EQL-CE: An Event Query Language for Connected Environment Management

Elio Mansour, Richard Chbeir, Philippe Arnould

EQL-CE: An Event Query Language for Connected Environment Management

Elio Mansour
Univ Pau & Pays Adour
E2S UPPA, LIUPPA
Mont-de-Marsan, 40000, France
elio.mansour@univ-pau.fr

Richard Chbeir
Univ Pau & Pays Adour
E2S UPPA, LIUPPA
Anglet, 64600, France
richard.chbeir@univ-pau.fr

Philippe Arnould
Univ Pau & Pays Adour
E2S UPPA, LIUPPA
Mont-de-Marsan, 40000, France
philippe.arnould@univ-pau.fr

ABSTRACT

Recent technological advances have fueled the rise of connected environments (e.g., smart buildings and cities). Event Query Languages (EQL) have been used to define (and later detect) events in these environments. However, existing languages are limited to the definition of event patterns. They share the following limitations: (i) lack of consideration of the environment, sensor network, and application domain in their queries; (ii) lack of provided query types for the definition/handling of components/component instances; (iii) lack of considered data and datatypes (e.g., scalar, multimedia) needed for the definition of specific events; and (iv) difficulty in coping with the dynamicity of the environments. To address the aforementioned limitations, we propose here an EQL specifically designed for connected environments, denoted EQL-CE. We describe its framework, detail the used language, syntax, and queries. Finally, we illustrate the usage of EQL-CE in a smart mall example.

CCS CONCEPTS

• **General and reference → General conference proceedings**; • **Information systems → Query representation**; • **Theory of computation → Grammars and context-free languages**; • **Computer systems organization → Sensor networks**.

KEYWORDS

Event Query Language, Internet of Things, Sensor Networks

ACM Reference Format:
Elio Mansour, Richard Chbeir, and Philippe Arnould. 2019. EQL-CE: An Event Query Language for Connected Environment Management. In *15th ACM Symposium on QoS and Security for Wireless and Mobile Networks (Q2SWinet '19), November 25–29, 2019, Miami Beach, FL, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3345837.3355950

1 INTRODUCTION

Recent years have witnessed the rise of smart connected environments where sensor networks are deployed in specific environments and produce valuable data for high level applications (e.g., improving manufacturing processes in smart factories, reducing the energy footprint of smart buildings, helping drivers avoid traffic congestion in smart cities). The aforementioned applications have different objectives. However, in order to achieve their goals, they all need to define and detect specific key events (e.g., machinery faults, traffic congestion, energy wastes). Event Query Languages (EQL) have been proposed in many works [1, 15] as a means for event definition prior to detection. EQL allow users to express how the targeted events are defined (i.e., event describing features, patterns). However, existing works [2, 3, 6–9, 11] suffer from various limitations:

(1) *Lack of considered components.* In addition to events, one might need to handle other components in a connected environment (these works heavily focus on defining events). For instance, managing the infrastructure (e.g., locations, spatial ties), the sensor network (e.g., sensors, observations), and additional descriptions related to the application domain (e.g., industrial, environmental).
(2) *Lack of considered query types.* In addition to defining components, one should be able to modify previously defined components (e.g., altering, renaming, or dropping), and manipulating component instances (e.g., selecting, inserting).
(3) *Lack of considered datatypes.* It is important to integrate the diverse data and datatypes needed for the definition of specific events (e.g., for scalar, multimedia sensor observations).
(4) *Handling environment dynamicity.* In a dynamic environment, mobile sensors can enter/leave the network or change locations at any time. Since events rely on sensors and their observations, event queries need to cope with the changes.

In addition, some works [4, 5] are not re-usable in different setups due to their reliance on a specific data model-based syntax.

In this paper, we propose an Event Query Language specifically designed for Connected Environments. Our proposal, denoted EQL-CE, considers the entire connected environment components. This entails defining the infrastructure, its sensor network, the targeted events, and the application domain. Our proposal provides the common query types needed for the definition of components and the management of their instances. Also, EQL-CE integrates various datatypes, and copes with the dynamicity of the environment by re-writing/updating queries that became obsolete due to sensor mobility (to be detailed in a separate work). Finally, EQL-CE uses EBNF (Extended Backus-Naur Form), a generic and easy to parse language in order to allow re-usability in various contexts.

In the following, we present our motivating scenario in Section 2, and present some background on EBNF in Section 3. Then, we briefly describe the framework of EQL-CE and detail its syntax in Section 4. An illustration example, and the experimental protocol

Q2SWinet '19, November 25–29, 2019, Miami Beach, FL, USA

ACM ISBN 978-1-4503-6906-0/19/11…$15.00
https://doi.org/10.1145/3345837.3355950

are provided in Section 5. We compare related work on EQL in Section 6. Finally, Section 7 concludes the paper and discusses future research directions.

2 MOTIVATING SCENARIO

Consider the following scenario that illustrates a smart mall. Please note that this example does not summarize all needs found in a connected environment/event detection application scenario. It is only used to highlight the main needs and challenges related to this work. Figure 1 details the infrastructure's location map, and individual locations (e.g., shops). The mall is equipped with a sensor network having static/mobile sensors that produce various observations (e.g., temperature, video). A mall manager would like to adopt an existing EQL capable of:

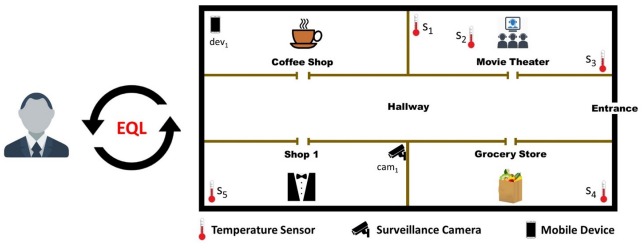

Figure 1: The Smart Mall

- Need 1: Providing common/basic querying functionality. This entails defining connected environment components (e.g., environment, sensor network, events, and application domain). This also includes interrogating the connected environment (e.g., selecting sensors, retrieving sensor observations, detecting events).
- Need 2: Coping with the connected environment changes. For instance, integrating a newly added shop into the mall's location map and detecting events in the new area. Similarly, adding more sensor instances in the mall in order to detect new events or better detect previously defined events (e.g., average values for indoor temperature). Adding components related to the environment/sensor network is needed.
- Need 3: Handling various datatypes. This entails covering scalar and multimedia sensor observations (e.g., textual temperature values, video surveillance footage). The mall manager needs to define different events (e.g., intrusion detection, indoor overheating). To do so, the EQL should be able to manipulate different data and their respective datatypes.
- Need 4: Coping with the dynamicity of the environment. The mall manager relies on clients' mobile phones as mobile sensors in the network. This allows dynamic sensing, and improves coverage without adding many static sensors and increasing the costs. However, mobile sensors change locations and enter/exit the network. Since the defined events rely on sensed data, some event definitions might become obsolete over time. The EQL should be capable of coping with this issue.

Existing EQL mainly focus on event definition, and do not handle other connected environment components. To address needs 1 and 2, one might use another language that integrates different functionality and handles the environment/sensor network changes. However, in this case, the manager will have to use various languages with different syntax. A more appropriate solution might be to extend the capabilities of the EQL to provide a means for defining the structure of various components related to the environment, sensor network, targeted events, and application domain (cf. Need 1-2). The EQL should also handle scalar/multimedia observable properties and sensor observations (cf. Need 3). In addition, the EQL should be capable of re-writing/re-defining obsolete event definitions (i.e., replacing sensors/observations that are missing due to sensor mobility) in order to cope with the dynamicity of the environment (cf. Need 4). Finally, this should be done using a generic, technology independent syntax that could be parsed into various data model-based languages to ensure re-usability. However, when considering all of the above, the following challenges emerge:

- Challenge 1: How to consider various query types to cover all the required functionality?
- Challenge 2: How to consider different components (i.e., environment, sensor network, events, and application domain) in the queries?
- Challenge 3: How to define the structure of both scalar and multimedia data?
- Challenge 4: How to detect obsolete event definitions? How to redefine these events by replacing missing sensors/data?
- Challenge 5: How to establish a generic/re-usable syntax that is independent from the underlying infrastructure?

In the following section, we provide some background on the EBNF syntax used in our proposal EQL-CE.

3 BACKGROUND & PRELIMINARIES

A syntactic metalanguage is useful whenever a clear formal description and definition is required. EBNF is defined by the International Organization for Standardization (ISO 14977[1]). It proposes a notation for defining the syntax of a language using rules. Each rule names part of the language (called a non-terminal symbol) and then defines its possible forms. A terminal symbol is an atom that cannot be split into smaller components of the language. EBNF extends the original BNF to avoid lengthier rules by adding notations for options and repetitions. Furthermore, EBNF includes mechanisms for enhancements, defining the number of repetitions, excluding alternatives, and adding comments. The following resumes the main characteristics of EBNF:

- Terminal symbols of the language are quoted so that any character, including one used in EBNF, can be defined as a terminal symbol of the language being defined
- The [] symbols indicate optional rules/statements
- The {} symbols indicate repetition
- Each rule has an explicit final character so that there is never any ambiguity about where a rule ends
- Brackets group items together. It is an obvious convenience to use the ( ) symbols in their ordinary mathematical sense

Table 1 details the main EBNF notations and their usage.

[1]https://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf

Usage	Notation	Usage	Notation
Definition	=	Grouping	(...)
Concatenation	,	Terminal String	...
Termination	;	Terminal String	' ... '
Alternation	|	Comment	(* ... *)
Option	[...]	Special Sequence	? ... ?
Repetition	{ ... }	Exception	-

Table 1: EBNF Notations

4 EQL-CE PROPOSAL

Here, we describe our EQL-CE [13] framework and briefly discuss its two layers. Then, we focus more on the syntax, and detail the structure of various connected environment components.

4.1 EQL-CE Framework

EQL-CE has two layers: (i) the logical layer allows the construction of generic queries; and (ii) the physical layer parses the queries into a data model-specific language (e.g., SQL, SPARQL) and executes the parsed queries. Figure 2 shows an overview of EQL-CE. In the following, we discuss each layer.

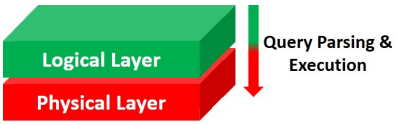

Figure 2: EQL-CE Overview

4.1.1 The Logical Layer. The top layer of EQL-CE, denoted the logical layer, allows users to compose/design their queries. The process starts by choosing a specific query type. To cover various common queries (cf. Criterion 1 - Section 6), we provide the following: (i) the Component Definition Language defines the structure of components. Various query types are included in this group (e.g., CREATE, ALTER, RENAME, DROP); and (ii) the Component Manipulation Language handles component instances. Here we propose the following query types: SELECT, INSERT, UPDATE, and DELETE. Any chosen query can be used with various connected environment components (cf. Criterion 2 - Section 6). Also, one could define the structure of scalar/multimedia data using EQL-CE (cf. Criterion 3 - Section 6). Finally, a query optimizer discovers/re-writes obsolete queries to cope with the dynamicity of the environment (cf. Criterion 4 - Section 6). This module is not detailed here, will leave it for a dedicated separate work.

4.1.2 The Physical Layer. The bottom layer of EQL-CE saves the received queries in a dedicated storage unit for future use. Then, it parses the aforementioned queries into a specific syntax depending on the underlying data model (e.g., SQL, SPARQL). Finally, the parsed query is saved and sent to the query run engine where it is executed. If needed, external functions, methods, or algorithms are called (e.g., for string comparison, mathematical functions).

4.2 EQL-CE Syntax

Here, we detail the syntax that describes the structure of the various connected environment components. To avoid redundancies, we state here that each component has a unique identifier.

4.2.1 Environment Component Syntax. Sensors and sensor networks are hosted on platforms. Each platform has a unique identifier and a type (cf. Syntax 1).

Syntax 1: Defining the structure of a Platform

```
CREATE PLATFORM ( [ ID ] <platform_id>
[ , [ TYPE ] <type> = 'infrastructure' | 'device' ] ) ;
```

We define two types of platforms: (i) infrastructures; and (ii) devices. Infrastructures represent physical, real world environments (e.g., office, building, forest). Each infrastructure has a location map to describe spatial features, and a set of hosted platforms such as other infrastructures or devices (cf. Syntax 2). A location map has a set of distinct locations (cf. Syntax 3), and each location has a description that details its geometric shape, coordinates in space, and a set of spatial relations with other locations (cf. Syntax 4). We also allow users to define external spatial relations (from the application domain) and use them for inter-location ties.

Syntax 2: Defining the structure of an Infrastructure

```
CREATE INFRASTRUCTURE ( [ ID ] <infrastructure_id>
[ , [ LOCATION MAP ] <location_map_id> ] [ , { [ PLATFORM ] <platform_id> }]) ;
```

Syntax 3: Defining the structure of a Location Map

```
CREATE LOCATION MAP ( [ ID ] <location_map_id>
[ , { [ LOCATION ] <location_id> } ] ) ;
```

Syntax 4: Defining the structure of a Location

```
CREATE LOCATION ( [ ID ] <location_id>
[ , [ DESCRIPTION ] <description_id> ]
[ , { ( [ RELATION ] <relation> ,  [ LOCATION ] <location_id> ) } ] ) ;

<relation> = 'contains' | 'covers' | 'crosses' | 'equals' |'includes' |
'isAbove' | 'isBelow' | 'isCloseTo' | 'isDisjointWith' | 'isFraFrom' |
'isLeftOf' | 'isRightOf' | 'overlaps' | 'touches' | <spatial_relation_id> ;
```

Devices (cf. Syntax 5) are also considered platforms since they are capable of hosting sensors. We describe their hardware, software, and provided services in Syntax 6. The descriptions for hardware might include information about the device's power supply, memory, processor, network interface, and expansion cards where sensors are embedded. The description of software might include details about the operating system. And finally, the services descriptions might specify the provided functionality, and input/output.

Syntax 5: Defining the structure of a Device

```
CREATE DEVICE ( [ ID ] <device_id> [ , { [ HARDWARE ] <hw_id> } ]
[ , { [ SOFTWARE ] <sw_id> } ] [ ,  { [ SERVICE ] <service_id> } ] ) ;
```

Syntax 6: Defining the structure of a device's Hardware, Software, and Services

```
CREATE HARDWARE ( [ ID ] <hw_id> [ , [ DESCRIPTION ] <description_id> ] ) ;

CREATE SOFTWARE ( [ ID ] <sw_id> [ , [ DESCRIPTION ] <description_id> ] ) ;

CREATE SERVICE ( [ ID ] <service_id> [ , [ DESCRIPTION ] <description_id> ] ) ;
```

4.2.2 Sensor Network Component Syntax. When considering the sensor network, many components can be defined. For the sake of brevity, we choose here to detail the structure of observable properties in the environment (cf. Syntax 7), sensor observations (cf. Syntax 8), and sensors (cf. Syntax 9). Various properties can be monitored in a connected environment (e.g., temperature, noise, humidity). Some are scalar (i.e., textual) and others multimedia (i.e., audio, video, images). And each property is linked to a set of sensor observations. Each observation, has a description (e.g., unit of measurement), data value (if scalar) or a data object/file (if multimedia) alongside a datatype. Finally, each observation is mapped to a set of metadata tag/value pairs.

Syntax 7: Defining the structure of a Property

```
CREATE PROPERTY ( [ ID ] <property_id>
[ , [ TYPE ] <type> = 'scalar' | 'audio' | 'image' | 'video' ]
[ , { [ [ SCALAR | MEDIA ] OBSERVATION ] <observation_id> } ] ) ;
```

Syntax 8: Defining the structure of an Observation

```
CREATE [ SCALAR | MEDIA ] OBSERVATION (
[ ID ] <observation_id>
[ ,  [ DESCRIPTION ] <description_id> ]
[ , ( [ DATA VALUE ] <data_value_id> | [ DATA OBJECT ] <data_object_id> ,
      [ DATATYPE ] <datatype_id> ) ]
[ , { ( [ METADATA TAG ] <metadata_tag> : [ METADATA VALUE ] <metadata_value> )
} ] ) ;
```

Finally, we define static or mobile sensors. Each having (i) a description; (ii) a location history record containing a set of location/time interval pairs; (iii) a coverage area history record containing a set of coverage area/time interval pairs; (iv) a set of sensed properties/produced observations; and (v) the platform in which the sensor is embedded/hosted. We should mention that the location/coverage area records must contain at all times a current value for the sensor location and coverage area (i.e., a location/coverage area with an ongoing time interval).

Syntax 9: Defining the structure of a Sensor

```
CREATE SENSOR ( [ ID ] <sensor_id> [ , [ TYPE ] <type> = 'static' | 'mobile' ]
( [ , WITH
    [ , { [ DESCRIPTION ] <description_id> } ]
    [ , [ LOCATION HISTORY ] <location_history> =
        { ( [ LOCATION ] <location_id> , [ TIME INTERVAL ] <ti> ) } ]
    [ , [ COVERAGE HISTORY ] <coverage_history> =
     { ( [ COVERAGE AREA ] <coverage_area_id> , [ TIME INTERVAL ] <ti> ) } ]
] )
( [ , SENSING { [ PROPERTY ] <property_id> } ] )
( [ , PRODUCING { [ OBSERVATION ] <observation_id> } ] )
( [ , HOSTED ON [ PLATFORM ] <platform_id> ] ) ) ;
```

4.2.3 Event Component Syntax. We define an event (cf. Syntax 10) by assigning to it what we called an event space, an n-dimensional space where each dimension represents an event describing feature. In addition, since the events are detected based on sensor data, we map a set of contributing sensors to each event definition. When defining the event, one might choose a specific set of sensors, or any available ones that fit the event needs.

Syntax 10: Defining the structure of an Event

```
CREATE EVENT (
[ ID ] <event_id>
[ , [ EVENT SPACE ] <event_space_id> ]
[ , USING { [ SENSOR ] <sensor_id> } ]
) ;
```

The event space (cf. Syntax 11) contains a set of features each having some related conditions (e.g., *temperature* feature with a condition *greater than 35°C*). Finally, all observations belonging to an event are found within its space.

Syntax 11: Defining the structure of an Event Space

```
CREATE EVENT SPACE (
[ ID ] <event_space_id>
[ ,
    { (
        [ FEATURE ] <feature_id>
        [ , [ CONDITION ] <condition_id> ]
    ) }
]
[ , { [ OBSERVATION ] <observation_id> } ]
) ;
```

4.2.4 Application Domain Component Syntax. Since event features are better defined by an expert. We leave the feature syntax (cf. Syntax 12) to the application domain part. A feature is represented as a dimension in the event space. Therefore, each feature has a specific datatype for its values, a function that measures the distance between two instances belonging to the same feature, a default value, and a description. We provide a set of basic features, and leave the definition of advanced/complex features to domain experts.

Syntax 12: Defining the structure of a Feature

```
CREATE FEATURE (
[ ID ] <feature_id>
[ ,  [ DATATYPE ] <datatype> = 'integer' | 'float' | 'boolean' | 'date' |
'time' | 'date time' | 'character' | 'string'
]
[ ,  [ DISTANCE MEASURE ] <distance_measure_id> ]
[ ,  [ DEFAULT VALUE ] <value> ]
[ ,  [ DESCRIPTION ] <description_id> ] ) ;
```

The application domain experts also define the constraints related to each feature. Syntax 13 defines a condition as a set of statements each having operands and an operator. We provide various operators and allow users to import external operators/functions.

Syntax 13: Defining the structure of a Condition

```
CREATE CONDITION ( [ ID ] <condition_id> [ , { STATEMENT <statement_id> } ] ) ;

CREATE STATEMENT ( [ ID ] <statement_id> ,
( [ OPERAND ] <operand_id>, [ OPERATOR ] <op> [ , [ OPERAND ] <operand_id> ] ));

CREATE OPERAND ( [ ID ] <operand_id> ,
( [ TYPE ] <type> = 'Temporal' | 'Spatial' | 'Other' , [ VALUE ] <val> ) ) ;

<val> = <string> | [ LOCATION ] <location_id> |
        [ TIMESTAMP ] <ts> | [ TIME INTERVAL ] <ti> ;

<op> = [ COMPARISON ] <cop> | [ TEMPORAL ] <top> |
       [ SPATIAL ] <sop> | FUNCTION <function_id>

<cop> = '=' | '<=' | '>=' | '<' | '>' | 'not' ;

<top> = 'hasBegining' | 'hasEnd' | 'inside' | 'intervalAfter' |
        'intervalBefore' | 'intervalContains' | 'intervalDisjoint' |
        'intervalDuring' | 'intervalEquals' | 'intervalFinishedBy' |
        'intervalFinishes' | 'intervalIn' | 'intervalMeets' |
       'intervalMetBy' | 'intervalOverlappedBy' | 'intervalOverlaps' |
        'intervalStartedBy' | 'intervalStarts' |
        [ TEMPORAL RELATION ] <temporal_relation_id> ;

<sop> = 'contains' | 'covers' | 'crosses' | 'equals' | 'includes' |
        'isAbove' | 'isBelow' | 'isCloseTo' | 'isDisjointWith' |
        'isFraFrom' | 'isLeftOf' | 'isRightOf' | 'overlaps' |
        'touches' | [ SPATIAL RELATION ] <spatial_relation_id> ;

[ FUNCTION ] <function_id> ;
```

Finally, since the application domain description differs from one context to another, one needs a generic definition of application domain components and relationships that could be instantiated in any context. Therefore, we define a component named *Concept* (cf. Syntax 14), and an inter-concept relationship, denoted *Relation* (cf. Syntax 15). Relations can also be used between environment, sensor network, or event components.

Syntax 14: Defining the structure of a Concept

```
CREATE CONCEPT ( [ ID ] <concept_id> [ , { ELEMENT <element_id> } ] ) ;

ELEMENT [ ID ] <element_id> = COMPONENT <component_id> |
                              ATTRIBUTE ( <name>, <datatype> ) ;
```

Syntax 15: Defining the structure of a Relation

```
CREATE [ <name> ] RELATION ( [ ID ] <relation_id>
[ , { ( CONCEPT SOURCE <concept_id> ,
        CONCEPT TARGET <concept_id> ) } ]
[ , { ( COMPONENT SOURCE <component_id> ,
        COMPONENT TARGET <component_id> ) } ] ) ;
```

To conclude, one can define the structure of various connected environment components (cf. Criterion 2 - Section 6). Also users can rename, drop, or even alter the structure of a previously defined component[2].

5 ILLUSTRATION & EXPERIMENTAL SETUP

In this section, we rely on the component definitions provided in Section 4 to illustrate the usage of EQL-CE in the Smart Mall (cf. Figure 1). Then, we discuss the evaluation of the language.

[2]Syntax details for all definition queries is provided on the following link https://github.com/eliomansour/EQL-CE/blob/master/EQL-CESyntax.pdf

5.1 EQL-CE Queries

5.1.1 Environment Queries. The mall is a platform of type infrastructure. It has a location map containing five locations (the Hallway, Movie Theater, Grocery Store, Coffee Shop, and Shop 1). The locations and location map are instantiated in queries 1 and 2 respectively. Each location instance specifies the relation between the location and its neighbours.

Query 1: Inserting the Locations

```
INSERT LOCATION ( 'Movie_Theater' ,  { ( 'touches' , 'Hallway' ) ,
( 'isRightOf' , 'Coffee_Shop' ) , ( 'touches' , 'Coffee_Shop' ) } ) ;

INSERT LOCATION ( 'Coffee_Shop' ,  { ( 'touches' , 'Hallway' ) ,
( 'isLeftOf' , 'Movie_Theater' ) , ( 'touches' , 'Movie_Theater' ) } ) ;

INSERT LOCATION ( 'Shop_1' ,  { ( 'touches' , 'Hallway' ) ,
( 'isLeftOf' , 'Grocery_Store' ) , ( 'touches' , 'Grocery_Store' ) } ) ;

INSERT LOCATION ( 'Grocery_Store' ,  { ( 'touches' , 'Hallway' ) ,
( 'isRightOf' , 'Shop_1' ) , ( 'touches' , 'Shop_1' ) ,
( 'isAcrossOf', 'Movie_Theater') } ) ;

INSERT LOCATION ( 'Hallway' ) ;
```

Query 2: Inserting the Location Map

```
INSERT LOCATION MAP ( 'Mall_Location_Map' ,
{ 'Movie_Theater', 'Coffee_Shop', 'Shop_1', 'Grocery_Store', 'Hallway' } ) ;
```

The smart mall infrastructure hosts a mobile device dev_1. The device instance is shown in Query 3. We do not detail the representation of the hardware, software, or services. We only specify that a temperature sensor ('s_6') is embedded on the device.

Query 3: Inserting the Device, its hardware, software, and provided services

```
INSERT DEVICE ( 'dev_1' ,
{ 'dev_1_Hardware' } , { 'dev_1_Software' } , { 'dev_1_Service' } ) ;

INSERT HARDWARE ( 'dev_1_Hardware' , { 's_6' } ) ;

INSERT SOFTWARE ( 'dev_1_Software' ) ;

INSERT SERVICE ( 'dev_1_Service' ) ;
```

Since all infrastructure elements have been instantiated, one can now insert the infrastructure instance (cf. Query 4).

Query 4: Inserting the Mall Infrastructure

```
INSERT INFRASTRUCTURE ( 'Mall_Infrastructure' , 'Mall_Location_Map' ,
                        { 'dev_1' } ) ;
```

The mall infrastructure and mobile device host sensors. Therefore, they are also considered platforms. Queries 5 and 6 insert the two platform instances.

Query 5: Inserting the Mall Platform

```
INSERT PLATFORM ( 'Mall_Platform' , <type> = 'infrastructure' ) ;
```

Query 6: Inserting the Device Platform

```
INSERT PLATFORM ( 'dev_1' , <type> = 'device' ) ;
```

5.1.2 Sensor Network Queries. In regards to the sensor network related queries, we begin by instantiating the two properties that are currently monitored in the example: (i) the scalar property temperature; and (ii) the multimedia property video. This is shown in queries 7 and 8 respectively.

Query 7: Inserting the temperature property

```
INSERT PROPERTY ( 'temperature_property' , <type> = 'scalar' ) ;
```

Query 8: Inserting the video property

```
INSERT PROPERTY ( 'video_property' , <type> = 'video' ) ;
```

These properties are monitored by various sensors. Six temperature sensors exist in the smart mall. $s_1, s_2, ands_3$ are deployed in the movie theater, s_4 in the grocery store, s_5 in Shop 1, and s_6 is the only mobile sensor deployed on dev_1 which is currently located in the Coffee Shop. A surveillance camera cam_1 is deployed in Shop 1. Query 9 instantiates all the aforementioned sensors. Note that the mobile sensor s_6 has a previous/current location/coverage area.

Query 9: Inserting all sensors

```
INSERT SENSOR ( 's_1' , <type> = 'static'  , WITH (
<location_history> = { ( 'Movie_Theater' , '19-04-2019 11:44:27 ; now' ) } ,
<coverage_history> = { ( 'Movie_Theater' , '19-04-2019 11:44:57 ; now' ) } ) ,
SENSING ( { 'temperature_property' } ] ) , HOSTED ON ( 'Mall_Platform' ) ) ;

INSERT SENSOR ( 's_2' , <type> = 'static'  , WITH (
<location_history> = { ( 'Movie_Theater' , '19-04-2019 11:54:27 ; now' ) } ,
<coverage_history> = { ( 'Movie_Theater' , '19-04-2019 11:55:27 ; now' ) } ) ,
SENSING ( { 'temperature_property' } ] ) , HOSTED ON ( 'Mall_Platform' ) ) ;

INSERT SENSOR ( 's_3' , <type> = 'static'  , WITH (
<location_history> = { ( 'Movie_Theater' , '19-04-2019 11:42:27 ; now' ) } ,
<coverage_history> = { ( 'Movie_Theater' , '19-04-2019 11:43:27 ; now' ) } ) ,
SENSING ( { 'temperature_property' } ] ) , HOSTED ON ( 'Mall_Platform' ) ) ;

INSERT SENSOR ( 's_4' , <type> = 'static'  , WITH (
<location_history> = { ( 'Grocery_Store' , '19-04-2019 11:44:27 ; now' ) } ,
<coverage_history> = { ( 'Grocery_Store' , '19-04-2019 11:44:57 ; now' ) } ) ,
SENSING ( { 'temperature_property' } ] ) , HOSTED ON ( 'Mall_Platform' ) ) ;

INSERT SENSOR ( 's_5' , <type> = 'static'  , WITH (
<location_history> = { ( 'Shop_1' , '19-04-2019 11:54:27 ; now' ) } ,
<coverage_history> = { ( 'Shop_1' , '19-04-2019 11:55:27 ; now' ) } ) ,
SENSING ( { 'temperature_property' } ] ) , HOSTED ON ( 'Mall_Platform' ) ) ;

INSERT SENSOR ( 's_6' , <type> = 'mobile'  , WITH (
<location_history> = { ( 'Coffee_Shop' , '19-04-2019 11:42:27 ; now' )
( 'Shop_1' , '19-04-2019 10:43:27 ; 19-04-2019 11:23:20' ) } ,
<coverage_history> = { ( 'Coffee_Shop' , '19-04-2019 11:43:27 ; now' ) ,
( 'Shop_1' , '19-04-2019 10:43:27 ; 19-04-2019 11:23:20' ) } ) ,
SENSING ( { 'temperature_property' } ] ) , HOSTED ON ( 'dev_1' ) ) ;

INSERT SENSOR ( 'cam_1' , <type> = 'static'  , WITH (
<location_history> = { ( 'Shop_1' , '19-04-2019 11:25:14 ; now' ) } ,
<coverage_history> = { ( 'Shop_1' , '19-04-2019 11:25:14 ; now' ) } ) ,
SENSING ( { 'video_property' } ) , HOSTED ON ( 'Mall_Platform' ) ) ;
```

When the network becomes operational, sensors will start producing observations. The latter are sent to a middle-ware that will generate an insert query in order to push the observations into the data model according to the observation syntax (cf. Section 4). Then, the middle-ware will update both sensors and properties by mapping them to their related observations. Query 10 shows an insert query generated by the middle-ware for a temperature observation having a float value of '20.3' and two associated metadata for time and location of capture. Similarly Query 11 shows another temperature observation taken from the Coffee Shop. Query 12 instantiates a video observation taken from the surveillance camera in Shop 1. This observation includes the video recording file, temporal, location, and video length related metadata.

Query 10: Inserting a temperature observation taken in Shop 1

```
INSERT SCALAR OBSERVATION ( 'temperature_observation_1',
( '20.3' , 'float' ) ,
{ ( 'timestamp' : '19-04-2019 11:34:54' ) , ( 'location' : 'Shop_1' ) } ) ;
```

Query 11: Inserting a temperature observation taken in the Coffee Shop

```
INSERT SCALAR OBSERVATION ( 'temperature_observation_2',
( '19.3' , 'float' ) ,
{ ( 'timestamp' : '19-04-2019 11:44:27' ) , ( 'location' : 'Coffee_Shop' ) } ) ;
```

Query 12: Inserting a video observation taken in Shop 1

```
INSERT MEDIA OBSERVATION ( 'video_observation',
( 'recording.mpeg' , 'video' ) ,
{ ( 'timestamp' : '19-04-2019 11:35:14' ) , ( 'location' : 'Shop_1' ) ,
( 'duration' :  '123 s' ) ) ;
```

5.1.3 Event Queries. In order to give an example of how an event can be instantiated we define next an intrusion event in Shop 1. The mall manager relies on video sensor cam_1 for the detection of this event. He/She defines the event as a face detected by cam_1 in the Shop after 8 PM. Three features define this event: (i) time with a condition after 8 PM; (ii) location with a restriction to Shop 1; and (iii) a detected face with a Boolean value equals true. In this example, the manager uses our provided basic features for time, location, and detected face (cf. Query 13) where we only define the feature as an identifier assigned to a datatype. The manager also creates the required conditions for each feature (cf. Query 14). However, one can use the application domain queries to define more complex/advanced features/conditions if needed. Finally, query 15 details the event space, and query 16 instantiates the event.

Query 13: Inserting features for the intrusion event

```
INSERT FEATURE ( 'time_f' , 'date-time' ) ;

INSERT FEATURE ( 'location_f' , 'string' ) ;

INSERT FEATURE ( 'face_f' , 'Boolean' ) ;
```

Query 14: Inserting conditions for the intrusion event features

```
INSERT CONDITION ( 'condition_1' , { 'statement_1' } ) ;
INSERT STATEMENT ( 'statement_1' ,
                 ( cam_1.Observation.timestamp , After( '8 PM' ) ) ) ;

INSERT CONDITION ( 'condition_2' , { 'statement_2' } ) ;
INSERT STATEMENT ( 'statement_2' ,
                 ( cam_1.Location , Equals( 'Movie_Theater' ) ) ) ;

INSERT CONDITION ( 'condition_3' , { 'statement_3' } ) ;
INSERT STATEMENT ( 'statement_3' ,
                 ( cam_1.Observation , face_detected( 'true' ) ) ) ;
```

Query 15: Inserting an event space for the intrusion event

```
INSERT EVENT SPACE ('event_space_1' ,
{ ( 'time_f' , 'condition_1' ) , ( 'location_f' , 'condition_2' ) ,
  ( 'face_f' , 'condition_3' ) } ) ;
```

Query 16: Inserting the event definition

```
INSERT EVENT ( 'intrusion_in_shop_1' , 'event_space_1' ,
               USING { 'cam_1' } ) ;
```

5.2 Additional Queries

EQL-CE provides various query types. For the component definition language one could rename/drop (cf. Query 17), or alter (cf. Query 18) a component's structure. Note that one could alter a definition by adding, removing, or modifying its content.

Query 17: RENAME/DROP query examples

```
RENAME COMPONENT ( 'SENSOR' TO 'SENS' ) ;
DROP COMPONENT ( 'SENSOR' ) ;
```

Query 18: ALTER Query - Infrastructure example

```
ALTER INFRASTRUCTURE ( [ ID ]  <infrastructure_id> ,
ADD | REMOVE ( [ , [ LOCATION MAP ] <location_map_id> ]
               [ , { [ PLATFORM ] <platform_id> } ] ) ) ;

ALTER INFRASTRUCTURE ( [ ID ]  <infrastructure_id> ,
MODIFY ( [ , [ LOCATION MAP ] [ <name> ] <location_map_id> ]
         [ , { [ PLATFORM ] [ <name> ] <platform_id> } ] ) ) ;
```

In regards to the data manipulation language, one could also select, update, or delete component instances. Examples are shown in queries 19, 20, and 21 respectively.

Query 19: SELECT all platforms of type device

```
SELECT PLATFORM <platform_id>
FROM PLATFORM WHERE PLATFORM TYPE <platform_type> = 'device' ;
```

Query 20: UPDATE a platform - Change type to device

```
UPDATE PLATFORM CHANGE <platform_type> = 'device'
WHERE <platform_id> = 'Platform_1' ;
```

Query 21: DELETE all platforms of type device

```
DELETE PLATFORM WHERE <platform_type> = 'device' ;
```

5.3 Experimental Setup

We are currently implementing the EQL-CE query run engine that executes the queries. It is part of an online platform for event detection in connected environments. Since the development is still ongoing, we propose here the experimental protocol that we will use to evaluate the query language. We propose the following experiments:

- Query Cost Evaluation: Providing the user with the ability to define the entire connected environment allows him/her to manage all its components from scratch. However, this might be costly in terms of the number of 'steps' (i.e., queries) required to achieve a specific task/objective. In this experiment, we set a list of objectives (e.g., defining a platform, a sensor, a location map) and quantify the required query batch size and the total cost of achieving the task.
- Re-usability Evaluation: In this test we evaluate the physical layer's ability to parse EBNF into various other languages (e.g., SQL, SPARQL). We re-iterate this experiment for each query type (i.e., SELECT, INSERT, UPDATE, DELETE, CREATE, ALTER, RENAME, DROP).
- Performance Evaluation: In this test, we measure the runtime, and the resource consumption (CPU/RAM) when executing EQL-CE queries. We measure performance for individual queries and for batches (required for specific tasks).

6 RELATED WORK

To compare existing approaches, we propose the following criteria based on the challenges and limitations discussed in Section 2:

- Criterion 1. **Basic Querying:** Stating if the EQL allows common query types for component definition, and component instance manipulation (cf. Need 1).
- Criterion 2. **Component Coverage:** Denoting if the EQL is capable of covering the entire connected environment. This includes environment, sensor network, application domain, and event related components (cf. Need 2).
- Criterion 3. **Data Diversity:** Specifying if the EQL is capable of integrating various datatypes related to the scalar/multimedia sensed properties/sensor observations (cf. Need 3).
- Criterion 4. **Handling Environment Dynamicity:** Indicating if the EQL provides the means to modify the structure of previously defined components (e.g., events) in order to cope with the environment's dynamicity. This is useful in a dynamic setup, where sensor mobility causes gain/loss of data in certain areas of the environment (cf. Need 4).

In the following, we evaluate some works for each criterion (we do not detail here every existing EQL for the sake of brevity).

6.1 Basic Querying

Here we evaluate some works' capability to provide various query types to allow component definition, and management (cf. Need

1). In [8], the authors propose an intuitive event query language denoted SNOOP. They define three event attributes: (i) a name; (ii) a set of conditions (the pattern); and (iii) a set of actions to be triggered once the event is detected. SNOOP integrates operators for inter-condition relations (e.g., conjunction, dis-junction, and sequence) and represents repetitive events through the usage of the periodic/aperiodic operators. In [6], the authors propose a language denoted CeDR. In comparison with SNOOP, CeDR adds a WHERE clause for filtering statements and has a wider range of operators. Therefore, CeDR is considered more expressive in terms of event pattern description. CeDR also includes an event lifetime operator and a detection window operator. CQL[4] is another language that can be used for event definition/retrieval. CQL extends SQL by emphasizing on continuous data streams/queries. The authors add temporal operators, sliding windows, and window parameters to better handle continuous data.

Discussion: The aforementioned works are intuitive, practical, and allow various composition operators for event definition. However, [6, 8] mainly focus on the definition and retrieval of events and neglect other tasks such as updating definitions or inserting data (cf. Criterion 1). CQL provides various query types and covers different functionality since it extends SQL.

6.2 Component Coverage

Here we evaluate some works' capability to cover various connected environment components (cf. Need 2). The authors in [11] propose an event query language for data streams called SaSE. They include the WITHIN and RETURN statements to respectively declare sliding time windows and the required output. SaSE also allows event pattern operators (similar to CeDR) in a WHERE clause. ETALIS[3] is an EQL that describes events as rules. Its syntax is based on logic style formulas. The authors propose a set of temporal relations and composition operators to define the event patterns. The syntax of the rules is independent of any underlying data model. EP-SPARQL[2] extends SPARQL to provide an EQL for linked data management systems. It integrates event processing operators (e.g., sequence) into the SPARQL syntax. This work allows the definition of simple and complex event patterns.

Discussion: The aforementioned works cover the majority of temporal and composition operators. ETALIS and SASE provide a re-usable syntax, however EP-SPARQL is restricted to semantic data models since it extends SPARQL. In terms of component coverage, SASE and ETALIS heavily focus on events and do not consider other connected environment components (cf. Criterion 2). EP-SPARQL has the ability to define structures for various components (e.g., related to the environment, sensor network).

6.3 Data Diversity

Here we evaluate some works' capability to integrate different data and datatypes in their syntax (cf. Need 3). SPARQL-ST[14] extends SPARQL by adding operators for spatial/temporal queries. This covers the definition and manipulation of spatial shapes and temporal entities. [12] also extends SPARQL. It considers multimedia data, and media fragments. This language aims to improve semantic multimedia data retrieval. XChangeEQ[7] is a logic style based language. It allows: (i) data-related operations such as variable bindings and conditions containing arithmetic operations; (ii) event composition operators such as conjunction, dis-junction, and order; (iii) temporal and causal relations between events in the queries; and (iv) event accumulation, for instance aggregating data from previous events to discover new ones.

Discussion: All the aforementioned works provide expressive tools for event definition. Languages extending SPARQL [12, 14] are all user friendly since they are based on a known language. Even though, these works handle various datatypes for scalar data (e.g., integer, float, string, date), SPARQL-MM[12] is the only one that integrates multimedia data as well.

6.4 Handling Environment Dynamicity

Here we evaluate some works' capability to cope with the dynamicity of a connected environment (cf. Need 4). ESPER[9] is an implementation for event detection in database systems. The authors proposed an SQL-like syntax for event processing. Therefore, known operators such as CREATE, SELECT, INSERT, UPDATE, and DELETE are available for event definition and detection. ESPER also includes temporal operators and a specific statement for event definition (i.e., the pattern). In addition to the aforementioned advantages, this language has a fast learning curve since it is highly similar to traditional SQL. In [10], the authors present an extension of SPARQL to integrate temporal features (e.g., annotating triples with time stamps) for better querying of RDF triples over time. C-SPARQL[5] extends SPARQL to consider stream data in the queries. To do so, the authors integrate sliding time windows.

Discussion: The aforementioned works integrate temporal operators, and some of them consider data streams. Also, some provide users with update queries to modify the data over time. However, none provides the means to automatically discover and re-write queries that have become obsolete due to sensor mobility. This entails measuring sensor/data similarities in order to replace missing the elements with correct alternatives.

Since none of the mentioned works fully considers our entire list of criteria, we examine next the Extended Backus-Naur Form metalanguage (EBNF). We provide some background and preliminaries, discuss its usage, syntax, and notations.

7 CONCLUSION & FUTURE WORK

Many challenges emerge when considering an EQL for connected environments. Here, we addressed the issues of covering various components, query types, datatypes, and coping with the environment's dynamicity. Our proposal provides a generic and re-usable syntax. EQL-CE considers various connected environment components (e.g., environment, sensor network, events, and application domain), offers common query types (e.g., for definition and manipulation of components/instances), and allows the definition of scalar/multimedia data structures.

As future work, our first priority is to finish the implementation and evaluation of EQL-CE. Then, we would like to address the following issues. First, we aim to define the security/privacy related query types (e.g., access control). Then, we would like to develop the

query optimizer by integrating advanced spatial/temporal elements to the queries for specific event definitions. Moreover, we would like to evaluate our query re-writing engine and test it in various scenarios. Furthermore, we still need to consider composite events in EQL-CE by integrating event composition operators. Finally, we would like to test the language in a real connected environment setup.

REFERENCES

[1] Giuseppe Amato et al. 2015. Querying moving events in wireless sensor networks. *Pervasive and Mobile Computing* 16 (2015), 51–75.

[2] Darko Anicic et al. 2011. EP-SPARQL: a unified language for event processing and stream reasoning. In *Proceedings of the 20th international conference on World wide web*. ACM, New York, NY, USA, 635–644.

[3] Darko Anicic et al. 2011. Etalis: Rule-based reasoning in event processing. In *Reasoning in event-based distributed systems*. Springer, Berlin, Heidelberg, Germany, 99–124.

[4] Arvind Arasu et al. 2006. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal* 15, 2 (2006), 121–142.

[5] Davide Francesco Barbieri et al. 2009. C-SPARQL: SPARQL for continuous querying. In *The 18th international conference on World wide web-WWW'09*. ACM, New York, NY, USA, 1061–1062.

[6] Roger S Barga and Hillary Caituiro-Monge. 2006. Event correlation and pattern detection in CEDR. In *International Conference on Extending Database Technology*. Springer, Berlin, Heidelberg, Germany, 919–930.

[7] François Bry and Michael Eckert. 2007. Rule-based composite event queries: the language XChange EQ and its semantics. In *International Conference on Web Reasoning and Rule Systems*. Springer, Berlin, Heidelberg, Germany, 16–30.

[8] Sharma Chakravarthy and Deepak Mishra. 1994. Snoop: An expressive event specification language for active databases. *Data & Knowledge Engineering* 14, 1 (1994), 1–26.

[9] EsperTech. 2006. EsperTech. Chapter 5. EPL reference:Clauses. http://esper.espertech.com/release-5.3.0/esper-reference/html_single/index.html#epl_clauses. Accessed: 2019-02-07.

[10] Fabio Grandi. 2010. T-SPARQL: A TSQL2-like Temporal Query Language for RDF.. In *ADBIS (Local Proceedings)*. Citeseer, 21–30.

[11] Daniel Gyllstrom et al. 2006. SASE: Complex Event Processing over Streams. *CoRR* abs/cs/0612128 (2006), 407–411. arXiv:cs/0612128 http://arxiv.org/abs/cs/0612128

[12] Thomas Kurz, Sebastian Schaffert, Kai Schlegel, Florian Stegmaier, and Harald Kosch. 2014. SPARQL-MM-extending SPARQL to media fragments. In *European Semantic Web Conference*. Springer, Berlin, Heidelberg, Germany, 236–240.

[13] Elio Mansour, Richard Chbeir, and Philippe Arnould. 2019. EQL-CE: an event query language for connected environments. In *Proceedings of the 23rd International Database Applications & Engineering Symposium*. ACM, 7.

[14] Matthew Perry et al. 2011. Sparql-st: Extending sparql to support spatiotemporal queries. In *Geospatial semantics and the semantic web*. Springer, Boston, MA, USA, 61–86.

[15] Nicholas Poul Schultz-Møller et al. 2009. Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. ACM, New York, NY, USA, 4.