spChains: A Declarative Framework for Data Stream Processing in Pervasive Applications

Dario Bonino[*], Fulvio Corno

Politecnico di Torino, Dipartimento di Automatica ed Informatica - Corso Duca degli Abruzzi 24 - 10129 Torino, Italy

Abstract

Pervasive applications rely on increasingly complex streams of sensor data continuously captured from the physical world. Such data is crucial to enable applications to "understand" the current context and to infer the right actions to perform, be they fully automatic or involving some user decisions. However, the continuous nature of such streams, the relatively high throughput at which data is generated and the number of sensors usually deployed in the environment, make direct data handling practically unfeasible. Data not only needs to be cleaned, but it must also be filtered and aggregated to relieve higher level algorithms from near real-time handling of such massive data flows. We propose here a stream-processing framework (spChains), based upon state-of-the-art stream processing engines, which enables declarative and modular composition of *stream processing chains* built atop of a set of extensible *stream processing blocks*. While stream processing blocks are delivered as a standard, yet extensible, library of application-independent processing elements, chains can be defined by the pervasive application engineering team. We demonstrate the flexibility and effectiveness of the spChains framework on two real-world applications in the energy management and in the industrial plant management domains, by evaluating them on a prototype implementation based on the Esper stream processor.

Keywords: Complex Event Processing, Stream Processing, Sensor Data, Pervasive Applications

1. Introduction

Pervasive and Ubiquitous systems share a common vision of small, distributed, networked devices distributed at all scales, from body-level networks to smart environments, possibly spanning across the home boundaries. These devices interact with each other and with the surrounding environment to reach specific goals, e.g., setting the comfort level of a smart home depending on the home inhabitants, selecting the best route across jammed streets, etc. Such inter-device interaction might happen regularly, or on a sporadic basis, generating an ever increasing electronic traffic involving sensor measurements, device activations, sensor detections, etc. As the engineering and miniaturization of smart devices proceeds, the amount of generated information increases, requiring pervasive applications, and networks, to successfully and effectively handle data (maximum time frames may vary from few milliseconds to seconds or minutes, depending on the application).

For small sets of devices and sensors, data can be handled directly in near real-time, especially where the term "near real-time" refers to time frames of the order of seconds, e.g., in smart homes or energy grids. On the other hand, in all settings where the number of involved devices is relevant, or the interaction frequency is high (up to real-time systems), direct handling of data by pervasive applications easily becomes not convenient. First, direct elaboration of field-data requires high data throughput, adding strict requirements to applications whose focus is on completely different issues, e.g., user location or context detection. Second, binding direct field-data elaboration to single pervasive applications causes a proliferation and duplication of functionalities, making the approach less scalable and preventing optimization.

[*]Corresponding author.
Email addresses: `dario.bonino@polito.it` (Dario Bonino), `fulvio.corno@polito.it` (Fulvio Corno)

Complex Event Processing (CEP) [1, 2, 3, 4] has proved to be a viable solution for similar issues in the Business Process Management (BPM) and Operational Research (OR) fields [5, 6], by being able to extract meaningful and actionable information from continuous event streams: typical CEP engines [7, 8] are able to deal with data rates between 1,000 to 100k messages per second. CEP is specifically designed to provide applications with a flexible and scalable mechanism for constructing condensed and refined views of data. It relies on a number of techniques involving event-pattern detection, event abstraction, event hierarchies, and so on, and it has reached a rather high maturity with several tools already available and ready to be integrated as data processing layers. Most CEP engines require the manual definition of elaboration and detection patterns (queries), in an SQL-like syntax (e.g., as in [9, 10]) whose particular features depend upon the adopted CEP engine. This prevents non-expert stakeholders, e.g., energy managers or interaction designers, from directly defining data handling procedures.

We foresee an ever increasing role of CEP techniques in pervasive and ubiquitous applications enabling effective abstraction, filtering and dimensional reduction of sensor and device-level events. In such a context, we aim at offering a re-usable, high-level and modular solution to enhance CEP query writing for non-expert stakeholders.

We propose a stream-processing framework (spChains), built upon state-of-the-art stream processing engines, which enables declarative and modular composition of *stream processing chains* built atop of a set of extensible *stream processing blocks*. Each *stream processing block* encapsulates a single (parametrized) stream query, e.g., windowed average or threshold checking, and can be combined (cascaded) to other blocks to obtain complex elaboration chains, using a filter-and-pipe [11] composition pattern. Stream processing blocks are predefined in a library and designed to be application-independent, and they can readily be re-used in almost any processing task. Chains (i.e., connected sets of blocks), instead, are defined by the pervasive application engineering team (possibly non-expert in CEP query writing) and they enable complex filtering, elaboration and aggregation of data flows, thus relieving the pervasive application core from the heavy burden of near real time data handling. The general spChains architecture is implemented and made available as an open source Java library with a specific binding for the Esper stream processing engine; however, its design is independent from the underlying CEP system and can be based on custom built processing modules or may integrate different processing engines in the same application. We demonstrate the effectiveness of the spChains framework through a set of use cases based on real-world requirements in the energy management and in the industrial plant management domains.

The remainder of the paper is organized as follows: Section 2 introduces the spChains framework and the related components. Section 3 describes the spChains implementation and Section 4 presents first results for real-world use cases where the spChains framework has successfully been applied. Section 5 reports related works and, finally, Section 6 draws conclusions and proposes future works.

2. The spChains framework

The spChains framework supports the elaboration, combination and abstraction of environmental data coming from multiple sources (i.e., sensors) through chains of modular and re-configurable processing elements called stream processing blocks (see Figure 1).

On one hand (Figure 1, right), pervasive applications (listening to events delivered by *event drains*) are relieved from the heavy burden of data handling, and they only need to define the aggregation and detection patterns (in form of *stream processing chains*). On the other hand (Figure 1, left), spChains can perform correlation and elaboration of heterogeneous data flows conveyed by the underlying pervasive communication infrastructure (typically wireless sensor networks) and abstracted in form of *event sources*. The inner spChains architecture respects the well-known filter and pipe pattern: each component (filter), i.e., each *stream processing block*, has a set of inputs and outputs. The component reads streams of data on its inputs and provides streams of data on its outputs. A connector (pipe) conveys the stream data from one block output to another block input. The overall data flow starts from a source (*event source*) and reaches a sink (*event drain*) through a set of pipes and filters, thus forming an acyclic graph (avoiding convergence issues related to cyclic processing). Event sources and drains abstract sensors and pervasive applications, respectively, defining a standard way (Java interfaces in the provided implementation) of pushing/extracting events in/from the spChains framework, while blocks and chains realize the core data processing.

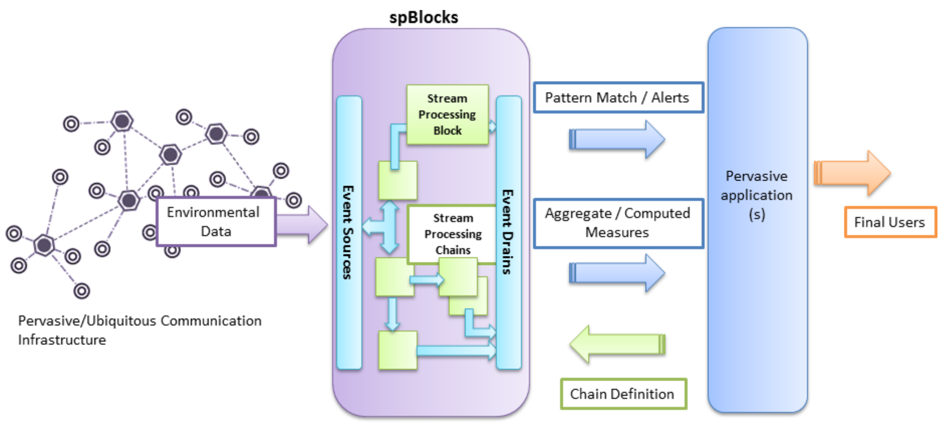


Figure 1: The spChains framework logic architecture.

2.1. *Stream Processing Blocks*

We define a *stream processing block* as a (software) component taking one or more event streams in input and generating one or more event streams as output. The output and input streams are correlated by means of a processing function, which, in general, is not linear (e.g., threshold) and/or with memory (e.g., a moving average). A block is:

- *transparent* if the block only filters out a subset of entering events, depending on some block parameter, while it is called *opaque* if it computes/generates new event values;
- *linear* if the processing function only applies linear operators to the input stream, while it is *not linear* otherwise;
- *with memory* if the computations made for generating the output stream depend on past events, while it is *without memory* if the processing function does not depend on past events.

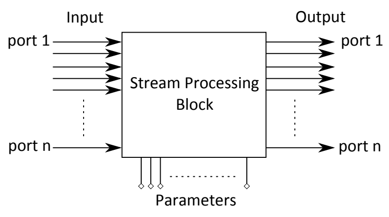


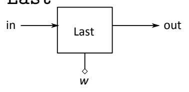
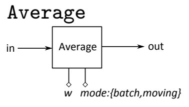
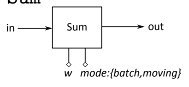
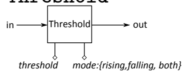
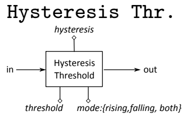
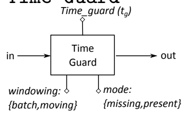
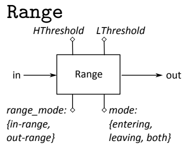
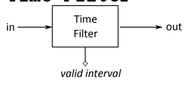
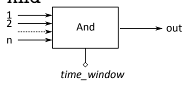
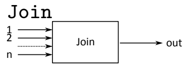
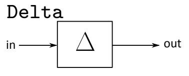
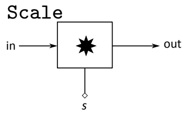
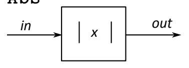
Figure 2: Generic Stream Processing Block

Figure 2 shows the general structure of a stream processing block. A stream processing block has a set of input ports, and a set of output ports, identified by unique port identifiers (mnemonic strings). Every port can only handle a specific type of event, i.e., it has an associated data-type, either real-valued or Boolean, that shall match the type of events received (generated) in input (output). A set of constant parameters can be defined to affect/tune the inner block functionality, e.g., values, window lengths, operating modes. Temporal computations performed by blocks can either be based on moving or batch windows. A moving window is a time window extending to the specified time interval into the past while a batch window buffers events and releases them every specified time interval in one update.

spChains blocks do not aim at representing the complete set of elaborations enabled by full CEP engines, instead they are focused on providing a flexible, reusable and easy-to-learn processing facility for non-experts, while retaining CEP optimization in single block implementations. Any block translation overhead is restricted to the initial composition phase while at runtime blocks perform computations as normal CEP queries thus achieving performances comparable to direct query writing.

Currently 11 basic blocks have already been defined, whose functionalities and features are briefly summarized in Table 1, while more complex boxes can be easily designed and integrated. Blocks can be instantiated by the pervasive application developer through a simple XML-based notation, called spXML (Figure 3).

Table 1: The spChains stream processing blocks.

Block name	**Type**	**Description**	**#IN**	**#OUT**	**#P**
Last 	transparent, linear, with memory	Given the time window *w*, whenever *w* expires this block provides as output the last received event within the window. Every time a filtering window expires, events in the window are dropped and a new batch filtering is started.	1	2	1
Average 	opaque, non-linear, with memory	It computes the average of all (real-valued) events received in a given time window *w*. It can either operate in *batch* or *moving* mode.	1	2	2
Sum 	opaque, linear, with memory	The Sum block is an opaque block computing the sum of all events received in a given time window *w*. It can either operate in *batch* or *moving* mode.	1	2	2
Threshold  in Threshold out threshold mode:{rising,falling, both}	opaque, non-linear, with memory	Generates a stream of Boolean events by monitoring one real-valued event stream for specific threshold passing. The Threshold block can work in 3 different threshold checking modes: rising (>), falling (<) and both.	1	1	2
Hysteresis Thr.  hysteresis in Hysteresis Threshold out threshold mode:{rising,falling, both}	opaque, non-linear, with memory	The Hysteresis Threshold block acts almost like the threshold block except for the threshold traversal detection, which exploits a tunable hysteresis parameter to smooth frequent near-threshold triggers that might make the output signal too unstable.	1	1	3
Time Guard  Time_guard (t_g) in Time Guard out windowing: {batch,moving} mode: {missing,present}	opaque, non-linear	Generates a Boolean event stream by monitoring a stream of Boolean or real-valued events for time limit/frequency compliance. It has two operating modes: *missing* and *present*.	1	1	3
Range  HThreshold LThreshold in Range out range_mode: {in-range, out-range} mode: {entering, leaving, both}	opaque, non-linear, with memory	The Range block is an opaque, non-linear block that checks real-valued input events against a range of accepted values (either in-range or out-range checking is supported).	1	1	4
Time Filter 	transparent, linear	The Time Filter is a linear, transparent block which acts as a time-based switch, allowing incoming events to pass through depending on the current time.	1	1	1
And 	opaque, non-linear, without memory	The And block is a non linear and opaque block which acts as a multi-port time guard. Given a time window t_w, usually short (around few seconds or less) the block provides one output event (Boolean) *iff* all input ports have received at least one event in t_w.	n	1	1
Join 	transparent, linear, without memory	The Join block is a linear, transparent block that multiplexes input events on different channels into a single output channel. It works with an event based paradigm: whenever an event arrives on any input port, it is automatically forwarded to the output port.	n	1	0
Delta 	opaque, linear, with memory	The Delta block is a linear, opaque block that computes the difference between pairs of consecutive events arriving on the block real-valued input port. Events participating in the difference are discarded one at time, i.e., the block works with a moving sampling window having a width of 2 samples.	1	1	0
Scale 	opaque, linear, without memory	The Scale block is a linear, opaque block that scales the value of input events by a given multiplying factor *s* defined as block parameter.	1	1	1
Abs 	opaque, not-linear, without memory	The Abs block is an opaque block, without memory, that simply provides as output (real-valued, positive) the absolute value of incoming real-valued events.	1	1	1

#IN = number of input ports, #OUT = number of output ports, #P = number of parameters.

```
<spXML:block id="Avg1" function="AVERAGE">
  <spXML:param name="window" value="1" unitOfMeasure="h" />
  <spXML:param name="mode" value="batch" />
</spXML:block>
```

Figure 3: Block instantiation in spXML.

3. Implementation

The spChains framework is implemented as an open source Java library, distributed with an Apache v2.0 license. It provides an abstract implementation of the logical modules of the architecture together with all the utilities needed to automatically verify and establish block connections as well as source-to-chain and chain-to-drain communications.

Starting from such abstract classes, we implemented the 11 blocks discussed in Section 2 by exploiting a state-of-the-art CEP engine called Esper [8]. Esper supports effective handling of up to 100k events per second, thus contributing a suitable performance to the spChains implementation. Block implementations are discovered at run-time by exploiting the Java Services pattern [12], which allows to easily provide new block implementations without affecting the core spChains library. Handling of real-events representing physical quantities, with explicit unit of measures is supported through the JScience(`http://jscience.org/`) library, which implements the JSR-275 standard Java specification.

Sources, drains, chains and blocks involved in pervasive applications are instantiated by simply writing a specific spXML description of the needed processing chains. Figure 4 shows the spXML specification of the chain reported in Figure 5(c) and istantiated in the context of the real-world experimentation with Eudata (see Section 4).

4. Experimentation

The spChains framework has been functionally tested in laboratory and then applied on 2 real-world case studies (in collaboration with 2 different software companies) respectively involving building energy management and industrial plant management and energy cost detection. In-laboratory experimentation involved two main steps: single block testing, to verify the actual implementation of desired processing tasks, and complex chain composition testing, to verify the ability to instantiate, connect and operate several stream processing chains, composed by a reasonable number of blocks. All the 11 blocks have been functionally tested and they have been (re)used to assemble and test over 48 different processing chains, with lengths going from 1 to 5 different processing blocks. The 48 chains were run simultaneously on a test machine (Intel Core i5, 4GByte of RAM) and they were able to support elaboration of 2 simultaneous event streams each delivering a new event every 2 ms.

4.1. Energy management

spChains is currently part of the “Greeny” service offered by Eudata[1]. Greeny provides multi-building, multi-sensor energy-consumption monitoring for energy managers. In the context of this project, spChains acts as data aggregation layer and performs real-time computation of non-direct measures, e.g., consumed calories given the temperature of the water incoming in the heating system and coming back from the building heaters. One new block has been developed to completely support the set of computations required by the project: the 2-channel sum, where events arriving on two input ports are summed iff they have a really small delay (less than 0.1s). The first integration of spChains and Greeny ran for 3 months without downtimes. Results were checked for consistency by Eudata, confirming the soundness of the provided implementation and the effectiveness of block-based event processing.

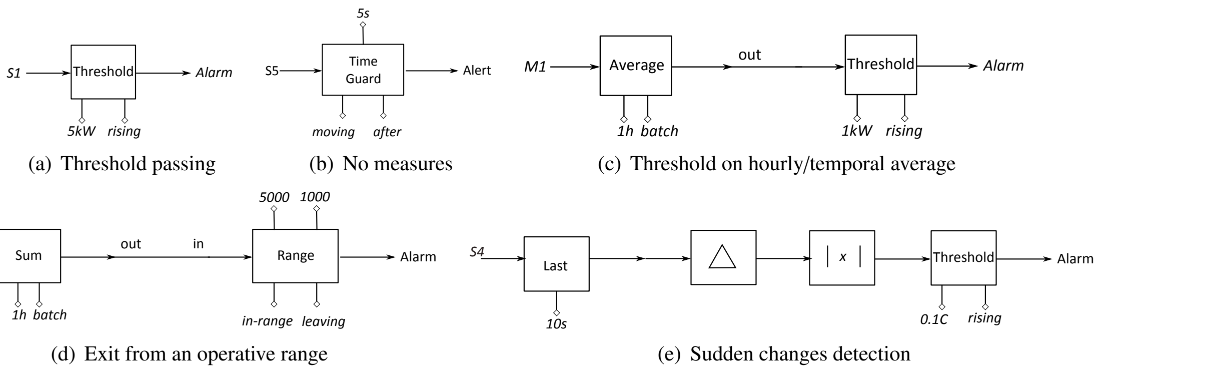
4.2. Industry plant management

spChains is also adopted by the JEERP (Java Energy-aware ERP) project, currently carried by Proxima Centauri[2], developer of the Oratio[3] open source ERP. In the context of the JEERP project, spChains is adopted as data aggregation

[1]http://www.eudata.biz/
[2]http://www.proxima-centauri.it
[3]http://www.oratio.it/

```
<spConfig:streamProcessingConfiguration>
 <spConfig:chains>
  <spXML:chain id="usecase1">
   <spXML:blocks>
    <spXML:block id="Avg1" function="AVERAGE">
     <spXML:param name="window" value="1" unitOfMeasure="h" />
     <spXML:param name="mode" value="batch" />
    </spXML:block>
    <spXML:block id="Th1" function="THRESHOLD">
     <spXML:param name="threshold" value="1" unitOfMeasure="kW" />
    </spXML:block>
   </spXML:blocks>
   <spXML:connections>
    <spXML:connection>
     <spXML:from blockId="Avg1" port="out" />
     <spXML:to blockId="Th1" port="in" />
    </spXML:connection>
   </spXML:connections>
   <spXML:input blockId="Avg1" port="in" id="in" />
   <spXML:output blockId="Th1" port="out" id="out" />
  </spXML:chain>
 </spConfig:chains>
 <spConfig:eventSources>
  <spConfig:eventStream id="M1" type="REAL"/>
 </spConfig:eventSources>
 <spConfig:eventDrains>
  <spConfig:eventStream id="Alarm" type="BOOLEAN"/>
 </spConfig:eventDrains>
 <spConfig:connections>
  <spConfig:connection>
   <spConfig:fromSource chainId="usecase1" inputId="in"
     source="M1" />
   <spConfig:toDrain chainId="usecase1" drain="Alarm"
     outputId="out" />
  </spConfig:connection>
 </spConfig:connections>
</spConfig:streamProcessingConfiguration>
```

Figure 4: Threshold on hourly average use case in spXML, see Figure5(c) for the corresponding block diagram.



(a) Threshold passing (b) No measures (c) Threshold on hourly/temporal average

(d) Exit from an operative range (e) Sudden changes detection

Figure 5: JEERP use cases.

and event pattern (alert) detection layer. While aggregation in JEERP is trivial as it only requires performing batch averages and delta computations on energy consumption data fed by field sensors, alerts are interesting, as they require more complex block compositions. Five alert use cases have been defined in the project: a) *Threshold passing*, e.g., if measurements coming from sensor *S1* exceed a predefined value (e.g., 5kW) raise an alarm; b) *No measures*, e.g., if sensor *S5*, which is configured to send new measures every 2s does not provide any measure for more than 5s, raise an alarm; c) *Threshold on hourly/temporal average*, e.g., if the hourly average of data coming from sensor *S1* exceeds the value of 37.2 $°C$ raise an alarm; d) *Exit from an operative range*, e.g., given the temperature data measured by sensor *S3*, raise an alarm if the temperature exits from the operative range $0 - 50$ $°C$; e) *Sudden changes detection*, e.g., given the flow data measured by sensor *S4* detect if in the last 10s the measured value is changed by more than $0.1\frac{m^3}{s}$. All of them have been successfully addressed with the provided block implementations, as summarized in Figure 5, and corresponding chains are currently integrated into the JEERP data handling layer.

5. Related Works

Effective data handling and management is attracting an increasing interest in the pervasive and ubiquitous computing community, due to the constantly growing amount of distributed sensors and devices participating in pervasive applications. As the order of magnitude of deployed devices increases, pervasive applications are loosing the ability to directly handle real-time data stream and new, data-centric techniques are being developed to offload this computation from (pervasive) applications to some kind of middleware having the computational capabilities to handle such massive data flows. In this context, the Solar middleware [13] provides a data-centric infrastructure to support context-aware applications, exploiting the filter-and-pipe [11] pattern. Similarly to the proposed spChains, Solar treats sensors as data stream publishers and applications as data stream consumers. Application developers explicitly compose desired sensor streams and transform low-level data into more meaningful context using operators (comparable to the the spChains Stream Processing Blocks). Solar defines operators as custom developed solutions ranging from simple logic AND to complex supervised machine learning whereas spChains exploits highly efficient stream processing engines, such as Esper, while also permitting non-stream solutions as the ones envisioned in Solar. Moreover, while Solar defines the infrastructure, leaving developers free to design and implement the needed processing operators, spChains aims at providing a standard, yet extensible, set of basic processing blocks ensuring re-usability of the approach across different application domains.

Obweger et al., in their "CEP Off the Shelf" paper [14] tackle this issue by proposing solution templates based on the SARI event processing framework. Solution templates offer well-proven, standardized event processing logic for common business needs with an underlying rationale pretty similar to the spChains motivations. Similarly to spChains, solution templates can be assembled from pre-defined, easy-to-use building blocks, in a way that abstracts from the underlying complexity. However, compared with spChains, templates are defined in terms of if-then-else rules, somewhat less expressive than the block composition offered by spChains.

In the Jeffery et al. [9] approach, CEP techniques are exploited for building sensor data cleaning infrastructures for pervasive applications. In the Extensible Sensor stream Processing (ESP) framework they propose, sensor data is cleaned by means of a pipeline defined through declarative mechanisms based on spatial and temporal data features. Two main differences and shortcomings can be identified with respect to spChains. First, processing components are defined as CQL [15] queries (supported by the STREAM [3] processing engine), which are difficult to compose and deploy for people without a deep knowledge of stream-processing languages, and that are difficult to reuse by being optimized for specific purposes, only. Second, the domain of application is much more restricted than the one targeted by spChains: while ESP is focused on data cleaning, spChains defines a general framework and a set of standard blocks that can be easily re-used and composed into application-specific processing chains.

6. Conclusions

We presented spChains, a declarative framework for data stream processing in pervasive applications. Thanks to the availability of a standard, yet extensible, set of 11 basic building blocks and to a flexible and easy-to-reuse composition mechanism based on block chains, with spChains applications can program their own event processing layer, without dealing with CEP query composition and without relying on a single CEP implementation. The Esper-based spChains implementation has been functionally tested and successfully applied to two real-world, commercial applications and is currently undergoing a more extensive test and deployment campaign. Results from pilot installations (Eudata and JEERP) confirm the approach feasibility and the actual capability of non-expert stakeholders to design and instantiate rather complex elaboration chains.

References

[1] D. C. Luckham, The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S. Zdonik, Aurora: a new model and architecture for data stream management, The VLDB Journal 12 (2003) 120–139. doi:10.1007/s00778-003-0095-z.

[3] T. S. Group, Stream: The stanford stream data manager, Technical Report 2003-21, Stanford InfoLab (2003). URL `http://ilpubs.stanford.edu:8090/583/`

[4] O. Etzion, P. Niblett, Event Processing In Action, Manning Publications and co., 2010.

[5] I. Kellner, L. Fiege, Viewpoints in complex event processing: industrial experience report, in: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems, DEBS '09, ACM, New York, NY, USA, 2009, pp. 9:1–9:8.

[6] C. Zang, Y. Fan, Complex event processing in enterprise information systems based on rfid, Enterprise Information Systems 1 (1) (2007) 3–23.

[7] N. Dindar, B. Güç, P. Lau, A. Ozal, M. Soner, N. Tatbul, Dejavu: declarative pattern matching over live and archived streams of events, in: Proceedings of the 35th SIGMOD international conference on Management of data, SIGMOD '09, ACM, New York, NY, USA, 2009, pp. 1023–1026.

[8] S. Oberoi, Esper Complex Event Processing Engine, Embedded System Engineering 16 (2) (2011) 28–29.

[9] S. R. Jeffery, G. Alonso, M. J. Franklin, W. Hong, J. Widom, Declarative support for sensor data cleaning, in: Proceedings of the 4th international conference on Pervasive Computing, PERVASIVE'06, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 83–100. doi:10.1007/11748625_6.

[10] W. Wang, J. Sung, D. Kim, Complex event processing in epc sensor network middleware for both rfid and wsn, in: Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on, 2008, pp. 165 –169. doi:10.1109/ISORC.2008.59.

[11] D. Garlan, M. Shaw, An Introduction to Software Architectures, Tech. Rep. CMU-CS-94-166, Carnegie Mellon University (January 1994).

[12] J. O'Conner, Creating Extensible Applications With the Java Platform, Tech. rep., Oracle, Java (2007).

[13] G. Chen, M. Li, D. Kotz, Data-centric middleware for context-aware pervasive computing, Pervasive and Mobile Computing 4(2008) (2007) 216–253.

[14] H. Obweger, J. Schiefer, M. Suntinger, F. Breier, R. Thullner, Complex event processing off the shelf - rapid development of event-driven applications with solution templates, in: Control Automation (MED), 2011 19th Mediterranean Conference on, 2011, pp. 631 –638. doi:10.1109/MED.2011.5983141.

[15] A. Arasu, S. Babu, J. Widom, The cql continuous query language: semantic foundations and query execution, The VLDB Journal 15 (2006) 121–142. doi:10.1007/s00778-004-0147-z.