GRANT AGREEMENT No 609035
FP7-SMARTCITIES-2013

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

Collaborative Project

Testing and Fault Recovery for Reliable Information Processing

Document Ref.	D4.2
Document Type	Report
Workpackage	WP4
Lead Contractor	UASO
Author(s)	D. Kuemper, T. Iggena, D. Puiu, M. Fischer, F. Gao, R. Toenjes
Contributing Partners	UASO, SIE, NUIG
Planned Delivery Date	M30
Actual Delivery Date	M30 (2016-02-29)
Dissemination Level	Public
Status	Final
Version	V1.0
Reviewed by	João Fernandes, Muhammad Intizar Ali

Executive Summary

To gain higher user acceptance and market penetration the reliability of smart city frameworks is an eminent characteristic, whereby the availability of smart city applications highly depends on the availability of appropriate, accurate, and trustworthy data. The CityPulse framework is organised in three consecutive iteratively applied processing layers, covering federation of heterogeneous data streams, large-scale IoT stream processing, and real-time information processing and knowledge extraction. To achieve reliability, CityPulse integrates knowledge-based methods with monitoring and testing at all stages of the data stream processing and interpretation. During design-time this requires testing of framework components for their behaviour regarding different quality levels of information. During run-time the approach includes monitoring of the reliability of data streams, monitoring of the information plausibility, and evaluation of extracted events on the information side. To process the quality indications, which are made during run-time, a fault recovery and a conflict resolution mechanism ensure fault-tolerant behaviour.

The work presented in this deliverable shows that, depending on the availability of individual data sources, various algorithms and processes can be used to evaluate city infrastructure data and infer decisions on data source usage. Therefore, spatio-temporal relations between events and continuous measurements are used to determine plausibility of the evaluated datasets. CityPulse ensures reliable information processing through four measures, namely Testing, Monitoring (Atomic and Composite), Fault Recovery, and Conflict Resolution, which are iteratively described in this deliverable. Furthermore, tools to visualise and evaluate the performance of these measures are presented.

Contents

Figures

Tables

Listings

Abbreviations

ACEIS	Automated Complex Event Implementation System
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
CDF	Cumulative Distribution Function
CP	CityPulse
CSV	Comma-Separated Values
DCT	Discrete Cosine Transformation
GUI	Graphical User Interface
HTTP	Hyper Text Transfer Protocol
IDCT	Inverse Discrete Cosine Transform
IoT	Internet of Things
JSON	JavaScript Object Notation
k-NN	k-Nearest Neighbour
ML	Machine Learning
N3	Notation 3

QoI	Quality of Information
RM	Resource Management
SPARQL	SPARQL Protocol And RDF Query Language
SUT	System Under Test
WSN	Wireless Sensor Network

1. Introduction

The reliability of smart city applications is an eminent property to achieve growing acceptance by citizens. This results in a high demand of monitoring and testing procedures to implement high-availability services and also a demand for compensatory mechanisms in case of failures. Testing in the research area of smart cities, involving the Internet of Things (IoT), utilises various testing methods like test first [Andrea 2007], model based testing or fuzz testing [Godefroid 2008] to achieve a reliable availability of sensor data and applications with a suitable testing effort. Especially the effort for creating the tests with the focus on dynamic city applications that rapidly change due to complex requirements in new smart city scenarios is a challenging task. Therefore, an adapted test data generation [Fischer 2012] is used to reach high efficiency. Nowadays, during the development of any smart city application, the testing is done in the following three steps: 1) describing the sensor data flow of the application corresponding to use cases; 2) defining test data with regards to the described data flows and 3) interpreting the results after tests execution. This deliverable introduces the test methodology applied in the CityPulse project. The methodology utilises features of the CityPulse framework, that allow for an iterative test process to determine the minimum information quality threshold for a reliable application execution.

In contrast to testing applications before their deployment, the monitoring of various smart city data streams during run-time requires the integration into a running system. Determining the sensors' accuracy in relation to its spatial-temporal location is a continuous task needed along the applications' life-cycle [Charef 2000]. Monitoring various data streams can be used as input to identify correlations between data sources. While testing and monitoring applications an observer compares the output of the system under test (SUT) against expected behaviour and values. The main difference to testing is that the observer does not actively stimulate the SUT in order to trigger a specific feature. Instead the observer has to wait until the feature is triggered by a real user, an external sensor, or an application. For this reason, monitoring is also named passive testing [Merayo 2012]. CityPulse utilises a two-staged monitoring approach, to evaluate the quality of smart city data streams during run-time. Incoming observations are annotated with quality metrics. These Metrics describe the quality level of a data stream and allow consumers (e.g. applications) to evaluate if this stream satisfies the requirements or if the consumer needs to switch to alternative data streams. Furthermore, the monitoring can be useful to determine the cause of an error. This deliverable defines the monitoring process in each of the two stages and explains the annotation process and how the quality metrics are interpreted. The monitoring component can only be used to find failures and errors in data sources. To solve identified issues additional processing has to be done. Therefore, the CityPulse framework offers additional solutions to deal with faulty data streams. To replace wrong or missing values online learning algorithms, enabling estimations for those values, are used. In case of non-replaceable values, failing data streams, or contradictory information sources, CityPulse offers a solution to automatically switch between available data streams depending on the requirements of a smart city application.

The remainder of this deliverable is structured as follows: Section 2 presents the state of the art and derives requirements for the reliable information processing components. Section 3 presents an integrated monitoring and testing approach to support scalable evaluation of data sources and

smart city applications. Section 4 describes response mechanisms for short and long term fall-back strategies in case of failing data sources. In Section 5 the implementation characteristics of the previously described components are presented. Section 6 gives a first outlook on initial experiments in the running framework. A conclusion and outlook are discussed in Section 7.

2. State of the Art and Requirements for Smart City Frameworks

2.1 Monitoring and Testing

In the following we discuss test case stimuli generation that is based on error sources for smart city external resources. The error-free execution of smart city services and applications relies on reliable and robust data sources. However, it would be irrational to assume that a large-scale network of data sources can operate error-free in long term. Especially when a large number of inexpensive sensors are deployed, faulty sensor readings caused by incorrect hardware design, improper calibration, or low battery levels have been observed [Ramanathan 2006][Tolle 2005][Werner-Allen 2006]. CityPulse acknowledges this fact by providing test-methodologies, which allow an application developer to estimate the robustness of his/her application under varying reliability of the used data sources. The goal is to use QoI metrics, evaluated through the quality monitoring, and be able to provide a statement about reliable operational range for the application. The stimuli in the test cases for the test-methodologies are a series of sensor readings with artificially induced errors of increasing magnitude. In order to avoid a test case explosion as well as concentrate on realistic errors this chapter will discuss different types of error sources for external resources and their relevance in a smart city context.

The source of an error depends on the type of the resource. For resources measuring physical phenomena in the real world an error shall be defined as the difference of the actual value and the measured value. The difference is always unequal to zero due to bias uncertainties in the measurement process and performance characteristics of the measurement equipment. [NHB2 2010] distinguishes the following three different types of characteristics (static, dynamic, other), depending on the state of the input signal.

Static characteristics provide an indicator for a steady-state input signal at one particular time. Typical static characteristics are the Zero-Offset, Noise and the Sensitivity. Zero-Offset describes a non-zero output of a measurement equipment for a zero input. A Zero-Offset is assumed to be constant for all input levels and thus provides a fixed error that can easily be reduced during calibration. A complete elimination of the Zero-Offset is not possible, because there is no way of knowing the true value of the offset. Sensitivity is defined as the ratio between an output signal to the corresponding input signal. The static characteristic describes how small variations in the input signal can be detected in the output signal. Noise describes a random offset between measurements of a constant input signal. Figure 1 depicts the principal of the static errors.

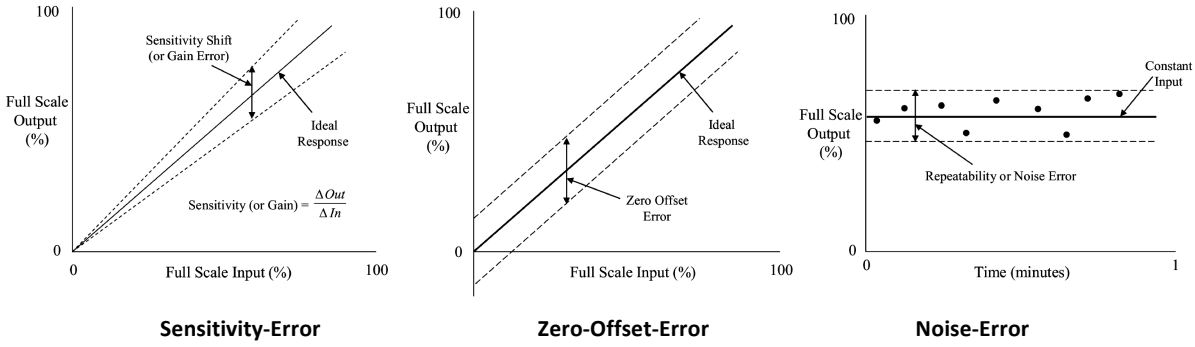

Figure 1: Schematic Representation of Static Error Types [NHB2 2010]

Dynamic characteristics describe the responses of measurement equipment for changing input signals over time. They include Response Time, and Damping Ratio (Figure 2). Response Time states the time the measurement equipment needs to adjust to a variation in the input signal. The time constant τ is the specific time required for the output signal to reach 63.2% of its final value. Damping Ratio is a measure of how energy from a rapid change in an input signal is dissipated with the measurement equipment. An under-damped device will overshoot the final output value, while an over-damped device requires more time to reach the final value.

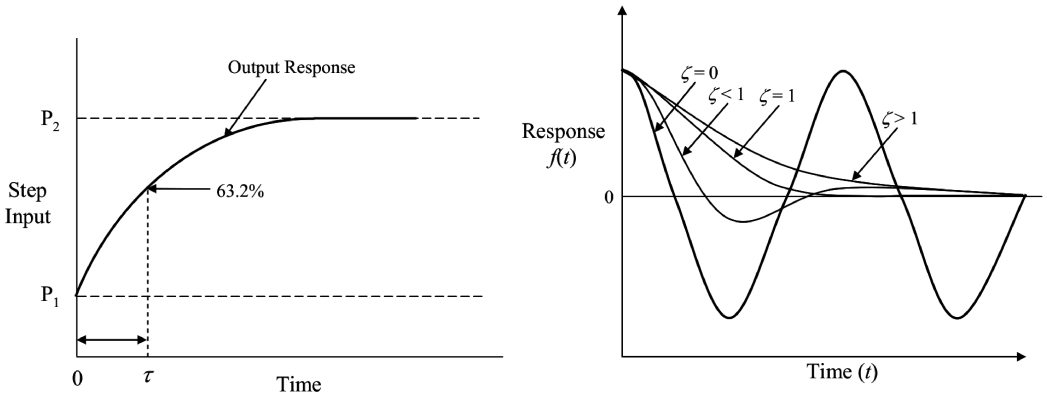

Figure 2: Schematic Representation of the Dynamic Error Types [NHB2 2010]

The third group of characteristics in [NHB2 2010] is named as '*other*' and describes those physical aspects of a measurement process, which neither fit in static nor dynamic characteristics. The *'other'* group includes environmental operation conditions, such as operating temperature, humidity, and pressure or Span, Deadband, and Threshold. Deadband is the lowest measurable input value, Threshold is the highest measurable value, and Span describes the difference between both (see Figure 3).

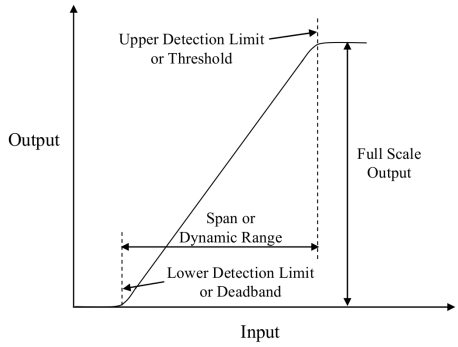

Figure 3: Schematic Representation of Deadband, Threshold and Span [NHB2 2010]

Above-mentioned errors are just an excerpt of errors described in [NHB2 2010], but show that although many causes for an error exist, the effects on the output signal are quite limited. Static and other characteristics provoke an offset in the value of the output signal while dynamic characteristics provoke a delay in the delivery of the output signal. The only difference is the magnitude and the occurrence probability. Therefore, the generation of the stimuli for the test cases can be reduced to the specification of the magnitude of the offset and the delay as well as a function for the probability of occurrence in an iterative algorithm.

In [NHB2 2010], authors' intention is to describe sources of errors in measurement systems to guide metrology experts in the selection of measurement and test equipment and help device manufactures in describing the limitations of their products. [Sharma 2010] on the other hand categorised sensor faults for automated fault detection in wireless sensor networks (WSN) by defining three fault models, namely CONSTANT, SHORT and NOISE. The study is driven by real world sensor readings and focuses on the patterns of the output signals. The investigations have the viewpoint of an output signal consumer and acknowledge the absence of a ground truth, thus an error is only a deviation from an expected output signal. They include outliers, sudden spikes in the output signal (SHORT); the 'stuck-at' phenomena, where the output signal does not change for a large number of successive readings (CONSTANT); and random variances of successive sensor readings (NOISE). With the exception of NOISE errors, these errors are unexpected by measurement equipment manufacturers, unlike those described in [NHB2 2010]. The authors stated, that due to the absence of a ground truth, a large number of sensor readings can be subjected to errors and still exhibit normal patterns, making it impossible to detect the error without a proper model of the expected sensor behaviour. Similarly, to the works of [NHB2 2010], the effects to the output signal are limited to an offset between true value and measured value. In contrast to [NHB2 2010], the amount of time the error affects the output signal is of interest, rather than a delay in the delivery of the output signal. Jaeger et al. [Jaeger 2014] propose the use of a neural network to provide a general solution covering typical sensor faults. They also present 3 error models named "Delay", "Offset" and "Stuck-at", which can be mapped to SHORT, NOISE and CONSTANT categories by [Sharma 2010]. In addition, they introduce the "omission" error, where individual sensor readings are delayed indefinitely.

The second type of external resources measures quantities indirectly by the aggregation or interpretation of other sensor readings. The physical phenomena used to deduce a quantity may vary between different cities. For example in the CityPulse project both, the city of Aarhus and the city of Brasov, provide data streams for the number of vehicles on a road. However, the city of Brasov uses footage from CCTV cameras to count the number of vehicles, while the City of Aarhus captures signals from mobile phones. A direct access to the underlying sensor network and its sensor readings is not available. Error sources, as in the first type, may be compensated or masqueraded in the aggregation/interpretation process, which poses itself an error source caused by software implementation faults. In addition, the frequency of those signals may be much smaller than those of the underlying sensors, requiring them to malfunction for a longer period of time to influence the output signal significantly. Also, events may be left out if occurred too shortly for the aggregation algorithm. As an example, a traffic jam may occur only as an increase in traffic if dissolved too quickly. Identifying faulty data relies on probabilistic methods, time series analysis or comparison with spatial-temporal correlated streams. Like for the first type (errors in measurement equipment) we are more interested in the effects on the output signal (the output signal here refers to the output of the aggregation method). As in the first type we can identify a delay and an offset effect.

In summary, the effects of the different kinds of errors are very limited for external resources measuring physical phenomena in the real world. We showed, that the effects of errors can be described by three dimensions: delay, offset and duration. To use these findings in the generation of input stimuli for the test cases, we derive an error-meta-model, which will be discussed in Section 3.2.2.

The third type of external resources in a smart city context encompasses data provided by human interaction. These resources can be dynamic in nature, such as social media streams i.e., Twitter, or rather static such as timetables for bus schedules. While the latter is produced in complex processes over a quite long period of time involving multiple humans, the former is produced spontaneously and often reflects the opinion of an individual. However, contradictory information between both may indicate the presence of an unexpected event.

The assessment of errors for social media data streams is quite complex due to two reasons. Firstly, the data is provided in natural language text and lacks any numerical values. In the bus example, often, a vague statement like "The bus is late" is provided, without a mention of bus number or station. Secondly, a mapping between the difference of expected and actual arrival time and the word "late" needs to be done. An in-depth analysis of the use of social media in a smart city framework is done in Deliverable 3.4 [CityPulse-D3.4].

Monitoring Requirements

To achieve reliability of smart city applications, CityPulse integrates systematic monitoring within its framework. To achieve a quick reaction in case of contradictory information or failing sensor streams the evaluation process (Atomic Monitoring) has to be integrated close to the virtualisation of incoming observations. An early calculation of estimated values in case of sensor failures allows the framework to remain operational. Furthermore, network-related quality metrics need to be monitored on a low level, as they cannot be evaluated later on. The results have to be transparent to

assist developers to determine an optimal balance between the most efficient and the most reliable and dependable solution. During run-time, data streams are monitored to detect violations against required QoI for an application and/or data inconsistencies. This information can then be utilised to identify and isolate unusual data patterns within the streams or adapt the data acquisition (sensors used, sampling rate etc.) to meet the QoI requirements of an application. The monitoring must employ iterative algorithms in order to scale linear with the number of deployed sensors within a smart city and keep its real-time properties.

2.2 Quality Annotation

In Deliverable 4.1 [CityPulse-D4.1] we proposed the CityPulse QoI ontology as a data model to store application independent quality information for data streams. This ontology has been documented and published for public use at [Quality Ontology]. Further explanations regarding state-of-the-art for QoI, the definitions for QoI metrics and the design of the ontology can be found at [Kolozali 2016].

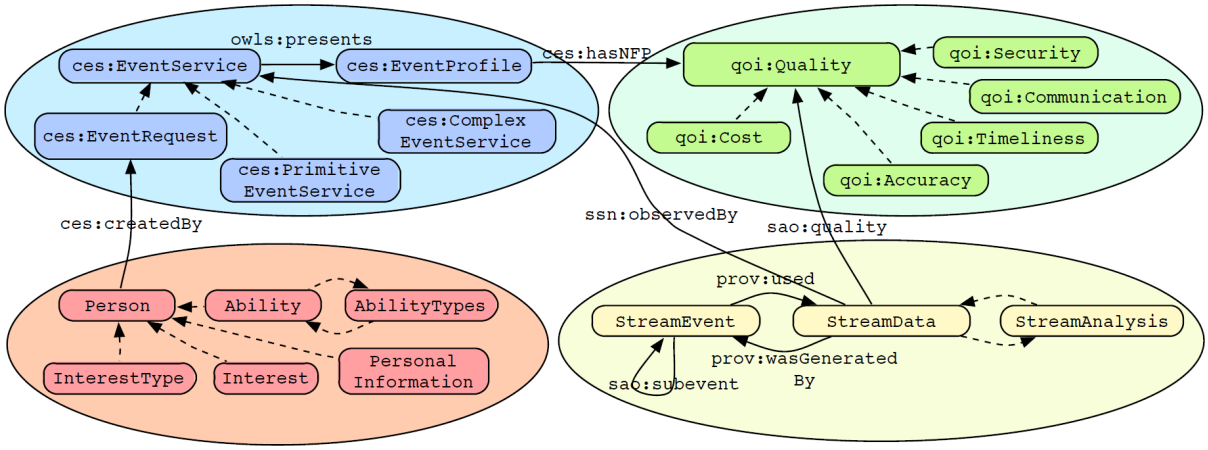

Figure 4: CityPulse Ontology Model [Kolozali 2016]

As a short introduction to the ontology, Figure 4 depicts a simple view of the CityPulse information model. The QoI Ontology (highlighted in green colour) is reduced to its main categories Accuracy, Communication, Cost, Security, and Timeliness. For the implementation of the ontologies into the CityPulse framework a tool, called SAOPY, to create Python programming language modules from ontology files is used (see [SAOPY]). This enables the CityPulse framework developers to directly use the ontologies within a Python environment.

2.3 Conflict Resolution

The task of the Conflict Resolution is to find more reliable or trustworthy streams in cases when two or more functional equivalent streams are reporting different values. To achieve this goal, the Conflict Resolution component has the following requirements:

- The component must be able to obtain the quality of the streams efficiently, especially the accuracy of the streams.
- The component must be generic enough to allow usage of different algorithms to determine, which streams are more reliable.

The first requirement can be addressed by invoking the QoI aggregation and estimation methods provided by the Data Federation component, as described in D3.2 [CityPulse-D3.2]. The second requirement is addressed in the implementation by a modular software design.

2.4 Fault Recovery

The Deliverable D4.1 [CityPulse-D4.1] presents the State of the Art in the related domain of fault recovery. The description is concentrated mainly around the standard machine models (more precisely the model is generated using datasets and it is not updated if the distribution of data is changing with the time). Since, we have updated the requirements of the Fault Recovery component (see Section 2.4) and modified the implementation to operate similar to an online learning algorithm, we briefly present below the State of the Art for this particular domain.

The performance of Machine Learning (ML) model relies heavily on the availability of a representative learning dataset. The creation of such a representative dataset is in most of the cases expensive and time consuming. As a result of that, such datasets usually become available in small batches at different times. As a consequence, it is necessary to incrementally update the existing ML model to accommodate new data while retaining the information obtained from old data [Polikar 2001].

Several algorithms with various interpretations of the phrase *incremental learning* have been proposed in the domain literature. In some cases, the incremental learning principles have been used to select the most informative training samples [Engelbrecht 2001] or a version as incremental construction of support vector machines [Shilton 2001]. Grippo suggested a form of controlled modification of classifier weights (typically by retraining with misclassified instances) [Grippo 2000].

In [Andonie 2003] Andonie et al. propose an incremental learning system for general classification and nonparametric estimation of the probability that an input belongs to a given class. The architecture of the network is able to incrementally grow and to sequentially accommodate input-output sample pairs. Each training pair has a relevance factor assigned to it. This factor is proportional to the importance of the respective pair in the learning process. Using a relevance factor adds more flexibility to the training phase, allowing ranking of sample pairs according to the confidence in the information source. The training sequence may include sample pairs from sources with different levels of noise.

The main objective of the Fault Recovery component is to increase the stability of the system by providing estimated measurements when the quality of a stream is modified (in most cases when the quality drops).

The requirements for Fault Recovery component have been presented in the Deliverable D4.1 [CityPulse-D4.1]. During the development of the Fault Recovery component we found the possibility to use a generic algorithm, which can work for most of the numeric data sources. In the consequence the following two requirements (specified in [CityPulse-D4.1]) are outdated:

- The component has to be generic and the domain expert is the one who provides the algorithm for emulating different data streams.

- The algorithms for emulating different stream data have to be stored in the knowledge database and the component must be able to identify/execute the model based on the fault recovery requests.

They are replaced by the following requirements:

- The component is integrated into the Data Wrapper.
- When the Data Wrapper is initialised it should accumulate historic data (when the quality of the stream is high), which will be used later on to generate the estimation (when the quality of the stream is low).
- The domain expert should be able to turn on/off the fault recovery functionality.

3. Monitoring and Testing

This section introduces the concepts of the monitoring and the testing developed in the CityPulse project. Monitoring deals with the evaluation of information quality during runtime. To meet the real time requirements in a smart city, the monitoring is divided into two separate stages. Figure 5 depicts the interaction of the coloured components in the Architecture. The first component, named Atomic Monitoring (blue square), observes incoming observations on isolated data streams and performs rudimentary but high performance, real-time sanity checks. The second component, named Composite Monitoring (yellow square) validates detected events by investigating correlations between spatial-correlated streams. Since the latter case is computationally more complex as compared to the former, the process is only triggered by new events and does not satisfy real-time requirements. In contrast to monitoring, where the system only observes the behaviour of external resources, in Testing (red square) we describe measures to generate artificial stimuli, with the intent to identify the minimum information quality to successfully execute smart city applications.

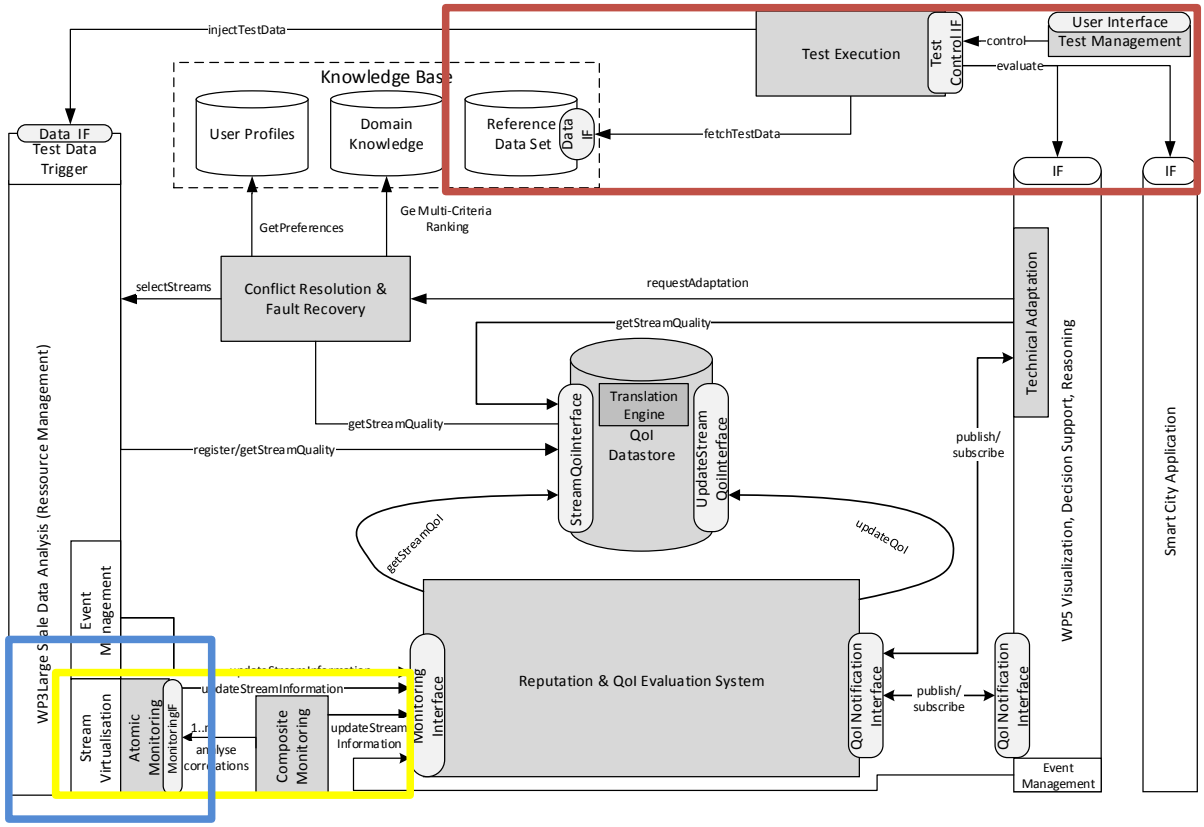

Figure 5: Monitoring and Testing within Work Package Architecture

3.1 Monitoring Concept

The availability of smart city applications highly depends on the availability of appropriate, accurate, and trustworthy data. This includes the availability of necessary data sources as well as accessing their QoI descriptions. The reliability of the extracted sensor information has to be monitored during run-time. Monitoring methods are used to compare the information quality of data streams with the QoI-requirements of an application. To get a continuous view of sensor stream qualities two

different monitoring components have been designed. The Atomic Monitoring component focuses on sensor information of an individual sensor stream whereas the Composite Monitoring component analyses inter-dependencies between various sensor streams to evaluate the plausibility of information.

The separation into two components ensures a constant real-time quality annotation for basic quality parameters and a more complex proof of correctness to rate the trustworthiness of data sources, as it probably could not to be done in real-time.

3.1.1 Quality Ontology Extension/Changes

The Quality Ontology introduced in Section 2.2 has been slightly adapted to be used with the Atomic Monitoring. The first version of the ontology only allowed having one quality value for each metric. However, the planned QoI metrics are able to calculate an absolute and a normalised quality value. The Quality Ontology (see Figure 6) has been extended to gain the ability to store both values.

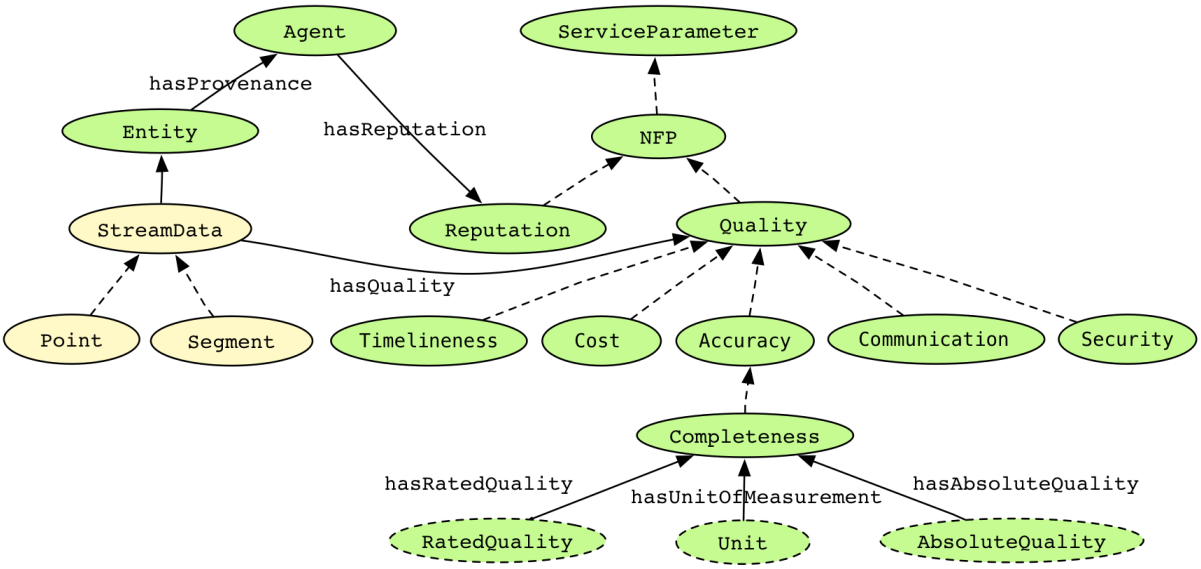

Figure 6: Extended QoI Ontology

Figure 6 shows the extension of the metric Completeness, which is a subcategory of Accuracy. Through the new data type properties, *hasAbsoluteQuality* and *hasRatedQuality*, it is now possible to store the data calculated by the Atomic Monitoring (see Section 3.1.2). The absolute quality is intended to hold the measured values, i.e. 1s for Latency or 0.33Hz for the Frequency. In addition, the unit for the absolute quality is given via the *hasUnitOfMeasurement* property. In the case of Latency it could refer to http://purl.oclc.org/NET/muo/ucum/unit/time/second. As every QoI metric calculates an additional normalised value with the help of the Reward and Punishment algorithm (see [CityPulse-D4.1 chapter 5.3.2]) this value is stored with the *hasRatedValue* property.

3.1.2 Atomic Monitoring

The Atomic Monitoring is responsible for the real-time quality annotation of newly fetched sensor observations. To fulfil the real-time requirements, it includes only basic QoI checks based on a description for each data stream. A short introduction of the functionality was first given in

[CityPulse-D4.1 chapter 5.3]. The atomic QoI value computation will be described in detail in the following paragraphs to help application developers and stream providers to consume and provide individual data streams.

Every sensor/data stream has to provide a sensor description, containing basic parameters of the sensor and its provided data fields. In Listing 1 a sensor description is depicted with all possible options supported by the Atomic Monitoring.

```
1  sensordescription.maxLatency = 2
2  sensordescription.updateInterval = 60
3  sensordescription.fields = ["v1", "v2", "v3"]
4  sensordescription.field.v1.optional = True
5  sensordescription.field.v1.dataType = "int"
6  sensordescription.field.v1.min = 0
7  sensordescription.field.v1.max = "@v3"
8  sensordescription.field.v2.dataType = "datetime "
9  sensordescription.field.v2.format = "%Y-%m-%dT%H:%M:%S"
10 sensordescription.field.v3.dataType = "int"
11 sensordescription.field.v3.min = 0
12 sensordescription.field.v3.max = 100
13 sensordescription.field.v3.optional = False
```

Listing 1: Sensor Description Example

The first two lines of the listing are describing update parameters of the whole stream. The maximum latency indicates the maximum amount of time in seconds, which is allowed for this stream. The second line describes the interval the stream is updated in seconds, meaning the stream delivers an update every 60 seconds in the example above. The *fields* parameter in the third line indicates how many individual data fields the stream provides by enumerating their names. In this case there are the fields *v1*, *v2*, and *v3*, each with an individual description.

Besides the options affecting the whole stream, there are attributes only related to single data field. These are described within the lines 4 to 13. Field *v1* is stated as an *optional* parameter in line 4. If a field is marked as optional an observation of a data stream might omit this field without affecting the QoI metric Completeness. This feature is helpful for information, which could not be delivered within the data stream at any time. The *dataType* parameter specifies the data format that a field should have. In the case of *v1* all values of the type integer are accepted. Line 6 and 7 define the range of *v1* with a maximum and a minimum value. While the minimum is described as a simple value, the maximum value is dynamic and depends on another field in the observation. The given *@v3* indicates that the maximum is determined by the value of *v3*. This allows dynamic value ranges during the runtime of a data stream without the need of modifying the sensor description. In addition to the simple integer *dataType v2* is stated to have the *dataType datetime*, which indicates that the value of *v2* is a date. The format of the date is specified by another attribute named format. The last field *v3* is an integer value in the range of 0 to 100. The optional parameter set to false indicates that this value for this field has to be present in every observation of the data stream. A real-world example for an existing data stream is provided in Section 6.

The sensor description is the anchor point for the QoI calculation of the Atomic Monitoring. Currently, Atomic Monitoring calculates the following metrics:

Age

The QoI metric Age ensures, that an observation was made within a certain time frame before being delivered to the CityPulse framework. In technical terms it calculates the difference of the current time to a timestamp delivered within an observation. If this difference is too large (the observation is too old) the QoI metric Age is lowered. By default, the *updateInterval* provided in the sensor description is also considered as maximum age.

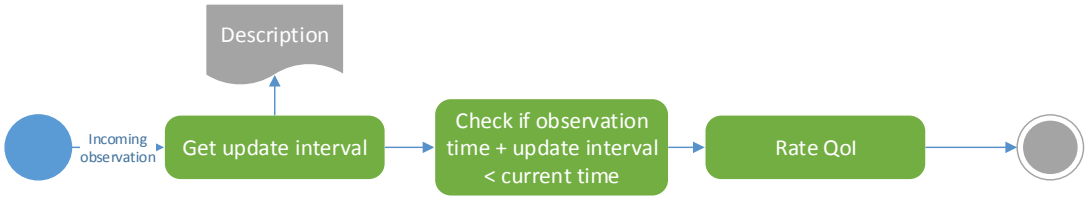

Figure 7: Determination of QoI Metric Age

Completeness

The Completeness of observations within a data stream is calculated by a comparison between the received data and the annotation of the stream in form of the sensor description. For the aforementioned example in Listing 1 the received data will be checked for containing the fields *v1*, *v2*, and *v3*. As pointed out, the QoI of the data stream will not be decreased if *v1* is missing as this field was annotated as being optional. An additional check ensures that the data fields contain values different from empty strings, *Null* values or *NA*. The value of the Completeness is indicating the number of fields an observation contains.

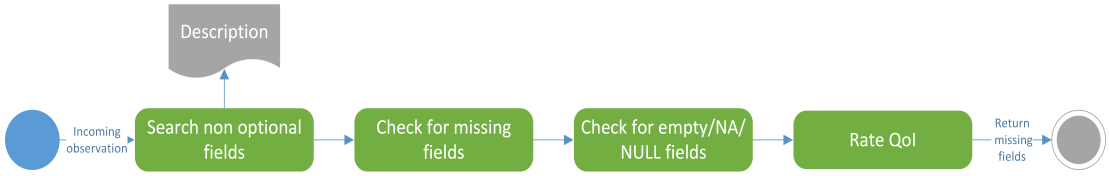

Figure 8: Determination of QoI Metric Completeness

Correctness

This QoI metric checks the delivered values within the observation to the dataType, *min*, and *max* parameters. The QoI is lowered if one of these values differs from the description.

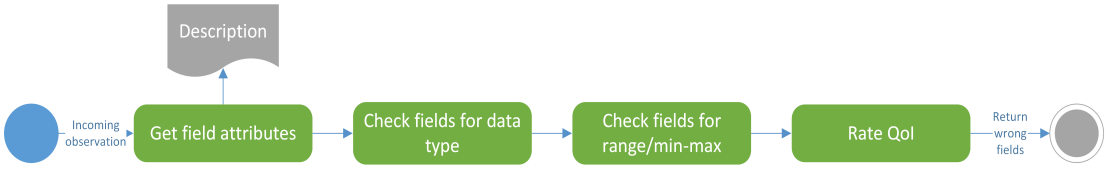

Figure 9: Determination of QoI Metric Correctness

Frequency

The parameter *updateInterval* from the sensor description indicates how often the framework should expect an update from the data stream in form of an observation. If there is no update or the update is received later than expected this QoI metric will be lowered.

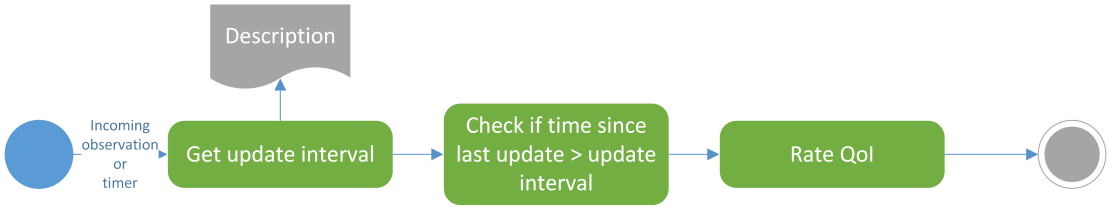

Figure 10: Determination of QoI Metric Frequency

Latency

For observations pulled by the CityPulse framework, the network latency is measured, to determine the amount of time required to transfer an observation. To rate the Latency QoI metric, the value is compared with the *maxLatency* stated in the sensor description. For pushed observations this QoI metric is left out, as it would require synchronising the clocks of the transmitting system and the CityPulse framework.

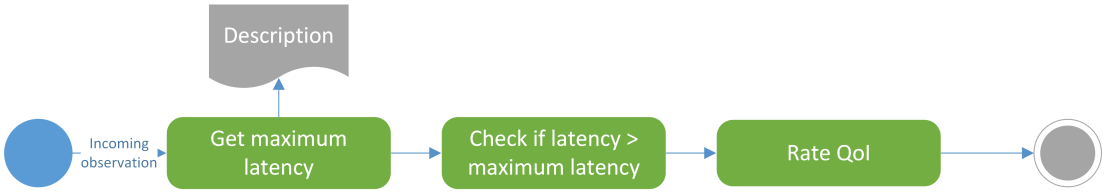

Figure 11: Determination of QoI Metric Latency

3.1.3 Composite Monitoring

The main challenge in evaluating the correctness and information quality of heterogeneous data sources in smart city environments is a missing ground truth for comparing results. If there is no exactly planned infrastructure it is a complex process to determine, which sensor measurements are correct in case of contradictory information. CityPulse aims at predicting the plausibility of events by pursuing a monitoring approach that analyses sensor values of related sensors of different kinds. This is realised in addition to the Atomic Monitoring and called Composite Monitoring. This approach helps to determine if outliers in sensor readings are due to defective sensors or can be explained by similar information from related sensors.

For example a traffic jam can be evaluated as follows:

- Detected by:
  - Traffic sensor shows extensively slower traffic speed
  - Amount of passing cars is lowered extensively
- Possible Evaluation:
  - Analysis of consecutive traffic sensors on a road

Contrary to the Atomic Monitoring the Composite Monitoring does not only use the current value of one data stream. It utilises historic time series of various dependent sensor streams. Therefore, a dedicated live-evaluation for every sensor measurement is not possible. It is used in case of significant sensor events or for manual maintenance. Thus, the Composite Monitoring is triggered by events from the Event Detection Component, by the Atomic Monitoring (in case a QoI metric drops significantly) or for a manual event or stream evaluation by Quality Explorer. To evaluate a reported event, in the first step, sources that were used to create the Stream Event (see Figure 4) are identified. Based on the category (e.g. Temperature, Parking, Traffic) of these sources further streams/sensors that are located nearby can be identified. It is fair to assume, that they may also be affected by the event and thus should show a similar behaviour. An impact model that defines various data streams and relations between their values is used to identify relevant sensors. It also describes a propagation model of the affected phenomena. For example, traffic propagates along the roads whereas noise propagates in every direction.

Figure 12 describes the evaluation process of the Composite Monitoring. The goal is to determine correctness value (C_e) for event (e). A set of correlating data streams (S_e) is used as validation source. The Figure shows the 4 phases of the validation process:

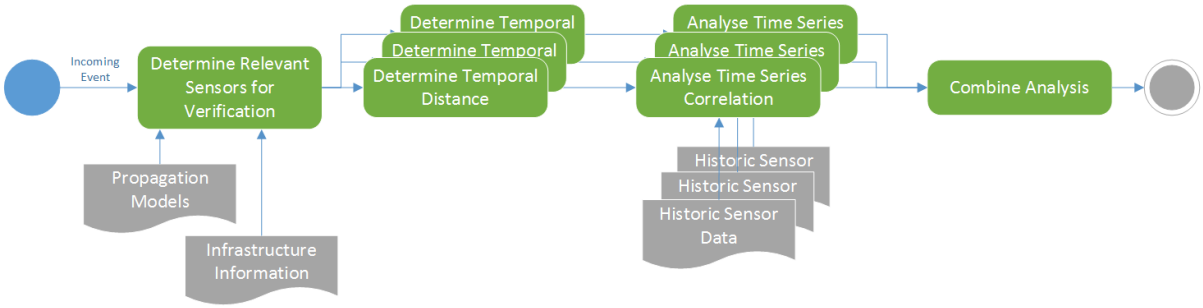

Figure 12: Activity Diagram of the Composite Monitoring Process

1. Determine relevant sensors out of the set of all Streams (s)
   a. Find spatially correlated streams by using a suitable distance model (dm) which describes the means of propagation of the event (air/street) (see Section 3.1.4)
2. Determine temporal distance by analysing the
   a. Direction (d) of expansion
   b. Propagation velocity (v)
   c. Range (r) of impact as function of *dm*

3. Compute the correctness for each correlated stream by applying
    a. V_s as the set of validator functions for e and each stream $s \in S_e$
    b. τ_s as set of temporal direction (is the change in s a result of *e*, or is it a cause for *e*?)
4. Combine all partial correctness values by using the:
    a. Set of weights (W_s) for each stream $s \in S_e$
    b. A combination function (∑), e.g. min, mean

As a result, we get the combined $C_e = (S, dm, d, v, r, V_s, \tau_s, W_s, \sum)$.

An example event created by the Event Detection Component is shown in Listing 2, detecting a moderate traffic jam reported by a sensor. The Composite Monitoring is triggered by an event and uses the location of the event description to determine relevant sensors that are located nearby (see Figure 13). This example uses the Euclidian distance to select the 4 nearest sensors.

```
1  sao:a26db0a4-20ca-4f4c-b553-a799200d58ca a ec:TrafficJam ;
2        ec:hasSource "SENSOR";
3        sao:hasLevel "1"^^xsd:long;
4        sao:hasLocation [ a geo:Instant;
5              geo:lat "56.18244908701999"^^xsd:double;
6              geo:lon "10.1972915214958"^^xsd:double
7        ] ;
8        sao:hasType ec:TransportationEvent ;
9        tl:time "2016-02-12T13:57:07.001Z"^^xsd:dateTime .
```

Listing 2: Traffic Jam as Event Example

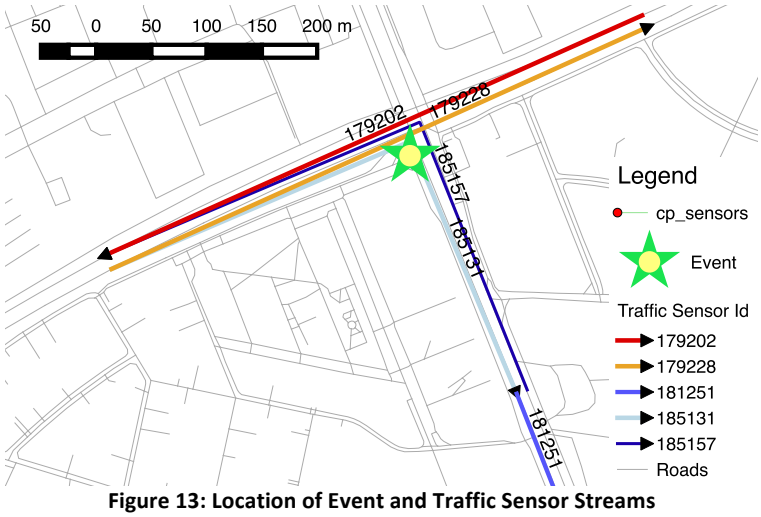

Figure 13: Location of Event and Traffic Sensor Streams

Since the event is directly located in the sensor measurement area, the temporal distance has no impact on the measurements. After analysing the time series (displayed in Figure 14) it can be evaluated that at the event time the traffic sensors (179202 and 179228) are detecting a relatively slow traffic movement for this road (compared to the average and the daily minimum). Since these 2 independent sensors are showing a similar pattern it can be assumed that the sensor measurement is plausible.

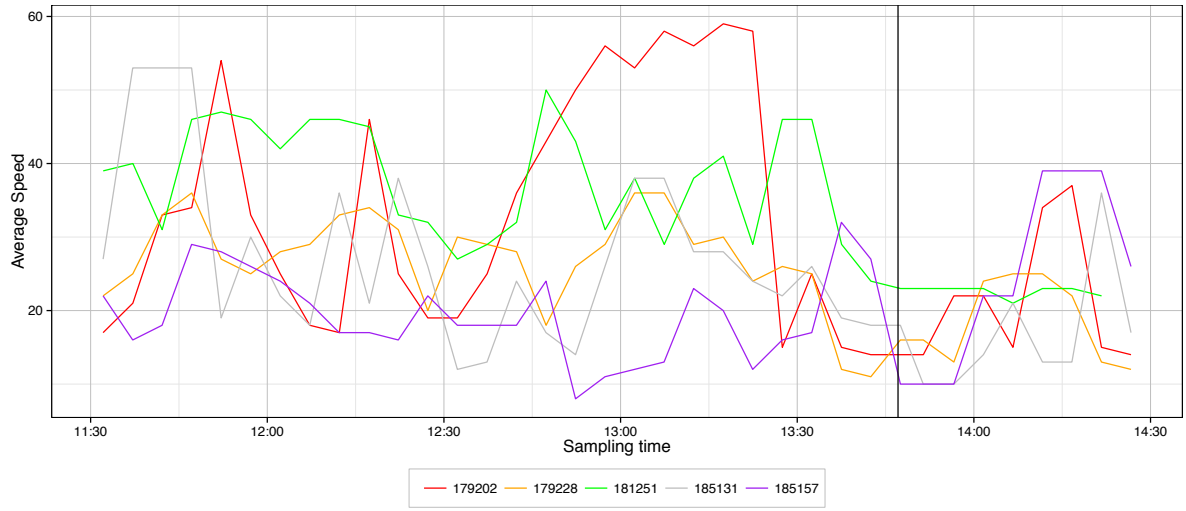

Figure 14: Unfiltered Time Series of Traffic Sensor Streams and the Detected Event

Due to seasonal patterns and low measurement frequencies, processing of raw input data shows a significant improvement in evaluating the correlations between data streams. The following methods are used (and visualised in Figure 15) as a pre-processing before the calculation of correlations between the data streams based on the individual scenarios:

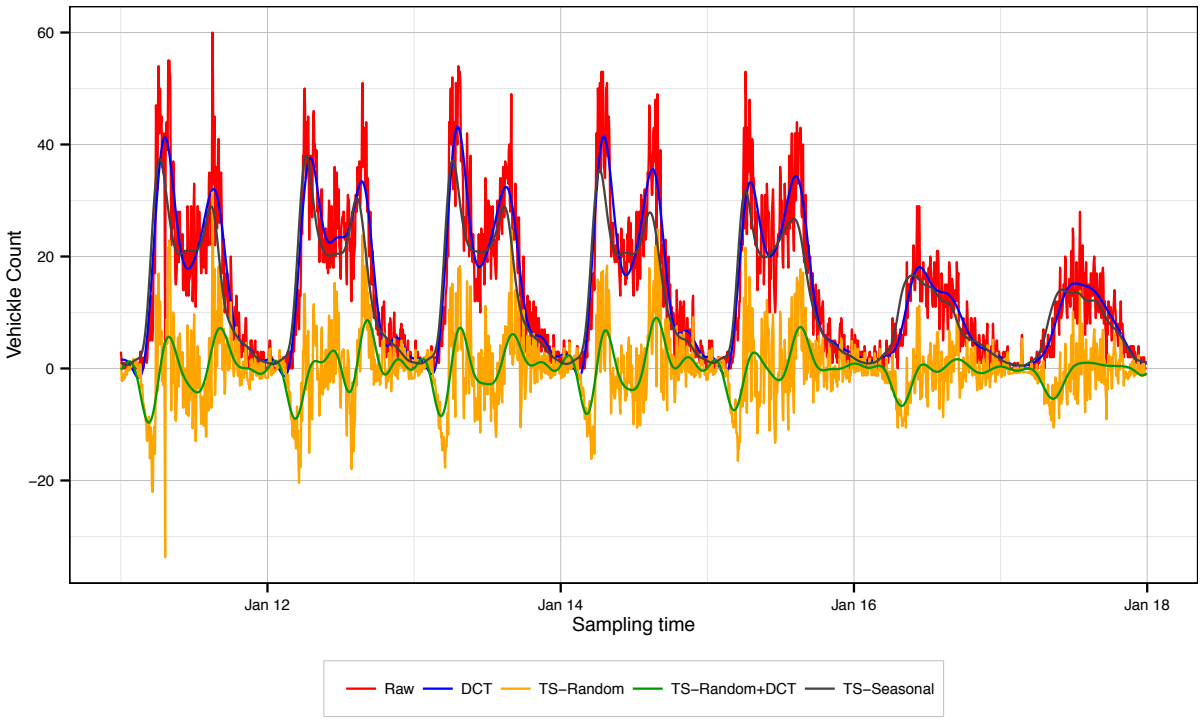

Figure 15: Pre-processing for Time Series Data before Correlation Calculation

Unmodified input data (Raw) has limited significance due to seasonal time series patterns but can be useful to evaluate detected events since only a small amount of data has to be available. For example, in order to compare the last 15 minutes (e.g., 3 measurements surrounding the event in Figure 14).

Discrete cosine transform (DCT): The discrete cosine transform (DCT) expresses a finite sequence of data points as a sum of cosine waves with different frequencies and amplitudes. The implemented discrete cosine transform filter depicted in Figure 16 uses the DCT to convert a signal (1) to an ordered sequence of frequencies and associated amplitudes (2). This sequence is then multiplied by a curve signal (3) to cancel out high frequencies (4). In the end the invers discrete cosine transform (IDCT) is used to convert the low pass filtered sequence back to a smoothed signal (5).

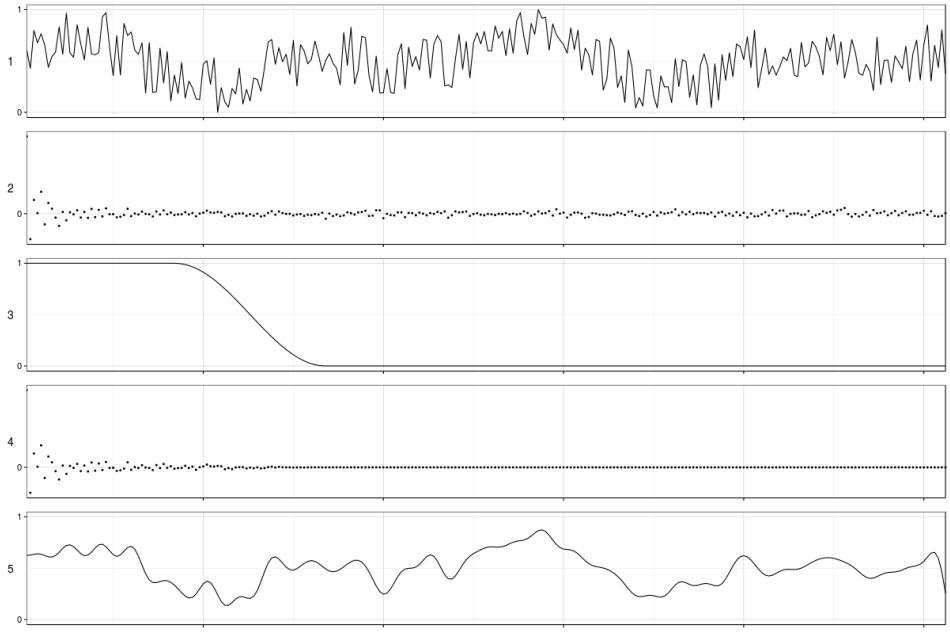

Figure 16: Applied DCT Smoothing of Time Series

The smoothened representation of the relevant sensor data leads to a better comparability of the time series regarding correlation algorithms.

Random Part of Time Series (TS-Random): The extraction of the seasonally adjusted random part of the time series decomposition allows to identify irregular changes in the time series. Whereas during the night there may be no traffic in a measurement interval, during daytime it could be an outlier that is caused by a traffic jam or defective sensor.

Random Part of Time Series + DCT (TS-Random + DCT): The combination of TS-Random and DCT allows easy comparability of acute and irregular events between time series.

Seasonal Part of Time Series (TS-Seasonal): Figure 15 also shows the seasonal part that was extracted from the time series

The results of the following Section 3.1.4 will show that due to seasonal patterns and low measurement frequencies a pre processing of raw input data shows a significant improvement in the evaluation of the correlations between data streams.

3.1.4 Spatial Dependencies

Reasoning in cities depends heavily on the spatial context. While temperature will be similar in the neighbourhood, noise propagation will depend on shielding buildings and traffic flows on road networks, on-going construction work, traffic density etc. Hence, spatial reasoning requires appropriate distance measures. However, the integration of large amounts of data sources and the computational complexity demand for efficient methods to provide the requested information in (real) time. This sections will a) discuss different distance measures, b) outline an iterative approach to bring in context knowledge constraining the search space and c) employ spatial interpolation to increase reliability. Interpolation of spatial data typically utilises variations of the Euclidean distance to find suitable sources of neighbouring sensors and to weight their impact [Zimmermann 1999]. Alternative weighting methods are utilising further sensor data (like wind data) [Mesnard 2013][Li 2014]. We propose the use of available infrastructure knowledge for sensor data analysis.

The Euclidean distance between two locations usually defines a brief coherence if for example certain events affect nearby entities or persons. However, applied in a complex city environment this metric does not represent the relevance of nearby events. Road-networks and encapsulated public transportation systems have to be considered since the beeline often does not reflect possible ways or distinct connections that can be used to reach or affect another location. Figure 17 shows the different distance models that can be used in the space of a city:

a) The direct radial propagation of an effect issued on a specific location that results in a Euclidean distance as a distance model.
b) The consideration of a directed street graph enables a more distinct model for measuring a distance, interpolating traffic related data and calculating the propagation of traffic-related events
c) Public transportation systems like trains allow only distinct boarding and exit stops, which further decrease the flexibility of the public space and reduce the details of the utilised graph.

a) Euclidean Distance **b) Shortest Path Distance** **c) Reduced Graph Distance**

Figure 17: Comparison of Spaces to Determine Distances

Let a distance function d (e.g., Euclidean on a metric projection where 1 unit is 1 meter) be defined on space X. K is a set of indices and the tuple $(P_k)_{k \in K}$ of nonempty subsets (the locations/events) in X. The Voronoi cell, R_k, associated with the site P_k is the set of all points in X whose distance to P_k is not greater than their distance to the other sites P_j, where $j \neq k$. The gapless Voronoi diagram is defined as

the tuple of cells $(R_k)_k \in K$. Figure 18 shows the Voronoi cells for traffic flow sensors, which are deployed in the road network of the city. The Voronoi cells illustrate that the nearest traffic sensor is unlikely to represent the condition of the illustrated street segments.

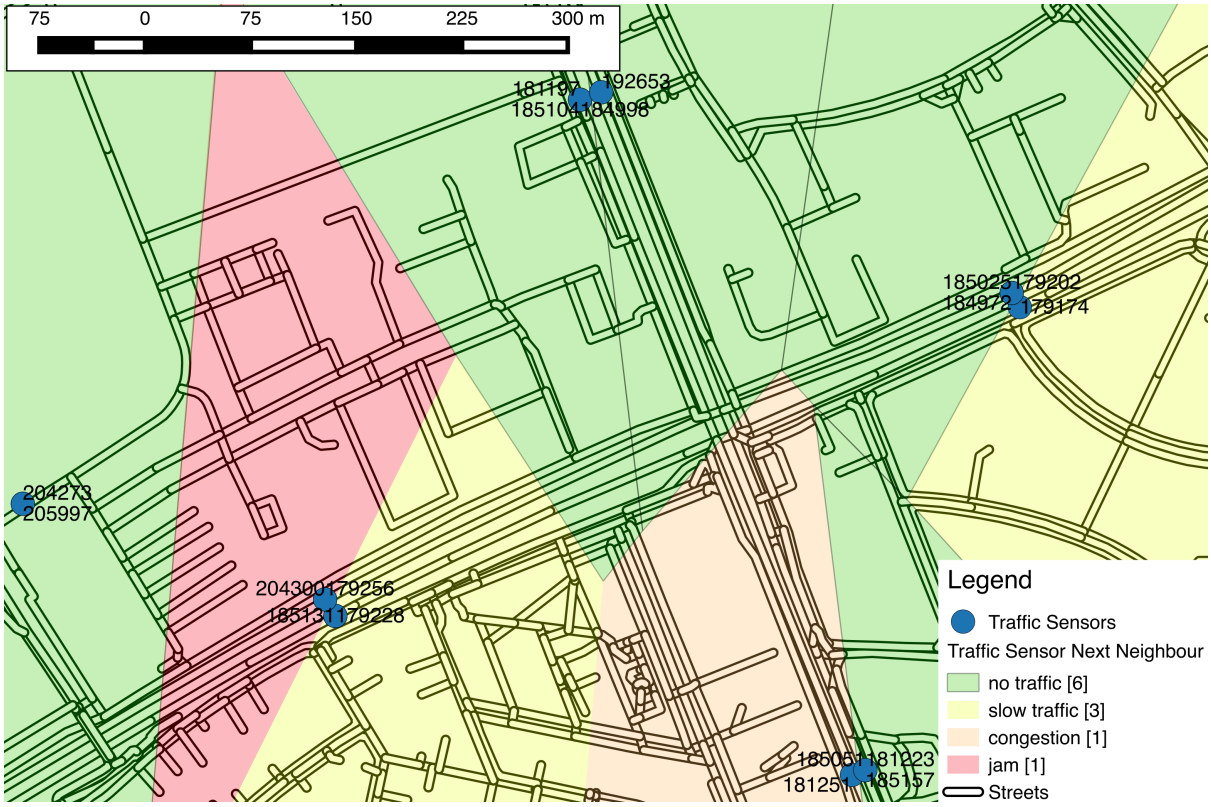

Figure 18: Voronoi Diagram - Depicting the Nearest Traffic Sensor

Distance Measure Accuracy

To find locations such as nearest parking space or closest hospital an iterative strategy is suggested reducing candidates and increasing accuracy and computational complexity stepwise. While the computation of the Euclidean distance is fast, the consideration of the road network is more accurate. In a first step the Euclidean distance can reduce candidates. But for exact navigation in a city the roadmap is needed.

To analyse the accuracy limits for a real scenario an infrastructure dataset from the city of Aarhus in Denmark was investigated, containing a routable graph, 3 hospitals, 13 pharmacies, 25 ATMs, 36 toilets, 45 waste baskets and 288 parking places were included in the examined area. For random locations in the city the Euclidean distance and the routing distance, calculated with a shortest path algorithm in the cities street network, have been compared. Table 1 shows an exemplary result of the experiment.

Order	Object	From(lon lat)	Id	EuclDist	RouteDist
1	hospital	10.19062 56.19125	52592	2665m	3422m
2	hospital	10.19062 56.19125	10646	2694m	3046m

Table 1: Example Results for Misleading Distances

The column *From(lon lat)* describes the starting point of the search; *Id* is the internal database identification of the hospital that was found in the search; *EuclDist(m)* shows the Euclidean distance between the starting point and the hospital in meters; *RouteDistance(m)* shows the shortest path, using the street network of the city in meters. In this example, the first hospital appears to be the nearest one by utilising the simple Euclidean distance. Taking the road network into account, the second hospital is 11% nearer then the first one. Figure 19 shows the overall results of the experiment. It depicts the error ratio in a list of the 2-5 nearest objects. If the Euclidean distance order was correct an entry is marked as correct. If the routing distance alters the order, the route is marked as incorrect. The experiment was repeated with 20000 random locations for the 6 objects. Since there are only 3 hospitals available, the hospital experiment was capped at a list length of 3. For this experiment one-way streets and vehicle restrictions have been ignored.

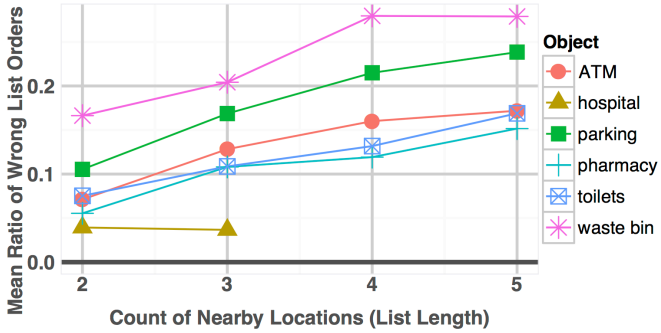

Figure 19: Evaluating Euclidean Distance with Actual Road Distances

The distance difference depends on terrain and road interconnectivity, based on the length of street segments and the street density. These static measures can be computed in advance and are used to control the stepwise selection and computation of distance measures in (real-) time.

Sensor data is often correlated and can be combined to increase reliability. Figure 20 shows the Pearson correlation between parking garage usage and close traffic sensors. The grey variance area of the regression points out the necessity to use shortest path distance for reliable sensor fusion. Selecting sensor pairs by shortest path results in higher correlation (Pearson correlation, Minkowski distance and Bray–Curtis dissimilarity) than using the Euclidean distance.

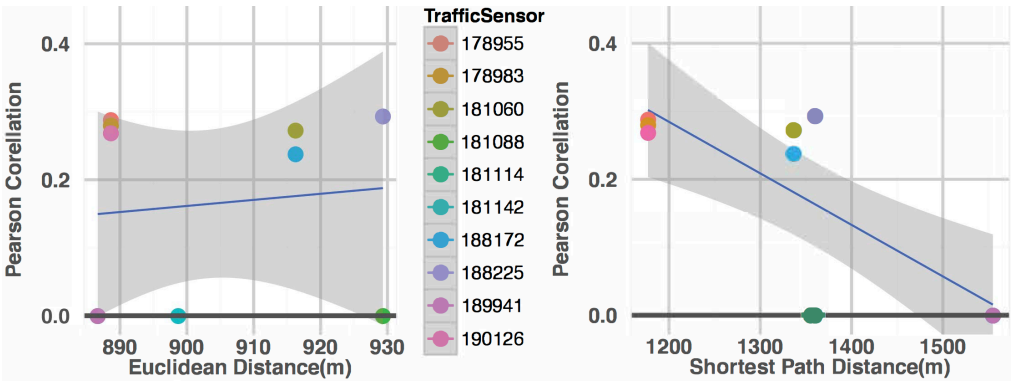

Figure 20: Pearson Correlation Between Traffic of Parking Garage and Nearest Traffic Sensors

To evaluate the applicability the distance metric results in the context of sensor data a dataset with 12 months of traffic data was used. The result is documented in Figure 21 and Figure 22 and described in the following paragraphs. The time series (vehicle count and average speed) of 449 traffic sensors was pairwise compared using the following similarity metrics:

- Pearson correlation
- Spearman correlation
- Kendal correlation
- Reward and Punishment based boundary algorithm (see [Deliverable 4.1] Section 5.3)
- Hamming Distance

The resulting 201152 similarity values (e.g. correlation coefficient) have been calculated for each combination of reports and every week in the given period with all of the above data modifications.

Afterwards these result have been analysed against the following set of distance metrics described in Table 2:

Keyword (for Figure 21 and Figure 22)	Description
euclidean_distance	Direct Euclidean distance between two sensors in meters.
route_distance	Route distance between two sensors in meters.
route_duration	Travel time by car of the route between two sensors.
route_steps	Number of steps (road crossover, turns, etc.) of the route between two sensors.
directed_duration	Additive composition of the duration and the angle between two sensors.
Sensor_hops	Number of sensors to jump over to get from one sensor to another. Equal to number of sensors if it is not possible to reach another sensor.

Table 2: Applied Distance Metrics

For every sensor the calculated similarity metrics have been correlated (with Pearson correlation) against the different distance metrics. The utilisation of matching metrics of the time series shows a significant effect on the correlation. Figure 21 shows the results of the experiment. The majority of correlation values are negative since the similarity between traffic sensor time series increases with shorter distances.

Figure 22 depicts the different correlations with the consideration of a time offset, which describes the time shift between different time series that were used to enable the modelling of propagation speed between sensors (route duration). Before the first similarity metric step, the compared time series was shifted regarding the estimated time delta of the propagation of the traffic.

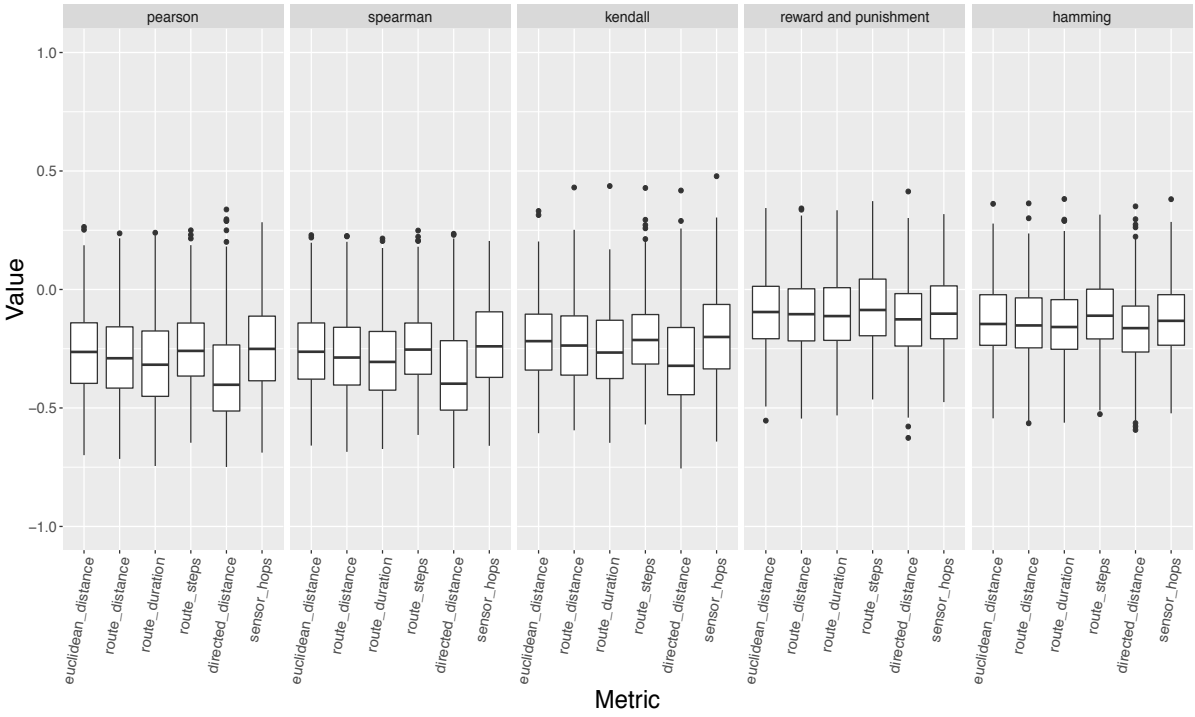

Figure 21: Correlation of Sensor Dependence against Different Distance Models

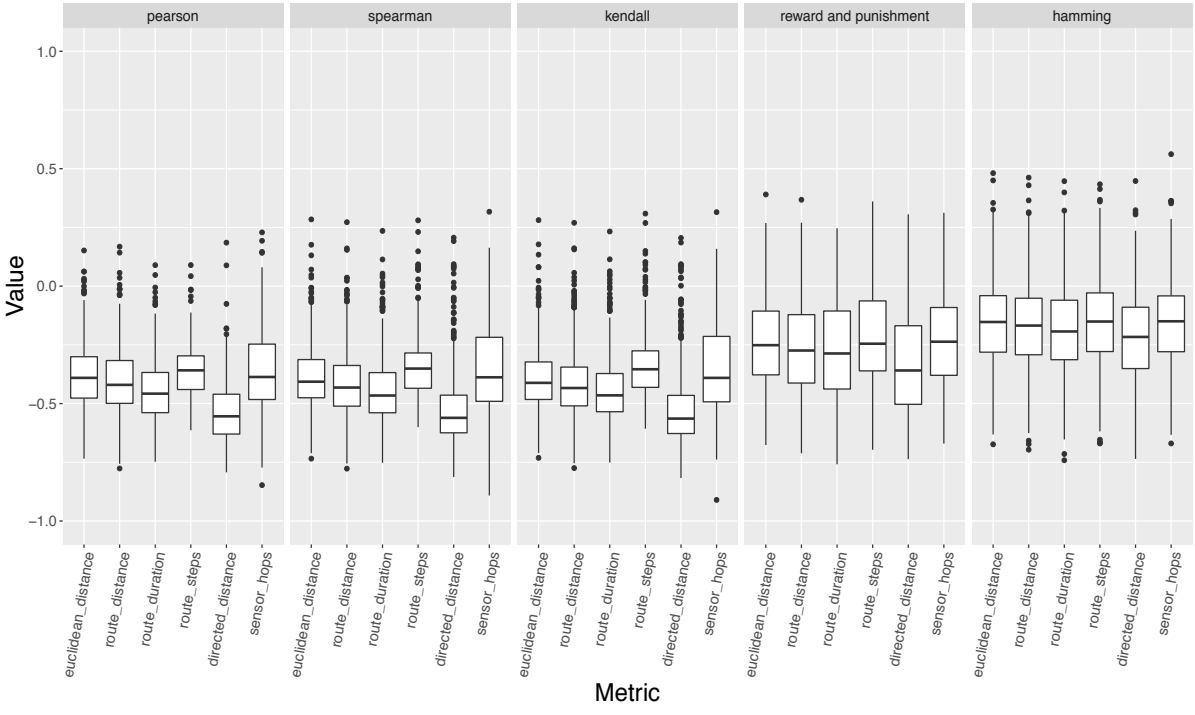

Figure 22: Correlation Results Optimised by Duration Offset Shift

The results show that the consideration of the propagation of events (time shift/offset) is an eminent step to compare e.g. traffic related sensor information. The offset-pre-processed time series show higher correlation coefficients (negative) and a smaller variance. The utilisation of the shortest path algorithm also shows a significant better (negative) correlation than the standard usage of Euclidean distance. The directed distance approach even tops this approach since it also takes the direction of the propagation into account.

3.2 Testing Concept

3.2.1 Smart City Application Testing Concept

The goal in CityPulse is to test the reliability of smart city applications with respect to the reliability of the external resources required for its execution. For this a series of test cases has to be generated, where each test case 'simulates' a less reliable deployment of external resources as the former. Since a true ground truth is not accessible, the first test case (TC_0) in such a series uses unmodified historic data, which are represented by the Reference Dataset from the CityPulse project [CityPulse-D2.3], and acts as ground truth. In each following test case the output of the CityPulse framework is recorded. The distances between the inputs of a test case to TC_0 and the distances of the outputs of same test cases must correlate or lie below a threshold in order to pass a test case.

The execution of a test case utilises the replay mode of the Resource Management, which is already capable of using historic datasets instead of live data. A setup for a test case is therefore the replacement of the original historic datasets with the manipulated, degenerated datasets. To limit the number of possible external resources involved in a test case an application profile states the types of resources it uses during execution. A geo-spatial search for relevant external resources for a specific test scenario will further reduce the test case input space. Both measure reduce the test execution time significantly. Figure 23 illustrates the test execution process and highlights the loop, in which the ground truth is degenerated up until the application output changes sufficiently to be able to make a statement about its robustness. For the sake of simplicity, the loop's end condition in case the maximum number of iterations was executed is not shown in the figure. The next section will describe in more detail the degeneration process of the historical data. In contrast to monitoring the testing is executed during the design-time of an application respectively before the deployment of an application.

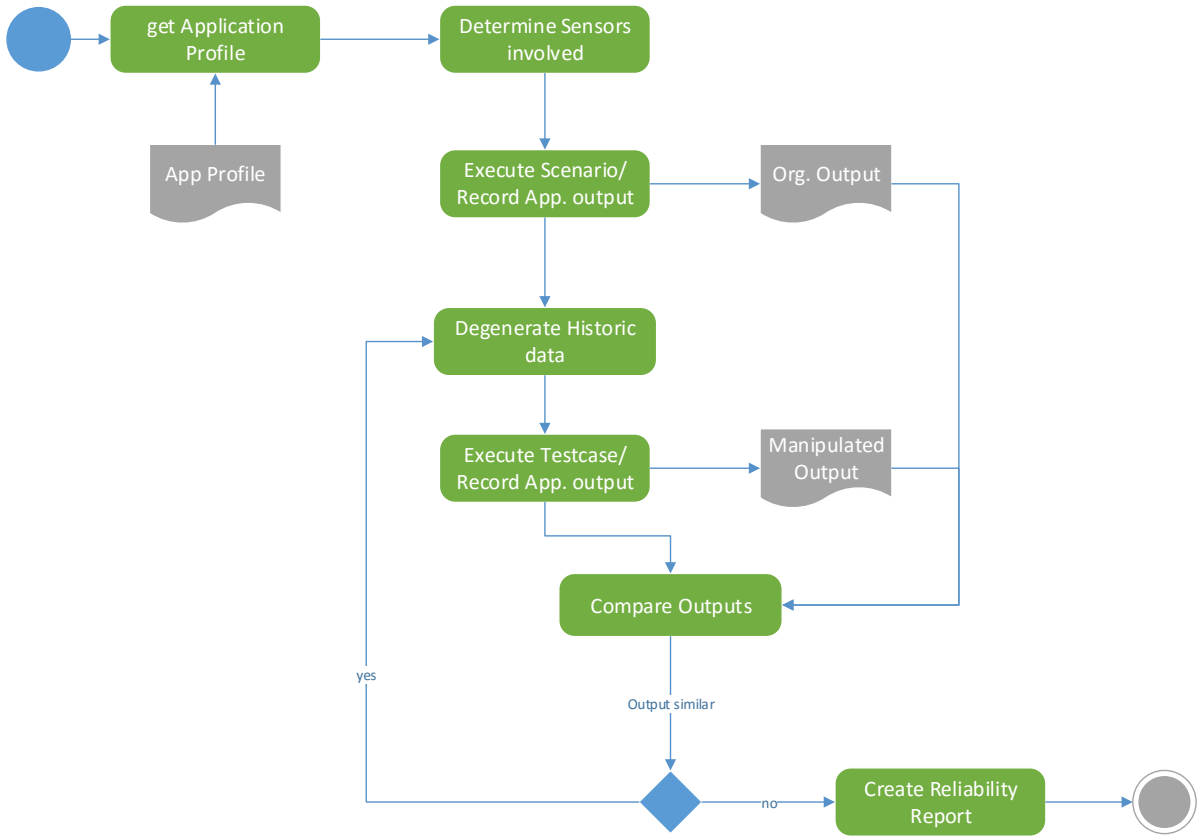

Figure 23: Execution of Test Cases

3.2.2 Degeneration of Input Data for Iterative Test Case Derivation

In this section we define the test case generation process. The process uses various error models to generate the test case stimuli for a set of test cases (*TC*).

We define *TC* as a tuple *(E, S, H, $P_{e,s}$, n, sr, A, Ω)*, where *S* is the set of sensors involved, *E* is the set of error models applied to each sensor $s \in S$ for each test case TC_i in *TC*, with *i = (0,n).*

$P_{e,s}$ denotes activation probability functions for an error $e \in E$ and a sensor $s \in S$. n is the total number of iterations and thus the number of test cases.

H denotes the historic datasets, which act as ground truth.

The statement *H(s, τ)* denotes the historic value of sensor *s* at a specific time τ. The tests are executed in a discrete system, where each step (called tick) jumps forward in time.

The sampling rate *sr* denotes the difference in time between two successive ticks. *A* and Ω represent the start and end date respectively.

Let $P_{e,s}$: $i \rightarrow (0, 1)$, with $i = (0, n) \in \mathbb{N}$, be the activation probability function for an error e and the sensor s. The function returns the probability that e will be activated at iteration i. For example, with a value of 0.1 there is a 10% chance that the error will be activated. For the realisation we use a pseudo random number generator with uniform distribution. If the random number is lower than or equal to the activation probability the error gets activated.

An error $e \in E$ is defined as $e = (v', \Delta t, d)$. The triple represents the three dimensions of effects an error can influence the output signal of a measurement equipment, as identified in Section 2.1. Here v' denotes the value change in the offset dimension for a specific sample, Δt is the duration (number of ticks) how long the error is active in the duration dimension. An error e for a sensor s is said to be active if the activation probability function $P_{e,s}$ activates it or the last activation was less then Δt ticks ago. An activation probability function should be monotonically increasing for subsequent test cases in order to test increasingly unreliable sensors. The parameter d specifies the number of ticks the new value v' is delivered late in the delay dimension.

With these definitions the generation of test case stimuli for one test case can be described as follows:

```
1  While τ = A + (sr * tick) < Ω:
2          For each sensor s in S:
3                 v = H(s, τ)
4                 for each error e in E:
5                       activate e, if Pe,s = true  e not active
6                       v' += apply e on v if e is active
```

Listing 3: Test Case Stimuli Definition

The new values v' substitute the original values at $H(s,\tau)$ for each sensor and form this way the test case TC_i. The process is repeated for n iterations, leading to test cases with increasing unreliable sensor, if the activation probability function is monotonically increasing.

4. Conflict Resolution and Fault Recovery

This section describes the concept behind the Conflict Resolution and Fault Recovery (see Figure 24 orange square). Conflict Resolution's intention is to find the most accurate data streams in case of contradictory observations. The Fault Recovery will estimate observations in case the quality of a data stream has dropped or observations were omitted.

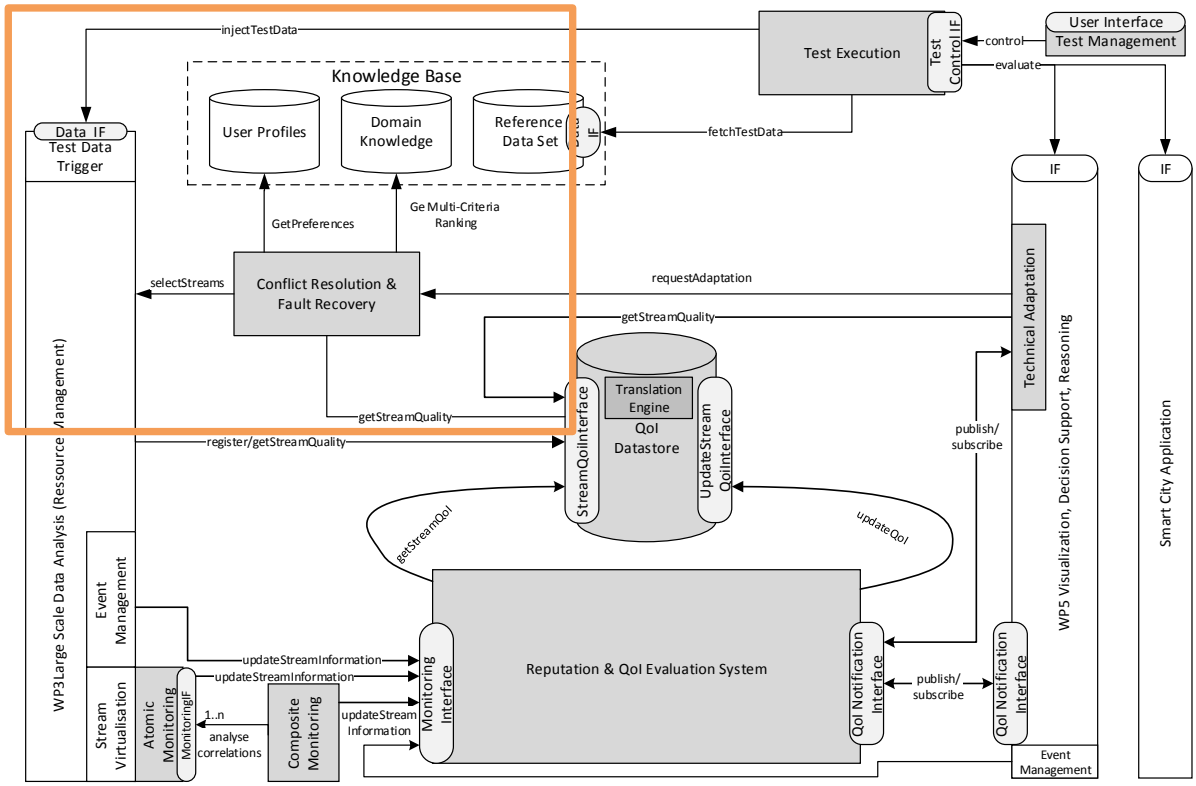

Figure 24: Conflict Resolution and Fault Recovery within the Work Package Architecture

4.1 Conflict Resolution

When the Conflict Resolution component receives a request for solving conflicts, it sends the relevant stream ids to the Data Federation component, which will query the stream meta-data to find the stream quality values and compare them. In cases where the conflicting streams are composite, the Data Federation component is responsible to aggregate and estimate their quality values using algorithms described in Deliverable 3.2 [CityPulse-D3.2].

Once the stream quality values are obtained, the Conflict Resolution component can simply choose the stream with highest accuracy value for the user, or the stream with the highest overall QoI value computed from multiple quality dimensions. Alternatively, it can group the streams based on the values they report, and calculate the believability of the values based on aggregating their accuracy values, i.e., given a set of conflicting streams S and total set of streams $\dot{S}$, a value function $v: S \rightarrow V$ gives the value of a stream $s_i \epsilon S$, a grouping function $g: S \rightarrow G$ is defined, where $G \subseteq \dot{S}$ and $\forall s_i^j \epsilon G_i, v(s_i^j) = C$, where C is a constant value. Then the best stream s_k is given by the function $max\{\prod_{s_j^k \in G_j} acc(s_j^k)\}$ where the function *acc* gives the accuracy of a stream.

Both solutions of resolving the conflicts consider only the latest quality observations. There can be more sophisticated ways of deciding which streams to believe, for example, by looking into the historical quality patterns. A third-party developer can extend the Conflict Resolution component to incorporate his/her own algorithms to solve the conflicts.

4.2 Fault Recovery

In Deliverable D4.1 [CityPulse-D4.1], at Section 6.3 Missing Values Estimation is presented as the core functionality of the Fault Recovery component. It is based on k-NN (k-Nearest Neighbour) algorithm [Aha 1991] and uses reference historic dataset for identifying similar situation with the current one, when an observation is missing or the quality of the stream is low. After it identifies the similar situations it generates the estimated value using the reference dataset.

In the first version of the Fault Recovery component the domain expert was responsible to collect historic data and to filter it in order to select the most appropriate reference dataset. The filtering procedure involved data cleaning and deletion of similar situations. This process is usually tedious and we have investigated the possibility to run the component in "online mode". More precisely the Experiences Dataset (see Figure 25 or [CityPulse-D4.1]) can be modified using the stream data when the QoI metrics of the stream are high. We have investigated two methods to implement the Experiences Dataset Updater module of the Fault Recovery. The performances of these methods are presented below.

Figure 25 depicts the updated architecture of the Fault Recovery component. The Observation Estimator module implements the k-NN algorithm and its operation has been presented in the Deliverable D4.1 [CityPulse-D4.1]. This module is triggered by the Data Wrapper (using the "Request estimation command") when a new estimation of the observation is needed due to a low quality of the stream. The component uses the k-NN algorithm and the Experiences Dataset in order to generate an estimation.

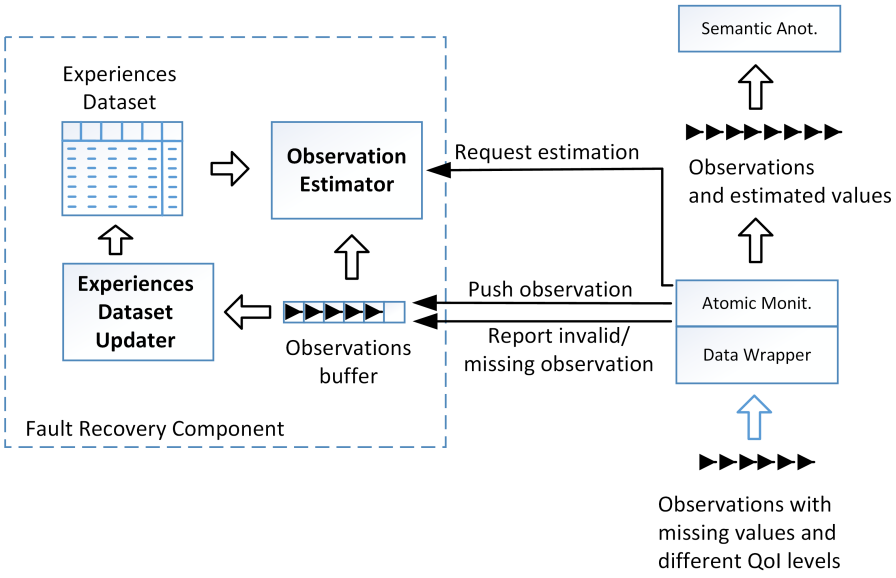

Figure 25: Fault Recovery Architecture and Integration Pattern into the Data Wrapper.

The observations buffer is implemented as a vector, which holds the latest observations generated by the data stream, using the command "Push observation". A null value is included in the buffer when an invalid or missing observation (using the command "Report invalid/missing observation") is reported.

Next the Experiences Dataset Updater module monitors the observation buffer and update the Experiences Dataset based on one of the following strategies:

- Check if a similar experience (complete sequence of observations) already exists in the Experiences Dataset and if not add it.
- Keep the last N sequences of observations (where by sequence of observation we refer to the content of the observation buffer when it does not contain any missing values).

As is presented in Figure 25 the Fault Recovery component is integrated into the Data Wrapper. The Data Wrapper is responsible to fetch the data from the data source. Next the Atomic Monitoring component performs the quality test on the observation. If the quality is on a high level, it triggers the "Push observation" function of the Fault Recovery. If not, it reports the invalid measurement and requests an estimation. At the end the Semantic Annotation module perform the semantic annotation of the observation.

In order to test the performance of the component and identify the best update strategy we have used the traffic and parking data from city of Aarhus. Below we present the results of the various experiments we have performed using the traffic sensor data[1] between 2014-02-13 and 2014-11-13. The dataset contains about 77000 observations.

Figure 26, Figure 27, and Figure 28 present the results for the following test configurations:

- Case A: Observations buffer length 20 and "Check if a similar experience already exists in the Experiences Dataset and if not add it into" update methodology;
- Case B: Observations buffer length 5 and "Check if a similar experience already exists in the Experiences Dataset and if not add it into" update methodology;
- Case C: Observations buffer length 5 and "Keep the last 100 measurements" update methodology.

[1] Sensor ID: 158324
[2] http://www.odaa.dk/dataset/parkeringshuse-i-aarhus

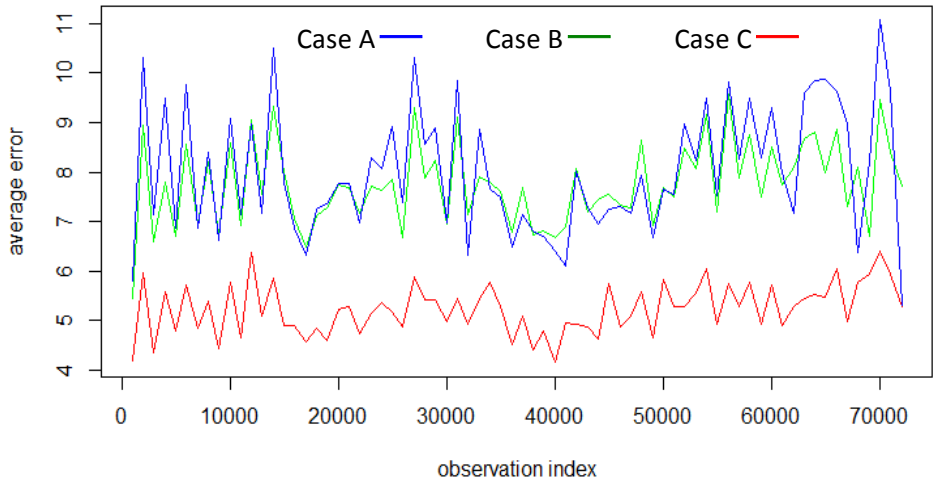

Figure 26: Average Error Comparison

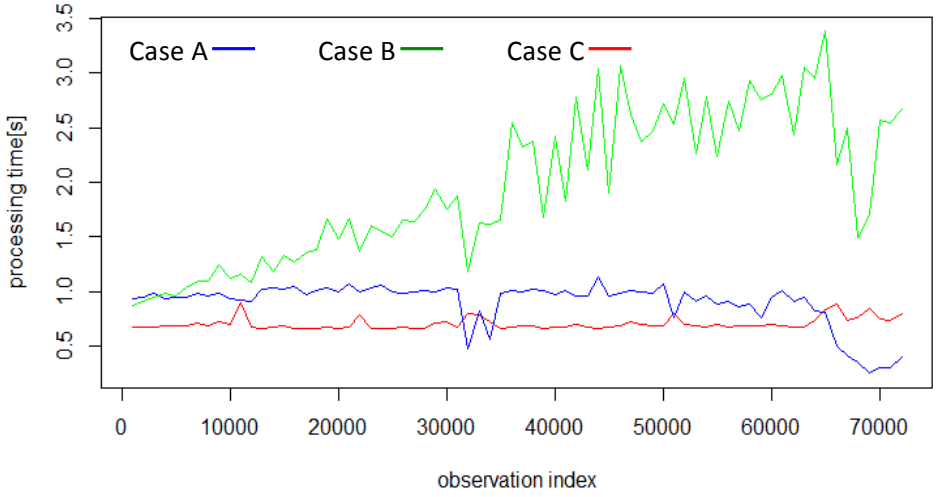

Figure 27: Processing Time Comparison

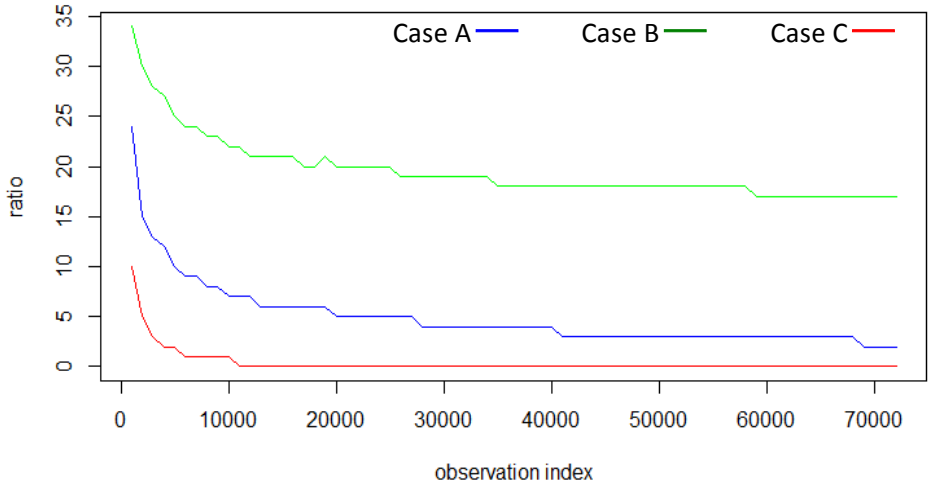

Figure 28: Memory Consumption Comparison

During the tests the observations from the dataset have been sent to the Fault Recovery component. Before sending an observation the Fault Recovery component is triggered to estimate an observation. This way we can compute the precision error. After 1000 observations have been delivered to the Fault Recovery we compute the average precision error, memory consumption (representing the size of the Experiences Dataset divided by the size of the test dataset and multiplied by 100), and the time needed to process the 1000 observations.

After analysing the average error (see Figure 26) we can conclude that it is enough to use the latest 100 sequences of observations in order to do a sufficient prediction. As one can see case C offers better estimation precision than Case A and B. According to Case C description, the algorithm is going to have constant size for the Experiences Dataset, which implies less processing to be done (demonstrated by the graph from Figure 27) and less memory to be used (visualised in Figure 28).

As a conclusion we have configured the Fault Recovery component to work according to case C description.

5. Implementation

This section describes the implementation of the work package 4 components namely the Conflict Resolution, the Fault Recovery, and the Quality Monitoring. In addition, the interaction between the Atomic Monitoring and the Fault Recovery is described in detail. The whole CityPulse framework is released as open source. Table 3 provides links to retrieve the source code of all components. We plan to provide continuous updates of the components and develop future revisions.. This approach enables the project to support the open source community by delivering fully working solutions, which can be used/extended for other projects.

Component	Github Address	Further Information
Resource Management	https://github.com/CityPulse/CP_Resourcemanagement	Contains Fault Recovery and Atomic Monitoring. Needed for online mode of Quality Explorer.
Atomic Monitoring	https://github.com/CityPulse/CP_Resourcemanagement/tree/master/Quality	Is directly integrated into Resource Management.
Composite Monitoring	https://github.com/CityPulse/CompositeMonitoring	In contrast to the Atomic Monitoring this is a separate component.
Fault Recovery	https://github.com/CityPulse/CP_Resourcemanagement/tree/master/virtualisation/wrapper/faultrecovery	Is directly integrated into Resource Management.
Quality Explorer	https://github.com/CityPulse/QualityExplorer	Contains Average Processing Times module. Various R libraries are needed. For online mode also a working Resource Management. Further details within README file on Github.

Table 3: Github Directories

5.1 Monitoring

As initially described in D4.1 the quality monitoring of data within the CityPulse framework follows a two-layered approach. This ensures a constant real-time quality annotation for basic quality parameters (Atomic Monitoring) and a correlation based proof of correctness to rate the trustworthiness of multiple data sources (Composite Monitoring).

5.1.1 Atomic Monitoring

The Atomic Monitoring is directly integrated into the Data Wrapper of the Resource Management. Therefore, it is implemented as a Python class, which can be internally used. A class named QoI-System manages the Atomic Monitoring. Each Data Wrapper integrates one QoI-System per sensor, which is done automatically by the AbstractWrapper class within the Resource Management (see [Puiu 2016] and [CityPulse-D5.3]). Figure 29 depicts the general structure of the module.

The several QoI-metrics are derived from a base class QoIMetric, which allows developers to easily write new metrics. The class, depicted in Listing 4, contains some class variables to calculate the quality value (line 6 -10), a reference to the Reputation-System it is contained in, and a name for the metric. The reference to the Reputation-System is used to access values of the sensor descriptions shown in the activity diagrams in Section 3.1.2. The *absoluteValue* and the *ratedValue* are used to store the current QoI values as described in the extended Quality Ontology shown in Section 3.1.1. QoI-metrics derived from the base class should mainly overwrite the update method. This method gets the data from the sensor data stream received by the Quality module. Depending on the type of metric, different data is processed in combination with the corresponding sensor description. As an example the implemented Completeness metric selects the list of fields contained in the incoming data and compares it to the list of fields the sensor should have according to the sensor description.

In contrast to the update method that should be overwritten in every QoI metric the *nonValueUpdate* method should be rewritten only for those metrics, which are time relevant. In case of the CityPulse framework the quality value of the Frequency metric should be lowered if no update has been received, but without lowering further quality values, such as Completeness. To achieve this functionality a timer is used to call the *nonValueUpdate* method periodically in case no update from the data stream was received within the specified interval (*updateInterval + 10%*). Such functionality needs to be implemented in the *nonValueUpdate* method, because the base implementation only returns the current QoI values.

```
1   class QoIMetric(object):
2         def __init__(self, name):
3               self.name = name
4               self.repsys = property(**repsys())
5               self.absoluteValue = 0
6               self.ratedValue = 1.0
7               self.initialValue = 1.0
8               self.min = None
9               self.mean = None
10
11        def update(self, data):
12              return (self.absoluteValue, self.ratedValue)
13
14        def getStats(self):
15              return (self.min, self.mean)
16
17        def nonValueUpdate(self):
18              jsonOb = JSONObject()
19              jsonOb.absoluteValue = self.absoluteValue
20              jsonOb.ratedValue = self.ratedValue
21              jsonOb.unit = self.unit
22              return (self.name, jsonOb)
```

Listing 4: QoI Base Metric

The QoI-System itself is mainly a managing module to initialise the inner class Reputation-System with the quality metrics for the sensor data stream and the description for the stream. It is responsible to set a list of QoI-metrics for the system. In addition, it is possible to add so called sinks, to direct the annotated data into other components of the framework. For example, it is possible to build an N3 (Notation 3) formed turtle graph and to save it to a triplestore independently from the Resource Management. The complete initialisation and the processing of incoming data is shown in a sequence diagram (see Figure 30).

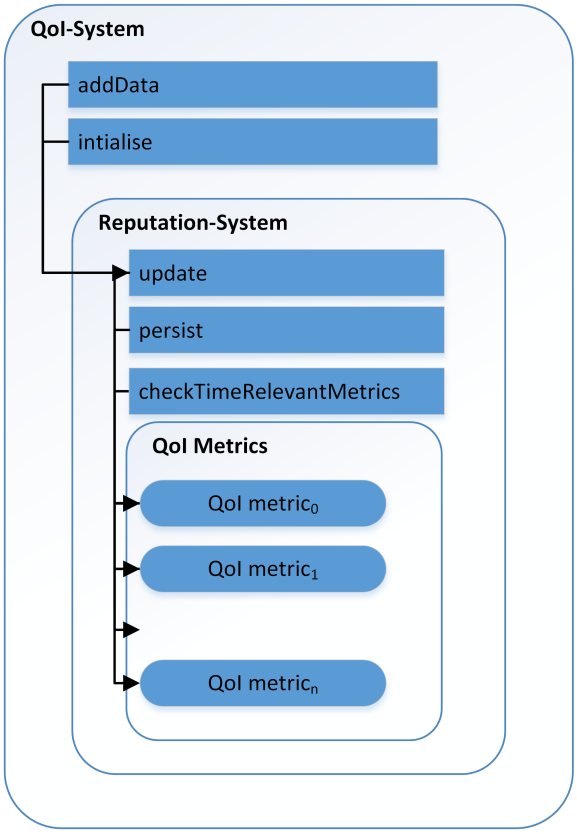

Figure 29: QoI System Structure

The Reputation-System class contains the list of QoI metrics and (optionally) added sinks set by the QoI-System. Its main responsibility is to keep a copy of the sensor description for the QoI metrics, to update the QoI metrics with incoming data, persist the data if needed, and to create a timer mechanism to support time relevant metrics. For this a timer executes the *checkTimeRelevantMetrics* method and iterates over the QoI metrics to update the time relevant metrics. The main activities of the QoI-System are shown in the following sequence diagrams:

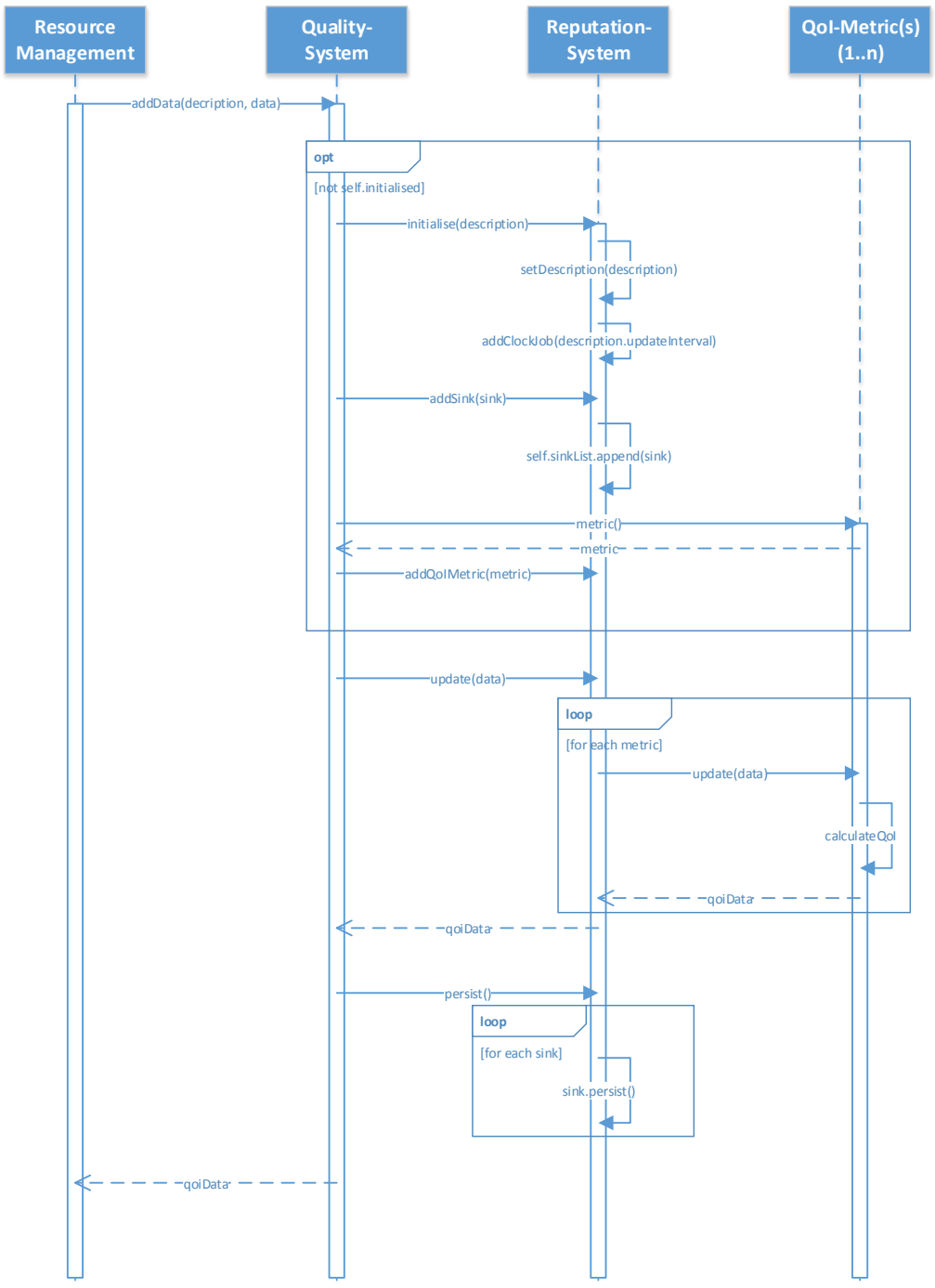

Figure 30: Sequence Diagram QoI-Update

Figure 30 shows the relation between the Resource Management and the Quality-System with its capsulated inner classes. In a first step the Resource Management sends data received from a data stream to the Quality-System as the main entrance point of the QoI module. The next step is to initialise the Reputation-System and its metrics if not done before. To execute time related QoI calculations an additional timer job is created. If the system is initialised, the received data is handled by each quality metric, which can than calculate the new quality values. Each QoI metric returns the current quality, which are collected by the Reputation System and sent back to the

Resource Management for further processing. For every sink set in the Reputation-System, the persist-method is called.

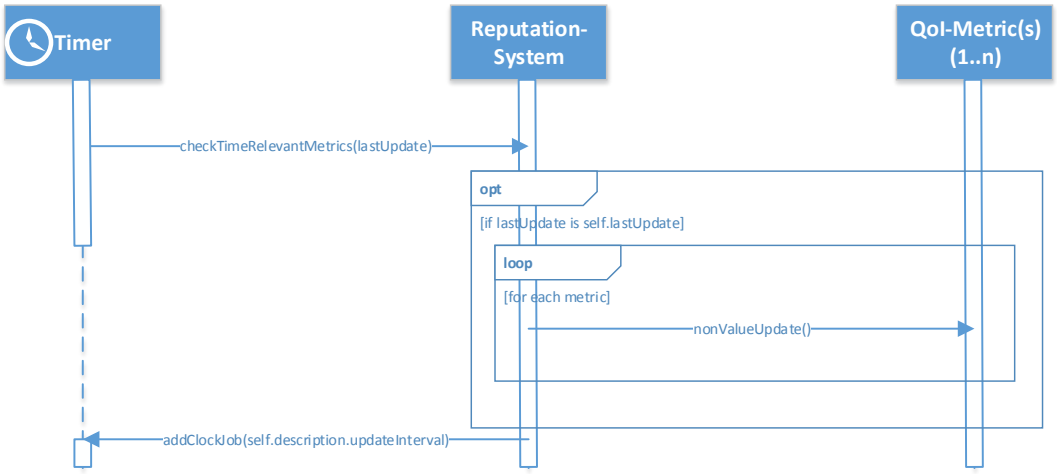

Figure 31: Sequence Diagram QoI-Update (Timer)

Figure 31 depicts the process of a timer based quality update. When the timer is expired it calls the *checkTimeRelevantMetrics* method with the timestamp of the last QoI update. The Reputation-System compares the time delivered by the timer with the time of the last update. If they are equal, there has not been an update and the *nonValueUpdate* method of the QoI metrics is called. After this process a new timer with the stream's *updateInterval* (plus a 10% safety buffer) from the sensor description is created.

Hourly Aggregation

The Conflict Resolution component in work package 5 (see [CityPulse-D5.3]) needs a quality rating for each data stream. As the Atomic Monitoring only features an observation based QoI annotation an additional module is implemented to rate data streams within an hourly interval. The functionality and implementation of this module is described in the following.

To calculate an average QoI value the single quality values of all observations for a data stream are stored in a cache with a capacity of one hour. Each data stream has a number of buffers equal to the number of configured QoI metrics. The quality information is normally stored in the framework's triplestore. The caching mechanism is an extension, which allows a faster access to average QoI values compared to extensive SPARQL queries. Due to the fact that the cache only stores numerical values the additional used memory is negligible.

There are two possibilities to access the average quality values. The first one is an HTTP based API provided via the Resource Management. The second one is to access the triplestore used by the Resource Management. A graph with the average QoI values is stored there every hour.

The API consists of two functions. The first function gets the last average quality values for all data streams registered to the framework. The average is calculated on the fly and not in fixed intervals.

Here the aggregation begins at the time of the request minus 1 hour. Details for the request are depicted in Table 4.

Function	getAllLastQualities	**Method**	GET
Description	Returns current/last QoI values calculated by the Atomic Monitoring for all streams registered to the Resource Management.		
Parameter	None		
Answer	If success		
	"list"	Contains a list of dict elements for every stream	
		"QoI metric name"	The name of the QoI metric (e.g. "Latency")
			CURRENT: States that QoI values are the current (last) ones.
			ratedValue: QoI value normalised to 0-1 by Reward and Punishment algorithm.
			absoluteValue: The absolute QoI value (e.g. 10 for latency
			unit: The unit of the QoI metric (e.g. "http://purl.oclc.org/NET/muo/ucum/unit/time/second" for latency)
		uuid	The uuid of the stream the QoI values belong to.
	If not successful.		
	status	In case an error occurred "Fail".	
	message	In case an error occurred a description of the error.	
	uuid	The uuid parameter as used in the request.	

Table 4: Last Qualities API 1

Listing 5 provides a possible answer for a request to *getAllLastQualities*. The example is based on a Romanian weather sensor reporting in an interval of 1 hour. The answer is structured as described in Table 4. The answer is shortened and depicts only one data stream.


```
[
   {
      "Latency":{
         "CURRENT":{
            "ratedValue":1,
            "absoluteValue":0.65055
         },
         "unit":"http://purl.oclc.org/NET/muo/ucum/unit/time/second"
      },
      "Completeness":{
         "CURRENT":{
            "ratedValue":1,
            "absoluteValue":5
         },
         "unit":"http://purl.oclc.org/NET/muo/ucum/physical-quality/number"
      },
      "uuid":"bcf90dc7-9da6-51a7-8b8e-da831a9918c8",
      "Age":{
         "CURRENT":{
            "ratedValue":1,
            "absoluteValue":2267.0
         },
         "unit":"http://purl.oclc.org/NET/muo/ucum/unit/time/second"
      },
      "Correctness":{
         "CURRENT":{
            "ratedValue":0.978947368421053,
            "absoluteValue":1
         },
         "unit":"http://purl.oclc.org/NET/muo/ucum/unit/fraction/percent"
      },
      "Frequency":{
         "CURRENT":{
            "ratedValue":0.7789473684210546,
            "absoluteValue":0.0002777777777777778
         },
```

```
            "unit":"http://purl.oclc.org/NET/muo/ucum/unit/frequency/Herz"
         }
      },
      {
         "Latency":{
            "CURRENT":{
               "ratedValue":1,
               "absoluteValue":0.118774
            },
            "unit":"http://purl.oclc.org/NET/muo/ucum/unit/time/second"
         },
         …
]
```


Listing 5: Last Qualities Answer 1

The second function is called *getQualityValues* and in contrast to the *getAllLastQualities* function it is configurable and has additional options, which are described in Table 5. In addition, a possible answer is shown in Listing 6.

Function	getQualityValues					**Method**	GET
Description	Returns QoI values calculated by the Atomic Monitoring for all streams registered to the Resource Management.						
Parameter	uuid	The UUID of a Data wrapper/stream. Multiple UUIDs can be provided as comma separated list.					
	types	Provides a possibility to get the average quality for timespan. Possible options are “HOURLY” and “DAILY”. Additional options “WEEKLY” and “MONTHLY” are implemented but not configured. The selected fields are added to the preconfigured “CURRENT” field for QoI values.					
	avg	Set this parameter to “True” to display average QoI values.					
	minimum	Set this parameter to “True” to display minimum QoI values.					
	maximum	Set this parameter to “True” to display maximum QoI values.					
Answer	If success						
	“list”	Contains a list of dict elements for every stream					
		“QoI metric name”	The name of the QoI metric (e.g. “Latency”)				
			CURRENT	States that QoI values are the current (last) ones.			
				ratedValue	QoI value normalised to 0-1 by Reward and Punishment algorithm.		
				absoluteValue	The absolute QoI value (e.g. 10 for latency		
			“type”	One type entry for every type in parameter “types”.			
				absoluteAvg	If parameter “avg==True” displays the average value for absolute QoI values.		
				ratedMax	If parameter “maximum==True” displays the maximum value for normalised QoI values.		
				ratedAvg	If parameter “avg==True” displays the average value for normalised QoI values.		
				absoluteMin	If parameter “minimum==True” displays the minimum value for absolute QoI values.		
				absoluteMax	If parameter “maximum==True” displays the maximum value for absolute QoI values.		
				ratedMin	If parameter “minimum==True” displays the minimum value for normalised QoI values.		
			unit	The unit of the QoI metric (e.g. “http://purl.oclc.org/NET/muo/ucum/unit/time/second” for latency)			
		uuid	The uuid of the stream the QoI values belong to.				
	If not successful.						
	status	In case an error occurred “Fail”.					
	message	In case an error occurred a description of the error.					
	uuid	The uuid parameter as used in the request.					

Table 5: Last Qualities API 2

In contrast to the first function the output in the listing below is related to parameters that are provided within the function call. The *CURRENT* values are shown in every case whereas the *HOURLY* and *DAILY* fields have to be selected as described in the function details table as well as the *xMax*, *xMin*, and *xAvg* fields.

```
[
    {
        "Latency": {
            "CURRENT": {
                "ratedValue": 1,
                "absoluteValue": 0.00036
            },
            "HOURLY": {
                "absoluteAvg": 0.0008980000000000003,
                "ratedMax": 1,
                "ratedAvg": 1,
                "absoluteMin": 0.000275,
                "absoluteMax": 0.003131,
                "ratedMin": 1
            },
            "DAILY": {
                "absoluteAvg": 0.0010711562499999998,
                "ratedMax": 1,
                "ratedAvg": 1,
                "absoluteMin": 0.000265,
                "absoluteMax": 0.086072,
                "ratedMin": 1
            },
            "unit": "http://purl.oclc.org/NET/muo/ucum/unit/time/second"
        },
        …,
        "uuid": "e5de76bc-e1db-5a68-9ff9-1deff0c41248"
    }
]
```


Listing 6: Last Qualities Answer 2

As an alternative to the API access an average QoI graph is stored every hour into the triplestore. Listing 7 shows an N3 formatted graph for one data stream, which is saved to the store. The average graph is named http://ict-citypulse.eu/store/citypulse/avgquality. As the data is saved in triples it is accessible via SPARQL. Beside this, the information contained in the graph is the same as accessed via API but it`s not configurable and provides only the average quality for the last hour. For easy usage with the Event Detection used in work package 5 of the CityPulse framework each average quality class is connected to a *ces:EventProfile*.

```
@prefix ces: <http://www.insight-centre.org/ces#> .
@prefix ct: <http://ict-citypulse.eu/city#> .
@prefix qoi: <http://purl.oclc.org/NET/UASO/qoi#> .

<http://ict-citypulse.eu/EventProfile-2586bf63-d256-59a0-bc29-9d04eb92dacb> a
ces:EventProfile ;
    qoi:hasQuality ct:hourly-avgAge-2586bf63-d256-59a0-bc29-9d04eb92dacb,
        ct:hourly-avgCompleteness-2586bf63-d256-59a0-bc29-9d04eb92dacb,
        ct:hourly-avgCorrectness-2586bf63-d256-59a0-bc29-9d04eb92dacb,
        ct:hourly-avgFrequency-2586bf63-d256-59a0-bc29-9d04eb92dacb,
        ct:hourly-avgLatency-2586bf63-d256-59a0-bc29-9d04eb92dacb .

ct:hourly-avgAge-2586bf63-d256-59a0-bc29-9d04eb92dacb a qoi:Age ;
    qoi:hasAbsoluteQuality "53.200" ;
    qoi:hasRatedQuality "1.000" ;
    qoi:hasUnitOfMeasurement <http://purl.oclc.org/NET/muo/ucum/unit/time/second> .

ct:hourly-avgCompleteness-2586bf63-d256-59a0-bc29-9d04eb92dacb a qoi:Completeness ;
    qoi:hasAbsoluteQuality "2.000" ;
    qoi:hasRatedQuality "1.000" ;
    qoi:hasUnitOfMeasurement <http://purl.oclc.org/NET/muo/ucum/physical-quality/number> .

ct:hourly-avgCorrectness-2586bf63-d256-59a0-bc29-9d04eb92dacb a qoi:Correctness ;
    qoi:hasAbsoluteQuality "1.000" ;
    qoi:hasRatedQuality "1.000" ;
    qoi:hasUnitOfMeasurement <http://purl.oclc.org/NET/muo/ucum/unit/fraction/percent> .

ct:hourly-avgFrequency-2586bf63-d256-59a0-bc29-9d04eb92dacb a qoi:Frequency ;
    qoi:hasAbsoluteQuality "0.017" ;
    qoi:hasRatedQuality "0.625" ;
    qoi:hasUnitOfMeasurement <http://purl.oclc.org/NET/muo/ucum/unit/frequency/Herz> .

ct:hourly-avgLatency-2586bf63-d256-59a0-bc29-9d04eb92dacb a qoi:Latency ;
    qoi:hasAbsoluteQuality "0.001" ;
    qoi:hasRatedQuality "1.000" ;
    qoi:hasUnitOfMeasurement <http://purl.oclc.org/NET/muo/ucum/unit/time/second> .
```

Listing 7: Average Quality N3 Graph

5.1.2 Composite Monitoring

The Composite Monitoring is triggered by events from the Event Detection, by the Atomic Monitoring (in case of QoI metric drops) or for a manual event or stream evaluation by the visual Quality Explorer interface. In addition to the Atomic Monitoring the Composite Monitoring utilises historic time series of various dependent sensor streams. Therefore, a direct access to sensor information in the Resource Management is necessary.

Figure 32 shows a sequence diagram for the plausibility evaluation of a stream. The Composite Monitoring receives a request to check the correctness of a sensor stream for a specific timespan. Based on the propagation and distance model of Section 3.1.4 a set of relevant sensors is identified to acquire information for comparison. Therefore, depending on the sensor's nature, distance models e.g., shortest path distance on the roads, Euclidean distance for radial propagation or infrastructure based metrics (e.g., change of road types) are used in the method *analyseInfrastructure* of the Geospatial Data Infrastructure (see Table 6). The distances between the selected streams and the *streamIDs* are returned to the Composite Monitoring component. In the next step the historic sensor datasets for those streams are fetched from the Resource Management.

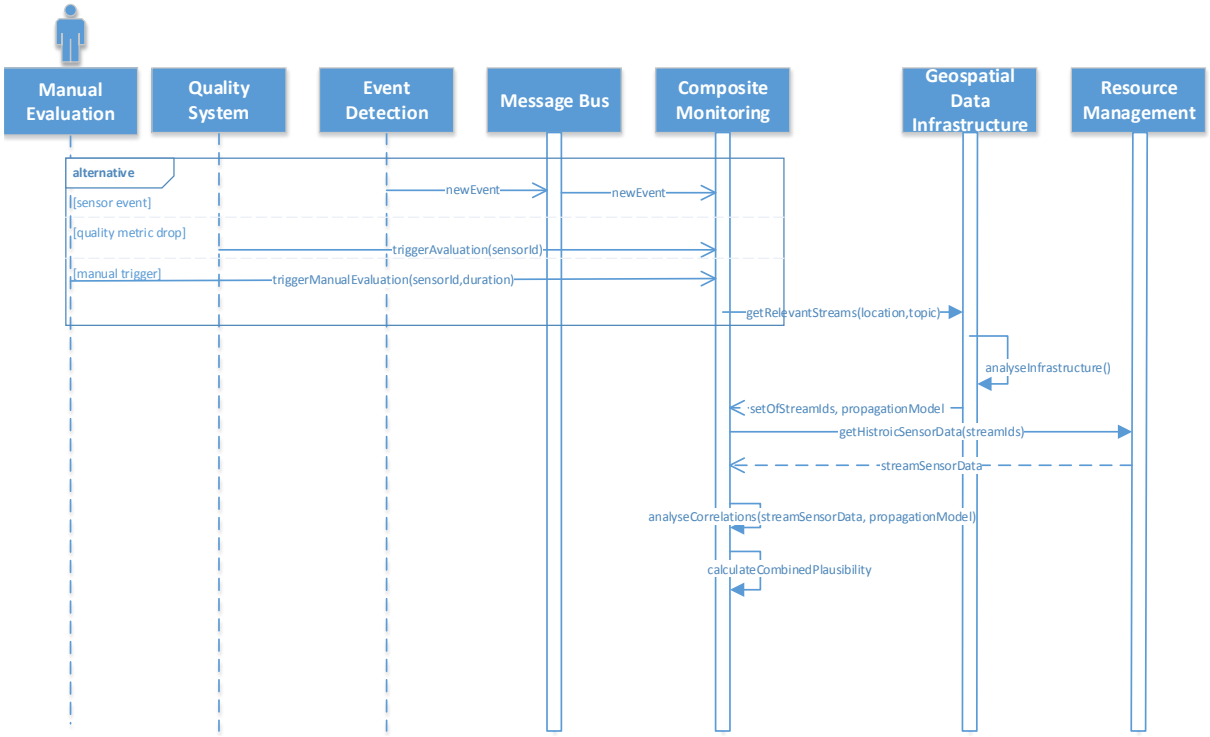

Figure 32: Sequence Diagram of the Composite Monitoring Process

Metric Parameter	Processing
euclidean_distance	Direct Euclidean distance between two sensors in meters regarding earth curvature.
route_distance	Route distance between two sensors in meters .
route_duration	Travel time by car of the route between two sensors.
route_steps	Number of steps (road crossover, turns, etc.) of the route between two sensors.
directed_duration	Additive composition of the duration and the angle between two sensors.
sensorHops	Number of sensors to jump over to reach from one sensor to another. Equal to number of sensors if it is not possible to reach another sensor.

Table 6: Implemented Distance Metrics in the Geospatial Data Infrastructure Module

The method *analyseCorrelations(streamSensorData, propagationModel)* offers a set of pre-processing options for the data series (see Table 7).

Identifier	Modification of data
raw	Unmodified input data. Limited significance due to seasonal time series patterns.
dct	Data with high frequency filter discrete cosine transform: DCT (data)
random	Data minus seasonal amount over given period: data – seasonal (data)
random_dct	Dct filtered random data: DCT (data – seasonal (data))
dct_random	Random of dct filtered data: DCT (data) – seasonal (DCT (data))
dct_random_dct	Dct filtered random of dct filtered data: DCT (dct_random)
offset_ ...	The time series of the second stream is shifted according to the estimated duration that is calculated with the propagation model.

Table 7: Time Series Pre-Processing Options

5.2 Conflict Resolution and Fault Recovery

5.2.1 Conflict Resolution

The Conflict Resolution component is implemented as a Java program and is hosted as a Websocket server. The code of the Conflict Resolution component is integrated within the Data Federation component as part of the Automated Complex Event Implementation System (ACEIS) middleware. Figure 33 illustrates the architecture of the Conflict Resolution.

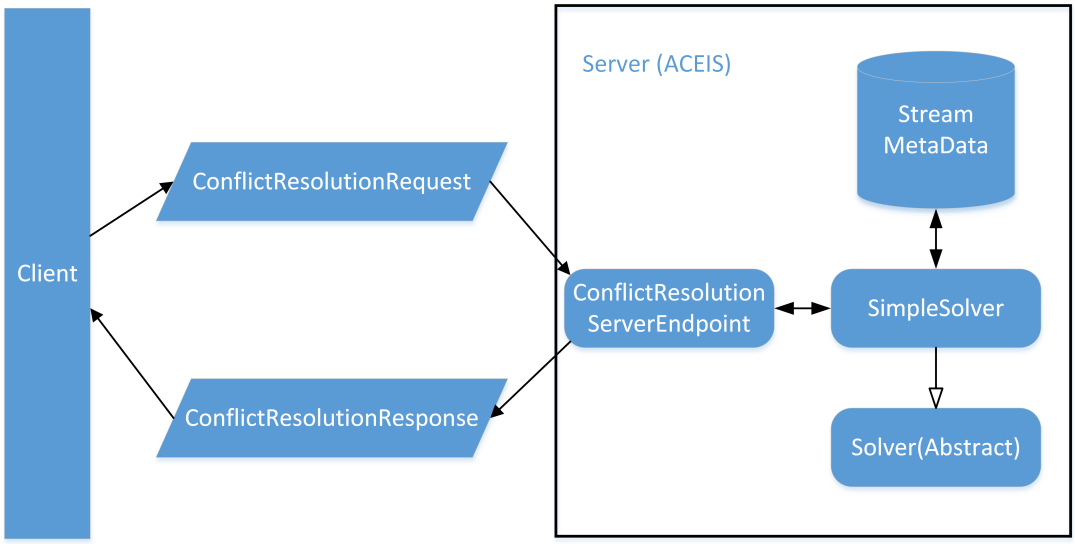

Figure 33: Architecture of Conflict Resolution Component

The Conflict Resolution requests and responses are generated by a client program, (e.g. the user interface) and formatted as JSON strings. The request contains a list of stream ids and the response is the stream id that is considered to be the most accurate/believable. The request is handled by the *ConflictResolutionServerEndpoint*, which instantiates a *SimpleSolver* to solve the conflicts. The *SimpleSolver* extends the abstract class *Solver*, and implements the *solve()* method in order to find out the most trustworthy stream. The *SimpleSolver* relies on the stream metadata cached by the ACEIS engine to retrieve the latest quality values for the data streams.

5.2.2 Fault Recovery

The Atomic Monitoring and the Fault Recovery are both integrated into the Resource Management and tightly bounded together. To clarify the interaction of both components there are two different use cases described in the form of sequence diagrams. The first diagram depicts the processing of incoming data and is the successor of Figure 30 in Section 3.1.2. The second diagram illustrates the case when no data is received and the entire dataset has to be recovered.

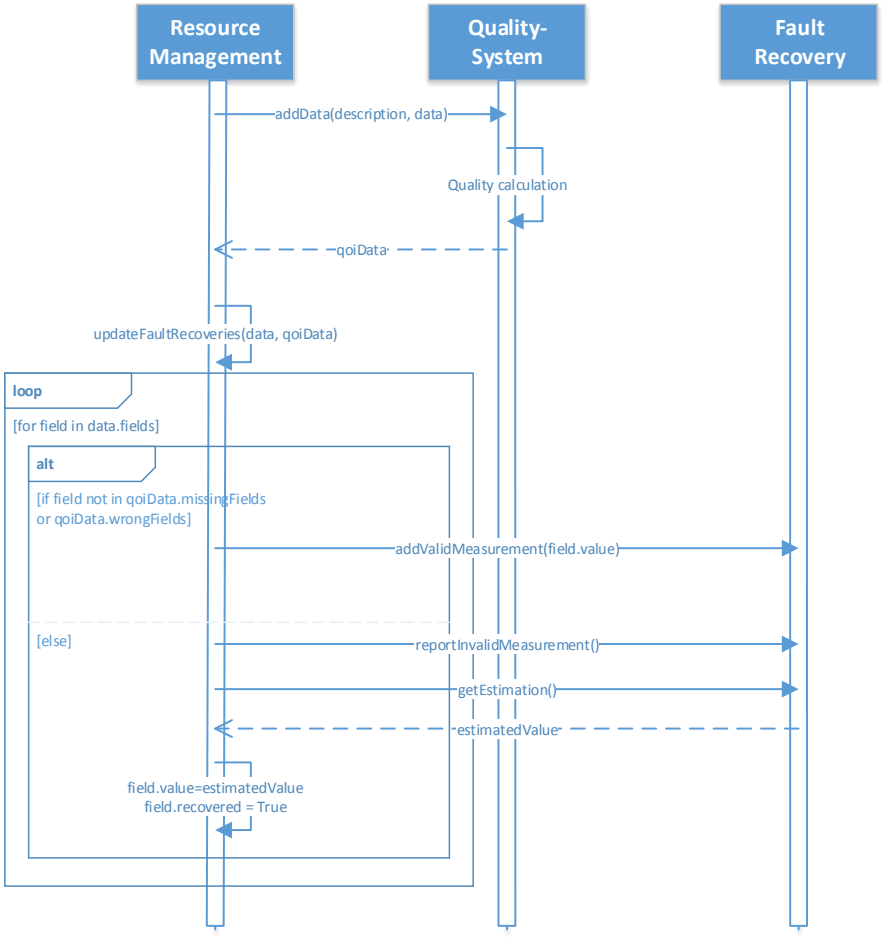

Figure 34: Sequence Diagram Recovery of Missing/Wrong Fields

The Resource Management acts as a handler for incoming data. As shown in Section 3.1.2 the incoming observations are delivered to the Quality-System to calculate the QoI. Therefore, in Figure 34 this quality check is abbreviated, as it is identical. The *qoiData* as the result of the quality check is then used by the Resource Management to update the Fault Recoveries for the data stream providing the current data. Each field in the received data (*data.fields*) is checked to be present in *missingFields* or *wrongFields* of *qoiData*. Its presence indicates that the field is missing (Completeness) or that this field is wrong (Correctness). In both cases the Resource Management tries to add or replace the value by using the Fault Recovery. In a first step the Fault Recovery is notified about an invalid measurement, to trigger an update process. In a second step an estimation is requested for the missing or wrong value. With the received estimated value, which is substituting the original value, the data field is marked as having a recovered value.

If the field is not missing or wrong its value is added to the Fault Recovery by *addValidMeasurement* method to indicate that it can be used to extend the training dataset.

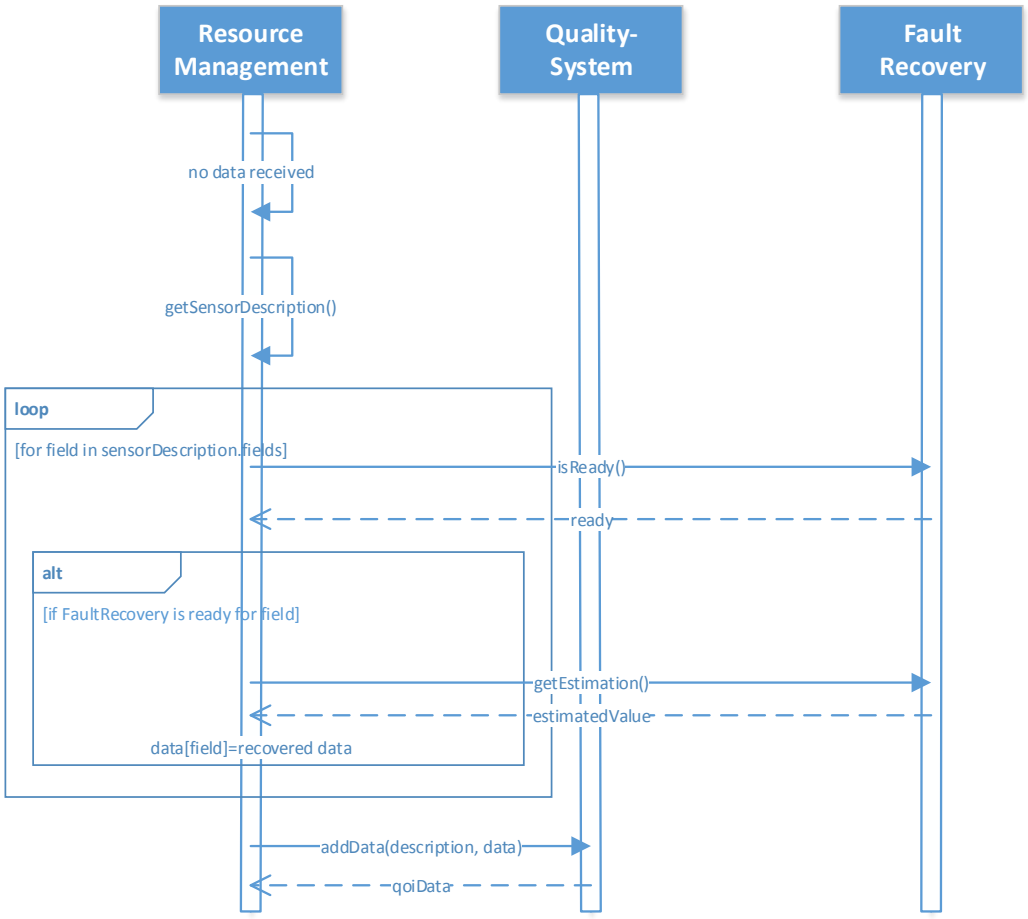

Figure 35: Sequence Diagram Recovery of Missing Observations

The second use case is shown in Figure 35. Within this use case the Resource Management receives no data for a data stream. Under these circumstances the Resource Management rebuilds an empty dataset with the help of the sensor description. As every data field, which should be contained in the dataset is described in the sensor description the Fault Recovery is used to recover values for it. This can only be done if the Fault Recovery is ready (i.e. has received enough valid measurements as training data). If it is ready, the Fault Recovery is requested to estimate the missing values. After filling up the empty dataset with the recovered values it is checked for its quality by the Quality-System.

6. Experiments/Performance Test

Atomic Monitoring

The Atomic Monitoring is directly integrated into the Resource Management as described in Section 5.2.2. The Quality Explorer allows users to see the current QoI level for all sensors. In the following section the quality values for a whole month (December 2015) are depicted to get an impression about the stream behaviour for a longer period. Here, the quality values of the Atomic Monitoring metrics in the Quality-System are exported into CSV files and processed with an R script in order to visualise the quality distribution. This is done for a parking and a traffic data stream for the city of Aarhus.

Parking

The first data stream within this experiment is the parking data[2]. It contains information about the capacity of a garage and the number of occupied spaces for 13 parking garages in Aarhus. New data is requested by the Resource Management every minute as the stream is annotated with an *updateInterval* of 60 seconds.

```
1   sensordescription.information = "Parking data for the City of Aarhus"
2   …
3   sensordescription.maxLatency = 2
4   sensordescription.updateInterval = 60
5   sensordescription.fields = ["vehicleCount", "totalSpaces"]
6   …
7   sensordescription.field.totalSpaces.propertyName = "ParkingCapacity"
8   sensordescription.field.totalSpaces.min = 0
9   sensordescription.field.totalSpaces.max = 10000
10  sensordescription.field.totalSpaces.dataType = "int"
11  …
12  sensordescription.field.vehicleCount.propertyName = "ParkingVehicleCount"
13  sensordescription.field.vehicleCount.min = 0
14  sensordescription.field.vehicleCount.max = "@totalSpaces"
15  sensordescription.field.vehicleCount.dataType = "int"
```

Listing 8: Parking Stream Description

The quality related part of the stream description is shown in Listing 8. The stream contains two fields, which are checked for their QoI. The values of the vehicle count within the parking garage are strongly related with the parking capacity. This is shown by the *@totalSpaces*, which indicates that there cannot be more cars in the garage than parking slots available.

[2] http://www.odaa.dk/dataset/parkeringshuse-i-aarhus

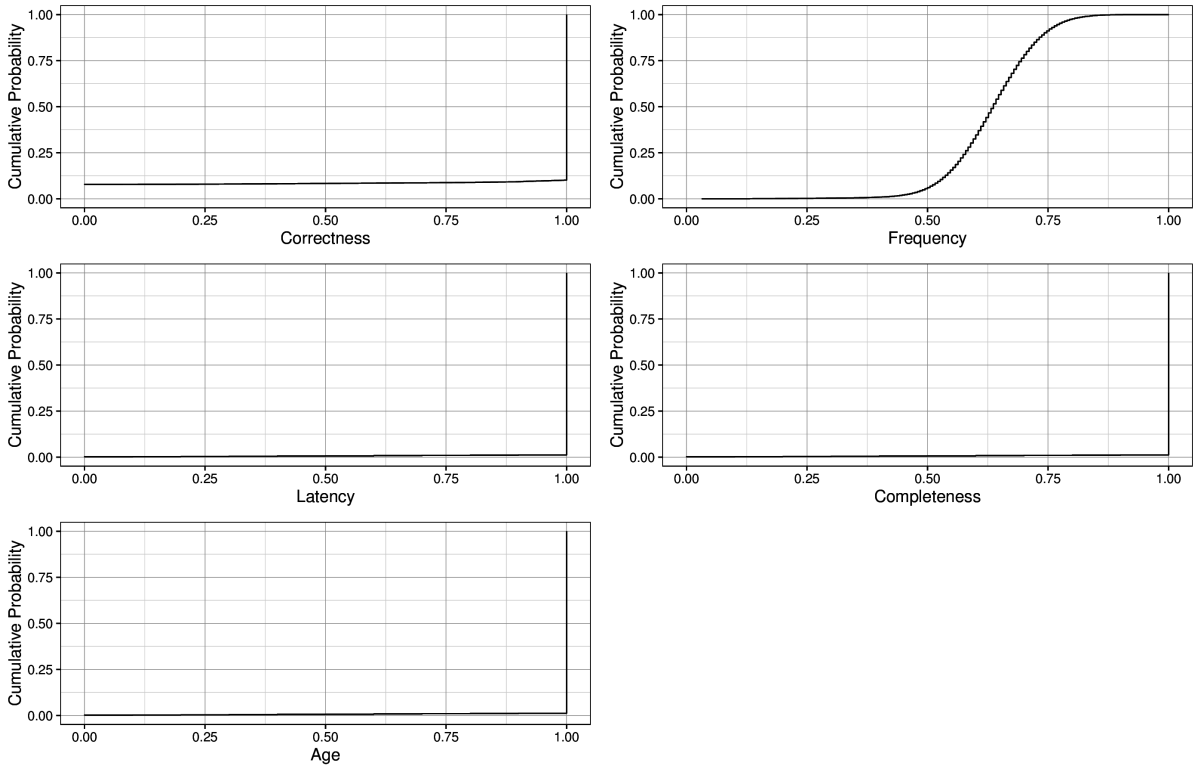

Figure 36: Parking Stream CDF

Figure 36 depicts the distribution of quality values cumulated for all parking garages. Whereas most of the Quality metrics show an absolute quality level of nearly 100% (1.0), there is a noticeable behaviour for the Frequency metric. This behaviour is also shown, when plotting every single data stream individually instead of creating a cumulated plot. It can be determined that the update interval for the sensors could not be met for nearly half of the time after inspecting the QoI values used to create the graphs. This behaviour could be explained by two possible reasons: the first one is slow connection to the sensor requiring too much time until an answer is delivered. The second one, which is probably the reason for the bad Frequency QoI is the replay of historic data. The historic data is not exactly within in the same interval as expected due to the sensor description's *updateInterval*, which causes the low Frequency metric values.

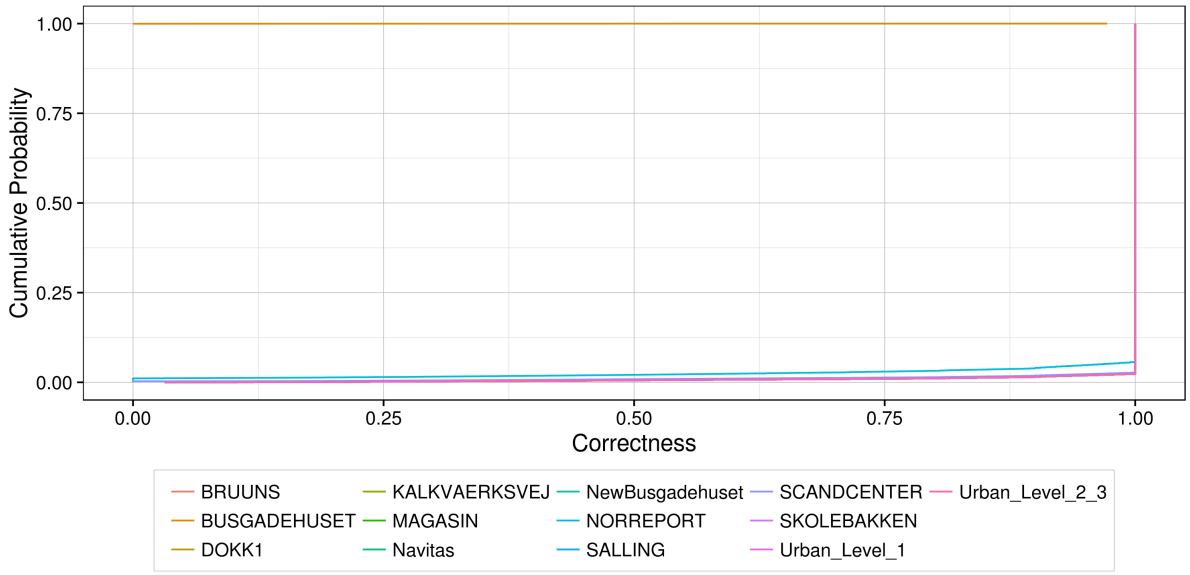

Figure 37: Parking Stream Correctness CDF

A better example for the Atomic Monitoring is the detailed view of the Correctness metric shown in Figure 37. One parking garage (Busgadehuset) has a QoI level of 0.0 for the whole time. The reason for this is the fact that the stream reports the number of occupied parking slots for this garage is larger than the number of available parking slots in this garage. For a second garage stream (Norreport) a slightly noticeable Correctness issue exists. In this case there are several observations with more cars reported to be in the garage than places available.

Traffic

The second data stream within the experiments contains data for 449 traffic sensors in the city of Aarhus. This stream is updated every 5 minutes via a pull request from the Resource Management to a CKAN open data platform[3].

```
1   sensordescription.information = "Road traffic for the City of Aarhus"
2   sensordescription.maxLatency = 2
3   sensordescription.updateInterval = 60 * 5
4   sensordescription.fields = ["TIMESTAMP", "avgSpeed", "vehicleCount",
    "avgMeasuredTime"]
5   …
6   sensordescription.field.avgSpeed.propertyName = "AverageSpeed"
7   sensordescription.field.avgSpeed.min = 0
8   sensordescription.field.avgSpeed.max = 250
9   sensordescription.field.avgSpeed.dataType = "float"
10  …
11  sensordescription.field.vehicleCount.propertyName = "StreetVehicleCount"
12  sensordescription.field.vehicleCount.min = 0
13  sensordescription.field.vehicleCount.max = 1000
14  sensordescription.field.vehicleCount.dataType = "int"
15  …
16  sensordescription.field.avgMeasuredTime.propertyName = "MeasuredTime"
17  sensordescription.field.avgMeasuredTime.min = 0
```

[3] http://www.odaa.dk/dataset/realtids-trafikdata

```
18  sensordescription.field.avgMeasuredTime.max = 1000
19  sensordescription.field.avgMeasuredTime.dataType = "int"
20  …
21  sensordescription.field.TIMESTAMP.propertyName = "MeasuredTime"
22  sensordescription.field.TIMESTAMP.min = "2012-01-01T00:00:00"
23  sensordescription.field.TIMESTAMP.max = "2099-12-31T23:59:59"
24  sensordescription.field.TIMESTAMP.dataType = "datetime"
25  sensordescription.field.TIMESTAMP.format = "%Y-%m-%dT%H:%M:%S"
```

Listing 9: Traffic Stream Description

Listing 9 provides the quality related part of the traffic stream description. This stream contains four different fields, which are checked for their QoI.

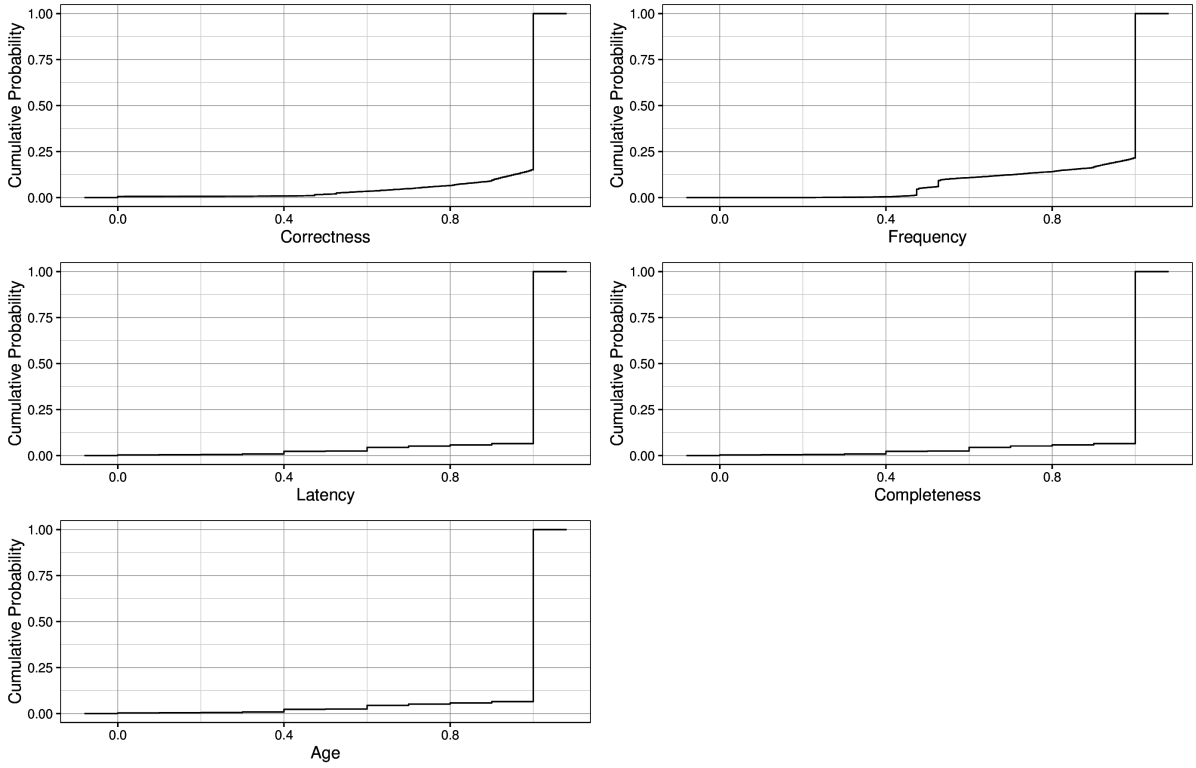

Figure 38: Traffic Stream CDF

Figure 38 depicts the cumulated QoI values for five different metrics. It is noticeable that there is no metric, which is fulfilled perfectly. Each of the metrics starts to decrease at a value of 0.4 for the QoI value. The Correctness and Frequency are slightly worse than the other metrics and are inspected in detail below.

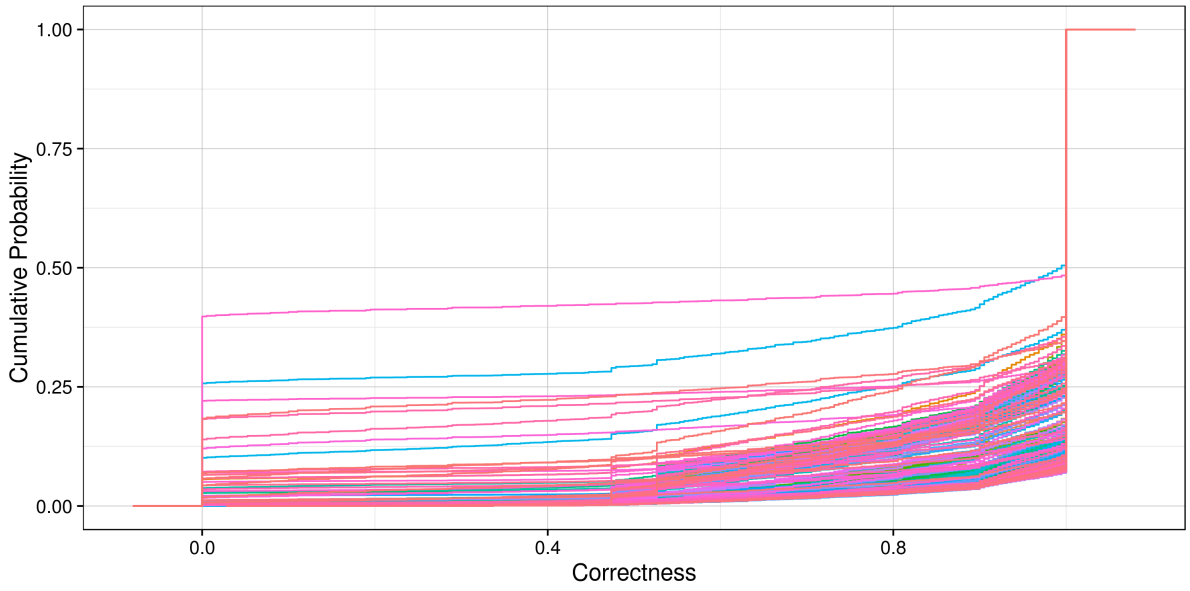

Figure 39: Traffic Stream Correctness CDF

Figure 39 depicts a detailed CDF for every single traffic sensor showing that most of the sensors have a relatively high Correctness level. This indicates that these sensors are delivering faulty data most of the time.

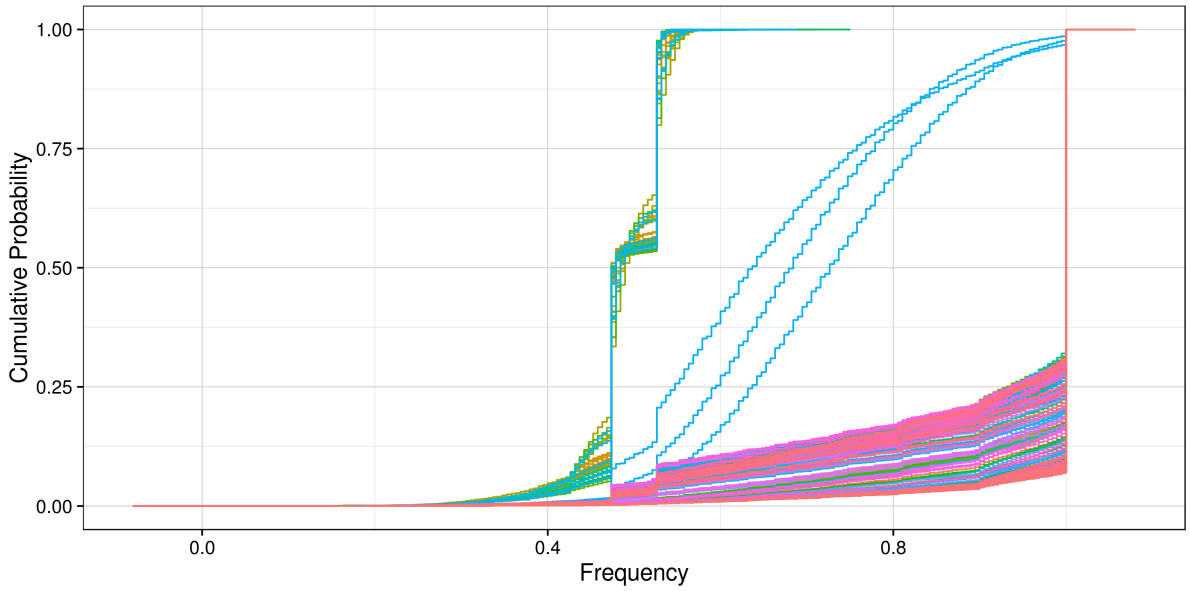

Figure 40: Traffic Stream Frequency CDF

Another interesting behaviour is shown in Figure 40, which contains the Frequency metric for the traffic data stream for every individual sensor. As it can be seen there is a high number of sensors, which are never reaching a 100% correct Frequency value (1.0). This is interesting as most of the sensors are much better and all of them are contained within the same data stream. With getting

the corresponding sensor ids a map of these sensors is plotted in Figure 41. It shows the maximum reached Frequency within the investigated month.

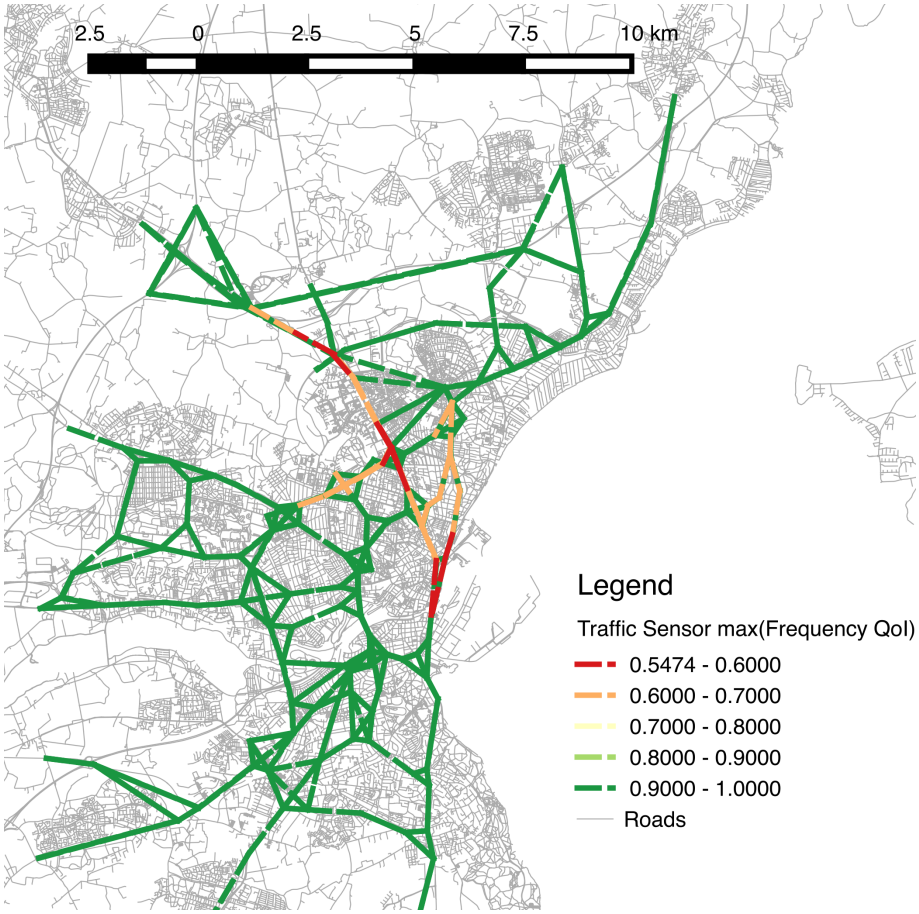

Figure 41: Traffic Stream Maximum Frequency Map

Especially the sensors within the harbour area of Aarhus and on one highway out of the city cannot provide updates frequently. This could indicate a possible failure within the infrastructure and should be inspected by the stream provider for the traffic data.

Quailty Explorer

While developing the Atomic Monitoring component it was hard to see the current QoI results for different data streams. Just checking the quality via the implemented APIs and with the usage of SPARQL and the triplestore is not very intuitive. To support the development process and to help stream providers and developers to identify, which data streams are not working within their annotated parameters, a tool called Quality Explorer has been developed. This tool is able to show the quality of all annotated data streams on a map. The usage of different colours for different quality levels enables developers and stream providers to simply check the QoI of a data stream.

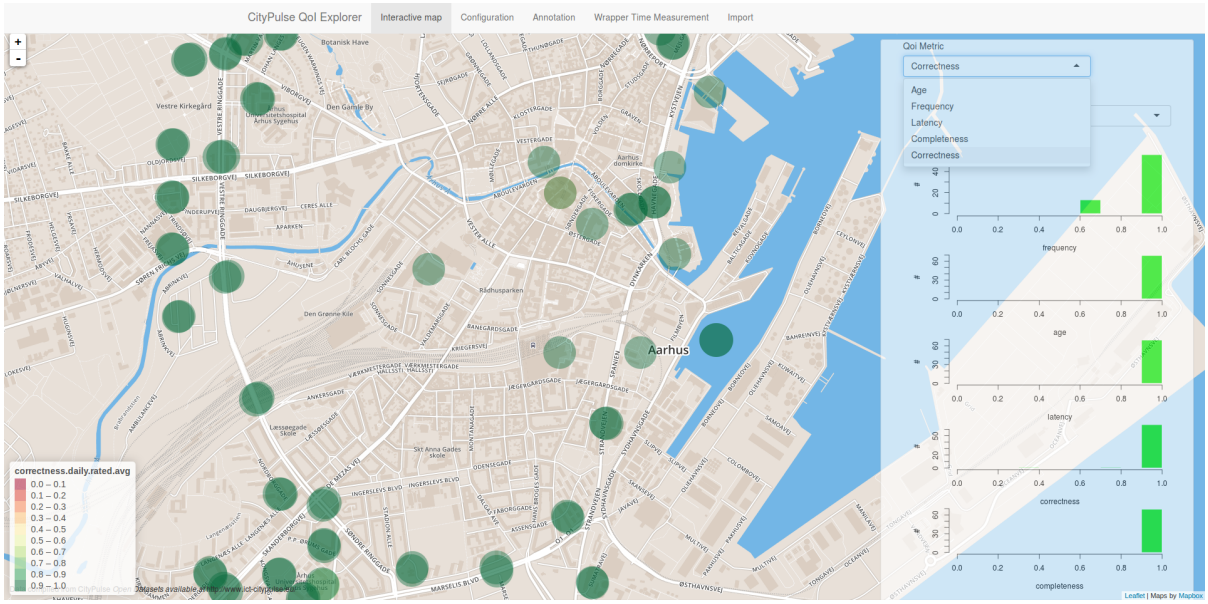

Figure 42: Quality Explorer

The tool itself is mainly a GUI for the API functions described in Section 5.1. It is possible to select the QoI metric, which should be displayed on the map. In addition, the aggregation interval can be set to the options supported by the API (normally *CURRENT*, *HOURLY*, and *DAILY*) as well as the average, minimum, or maximum values can be selected and shown on the map. To give an extra overview of the QoI values a histogram for each QoI metric is depicted showing the distribution of QoI levels.

Processing Time Measurement

While developing the QoI components and integrating them into the Resource Management an API to measure the time consumptions has been developed to ensure a fast processing time of the different modules and functions. The implemented statistics module has then been integrated into the Resource Management and provides the processing times for the different components. At the moment it is able to measure times for the URL connection to get the observations of a data stream, the Fault Recovery update, the Fault Recovery recovery mechanisms, the QoI calculation as well as the Resource Management coordinating each step.

The processing times of the different data sources of a data stream are aggregated and plotted as a layered pie diagram for a whole data stream. Additionally, a plot for a single sensor within a data stream is possible. With this diagrams and a histogram for the different measured functions, which are implemented into the different components, it is possible to detect processing speed limitations. This supports developers of the framework to fulfil the real-time requirements.

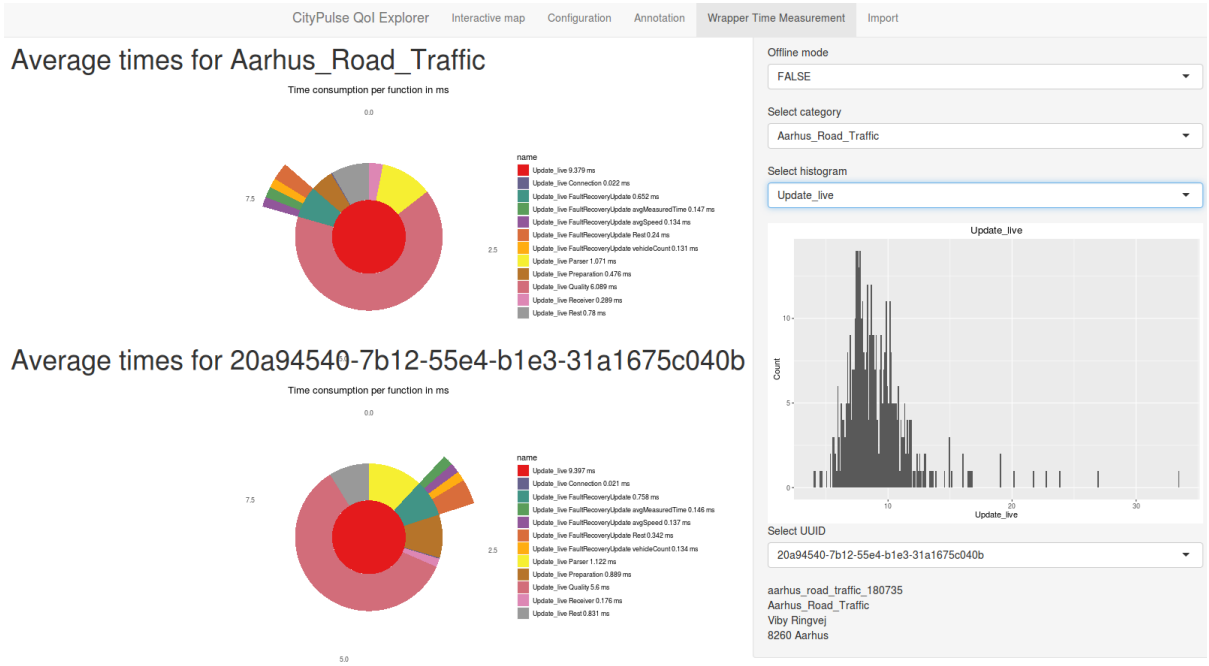

Figure 43: Average Processing Times

The Processing Time Measurement GUI is developed as an R Shiny[4] application and integrated as a separate tab into the Quality Explorer as shown in Figure 43. Besides the server address and the port where the Resource Management is running on, no additional information has to be provided.

[4] http://shiny.rstudio.com

7. Conclusion and Outlook

In this deliverable we discussed the measures of ensuring or increasing the reliability of smart city applications executed in the CityPulse framework. In total 4 different measures are described in this document, which include:

- Testing
- Monitoring
- Fault Recovery
- Conflict Resolution

Testing deals with the stability of smart city applications with a decreasing quality of sensory observations from deployed sensors. Initially, we describe the effects of faulty sensors on the output signal of measurement equipment. This information is then used to define an iterative test case generation process, where each successive test case stimulates the CityPulse framework with less reliable data streams (lower quality) than its predecessor. The process is repeated until the framework's output differs sufficiently from the original output (test run using original historic data) in order to make a statement about the quality threshold required by the application under test.

In contrast to Testing, the second measure Monitoring observes the QoI of incoming sensor observations during the run-time of the CityPulse framework. For scalability reasons and to be able to meet real-time requirements the monitoring is divided into a two-stage approach. The first stage, Atomic Monitoring, is responsible for single stream data quality metrics, such as Completeness, Age and Frequency. To achieve best possible performance, the Atomic Monitoring was integrated directly into the data streams Data Wrappers, a modular concept acting as entry point for external data resources into the CityPulse framework. The second stage, the Composite Monitoring, triggered only by detected events, rapidly dropping QoI values or through manual interaction, provides a multi information source plausibility evaluation scheme. For this correlating information sources within a spatio-temporal distance around the event are identified. The work here highlights the importance of a fitting distance model describing to propagation properties of a physical phenomenon along with an appropriate correlation method.

The third measure, the Fault Recovery acts as a fall-back solution in case of missing or low quality sensor observations. Here, the faulty observations are replaced by estimated values gathered from a k-NN algorithm. Compared to the previous Deliverable D4.1 a more efficient learning mechanism is introduced. As for the Atomic Monitoring, the Fault Recovery was integrated directly into the Data Wrappers for better performance.

As a last measure, Conflict Resolution is responsible to select similar but more reliable data streams in case the Fault Recovery cannot compensate faulty or missing sensor observations anymore. Using aggregated quality metrics over a period of time the Conflict Resolution is able to select the best data stream across multiple quality dimensions. A transparent adaptation allows smart city applications to seamlessly switch to a different data stream and continue its execution.

In conclusion, CityPulse not only acknowledges the fact that data collected in a smart city may be error-prone and incorrect for smart city applications, CityPulse expects that the data sources become unreliable over time. As a counter measure several actions to identify and react on varying information qualities have been investigated and integrated into the CityPulse framework. In the future we plan to investigate more closely the Composite Monitoring approach and to apply the concept to different domains required by the use cases of activity 6.2, such as environment or noise pollution.

8. References

Aha 1991 — Aha, D. W., Kibler, D., & Albert, M. K. (1991). Instance-based learning algorithms. Machine learning, 6(1), 37-66.

Andonie 2003 — Andonie, R., Sasu, L., Beiu, V "Fuzzy ARTMAP with relevance factor", Proceedings of the International Joint Conference on Neural Networks, 2003, pp 1975 – 1980.

Andrea 2007 — Andrea, J. (2007) Envisioning the Next-Generation of Functional Testing Tools. IEEE Software Journal, Volume 24, pp.58-66, 2007.

Charef 2000 — A. Charef, A. Ghauch, P. Baussand, and M. Martin-Bouyer, "Water quality monitoring using a smart sensing system", Measurement, vol. 28, pp.219 -224 2000

CityPulse-D3.2 — Gao, F. et al. (2014, August). CityPulse D3.2 - Data Federation and Aggregation in Large-Scale Urban Data Streams.

CityPulse-D3.4 — Farajidavar, N. et al. (2016). CityPulse D3.4 – Social Media Analysis (working title). **

CityPulse-D4.1 — Kuemper, D. et al. CityPulse D4.1 – Measures and Methods for Reliable Information Processing, February 2015.

CityPulse-D5.3 — Piui, D. et al. CityPulse D5.3 – Smart City Environment User Interfaces, 2016, February. **

CityPulse-D2.3 — Reference Dataset Website : http://iot.ee.surrey.ac.uk:8080, last visited 22.02.2016.

Engelbrecht 2001 — Engelbrecht A.P. and Brits R., "A clustering approach to incremental learning for feedforward neural networks," Proc. of Int. Joint Conf. on Neural Networks, vol. 3, pp. 2019-2024,2001.

Fischer 2012 — M.Fischer, R.Tönjes: "Generating Test Data for Black-Box Testing using Genetic Algorithms", 7th IEEE International Workshop on Service Oriented Architectures in Converging Networked Environments (SOCNE 2012), Krakau, Poland, Sep. 2012.

Godefroid 2008 — Godefroid, P., Levin, M. Y., Molnar, D., Automated whitebox fuzz testing, Proceedings of Network and Distributed Systems Security (NDSS), San Diego, USA, 2008.

Grippo 2000 — Grippo, L. "Convergent on-line algorithms for supervised learning in neural networks," IEEE Trans. on Neural Networks, vol. 1 1, no. 6, pp. 1284-1299, 2000.

Jaeger 2014 — Jäger, Georg, et al. Assessing neural networks for sensor fault detection. In: Computational Intelligence and Virtual Environments for Measurement Systems and Applications (CIVEMSA), 2014 IEEE International Conference on. IEEE, 2014. S. 70-75.

Kolozali 2016 — Kolozali, Sefki et al. Observing the Pulse of a City: A Smart City Framework for Realtime Discovery, Federation, and Aggregation of Data Streams. 2016. *

Li 2014 — L. Li, J. Gong, and J. Zhou, "Spatial interpolation of fine particulate matter concentrations using the shortest wind-field path distance," 2014.

Merayo 2012 — Merayo M. G., "Passive Testing of Timed Systems with Timeouts," in 2012 12th International Conference on Quality Software (QSIC), 2012, pp. 69 –78.

Mesnard 2013 — L. De Mesnard, "Pollution models and inverse distance weighting: Some critical remarks," Computers & Geosciences, vol. 52, pp. 459–469, 2013.

NHB2 2010	NASA. NASA Measurement Quality Assurance Handbook – ANNEX 2. In: Measuring and Test Equipment Specifications, 2010.
Polikar 2001	Robi Polikar, Jeff Byorick, Stefan Krause, Anthony Marino, Michael Moreton „Learn++: A Classifier Independent Incremental Learning Algorithm for Supervised Neural Networks" IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews, 2001, Volume:31, Issue: 4, pages 497 – 508.
Puiu 2016	Puiu, Dan, et al. CityPulse: Large Scale Data Analytics Framework for Smart Cities. In: IEEE Access. 2016. *
Quality Ontology	http://purl.oclc.org/NET/UASO/qoi
Ramanathan 2006	Ramanathan, Nithya, et al. Rapid deployment with confidence: Calibration and fault detection in environmental sensor networks. Center for Embedded Network Sensing, 2006.
SAO	http://iot.ee.surrey.ac.uk/citypulse/ontologies/sao/sao
SAOPY	http://iot.ee.surrey.ac.uk/citypulse/ontologies/sao/saopy.html
Sharma 2010	Sharma, Abhishek B.; Golubchik, Leana; Govindan, Ramesh. Sensor faults: Detection methods and prevalence in real-world datasets. ACM Transactions on Sensor Networks (TOSN), 2010, 6. Jg., Nr. 3, S. 23.
Shilton, 2001	Shilton, A., Palaniswami, M., Ralph, D. and Tsoi, A. C. Incremental training of support vector machines, Proc. of Int. Joint Conf. on Neural Networks, v01.2, pp. 1197-l202, 2001.
Tolle 2005	Tolle, Gilman, et al. A macroscope in the redwoods. In: Proceedings of the 3rd international conference on Embedded networked sensor systems. ACM, 2005. S. 51-63.
Werner-Allen 2006	Werner-Allen, Geoff, et al. Fidelity and yield in a volcano monitoring sensor network. In: Proceedings of the 7th symposium on Operating systems design and implementation. USENIX Association, 2006. S. 381-396.
Zimmermann 1999	D. Zimmerman, C. Pavlik, A. Ruggles, and M. P. Armstrong, "An experimental comparison of ordinary and universal kriging and inverse distance weighting," Mathematical Geology, vol. 31, no. 4, pp. 375–390, 1999.

*under review for publication

**in progress