem, and continuously updates the options when the selected one is no longer available or is no longer the "best" one according to the user 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.

An additional Visualization and Exploration capability is supported, which mostly provides storing and retrieval capabilities that are required for visual analytics through the CityPulse dashboard. This capability is entirely developed as part of the Visualization in Dynamic Smart City Environments activity (Activity 5.3) due at month M30.

6. Implementation Details

In this section we provide implementation details of the current functionalities of the User-Centric Decision Support component. A simplified UML representation of the physical deployment of the components is presented in Figure 9.

As detailed in previous sections, the User-Centric Decision Support functionality is composed by different modules.

The main module (*DSManager*) receives a request, to start the components needed for the reasoning task and to return as output the answers (or solutions) for that specific task.

The RequestRewriter module in addition is used to translate the **Functional Details**, as explained in details in Section 4.1, from Objects to *logic rules* that can be interpreted by the ASP solver. This module uses the mapping explained in that same section to perform the translation.

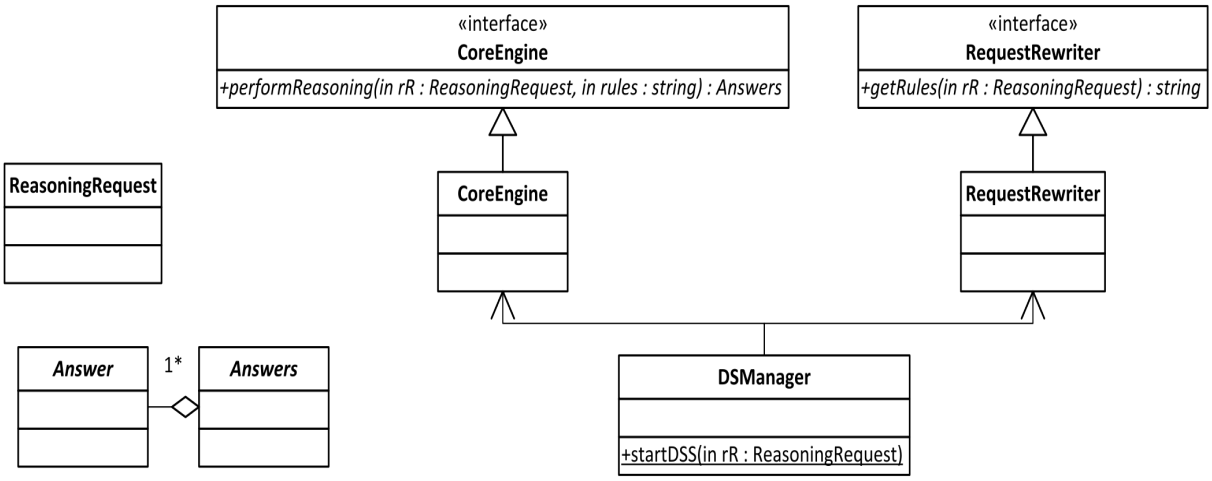

Figure 9 – Deployment Diagram for Real-Time Adaptive Urban Reasoning

In addition, the *CoreEngine* module, using the **EmbASP** framework, is able to invoke the ASP solver and to collect Answer Sets as plain Java objects.

Listing 1 is an extract of the logic rules used in the ASP program for the Travel Planner scenario.

```
1. input_get_routes(SP, EP, V, 5) :-    parameter("STARTING_POINT", SP),
                                    parameter("ENDING_POINT", EP), route_costMode(V).
2. route(@get_routes(SP, EP, V, N)) :- input_get_routes(SP, EP, V, N).
3. route_data(@get_routes_data(SP, EP, V, N)) :- input_get_routes(SP, EP, V, N).
4. max_pollution(@get_max_pollution(RouteID)) :- selected(RouteID).
5. 1 <= {selected(RouteID) : route((RouteID, _, _))} <= 1.
6. valueOf("TRAVEL_TIME", ApproximatedTime) :- selected(RouteID),
                                        route_data((RouteID, Time, _)),
                                        ApproximatedTime = Time / 60.
7. valueOf("DISTANCE", ApproximatedDistance) :- selected(RouteID),
                                        route_data((RouteID, _, Distance)),
                                        ApproximatedDistance = Distance / 100.
8. valueOf("POLLUTION", Pollution) :- max_pollution(Pollution).
9. parameter("STARTING_POINT","10.116919 56.226144").
10. parameter("ENDING_POINT","10.1591864 56.1481156").
11. :- violatedConstraint("POLLUTION").
12. violatedConstraint("POLLUTION") :- valueOf("POLLUTION", AV), 135 < AV.
13. #minimize{AV@2 : valueOf("DISTANCE", AV)}.
14. #minimize{AV@1 : valueOf("TRAVEL_TIME ", AV)}.
```

Listing 1 -Logic decision support rules for the Travel Planner scenario

In order to deploy this solution and implement the intended algorithm for decision support we used the well-known Guess/Check/Optimize paradigm so that the intended behaviour of the logic rules in the program are easy to understand, enrich and customize thanks to their declarative specification.

Rules 1-4 in Listing 1 collect the possible routes and other data about them using *external atoms* that invoke **external modules** to receive this information.

Rule 5 in Listing 1 is a so-called *choice rule* (the *Guess* part in the ASP approach) that selects one route per solution.

Rules 6-8 in Listing 1 project the specific atoms of the scenario into the "generic" atoms used in the translated **Functional Constraints** and **Preferences**.

Rules 9-14 in Listing 1 are automatically translated from the **Functional Details**. This includes mapping and translation of the following functional details:

- **Parameters** (rules 9-10), translated as simple *logic facts*;
- **Constraints** (rules 11-12), translated as *strong constraints* and also referred to as the *Check* part, which eliminates the solution produced in the *Guess* part that are violating any of the strong constraints;
- **Preferences** (rules 13-14), translated as *optimize statements*; the *Optimize* part which ranks the solutions to provide only those that are qualitatively better with respect to the optimization statements used.

6.1 Related Technologies

The Decision Support component relies on some existing tools and technologies, integrated, extended and deployed to achieve the objective of this component. We briefly discuss each of these tools and also elaborate on how these tools communicate with our components.

6.1.1 External modules

External modules are exploited by the Decision Support component directly inside the ASP program, using the so called "external atoms". By using this technique, the ASP reasoner is able to invoke the external modules interactively, only when needed and possibly using data that can be derived as a result of other reasoning tasks. This offers a very powerful compositionality property to our solution, and it also increases the scalability of the stream reasoning process for decision support.

In order to illustrate how external modules can be used, we provide in what follows some details on the Routing sub-reasoning process.

To support decision support in CityPulse, various tasks need the possibility to find adequate ways from one location inside the city to another. Several online services for simple path finding/routings are freely available on the Internet but do not allow individual routing tasks, e.g.:

- Utilising individual road metrics for routing
- Support of areas that should be avoided
- Provide the result as a number of alternative paths

To overcome these functional gaps, the OpenStreetMap (OSM) Data was used to generate an efficiently searchable database, which is able to perform alternative path finding algorithms on a routable graph. With the usual handling of street segments (*edges*), which are connected through intersections (*nodes*) in a routing component, properties like:

- Length of a segment
- Maximum driving speed on the segment
- One-Way Restrictions
- Accessibility for vehicles/bike cycles/pedestrians

are used to calculate the *cost* to pass an *edge* between two *nodes.* This information is stored in the geospatial database, hosting the OSM information. To enable application specific path finding/routing, an additional layer containing edge-based cost-multiplicators for distinct tasks is introduced. It allows marking edges, which should be avoided or preferred, with multiplicators that alter the cost, based on specific topics, e.g.:

- Pollution (which should be avoided)
- Traffic (which should be bypassed)
- Scenic routes (which could be preferred).

To find suitable routes between multiple locations, different shortest-path search algorithms are used. The Dijkstra-algorithm [12] can be used for a simple search, which does not have special requirements to the additional multiplicators. However, since it builds a tree from a source to any other node in the graph (until it finds the target/destination node) it is a quite time consuming way to find the route. If multiplicators (modificators of the heuristics) are limited to multiplicator>=1, a best-first-search A* Algorithm [13] can be used to reach the target more efficiently. A* uses a best-first search and finds a least-cost path from a given starting node to one target node. As A* traverses the graph, it builds up a tree of partial paths. In contrast to the Dijkstra algorithm, the priority queue orders the leaf nodes by a cost function, which combines a heuristic estimate of the cost to reach a goal and the distance travelled from the initial node. Specifically, the cost function is

$$F(n)=g(n)+h(n)$$

g(n) is the known cost of getting from the initial node to n. h(n) is a heuristic estimate of the cost to get from n to any goal node.

For the implementation, the native Java/Python clients for the following GDI components have been created:

- PostgreSQL database with Postgis extension
- PGrouting library

Figure 10 shows the result of a routing request returning 10 different routes with different cost between an origin and destination of a route search. To get these different paths, the multiplicators of the edges of each best path were increased after each iteration.

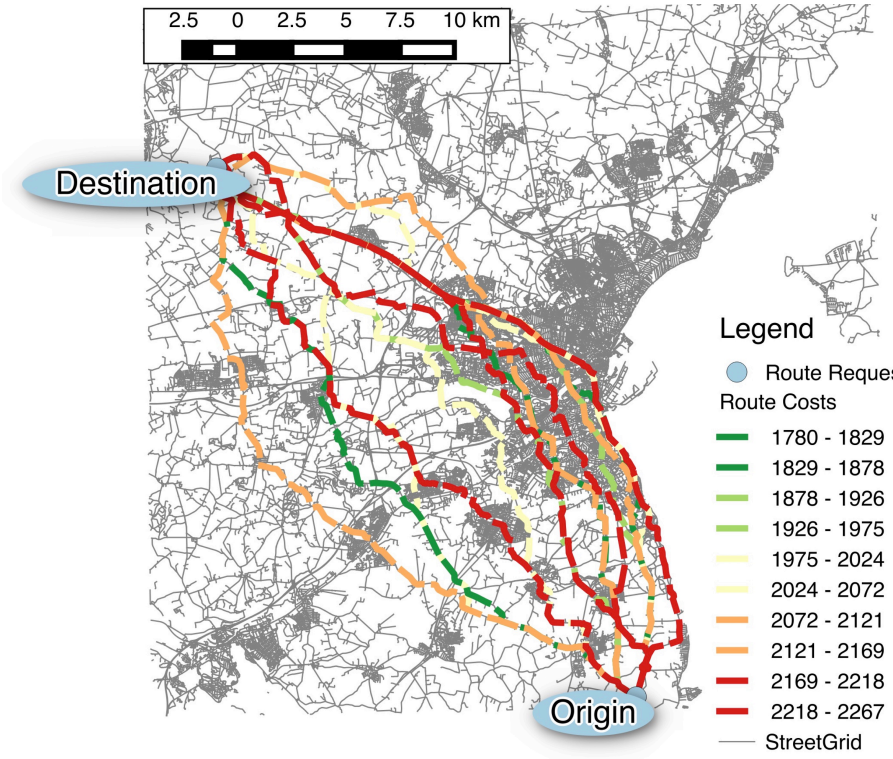

Figure 10 - Search of multiple paths between two locations

Figure 11 shows the same request with a (red marked) area, which should be avoided because of a severe interruption in traffic. Therefore the multiplicators of the edges are set to 15.

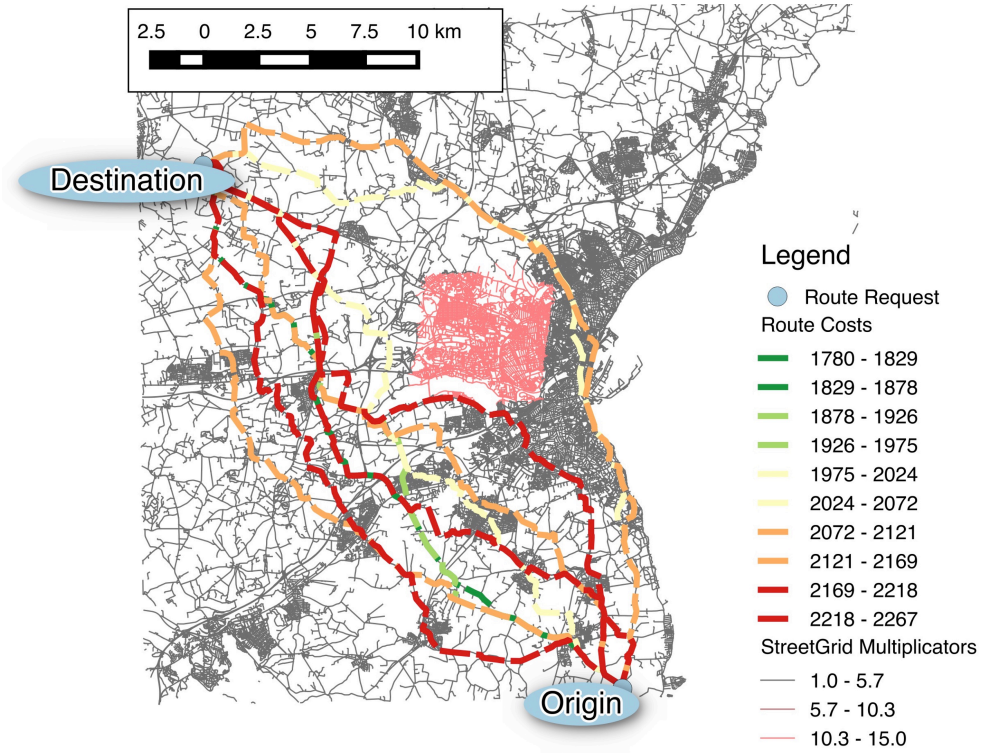

Figure 11 - Search of multiple paths between two locations with an avoidance area of 3km

6.1.2 Geospatial database

To overcome the limitations of semantic data storage and processing, a geospatially enabled database is used to implement efficient geospatial and spatiotemporal reasoning processes. This goal can be achieved by utilising an optimised search index for geospatial objects. Since most locations in the cities can't just be described as a set of two coordinates (e.g. longitude, latitude in the WGS84 system of GPS) it is necessary to support more complex geometric types like Linestrings and Polygons to represent roads, public places, polluted areas, etc. Processing these kinds of complex geometries needs a qualified geospatial indexing process to find nearby objects, process overlapping areas and generate adapted routing graphs.

CityPulse utilises a PostgreSQL database with a PostGIS extension to enable those technologies in the project. Using the Generic Index Structure (GIST) overcomes the limitation of the default index type B-Tree [14]. B-Tree indices are not lossy (inexact) in the way a GIST index can be. This means that while the GIST index only indexes the bounding box of the geometry, the B-Tree must index the entire geometry, which can often be larger than the index can cope with. Therefore a first selection of query-results can be found by simply using the min/max longitude instead of always comparing the full values/geometries. Figure 12 depicts the Architecture of the Geospatial Data Infrastructure (GDI) connection.

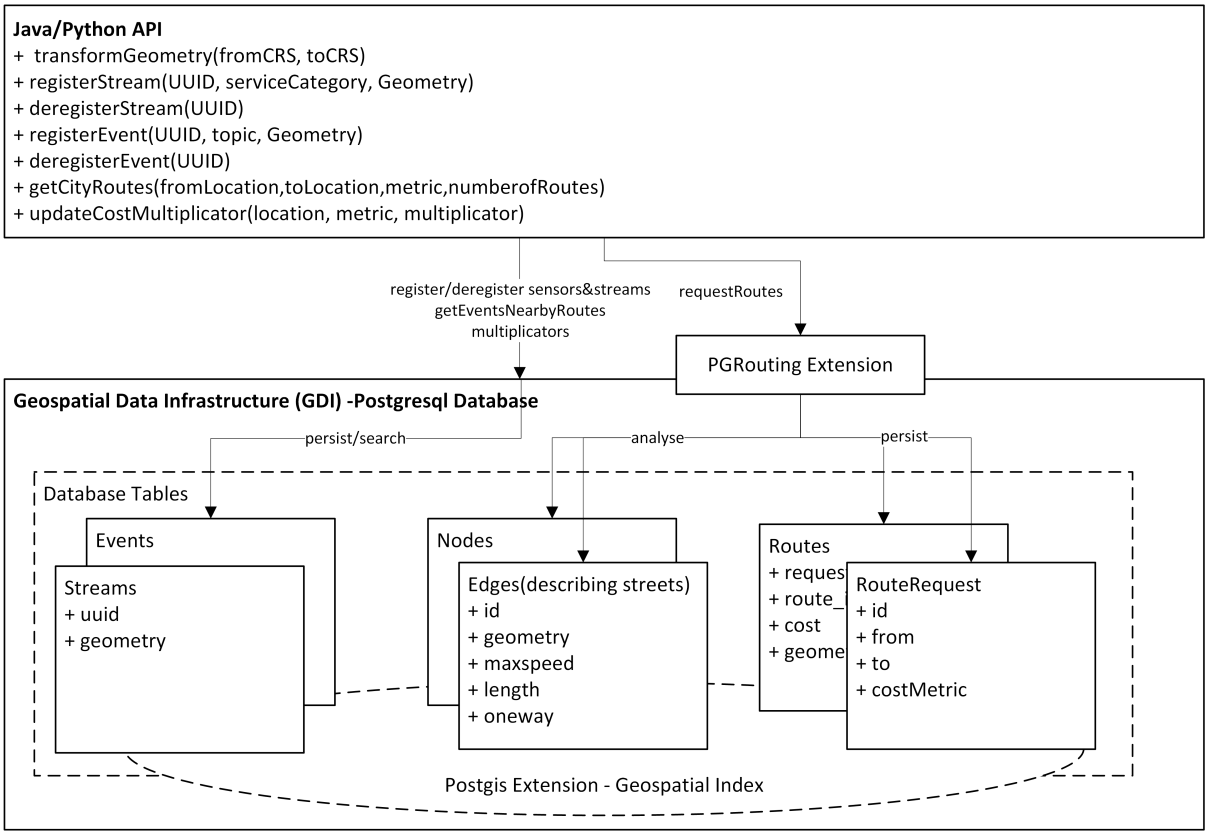

Figure 12 - Overview of the Geospatial Data Infrastructure

6.1.3 Clingo4

The core ASP solver used in the implementation of the Decision Support component of the CityPulse framework is Clingo4. Clingo4 [15] is an ASP solver that combines the grounder *gringo* with the solver clasp for computing stable models of ASP programs. The new clingo4 series provides new high-level constructs to deal with changes in the problem specification during the reasoning process (for example when data or rules are added, deleted or replaced). The grounding and solving steps in clingo4 are performed within a single integrated process to avoid redundancies in grounder and solver calls. On the declarative side, a flexible way to parameterise programs is also provided along with support for the ASP declarative input language and control capacities via embedded scripting languages Lua and Python. The separation of logic and control programs in clingo4 eliminates the need for special-purpose systems for incremental and reactive reasoning like iclingo [16] and oclingo [17].

All state-of-the-art ASP solvers follows, internally, a two-step approach better known as generate-and-test. First, a grounder generates a (finite) propositional representation of the input program; then a solver computes the stable models of the propositional program by exploring the search space to find solutions that are entailed by the program rules (test). This control loop is pre-defined and the search is guided by heuristics that allow little user control.

6.1.4 EmbASP

EmbASP[5] is a general framework for embedding ASP in complex systems, and in particular within applications for mobile devices. EmbASP is conceived in order to help developers at designing and implementing complex reasoning tasks in a straightforward way by means of ASP solvers. In the implementation of the core reasoning component used for decision support, the Clingo4 solver is embedded into EmbASP to facilitate the integration of external knowledge and the use of external modules as detailed later in this section.

6.1.5 Event Detection

The event detection represents the activity of leveraging the domain knowledge while processing in real time the flow of observations with the scope of identifying added value events. Event processing in real-time is referred to as complex event processing (CEP) that has originated from the work on an object-oriented language designed for system architectures prototyping with powerful event oriented semantics but has seen a sharp increase of interest and usage in the fields of BAM (*Business Activity Monitoring*) or BI (*Business Intelligence*).

The Event Detection component in CityPulse is implemented using the Esper CEP (Complex Event Processing) engine. This engine uses EPL (Event Processing Language) to implement event stream statements for event stream analysis. Event stream statements provide the windows, aggregation, joining and analysis functions and the CEP engine matches expected sequences of presence, absence of events, or combinations of events. ESPER includes also time-based correlation of events. The goal of the Event Detection component is to detect special situation on monitored area that are related to the city, such as traffic jam, parking status, pollution levels and more. These detected events are

[5] https://www.mat.unical.it/calimeri/projects/embasp/

used by the CF component as an input to identify relevant and critical events the DS (Decision Support) component should act upon.

6.2 User-centric Event Filtering Implementation

With respect to the implementation of the Contextual Filtering (CF) in Deliverable 5.1, we integrate this component with the Decision Support component via the user application. This makes our solution become a user-centric or user-driven system. Therefore, we recall here again the implementation of the CF component to highlight its interaction with the Decision Support component.

The CF component aims to provide critical events that are relevant for the user. This component filters both the list of events detected from the Event Detection component in Activity 3.4 and the contextual information of the user received from the application. The CF filters out unrelated events based on the City Event Ontology and the User Context Ontology. After that, it assigns a level of criticality to related events based on the information about the (dynamic) context of the. Figure 13 presents the sequence diagram to illustrate the interactions of the CF module with the different modules of the Decision Support Component.

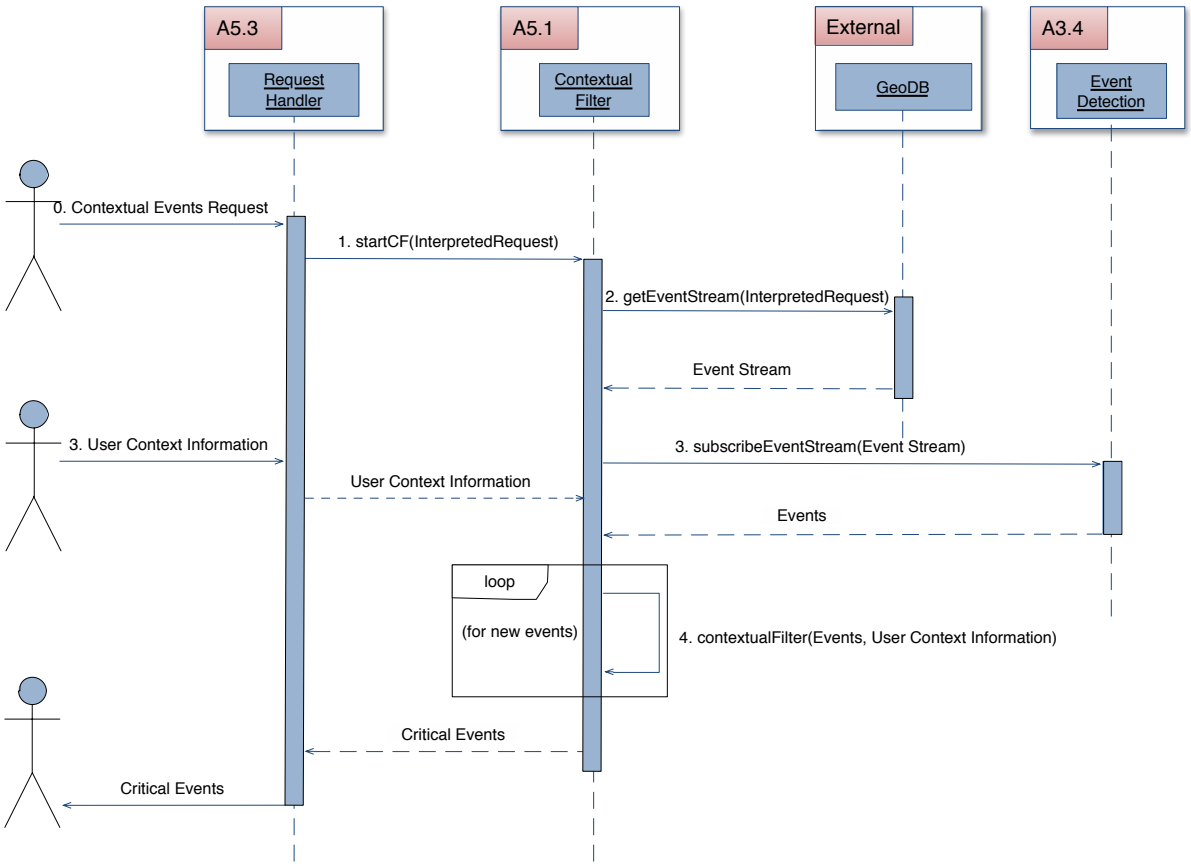

Figure 13 – Sequence Diagram for the Contextual Filtering component flow

A detailed step-by-step description is as follows:

- When a new ContextualEventsRequest is sent to the Request Handler to subscribe to critical events required by a user/application, an instance of the Contextual Filter is created.
- The instantiated ContextualFilter receives the InterpretedRequest, which is interpreted by the Request Handler with the ContextualEventsRequest as a parameter.
- The Contextual Filter sends a request to the GeoDB component to obtain the event streams from a certain area of the city or along a certain route, which is registered by the Event Detection component.
- After receiving the event stream from the GeoDB, the Contextual Filter starts subscribing to detected events from the Event Detection component. The Contextual Filter will receive detected events as a stream. At the same time, it also receives the contextual information of the user from the user/application as a stream.
- Whenever there is a new event detected by the Event Detection component, the Contextual Filter initiates the contextualFilter() function with the latest user's context information in order to filter and assign the most appropriate criticality (from 1 to 5) to the new event.
- If the new event is marked as critical, the user receives a notification and the option to decide whether to change the current solution and request a new one or not.

The ContextualEventRequest includes the ReasoningRequest and the answer or solution selected by the user among those provided by the Decision Support component.

6.3 User-Centric Decision Support Implementation

Figure 14 presents the sequence diagram detailing the interactions of the Decision Support modules with each other as well as with the external modules.

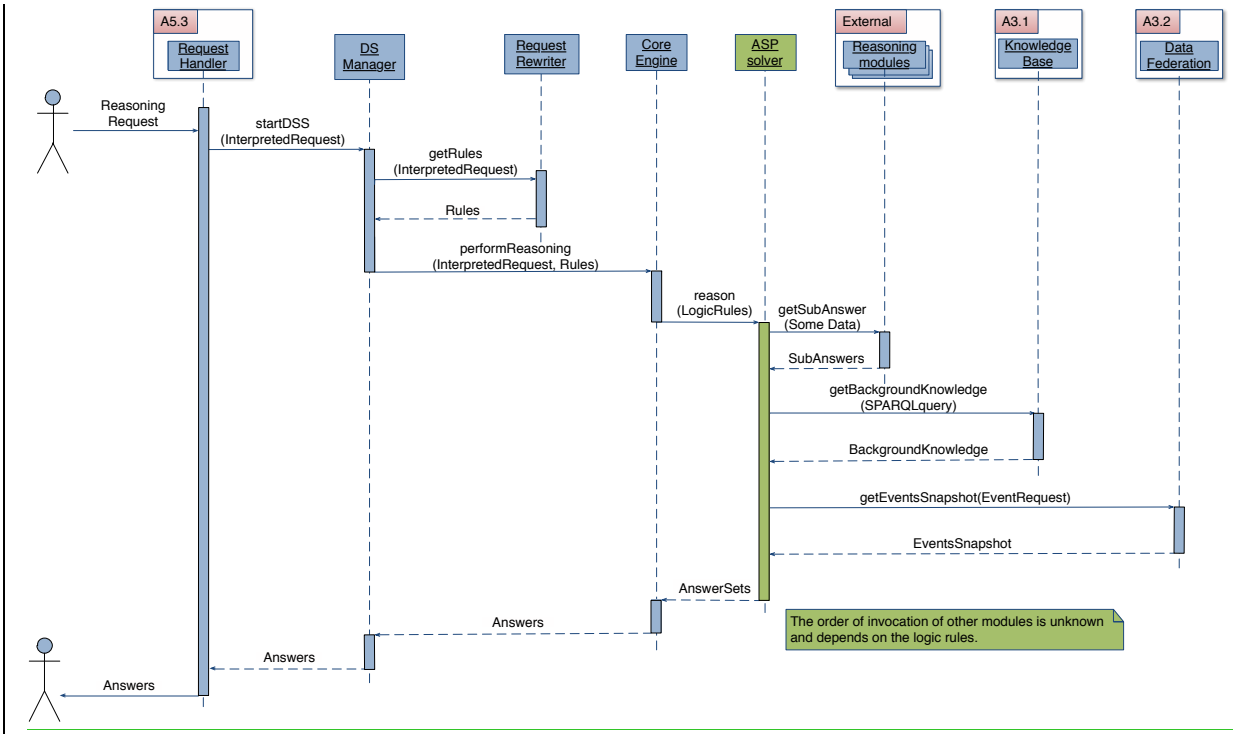

Figure 14 – Sequence Diagram for the DS component flow

A detailed step-by-step description is as follows:

- When a new *ReasoningRequest* is sent to the Request Handler to obtain candidate solutions required by the user/application, an instance of the **Decision Support System** is instantiated.
- The *ReasoningRequest*, which is sent to **Request Handler**, is interpreted as *InterpretedRequest* and used to start the **DS Manger**.
- **The DS Manager** sends this request to **Request Rewriter** to obtain the *logic rules*, which are automatically generated from the request. The method for automatic mapping is presented in Section 4.1.
- After receiving the rules from **Request Rewriter**, the **DS Manager** asks the **CoreEngine** to perform the reasoning by sending *InterpretedRequest* and *Rules* as parameters.
- The **CoreEngine** is a component that executes the ASP solver Clingo4 using the **EmbASP** framework.

  The ASP solver starts by collecting:
  - *Sub-Answers* from **External Modules**. *Sub-Answers* are partial answers that are performed externally for efficiency or privacy reasons. For example, in the Travel Planner scenario, the possible routes from starting point to ending point are computed externally.
  - *BackgroundKnowledge* from **Knowledge Base**. *BackgroundKnowledge* is static information stored in our triple stores and Data Bases. For example, in the Parking scenario, the locations of the parking spaces are part of background knowledge.
  - *EventsSnapshot* from **DataFederation**. *EventsSnapshot* are the latest values of events in the city, which help to produce answers based on the most updated

values. For example, in the Parking scenario, the availability of the parking spaces is recorded as event snapshot.

- The ASP Solver combines *Sub-Answers*, *BackgroundKnowedgle*, *EventSnapshot*, and Rules to compute the optimal answers (in form of *AnswerSets*) that are best matches to user constraints and preferences.
- These *AnswerSets* are processed by the **EmbASP** framework, and the Answers (as objects) are sent to the **CoreEngine** and then back to the user application for visualization.

7. Summary

In this deliverable we have documented the work carried out in the area of User-Centric Decision Support in Dynamic environments. This section presents our current outcomes, summarises the core novelties, describes the on-going work and outlines the next steps.

7.1 Outcomes

The user-centric DS functionality is achieved by normalizing user preferences and constraints, continuously capturing updates in form of relevant events, collecting partial solutions via sub-modules when necessary and producing ranked solutions presented to the user for decision making. We summarise the different outcomes in this process below:

- **Normalization of user preferences and constraints:** the component handling a request from a smart city application is in charge of manipulating user preferences and constraints as part of the request, and automatically represent them as declarative inference rules to be used by the core engine in computing the solution to the decision making task required by the application. The ability to seamlessly and automatically integrate these user-centric aspects within the decision support process is a key outcome.
- **Request partial solutions to sub-modules:** as part of the tasks performed by the CoreEngine component, the decision support capability of the CityPulse framework is able to request and embed partial solutions provided by external modules, in order to speed up and parallelize the complex computation of optimized decision support solutions. This outcome strongly rely in the interplay between the CoreEngine and the EmbASP framework described earlier in this document as part of related technologies
- **Continuous Monitoring Events update:** one of the input to the CoreEngine for embedding dynamicity handling within the Decision Support component are events updates. We ensure the strong interplay between the core reasoning engine and the Contextual Filtering component of the CityPulse framework has fast access to the most recent snapshot of events through the Data Federation component, which also makes sure the best information sources are considered.
- **Deduction process and decision tasks:** the actual deduction process for generating solutions to the decision support task required by the application is performed by combining background knowledge, event snapshots, user preferences and constraints and the DS module used within the application in a logic program represented by a set of rules. The fully declarative nature of the ASP framework used in the implementation of the DS capabilities makes it possible to

combine these rules and knowledge facts in a straightforward way, and enables full exploitation of the expressive power of ASP inference for constraint checking and preference-based deduction. This very same declarative feature is likely to simplify the extension of the DS modules provided within the CityPulse framework (namely routing, optimal selection among a set of alternatives and planning) and the creation of new modules.

7.2 Core Novelties

The core novelties of the CityPulse user-centric decision support strategy are summarized in this section.

In the DS process, the key novel capability we focused on for the CityPulse framework is the enhancement of existing semantic stream processing mechanism with more complex reasoning capabilities that go beyond the expressivity of stream query processing engines. Relating event streams to changing contexts requires dealing with incomplete and noisy input streams, conflicts, defaults, qualitative preferences, constraints and optimization problems in a declarative, knowledge-driven way. The use of Answer Set Programming to represent and reason about such knowledge is a clear advance in the expressivity of Web Stream Reasoning. The full integration of an ASP engine within the DS functionality in such a dynamic environment presents some challenges with respect to the scalability and expressivity trade-off. In order to make such computationally intensive reasoning possible, the DS component in CityPulse leverages the interplay between Continuous Query Processing and Event Filtering capabilities of the CityPulse framework with the Decision Support component in a pipeline that guarantees scalability by reducing the initial input to focus only on relevant information for the more complex reasoning tasks used in decision support.

7.3 On-going work

The scenario implementation has been mostly focused on using the DS Routing and the DS Optimized Selection among alternative modules. We are currently finalizing the planning module and we will exploit all three modules in a broader set of implemented scenarios, showing their flexibility to be adapted, and proposing guidelines for their combination.

We are also working on the specification of the functionalities exposed by the DS modules as APIs, which will be available as part of the outcome of Activity 5.3.

7.4 Next steps

The general idea underlying the design and development of algorithms for complex reasoning and decision support over web streams is to process dynamic data at different levels of abstraction and granularity, so that the amount of relevant information to be processed is filtered (and therefore reduced in size) as the expressivity of the reasoning (and therefore the computational complexity) increases.

Despite the potential of this approach applied in the Decision Support component of the CityPulse framework, the one-directional processing pipeline from query evaluation to non-monotonic reasoning presents some challenges and further exploration of the expressivity vs. scalability trade-

off is needed. Specifying parameters for this trade-off at design time does not achieve the best results due to the dynamic nature of web streams and their changing rate, quality and relevance. As a result of our preliminary analysis, we derived two main observations as a starting point for next steps:

1. We can find an optimal window size for a given streaming rate when the reasoning task as a fixed complexity, but as a next step we will work on characterizing in depth what are the effects of the complexity of the reasoning task on this trade-off, and how can this complexity be better characterized qualitatively and quantitatively.
2. Splitting the input in smaller chunks (even whit no parallelization of the reasoning process) provides better results in terms of processing times than processing all the input within the processing windows at the same time[6]. This split however is only possible if we assume input events are independent, which is not always the case. We are currently planning to investigate how can we characterize input dependencies and how can this assumption be relaxed to capture real-world use-cases and enable algorithms for input splitting to be designed for better scalability.

Advances and investigation around these aspects will lead to the design of heuristics for adaptation within the reasoning process in the CoreEngine, independently of the adaptive loop generated by the Contextual Filtering. Such loop helps in dealing with changes in the real world, but it does not act on the scalability of the reasoning algorithms itself, which is still highly dependent on the streaming rate, window size and complexity of the reasoning.

8. References

[1] A. Margara, J. Urbani, F. van Harmelen, and H. Bal, "Streaming the Web: Reasoning over Dynamic Data," *Web Semant. Sci. Serv. Agents World Wide Web*, vol. 25, pp. 24–44, 2014.

[2] E. Della Valle, S. Schlobach, M. Krötzsch, A. Bozzon, S. Ceri, and I. Horrocks, "Order matters! harnessing a world of orderings for reasoning over massive data," *Semant. Web*, vol. 4, no. 2, pp. 219–231, 2013.

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

[4] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus, "Querying rdf streams with c-sparql," *ACM SIGMOD Rec.*, vol. 39, no. 1, pp. 20–26, 2010.

[5] J.-P. Calbimonte, O. Corcho, and A. J. G. Gray, "Enabling ontology-based access to streaming data sources," in *The Semantic Web--ISWC 2010*, Springer, 2010, pp. 96–111.

[6] G. Antoniou, S. Batsakis, and I. Tachmazidis, "Large-Scale Reasoning with (Semantic) Data," in *Proceedings of the 4th International Conference on Web Intelligence, Mining and Semantics (WIMS14)*, 2014, p. 1.

[6] This is probably due to the grounding phase of the ASP solving process: the bigger the input, the higher the complexity of the combinatorial grounding of variables within the rules, leading to a much bigger size of the search space to be navigated in the solving step of ASP reasoning.

[7] H. Beck, M. Dao-Tran, T. Eiter, and M. Fink, "LARS: A logic-based framework for analyzing reasoning over streams," in *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[8] M. Gebser, T. Grote, R. Kaminski, P. Obermeier, O. Sabuncu, and T. Schaub, "Answer set programming for stream reasoning," *CoRR, abs/1301.1392*, 2013.

[9] T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits, *Dlv-hex: Dealing with semantic web under answer-set programming*. In: Proc. of ISWC, 2005.

[10] A. Mileo, A. Abdelrahman, S. Policarpio, and M. Hauswirth, "Streamrule: a nonmonotonic stream reasoning system for the semantic web," in *Web Reasoning and Rule Systems*, Springer, 2013, pp. 247–252.

[11] S. Germano, T.-L. Pham, and A. Mileo, "Web stream reasoning in practice: on the expressivity vs. scalability tradeoff," in *Web Reasoning and Rule Systems*, Springer, 2015, pp. 105–112.

[12] T. J. Misa and P. L. Frana, "An interview with edsger w. dijkstra," *Commun. ACM*, vol. 53, no. 8, pp. 41–47, 2010.

[13] A. V Goldberg and C. Harrelson, "Computing the shortest path: A search meets graph theory," in *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, 2005, pp. 156–165.

[14] S. Hwang, K. Kwon, S. K. Cha, and B. S. Lee, "Performance evaluation of main-memory R-tree variants," in *Advances in Spatial and Temporal Databases*, Springer, 2003, pp. 10–27.

[15] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, "Clingo= ASP+ control: Preliminary report," *arXiv Prepr. arXiv1405.3694*, 2014.

[16] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele, "Engineering an incremental ASP solver," in *Logic Programming*, Springer, 2008, pp. 190–205.

[17] M. Gebser, T. Grote, R. Kaminski, and T. Schaub, "Reactive answer set programming," in *Logic Programming and Nonmonotonic Reasoning*, Springer, 2011, pp. 54–66.