Bringing Complex Event Processing into Multitree Modelling of Sensors

Alexandre Garnier, Jean-Marc Menaud, Nicolas Montavont

Bringing Complex Event Processing into Multitree Modelling of Sensors

Alexandre Garnier[1], Jean-Marc Menaud[1], and Nicolas Montavont[2]

[1] ASCOLA Research Group, Mines Nantes / Inria / LINA UMR 6241
Nantes, France
{alexandre.garnier, jean-marc.menaud}@mines-nantes.fr
[2] Institut Mines-Télécom / Télécom Bretagne, Irisa
Rennes, France
nicolas.montavont@telecom-bretagne.eu

Abstract. The recent advances in the Internet of Things allow deploying a large variety of applications for smart cities, home automation or the industry of the future. These applications generate a large amount of data that can be challenging to manage; identifying and parsing this data become a prominent problem. In order to address this issue, we propose a multitree model for the sensor representation which matches the need for heterogeneous applications and user support. From there, we define a complex event processor based on a new language and grammar, in order to filter and identify user specific events. We show that we considerably reduce the size of queries by focusing on end-users knowledges as semantics for data streams.

Keywords: sensor networks, domain-specific languages, complex event processing

1 Introduction

From home automation to smart cities, from amateur weather stations to large deployments of smart power meters in datacenters, Internet of Things (IoT) applications target more and more end-users every day. The link between these users and the IoT is usually provided by applications deployed over a sensor network. A recurring problem concerning sensor networks is the heterogeneity, not only of sensors, but also of the protocols used to access them, often characterized by their low bandwidth and poor reliability. To address this issue, the notion of data streams has emerged, leading to a change of paradigm around data parsing, from traditional DataBase Management Systems (DBMS) to Complex Event Processing (CEP). Instead of processing stored persistent data through volatile queries, CEP parses volatile data stream as it comes through persistent queries. However, parsing the raw data issued from sensors remains a complex task. This is due to the endless nature of the data stream and its growing heterogeneity, which reflects the variety of networked things. In order to address this issue, ideally the data has to be adapted to the user's knowledge.

Given that the number of users to consider grows with the IoT coverage, the ability to provide a meaningful access to the data to each user is a prominent problem. If CEP alone is able to notify specific users with data they are interested in, it does not allow to pre-identify these data. To leverage this gap, some kind of semantics, or context, can be attached to the data, in order to assist users when identifying the information they need. To this end, various tracks have been followed, from ontology-based enrichment to context-aware solutions. While ontology-based solutions tend to be harsh to manage for non-expert end-users and do not directly address their needs, context-aware solutions usually lack interoperability between different user contexts. A good compromise could be to provide a cross-context modelling of the data. Such a model would provide a defining frame to the semantical enrichment, while avoiding context partitioning. In order to fully provide to users a simple yet effective access to the data, CEP should be merged with such a modelling of the data.

Our previous work, SensorScript [7], aimed at providing a cross-context modelling of the data. In this paper, we propose to enhance it with a complex event processor, addressable through a new Domain-Specific Language (DSL). The DSL focuses on pre-identified uses and their combinations, relying on end-users oriented knowledge. This allows to reduce lengths of queries an end-user can express as they are able to manipulate nothing more than what he considers to be relevant information. Moreover, decision making can be automated in order to address actuators specific features in addition to sensors data gathering.

The remainder of the paper is organized as follows. Section 2 studies existing work about data semantics and CEP. In section 3 we draw up the motivation for integrating CEP with SensorScript. Sections 4 and 5 introduce the model and the language which ensure complex event processing. Section 6 evaluates the language concision and the underlying query management through a demonstration scenario. Finally, section 7 concludes by presenting future work.

2 Related Work

Data identification has been a prominent track of research over the years.We can divide the existing work into two categories: data semantical enrichment on one hand; context aware data stream mining on the other hand. Several publications about context awareness for the IoT are discussed in [13]. In this paper we will focus on solutions which provide real-time processing of the data stream, as it is a strong requirement for complex event processors.

In [18] data semantics are provided by a separated knowledge base, which is a Resource Description Framework [9] (RDF) store. Thus event queries mix raw events extracted from the data stream and background knowledge retrieved from the knowledge base. That allows to establish relationships between the raw events. RDF knowledge base access is done through SPARQL [20] queries. This inevitably leads to hybrid queries, which mix SPARQL syntax with complex event paradigm. Use of such semantically enriched complex events is addressed in [17]. It relies on the notion of event stream from which raw data is pulled then

pushed back after semantical enrichment and event composition. Furthermore, a partitioning of enriched data stream mining operators is proposed for both CEP (*e.g.* filters or aggregators) and knowledge operators.

SCONSTREAM [10] aims at providing spatial enrichment over the data for the specific case of users tracking in home automation. Queries continually parse the raw data to generate spatialized events when triggered. UbiQuSE [16] proposes a more generic contextual framework for data mining queries. It relies on XML formalism for context-enriched data. Thus it uses XQuery [2] to express queries that address both real-time and historical data querying. This broadens the use of the DBMS, as it stores both contextual and historical data. These two solutions however rely on pre-existing solutions to bundle both data mining (being real-time or periodic) and context-awareness, which leads to hybrid querying over the data. COPAL [11] aims at providing a DSL to broaden the notion of context from sole location to handle processing environments in the case of distributed processing. This DSL provides a complex event processor in order to compose events through a declarative, and quite verbose, developer-centric syntax, in the sense that a user has to learn the underlying model before composing events. A common issue of these solutions remains in the context storage, generally based on a decoupled DBMS, which impacts the simplicity of queries. The runtime additional cost of addressing the DBMS and couple its information with the raw data is addressed by none of these publications.

Concerning CEP languages, other contributions mainly aimed at adapting Structured Query Languages (SQL) to manage data streams and event composition. CQL [1] is one of the first to do so. The change of paradigm from relational databases to complex event processing focuses on the notion of a relation. A relation is addressed in the *from* clause, like tables in SQL, and mapped over time windows to a finite set of data from. Other than time windows, partitioned windows can also be expressed, providing a partition over the data stream similar to the SQL *group by* clause. TinyDB [12] provides time windows with a dedicated additional clause rather than a mapping over data stream. It also allows to specify a recovery rate for queries execution, jeopardizing efficiency as there is no guarantee that the data will be updated at at least the same rate of the accesses defined in queries. Aiming at providing more flexibility over windows specification, Esper [6] provides the notion of pattern which orchestrates both time windows and data filtering with boolean operators. WildCAT [5] aims at coupling Esper Processing Language (EPL) with data context awareness through hierarchical contexts definition. However, this semantical enrichment of data operates as on overlay to Esper rather than being fully integrated within EPL.

Another track focused on declarative event specification. AmbientTalk [19] uses this concept for the actors within mobile ad-hoc networks, as a mean to leverage the problematics specific to these infrastructures. TESLA [4] formalizes an event specification language. Following AmbientTalk, REScala [15] and EventCJ [8] integrate such a formalism within object-oriented and functional programming. If these languages depart from traditional SQL, they however concentrate on addressing a larger scope rather than simplifying their syntax.

3 Motivation

SensorScript was based on a previous work: btrScript [14]. btrScript is a datacenter monitoring DSL inspired by XPath [3], in particular its queries which allow implicit pathfinding within a tree. Indeed the DSL is backed to two static trees to address both virtualized and physical aspects of a modern datacenter. In [7], we altered the underlying model to manage any number of configurable intricated trees, which allows SensorScript to address the diversity of sensor networks. The trees intrication of the model will be detailed in section 4.

The benefits of such a modelling are two-fold. On one hand it offers a good semanticization over the data by integrating it within tree contexts. On the other hand, queries remain concise as the model still relies on trees. Hence, we consider these features make SensorScript a strong candidate to be integrated within a complex event processor, as existing CEP languages suffer from verbose queries. This led us to deeply alter SensorScript data and query management, and to rethink key components of the DSL grammar.

4 Model

The model consists of two parts, which are the data modelling and the complex event processor. The data modelling consists on a multitree modelling of the data in which each tree corresponds to an end-user field of expertise, or a *context*. Figure 1 illustrates a multitree model with five different contexts. These contexts revolve around a conference site, with lighting monitoring and automation on one hand, and presentations' affluence tracking system on the other hand. These use cases are described in more detail through some examples in section 6.

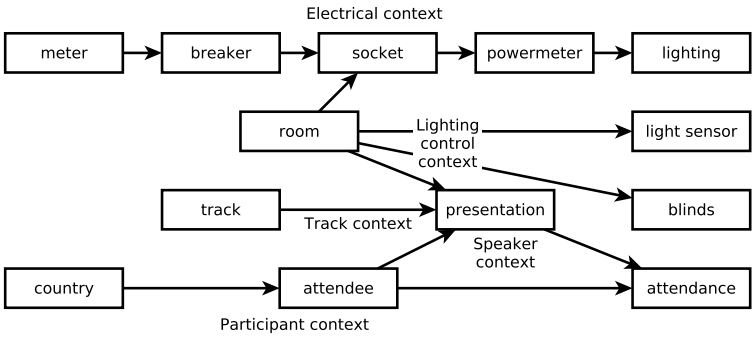

Fig. 1. Example of a multitree model

To set up complex event processing, we propose to change the paradigm on which is based traditional data management. Instead of considering the data as persistent, we assume data streams on which queries are considered persistent. These queries must be constantly aware of any data update. To achieve that, a naive solution would be to rely on a periodic queries-executing process. This is however unsatisfactory because of higher costs in terms of both efficiency

and responsiveness, according to the data stream rate. More realistically, both problems will happen at different times, due to the various underlying networks which are not all reliable, and the various number of queries impacting their execution time. Hence, the query model must react dynamically to data changes.

We lean on the multitree model to leverage data accessibility. The hierarchy of nodes within the multitree can and will be accessed through the queries, as it provides meaningful information about contexts, therefore users specific knowledges. As a matter of fact, queries results are updated on real time with the data stream, but also with changes of the multitree structure. Thus the multitree sets a semantical structure down. Queries rely on these semantics in order to access nodes based on the constitutive contexts information of the multitree model. To achieve that, we propose the query object model as illustrated in the UML class diagram of figure 2.

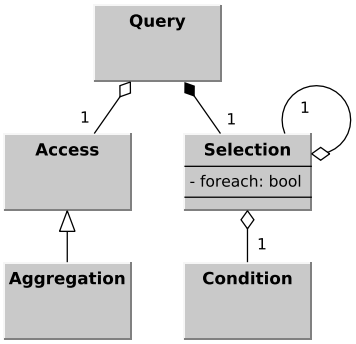

Fig. 2. Query object model

In our query object model, a query consists of three main concepts, which are the node *selection*, on which can be expressed a *condition*, and the optional *access* to selected node attributes or methods.

4.1 Selections

An arbitrary amount of sub-selections can be specified, as well as conditions optionally filter the nodes in each selection or sub-selection. Considering that nodes can be added, moved, removed, and conditions on them can change, selections will evolve with each change impacting its nodes.

4.2 Conditions

Conditions allow to filter the selected nodes. Two kinds of conditions can be expressed:

- conditions on attributes (specified by name): for each node of the selection, a comparison is done either between an access and a constant, or between two accesses over the node;
- conditions on connected nodes (specified by type): for each node of the selection, a boolean set operation is done on all the nodes of the given type that are accessible, upwards or downwards, from this node; for instance, we can restrict the selection of all sockets to the set of sockets with no powermeter.

4.3 Accesses

Accesses are made on each node of the selection, and can be delivered as is or aggregated (thus the *Aggregation* access inheritance).

5 Language

The main objective of the SensorScript language is to allow users to express CEP queries about their own field of expertise, regardless of the complexity of the whole underlying multitree model. As we saw in section 2, existing CEP DSL derive from SQL, thus require users to know more of the underlying model than what should be needed. In contrast, SensorScript comes with a language which leans on the multitree model and takes advantage of the relations between its nodes to provide implicit connections among them, regardless of the distance separating them in the model. As we want to keep the language as concise as possible, we choose to use character operators rather than english words based syntax, which we hope significantly reduces the verbosity of queries. Naturally, the language reflects the selection, access and condition concepts constituting a query as presented in section 4. It is essential however to keep the expression of these three concepts as simple as possible.

Listing 1. SensorScript simplified grammar

```
1:            Query → Selection(.Access)? |
                      Selection:Selection.AggregationMethod
2:        Selection → AtomicSelection({Condition})?(/Selection)?
3:           Access → SimpleAccess | AggregationMethod
4:        Condition → SimpleAccess Comparator SimpleAccess |
                      Condition BooleanOperator Condition |
                      (Condition, Duration)
5:     SimpleAccess → Attribute | SimpleMethod | Constant
6:       Comparator → = | != | < | > | <= | >=
7: BooleanOperator → & | | | ;
8: AtomicSelection → NodeName | NodeType
```

Non-terminal symbols
Grammar description operators
Terminal symbols

Considering these points, we propose a simplified grammar of the language in listing 1. We will go through the grammar rule by rule, following the non-terminals as they occur within rules. Rule 1 reflects that a query is either a *selection* or an *access* (simple or aggregated) over a selection.

5.1 Selection

The second rule shows that sub-selections over a selection are expressed by the slash operator between super and sub-selections. Selections are expressed either on node types or node names. For this reason, not only both names and types are unique, but also a name cannot be equal to a type. Considering a sensor network modelled with figure 1, the query listing the breakers of room 42, for instance, would be: `room42/breaker`

5.2 Condition

Rule 2 in listing 1 also introduces the expression of conditions, within braces operators, over selections. As shown in rule 4, conditions are either simple, consisting of comparisons on *accesses*, or composed of sub-conditions by boolean operators.

Besides traditional *and* and *or* operators, we introduce here the *sequence* operator ";", so that the condition `<selection>{A;B}` ensures that conditions A *then* B are met on nodes of the selection. That does not mean that B has to match after A is satisfied, but that, whether or not B was already satisfied when A matches, B must be checked *chronogically after* A matches for the condition to be met.

Another aspect of time management appears with time conditions, which are simply conditions checked over a duration of time, both of them expressed between parentheses and separated by a comma.

As an example, we consider that one wants to detect the room 42 powermeters that go through an electrical overload. This can be described as the powermeters that have a power consumption that outnumbers their capacity just before it drops to zero, which can be expressed with this query:

```
room42/powermeter{power > capacity; power = 0}
```

As preventing an electrical overload seems to be a better solution, one could create an alert of when a powermeter is soon to be overloaded, for instance when its power consumption remains close to its capacity (with a minimum charge of 90%) for at least one hour:

```
room42/powermeter{(power > capacity * 0.9, '1 h')}
```

5.3 Access

Accesses are done on each node of a selection, through the dot operator. The access of a query occurs on two occasions on runtime:

- when a node is added to the selection, access on it occurs systematically;
- for a node already in the selection, each node update that affects the access will trigger it.

Rules 3 and 5 in listing 1 show that they exist several possible accesses on nodes:

Attribute access for each node of a selection, the query will wait for the given attribute to be updated. For instance, to be notified of each power update from powermeters of room 42: `room42/powermeter.power`

Constant access this access allows to express constants, which is mostly useful for conditional expressions. As shown in rule 4, accesses within conditions are expressed without the dot operator. Considering our previous example, this corresponds to the zero in this query:

```
room42/powermeter{power > capacity; power = 0}
```

Method access for each node of a selection, the query will recall the method for each node update that might affect the method result. This will exclusively happen for method with parameters that correspond to attribute accesses. For methods with no parameter or only constant parameters, accesses are only provided when nodes are added to the selection and for these nodes only. Two types of methods exist:

- simple methods: similar to attributes accesses, they are called separately for each node of the selection. For instance, this is the get method, which is equivalent to an attribute access: `/room42/powermeter.get(power)`
- aggregation methods: on the contrary, aggregation methods provide a computation which occur on all nodes from the selection to produce one result only; an update on one node of the selection, as well as changes of the selection itself, will trigger the method to be called. As an example, let's consider that one wants to access the total consumption from room 42 powermeters: `room42/powermeter.sum(power)`

5.4 Foreach

A particular aggregation use case allows to partition the selection to provide a behavior similar to the *group by* clause in MySQL. This is what we call the *foreach* aggregation method access, expressed by the colon operator in rule 1. To explain how it is expressed, we will consider this example , and its equivalent in SQL:

SensorScript	**SQL**

`room42/breaker:powermeter.sum(power)`	`SELECT sum(powermeter.power)`
	`FROM breaker, powermeter`
	`WHERE breaker.room = 42`
	`AND powermeter.brId = breaker.id`
	`GROUP BY breaker.id`

We see here two selections around the colon operator, which are breakers from room 42 for the first one, powermeters for the second one. Besides, the sum method is called on the *power* attribute from powermeters. *De facto*, this query will follow power updates for each room 42 powermeter. But rather than summing the whole power consumption of the room, it will sum the power consumption *for each* breaker accessible from the room 42, considering sockets within a same room are attached to different breakers.

So, if we consider a query of the form `A:B.method(access)`, considering that A and B are selections, this means that *for each* node N from the selection A, the specified aggregation method will be called on nodes from selection B *accessible from* N (or the nodes corresponding to the `N/B`).

The difference in the concept's name with SQL is to reflect the way it is expressed and avoid confusion: the *group by* clause precedes an attribute, the *foreach* operator follows a node selection.

6 Evaluation

This section proposes to evaluate the language concision through some examples over the model from figure 1 and compare them with similar examples from the literature. Then we propose a scenario which reflects a more complex yet realistic use of the language. Both approaches focus only on syntactic concision of the language, performance evaluation will be subject to future work. Futhermore, we will specifically look at timed conditions management as they bring an additional constraint over the model dynamicity. Finally, we will highlight the limitations of SensorScript in terms of features, compared to other CEP languages.

6.1 Comparison with CQL

In the model from figure 1, more specifically around the *track*, *speaker* and *participant* contexts, we consider a conference for which name tags distributed to every attendee embed an RFID chip. For each presentation, they are invited to check in by swiping their name tag in an RFID reader. Speakers (which is a role that an attendee assumes for a presentation) also check in when beginning their presentation. Each room of the conference has its own RFID reader. Technically, the data stream is flowing with the presence of attendees in any of the conference rooms.

To keep things as simple as possible, we concentrate here on the three aforementioned contexts:

- the *participant* context, for attendees who attend a presentation;
- the *speaker* contexts, for the attendee who holds a presentation;
- the *track* contexts, that reflects the fact that presentations are part of a track of the conference.

These three contexts are directly inspired from the example of CQL [1]. This example considered an auction system, for which we propose the mapping table 1 in order to stick to our conference tracking system.

Table 1. Mapping to CQL example

Conference attendance monitoring model	**Auction system model**
Attendee	User
Presentation	Auction
Attendance	Bid
Country	U.S. state
Participant context	Bidding context
Speaker context	Seller context
Track context	Auction context

Table 2 gives a comparison between SensorScript and CQL queries based on the aforementioned mapping.

1. The first query allows to select presentations that occur after noon. It is conceptually very similar to the CQL query, as the condition between braces corresponds to the one declared in the *where* clause.

Table 2. Comparison with CQL

	SensorScript	**CQL**
1.	`presentation{starttime > '12:00'}`	`Select * From Ongoing Where starttime > '12:00'`
2.	`track{name.in('tr1', 'tr2')}/ attendance{checkintime > now() - 3600}.sum(1)`	`Select Count(*) From attendance[Range 1 Hour] Where trackname In ('tr1', 'tr2')`
3.	`presentation{starttime + duration > now()}`	`Select * From Ongoing Where pres_id Not In (Select * From Over)`
4.	`presentation{startime + duration > now()}/attendee`	`Select name, state From Present [Partition By attendee_id Rows 1] Where attendee_id Not In (Select * From Absent)`
5.	`presentation{(status='ongoing' ; status='over', '35 min')}`	`Select Istream(Over.pres_id) From Over[Now], Ongoing[Range 35 Min] Where Over.pres_id = Ongoing.pres_id`
6.	`presentation{status='over'}/ attendee.min(age)`	`Select Istream(Over.pres_it, A.age) From Over[Now], (Select attendee_id, age From attendee) [Partition by pres_id Rows 1] as A Where Over.pres_id = A.pres_id`

2. This second query aims at maintaining a running count of attendees to tracks 1 and 2 over the last hour. There is an important difference here as time windows can only be specified within conditions in SensorScript. This results in two conditions specified over the two sub-selections of the whole selection.
3. With this query we want to maintain a list of the current presentations. The main difference here is that SensorScript relies on attributes updated with the data stream over the nodes of the multitree, where CQL backs to table-like streams to manage the presentation state (*ongoing* or *over*).
4. Given that we want here to list the present attendees, we only need to add a sub-selection to the previous query with SensorScript, considering that a present attendee is an attendee that checked in a current presentation. On the other hand, CQL proposes a whole new, though significantly longer, query, based once again on streams that reflect presence or absence of attendees.
5. As presentations can be rescheduled during the conference, we consider now that speakers check-ins affect directly the state of presentations. This allows us to get a list of non-keynote presentations, as we can follow the presentations that started then stopped in a window range of less than 35 minutes.

We see here that the pathfinding mechanism of the language allows to get rid of any explicit join condition.

6. This last query keeps the age of the youngest speaker for completed presentations. An interesting point here relies on the multitree structure. As we saw in figure 1, the graph follows two routes from attendee to attendance, depending on whether the attendee is the speaker or assists to the considered presentation. In fact, the multitree allows *partially ordered sets* (or *posets*) in the graph, as long as absolute order can be decided between every couple of types from a *poset*. Actually, we rely on this property here to get the list of speakers. When following the orientation of connections between types, the *nearest* matching nodes are selected. Therefore, for presentations *attendee* nodes, this is the speaker. That said, if one wants to look at the age of the youngest audience member of completed presentations, this can be done with the following query, as *attendance* nearest *attendee* nodes are accessed through the participant context rather than the speaker context:
`presentation{status='over'}/attendance/attendee.min(age)`
In comparison, CQL requires both a nested condition and an explicit join.

As we can see, SensorScript expressions minimize the concepts that are specific to the language. In fact, selections, accesses and conditions are specified by operators rather than english words. Moreover, it simplifies multi-stream selections based on the implicit link provided between nodes by the multitree, compared to union specifications of SQL. Finally, conditions and timed conditions are expressed the same way, as the language was designed to implement them, whereas CQL introduces a new syntax dedicated to time windows.

6.2 Rooms lighting scenario

We propose here to consider the whole model from figure 1 in order to orchestrate all the sensors, aiming at automating the conference rooms lighting management. These different sensors allow to monitor light, participants presence and power consumption. The room lighting management addresses a typical problematic of home automation, which brings our example closer to both [17] and [11] examples. We also consider that blinds and powermeters are equipped with actuators. This will allow us to illustrate SensorScript's actions specification in order to address functions of these actuators.

In this scenario we will consider that ambient light of the conference has to be adapted according to several concerns:

– First of all, to save energy we would like the lighting of room to automatically turn off when all attendees have left it:

```
/room{!has(attendee)}/lighting/powermeter.turnoff()
```

We see in the model that two paths exist between the *room* and *attendee* types. However, only one condition is required here. In fact, on one hand, if a speaker is found using the shortest path, the condition is false without

having to check the assistance using the longer path. On the other hand, if the room has no speaker but still some people in the audience, listing these attendees will be the only path existing within the model, therefore it will be the shortest one. This saves us having to express and test the two different accesses within the condition.

- Second, for rooms with open blinds, when the daylight (measured by an outside light sensor named *daylight* for each room) falls below a certain threshold, we want the blinds to close and, if the room is not empty, the inside light to turn on. This is provided by the following two queries:

```
daylight{light < out_threshold}/blinds.close()
daylight{light < out_threshold}/attendee/room/lighting/
    powermeter.turnon()
```

  As a non-empty room is a room bound to at least one attendee, it is more efficient here to use implicit filtering on sub-selections (the `/attendee/room/` part of the query) rather than an explicit condition on the rooms.
- As ambient light, *i.e.* the light measured within a room, can incommode the readability of projected slides during a presentation, the following two queries propose to close the blinds, if open, and turn off the lights, if required, when a presentation starts:

```
presentation{status = 'scheduled' ; status = 'ongoing'}/
    blinds{status = 'closed'}.close()
presentation{status = 'scheduled' ; status = 'ongoing'}/
    lighting/powermeter{status = 'on'}.turnoff()
```

- Finally, we want the blinds to open or the light to turn on, according to daylight, when a presentation is over:

```
presentation{status = 'ongoing' ; status = 'over'}/
    daylight{light > out_threshold}/blinds.open()
presentation{status = 'ongoing' ; status = 'over'}/
    daylight{light <= out_threshold}/lighting/
    powermeter.turnon()
```

As we saw, actuators functions are called as methods on nodes within the language. The method specification is provided through inheritance over the *Node* class in the system, which implements the multitree nodes. For instance, listing 2 illustrates the way to specify the *open* method for blinds. The system is able to detect classes that extend the *Node* class, which allows it to use these classes to instantiate according typed nodes, *blinds* in this example.

Listing 2. Action specification example

```
public class Blinds extends Node {
[...]
  @SensorScriptMethod
  public boolean open() {
    // call to actuator switch to open the blinds
  }
}
```

6.3 Limitations

CQL [1], TinyDB [12] and Esper [6] take advantage of SQL to address both dynamic and persistent data. Considering SensorScript is designed over data streams only and ensures real-time processing of data, we could not afford to keep a history over the data. In fact, even timed conditions do not require a history to be checked. As we only have to make sure that the condition holds for the specified time, this is the unique information to keep during the lifetime of the condition, which is also discarded as soon as the condition is unsatisfied or the time is over. However, we do not aim at replacing traditional DBMS, that can be used in parallel, whether storing the whole data stream or data prefiltered by SensorScript.

Languages as AmbientTalk [19], REScala [15] or EventCJ [8] aim at integrating event specification into existing programming paradigms. In this sense, their scope extends far beyond the one studied here, as we focus on the multitree only as the underlying model of the language. Nevertheless this limitation is what gives SensorScript its concise language based on implicit model parsing.

7 Conclusions and Future Work

We presented here the evolution of SensorScript towards a language for complex event processing dedicated to sensor networks. While the model mainly relies on previous works, we highlighted how the new language builds on the multitree in order to provide complex event processing mechanisms. We are able to balance the syntactic concision of the language with a real-time complex event processor for sensor networks. By providing flexible selections over the nodes, with the possibility to filter them on complex conditions, possibly over a time window, we offer a strong alternative to traditional SQL used in the literature. Moreover, SensorScript does not focus only on data access. In fact it provides the possibility to widen the scope of the methods accessible on nodes to other features than sensors monitoring, including but not limited to addressing actuators functions. Finally we showed that SensorScript is able to address examples proposed in the literature, with simpler results than SQL, while highlighting its limitations, especially on history management.

Future works will focus on deploying SensorScript over a sensor network spread over two distant sites. This will allow us to test both scalability and performance. Another lead would focus on interfacing with a traditional DBMS in order to integrate history management.

References

1. Arasu, A., Babu, S., Widom, J.: Cql: A language for continuous queries over streams and relations. In: Database Programming Languages. pp. 1–19. Springer (2003)
2. Boag, S., Chamberlin, D., Fernández, M.F., Florescu, D., Robie, J., Siméon, J., Stefanescu, M.: XQuery 1.0: An XML query language (2002)

3. Clark, J., DeRose, S., et al.: Xml path language (xpath) version 1.0 (1999)
4. Cugola, G., Margara, A.: Tesla: a formally defined event specification language. In: Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems. pp. 50–61. ACM (2010)
5. David, P.C., Ledoux, T.: Wildcat: a generic framework for context-aware applications. In: Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing. pp. 1–7. ACM (2005)
6. EsperTech: Esper. `http://www.espertech.com/esper` (2015)
7. Garnier, A., Pottier, R., Menaud, J.M.: Sensorscript: a domain-specific language for sensor networks. In: International Conference on Future Internet of Things and Cloud (FiCloud-2015)
8. Kamina, T., Aotani, T., Masuhara, H.: Eventcj: a context-oriented programming language with declarative event-based context transition. In: Proceedings of the tenth international conference on Aspect-oriented software development. pp. 253–264. ACM (2011)
9. Klyne, G., Carroll, J.J.: Resource description framework (RDF): Concepts and abstract syntax (2006)
10. Kwon, O., Song, Y.S., Kim, J.H., Li, K.J.: Sconstream: A spatial context stream processing system. In: Computational Science and Its Applications (ICCSA), 2010 International Conference on. pp. 165–170. IEEE (2010)
11. Li, F., Sehic, S., Dustdar, S.: Copal: An adaptive approach to context provisioning. In: Wireless and Mobile Computing, Networking and Communications (WiMob), 2010 IEEE 6th International Conference on. pp. 286–293. IEEE (2010)
12. Madden, S.R., Franklin, M.J., Hellerstein, J.M., Hong, W.: Tinydb: an acquisitional query processing system for sensor networks. ACM Transactions on database systems (TODS) 30(1), 122–173 (2005)
13. Perera, C., Zaslavsky, A., Christen, P., Georgakopoulos, D.: Context aware computing for the internet of things: A survey. Communications Surveys & Tutorials, IEEE 16(1), 414–454 (2014)
14. Pottier, R., Menaud, J.M.: Btrscript: a safe management system for virtualized data center. In: ICAS 2012, The Eighth International Conference on Autonomic and Autonomous Systems. pp. 49–56 (2012)
15. Salvaneschi, G., Hintz, G., Mezini, M.: Rescala: Bridging between object-oriented and functional style in reactive applications. In: Proceedings of the 13th international conference on Modularity. pp. 25–36. ACM (2014)
16. Shaeib, A., Cappellari, P., Roantree, M.: A framework for real-time context provision in ubiquitous sensing environments. In: Computers and Communications (ISCC), 2010 IEEE Symposium on. pp. 1083–1085. IEEE (2010)
17. Textor, A., Meyer, F., Thoss, M., Schaefer, J., Kroeger, R., Frey, M.: An architecture for semantically enriched data stream mining. In: Proceedings of the First International Conference on Data Analytics, S. Bhulai, J. Zernik, and P. Dini, Eds., Barcelona, Spain (2012)
18. Teymourian, K., Paschke, A.: Enabling knowledge-based complex event processing. In: Proceedings of the 2010 EDBT/ICDT Workshops. p. 37. ACM (2010)
19. Van Cutsem, T., Mostinckx, S., Boix, E.G., Dedecker, J., De Meuter, W.: Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks. In: Chilean Society of Computer Science, 2007. SCCC'07. XXVI International Conference of the. pp. 3–12. IEEE (2007)
20. W3C: SPARQL 1.1 Overview. `http://www.w3.org/TR/sparql11-overview/` (2013)