SEVENTH FRAMEWORK PROGRAMME
THEME SECURITY
FP7-SEC-2009-1

Project acronym: *EMILI*

Project full title: Emergency Management in Large Infrastructures

Grant agreement no.: 242438

D4.1
A Survey on IT-Techniques for a Dynamic Emergency Management in Large Infrastructures

Due date of deliverable: 30/06/2010
Actual submission date:

Revision: Version 1

Ludwig-Maximilians University Munich (LMU)

Project co-funded by the European Commission within the Seventh Framework Programme (2007–2013)		
Dissemination Level		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Author(s)	Simon Brodt, Steffen Hausmann, Francois Bry, Olga Poppe, Michael Eckert (meanwhile TIBCO Software Inc.)
Contributor(s)	LMU

Remove the contents below from this page before submission

INTERNAL REVISION: V0.1

(please note that the final draft accepted by the general quality manager will receive the external "Version 1" as submitted to the EU; if the EU requires an improvement, i.e. an official "Version 2", then the drafts for this revision may receive several further INTERNAL revision numbers, to be maintained on this page)

Work package	WP 4: "Semantic Web Technology for Complex Events"
Task(s)	Task 4.1: "State-of-the-Art Analysis of Complex Event Processing"

Quality Manager	
Approval Date	
Remarks	

Internal Review	CWI
Approval Date	
Remarks	

Preface

This deliverable is a survey on the IT techniques that are relevant to the three use cases of the project EMILI. It describes the state-of-the-art in four complementary IT areas: Data cleansing, supervisory control and data acquisition, wireless sensor networks and complex event processing. Even though the deliverable's authors have tried to avoid a too technical language and have tried to explain every concept referred to, the deliverable might seem rather technical to readers so far little familiar with the techniques it describes.

Index

1 Introduction

The main goal of EMILI is the development of a new generation of intelligent data management and control systems in order to improve the security of large infrastructures. This includes a more efficient and reliable detection of critical and emergency situations and automatic as well as semi-automatic reactions of the system which support the operator of the critical infrastructure (CI) in such a situation. The assessment of the CI's condition and the execution of appropriate actions will be mainly based on the sensor readings and whenever reasonable on the output of appropriate simulations which enable more appropriate reactions based on the prediction of the conditions in the near future.

Thus the basis for an intelligent system for emergency management and decision support are good and reliable measurements which describe the physical conditions of the CI and are typically provided by sensors. However, the data gathered by the sensors is inherently noisy and unreliable. The noise is caused by measurement errors of the sensors, transmission and processing errors of the data provided by the sensors or even by people which are trying to confuse the system on purpose by manipulating single sensors. Therefore adequate methods for the cleaning of noisy sensor data are required within EMILI in order to provide reasonable and appropriate results. Section 2 summarizes the problems and possible approaches for cleaning noisy sensor data in the context of EMILI.

Before the assessment of the situation and the decision support based on the sensor data can be carried out, the data needs to be collected from the sensors. Supervisory Control and Data Acquisition (SCADA) systems and (wireless) sensor networks are two different approaches which perform this task. A SCADA system is basically used to integrate the data of various heterogeneous sensors and to provide the operator the overview of the current state or condition of the system based on single sensor values. Typically some kind of traditional hard wired network is used in a SCADA system to send the sensor measurements to the SCADA server. In contrast a wireless sensor network is a self organizing wireless network of small sensor nodes which is particularly suitable for harsh environmental conditions which make the installation of a traditional hard wired network either infeasible or too expensive. However, for our approach both systems can form the basis the collection of sensor measurements. Section 3 and 4 respectively provide an overview of the techniques which are used in SCADA systems and wireless sensor networks to collect the data from the devices of distributed industrial installations and infrastructures.

Based on the sensor data which is either provided by the SCADA system or the wireless sensor network, the single measurements can be combined in order to derive new knowledge on the condition of the CI and based thereon trigger appropriate reactions. In contrast to SCADA systems, not only single measurements are taken into account, but also the combination (and even absence) of multiple events is regarded. These properties of the system will be realized by means of complex event processing (CEP) and event condition action (ECA) rules. However, in order to specify the events and their dependencies which are representing such new knowledge in a way that is appropriate for the CEP engine, they need to be specified in a formal manner, that is a CEP query. The same applies of course for the specification of ECA rules as well. In recent years, a myriad of different CEP languages and engines have been proposed by the research community and the industry. Section 5 contains a summary and categorization of the currently available complex event query languages.

This document is focused on techniques for a dynamic emergency management in large infrastructures which are related to the detection of situations relevant for CIs. It is tightly coupled to the deliverable D3.1 (Use Case Requirement Analysis and Specification) and its annexes as the focus of each topics of this documents is adjusted to the requirements of the described use cases. While this document is regarding the detection of situations, the reaction to detected situations is considered within the deliverable D4.2 (State-of-the-Art Analysis of Event-Condition-Action Rules). It gives an introduction to Event-Condition-Action Rules and their use for emergency management based on real-life examples from the use-cases.

2 Data Cleansing

Data delivered by sensors is typically noisy and unreliable. Limitations of the sensors, measurement errors, poor measurement conditions, parallel activities causing interferences, transmission and processing errors and even malicious manipulations make it necessary to take a careful look at sensor signals. The proper handling of noisy and unreliable sensor signals is an important issue for the EMILI use cases, e.g. the metro use case described in deliverable 3.1 Annex B (See 'Sensors and Signals').

The task of improving, correcting and filtering the incoming data is called *data cleansing* or *data scrubbing*. These terms originally stem from data management in data base systems. The intention of data cleansing in this field is to achieve a consistent database state, where 'consistent' refers to more or less explicit and application dependent data models and consistency specifications. In the following we give an overview of data cleansing for complex event processing in the context of emergency management in general and with particular regard to our use cases. The overview does not claim to be comprehensive. The reason for this is that data cleansing is strongly application dependant and there is no "one size fits all" approach for the kind of data used in emergency management. The use case descriptions in the annexes of Deliverable 3.1, i.e. Annex A for the airport use case, Annex B for the metro use case and Annex C for the power grid use case, identify a number of points where noisy and unreliable sensor signals have to be treated. However the specific goals and tailored methods for data cleansing need further examination within EMILI.

2.1 Origin of Noise and Unreliability

The "noise" or "dirt" which should be eliminated while data cleansing may have different reasons. In the case of sensor signals the following reasons are probably the most common ones:

- **Limitations of Sensors.** The quality of the available sensors is typically limited by technical and economical constraints. Sometimes there are also physical limits to measurements. These limitations of sensors lead to fluttering and spontaneous errors in measurements. This kind of noise is typically stochastically distributed. Many sensor devices already try to eliminate these effects by themselves.
- **Interferences.** In the context of emergency management sensors typically have to work under poor, i.e. in non-laboratory conditions. The surveyed infrastructure is in use during the measurements are taken and thus other activities may interfere with the measurement. In the metro use case for example, one is interested in the current wind speed within a tunnel because this information is needed for controlling the ventilation. Anemometers are able to detect the current air flow at their positions. However the anemometers do not only register the continuous airflow through a tunnel but also local and temporal turbulence caused by moving trains or passengers. These interferences have to be removed to obtain the relevant information, i.e. the wind speed. Note that the sensors work perfectly fine from the point of physical measurement in this case. Therefore sensors are hardly

able to eliminate this kind of noise.

- **Failure of Sensors.** Due to defects, pollution or other reasons sensors fail from time to time. This results in missing or faulty signals. The sensor devices might be able to detect some of the failures. However the emergency management system should also be able to recognize faulty sensors.
- **Human Errors.** Some of the incoming data is not coming from sensors but from some kind of manual input. For example an operator may enter information based on his/her own observations or on a received emergency call. A passenger may press an emergency button or indirectly trigger an emergency signal by removing a fire extinguisher. Depending on the person where the information comes from, it is more or less reliable and may need further validation. Moreover there is always some risk of human errors when entering the information, i.e. typing errors for instance.
- **Manipulation.** Critical infrastructures are always threatened by vandalism, sabotage or even terrorism. Therefore one should account for malicious manipulations of the systems. This is particularly important when data cleansing is considered, because methods doing some kind of filtering are potentially vulnerable for manipulations. See Sec. 2.6 for an example.
- **Transmission Errors.** The communication in an emergency management system usually bases on more or less standard network components and protocols. Therefor the system should be able to cope with data losses due to transmission errors.

Which kinds of “noise” and “dirt” have to be considered strongly depends on the type of the incoming signals and on the way they are used in the application. Temperature values might be checked for impossible values, i.e. values outside of the physically possible range where the range may depend on the actual position of a sensor, for example a temperature below -20°C is quite impossible inside a metro station. Anemometers need some smoothing of the measurements to correctly determine the current wind speed. And optical smoke sensors might want to ignore very short alarms as they might be caused by some piece of dust entering the sensor accidentally.

2.2 Data Quality Criteria

The overall objective of data cleansing is called “data quality”. Data quality is specified depending on the characteristics of the applications considered. The data quality criteria usually considered are the following (in alphabetical order):

- **Accuracy:** Cumulative criterion defined in terms of integrity, consistency and density.
- **Completeness:** Refers to data without anomalies, e.g. no impossible values. The definition is application dependent.
- **Consistency:** States that the data is free of contradictions with respect to application dependent consistency condition or constraints (like an age not being over 120 years).

- **Density:** Ratio of the amount of missing data compared to the total amount of data which was expected
- **Integrity:** Cumulative criterion defined in terms of completeness and validity.
- **Uniqueness:** Refers to the number of (unwanted) duplicates in the data. Note that not all applications reject duplicates.
- **Validity:** Approximate value reflecting the relative amount of data satisfying integrity conditions or constraints.

2.3 Need for Physical Models

In the context of sensor signals the specification of the above criteria, particularly completeness, consistency and density, is closely related to the physics of the surveyed infrastructure. Therefore physical models of the infrastructure are often necessary for data cleansing. In the case of completeness for example a physical model could determine which values are possible at a certain location and under specific conditions. The cross-checking of different sensor signals which is a measure for finding and eliminating inconsistencies also depends on an accurate physical model. Finally physical models, in this case actually physical simulations, are required to meet the density criterion. The reason is that simulations can help to replace missing sensor data, based on the available one. This is an important issue as sensors could be destroyed during an emergency situation like fire (metro and airport use cases) or just are not sufficiently available (power grid use case).

How and which physical models are actually used obviously depends on the application, i.e. the kind and structure of the critical infrastructure and the concrete surveillance and emergency management tasks. Physical models will never reflect all physical aspects of an infrastructure but only those parts which are essential for emergency management. The power grid use case for instance, needs precise models for electricity, but it is not concerned about people movement or smoke and fire propagation which are important for the metro and the airport use case. For simplicity and efficiency, there will typically not be one comprehensive physical model for all physical aspects considered for some infrastructure and the physical model used for a particular aspect may be tailored to the infrastructure and not be designed for maximum generality. Even the way how physical models are brought into emergency management depends on the goal of the integration. The knowledge extracted from the physical model could for example be expressed by constraints (identifying anomalies), in form of logic-style rules (cross-checking) or be coded into a simulation (replacing missing data, prediction).

2.4 The Phases of Data Cleansing

- **Data Analysis / Auditing:** The first and already challenging task for data cleansing is the correct identification of the occurring kinds of “noise” and “dirt” in the data. This can be done on the semantic as well as on the data level. The analysis on the semantic level bases on the meaning and known properties and relationships of the data. This kind of analysis has typically to be done mostly manually and refers to human experience. In the case of sensor signals the meaning, properties and relationships of the data are usually given by physical values and laws. Therefore one important part of the semantic analysis is building appropriate physical models for the data. The analysis on the data level is frequently based on statistical methods which are able to recognize unusual or unlikely data. This is also referred to as outlier detection. Outlier detection has been widely examined on the field of databases. However the research on applying statistical methods to stream data is still in a very early phase.

- **Cleansing specification:** Based on the identification of “noise” and “dirt” in the previous step, a sequence of operations has to be defined to specify a suitable cleansing. This is can be done by a kind of workflow specification which makes it possible to consider partly cleansed data sets for further cleansing. The type of workflow needed depends on the application considered: Cleansing texts does not call for the same processes as cleansing sensor data for fire detection in a metro station or airport and the cleansing for alarm signals from electrical devices in power grids is different again. Cleansing workflows are verified for completeness (all cleansing needed is performed), correctness (the cleansing performed does not introduce inconsistencies), termination and efficiency (the cleansing can be performed in the time imparted on the data sets expected). Software verification methods are used in this task as far as possible. The specification of a data cleansing workflow is a static task, that is, it is performed when data cleansing is implemented.
- **Workflow execution:** The application of the formerly specified cleansing workflow to the data. Additional consistency checks are often performed at run time because a properly specified cleansing workflow, for example for efficiency reasons, does not always preclude inconsistencies.
- **Post-Processing and Controlling:** Final inspection of the cleansed data for checking its correctness. If needed, manual cleansing and/or a new cleansing process might be initiated.

2.5 Dimensions of Data Cleansing

Data Cleansing has different dimensions which are related to the different kinds of “noise” and “dirt” as discussed in Section 2.1.

- **Noise Reduction / Smoothing.** Stochastic noise or noise caused by interferences can often be treated by some kind of smoothing (like a moving average) or other noise reduction methods. The idea is to extract the relevant portion of the incoming signal. The incoming signal is modified but not removed. Thus the risk of loosing relevant information by this kind of cleansing is very low.
- **Filtering.** Faulty data due to sensor failures or human errors can often be recognized by cross-checking information from different sources. This is often based on physical

models. In a metro station for example one could use temperature and smoke sensors to detect fire. Cross-checking the signals from spatially close sensors of both kinds allows to detect unlikely or impossible measurements of a single sensor and could help to avoid false-alarms. However filtering has to be used carefully as it might be vulnerable for manipulations. Consider the following example: Under normal conditions it is physically impossible that one temperature sensor measures great heat, e.g. 800°C, and another sensor in only little distance returns a normal measurement of 20°C. In this situation the system might assume that the first sensor is faulty and decide to ignore its measurements until the sensor has been checked by some maintenance staff. This behavior could be exploited to manipulate the system. A small burner could be used to make the system believe that some of the sensors were broken. After that a real emergency, e.g. fire, is initiated but the system does not react, even though the sensors report the emergency, because the system considers the sensors unreliable.

- **Generation.** Data cleansing may also try to fill gaps in the data due to missing sensors or transmission errors. This task is likely to employ physical models. Filling up missing data can be very useful as it allows to draw an overall picture of the state of an infrastructure (see for example the “State Estimator” in the power grid use case). Such data should be marked as generated, virtual or simulated since it is not as reliable as real measurements.

2.6 Difficulties of Data Cleansing

Data cleansing is an important task for ensuring accurate results when using data from sensors. However data cleansing also introduces some difficulties and has to be applied carefully as it has a great impact on the whole emergency management system. The careless use of data cleansing is able to cause serious risks for an infrastructure.

- **Data losses:** Cleansing might yield losses. This is the most critical point in data cleansing (See the example in Filtering). A trade-off must be strived for between possibly losing data and possibly not sufficiently cleansing data. This trade-off depends on the application.
- **Physical models:** Cleansing of sensor data depends on physical models. Such models may be very simple: a simple speed equation suffices to recognize faulty data reporting the presence of tracked vehicle at a given place and given time. Person or crowd movements, smoke and fire propagation might, but must not, need more involved mathematical models.
- **Efficiency:** When the data to cleanse is to be used in real-time applications, like in emergency management application, then efficiency is an sensitive issue. A trade-off between data quality and cleansing timer must be strived for. The trade-off depends on the application and its time constraints.

2.7 Conclusions

This section provided an overview of data cleansing in general. Regarding EMILI, data cleansing is an issue that mostly depends on the goals, approaches and data collected in the use cases. An “one size fits all” cleansing does not exist so far for the kind of data used in emergency management (as opposed for example to generic well established methods, data sets and algorithms for text cleansing).

Current state-of-the-art SCADA systems are already capable of cleaning sensor data (c.f. deliverable 3.1 annex b and annex c respectively). However, if the data cleaning capabilities provided by the SCADA systems of the use case partners are sufficient for our purposes or whether methods for data cleaning need to be integrated into the CEP engine is still an open question. The benefit of data cleaning inside a CEP engine is that the engine integrates the data of all (sub)system of the infrastructure whereas the data cleaning in a SCADA system is limited to a certain (sub)system. The additional data that is available to the CEP engine might be used to yield better data cleaning results.

For further information on data cleansing see [84, 45, 74, 35, 36, 70, 53, 1].

3 Supervisory Control and Data Acquisition

Supervisory Control and Data Acquisition (SCADA) Systems [9] are used for the monitoring and controlling of large and distributed industrial installation and infrastructures such as factories, manufacturing lines, power plants and networks, oil and gas pipelines, and facilities in transport systems (e.g., airports, train and metro stations). They collect and interpret data from the various devices in such an infrastructure and provide an overview of its current condition to the operator which is responsible for the maintenance and safety of the infrastructure. Furthermore, SCADA systems offer a centralized interface to the operator which allows him to intervene and alter the current processes of the infrastructure's components. Such an manual reaction of the operator is required in case the automatic regulations of the components are not appropriate for a certain situation and thus require adaptations (for instance during an emergency or ongoing maintenance work).

A SCADA system typically consists of the following components [61]:

- **Remote Terminal Units (RTUs)** are connected to the sensors and devices of the system and are in charge of convering the (possibly) analog values of the sensors to digital values. They are also able to receive simple commands and thus alter the state of connected devices.
- **Programmable Logic Controllers (PLCs)** are specialized digital computers that have been designed to operate under the particular conditions of process control such as particular high or low temperatures, vibrations and high humidity. Both RTUs and PLCs are programmable which implies that their behavior can be adopted by exchanging the code that they are running.
- **Master Terminal Unit (MTU)** is connected to the communication infrastructure of the SCADA system and collects all data that is provided by the RTUs.
- **Human Machine Interface (HMI)** presents the data collected by the MTU to the operator. It offers means for the operator to monitor the condition of the infrastructure and to send commands to its devices.
- **Communication Infrastructure**. Messages which are exchanged between the RTUs and the MTU are send over some kind of communication infrastructure, for instance a wireless or hard wired network.

Intelligent RTUs and PLCs are connected to the devices of the infrastructure in order to establish an interface between the physical devices and the SCADA system. RTUs and PLCs typically serve two main purposes, namely to gather the measurements which are describing the current condition of the device they are connected to and the regulating the operation of the devices.

Under normal conditions, the SCADA system works on a supervisory level, that is, it does not influence the behavior of single components. Instead the RTUs and PLCs are responsible for the automatic execution of control actions that are regulating the operation of the physical devices of the infrastructure. However, in case an operator needs to intervene in the behavior of one or multiple devices, for instance when something goes wrong and the (predefined) reactions of the RTUs and PLCs are no longer appropriate for the new situation, the operator can use the HMI of the SCADA system to override the controls of the RTUs and PLCs and thus influence the situation as it is required to cope with the changed conditions.

In addition to the supervisory methods which are provided by SCADA systems, they offer means for trending and analytical processing [27]. To this end a log of respectively an archive of all measurements, alarms and taken actions is stored in a database for the subsequent off-line analysis.

SCADA systems face issues of heterogeneity. Various different devices from different manufacturers may be employed in an infrastructure which leads to a plurality of different and often proprietary protocols that have to be supported by the SCADA system. Recently, this issue is approached by building on open, Web-based standards for data formats and communication protocols. All the proprietary protocols which appear in an infrastructure are translated to a single, uniform and open format which is then used for the communication between the SCADA system and the other components. An example of such an protocol is the Facility Control Markup Language (FCML) [20], which provides an XML format HTTP-based communication.

Complex event processing (CEP) in SCADA systems can be used to detect higher level events which are derived from multiple lower level events or rather measurements. By combining multiple measurements, not only the condition of a single device, but rather the condition of a complete area or component of the infrastructure can be reflected. Since the measurements of various different subsystems are integrated in a SCADA system, a CEP engine based on a SCADA system has a complete view of the system, in contrast to the RTUs and PLCs which are only aware of the components they are directly connected to. This is important since the different subsystems of an infrastructure can easily interfere during an exceptional or emergency situation. For instance during a fire, the heating, ventilating, and air conditioning (HVAC) system can easily interfere with the smoke extraction system.

During an exceptional or emergency situation many components of the infrastructure are likely to report warnings and abnormal measurements in a small period of time (avalanche of alarms). Many of those warnings contain only little and/or highly redundant information which is rather useless for the operator in such a situation. Consequently the few warnings which need the immediate attention of the operator are easily overseen. The aggregated view of the infrastructure's condition, which is provided by the CEP engine in form of complex events, contributes to support the operator during such a situation. The aggregated view is an abstraction of the actual infrastructure and only events which are related to the change of a component's condition are presented to the operator. In this way, the operator can concentrate on fewer but more meaningful events.

4 Wireless Sensor Networks

4.1 Overview

A wireless sensor network (WSN) commonly consists of a potentially large number of small sensor nodes which form a self-organizing wireless network. The nodes of the network are small devices equipped with an energy supply, a processor, a communication unit and one or several sensors. Their main task is to measure the conditions of their environment and to send the measured data towards an information sink for further processing.

The size of sensor nodes can range from a few millimeters to several centimeters [85]. Due to their small size, sensor nodes are heavily constrained in processing power, available memory and communication bandwidth and range. This has severe consequences for the network: each sensor node can only communicate with its neighboring nodes and data is sent to remote nodes by passing it from node to node until it reaches its destination. Complicated computations cannot be carried out by the nodes and only a small amount of data can be kept in memory.

Typical application fields of WSN are military applications for the surveillance and tracking of enemies [95, 67], environmental and habitat monitoring [97, 66], monitoring of factories and machines [57] and monitoring of health conditions [10].

Energy consumption is an important issue for WSNs, since the lifetime of a sensor node is limited by the amount of available energy. Therefore one main interest of the research community is to find ways to reduce the energy consumption of the nodes of the network. This includes the development of efficient network protocols which enable a reliable and robust communication between nodes while keeping the communication between all nodes of the network low and techniques like data aggregation and compression which also reduce the communication between nodes.

Other topics of interest for WSNs are time synchronization, securing communications between nodes and (spatial) localization of nodes. [6, 103] are survey on WSN in general which provide an comprehensive overview on these and other WSN related topics.

4.2 Categorization of applications

In general WSNs are used to acquire data in a distributed and fault tolerant fashion in environments which are infeasible or too expensive to be monitored by sensors that are connected to a traditional network. Compared to traditional networks WSNs are cheap and easy to set up, robust against the failure of single components of the network and are thus appropriate to be used in rather harsh environments.

WSNs can be classified into one of the following categories depending on how the gathered data is processed:

Data gathering for off-line analysis. The main purpose of applications belonging to this category is to collect data of sensors with a relative low data rate for subsequent analysis on a regular PC. The data can either be distributed among all nodes of the network or be stored in a dedicated node. In order to gain good results from the analysis the focus of this kind of WSNs is to provide accurate and complete measurements.

Tolle et al. [97] used such a WSN to monitor temperature, relative humidity and light levels at different heights of a redwood tree. The sampling rate of the sensor nodes was set to 5 minutes and the reported measurements where sent to a gateway node which stored them in a database for later analysis.

Monitoring and event detection. A major characteristic of applications of this category is that the processing of the data is done inside the network, that is in the sensor nodes and not just in the gateway node or even later. For these type of applications not every measurement is of importance but rather aggregated values and certain measurements which indicate a special situation. Therefore the network is not expected to transmit all measurements to the information sink which makes it possible to filter and aggregate values in the sensor nodes what leads to a reduced communication and thus saves energy.

Werner-Allen et al. [99] used a WSN to monitor the activity of an active volcano. In order to reduce the data rate of 100 Hz averaging filters were applied on the measurement in the sensor nodes. Only when the difference of the two averages exceeded a threshold an event was generated and sent through the network to the base station.

Data gathering for on-line analysis. These types of applications use WSNs as a lower level tool for the real-time collection of sensor measurements. The analysis of the data is carried out on a high performance machine which is connected to a gateway node of the WSN. A WSN is preferred for this type of applications, since it is easier to set up than a traditional network and does not require an expensive wiring. Therefore the network is well structured and contains only a few nodes at well chosen locations. Typically all sensor nodes communicate directly with a gateway node which is in turn connected to a traditional network. The WSN is thus optimized for high throughput and low latency while the reduction of communication is of lower concern.

Krishnamurthy et al. [57] employed a WSN of this kind for vibration analysis in an industrial factory to monitor the health of the equipment. In order to obtain an adequate resolution of the measurements a sample rate between 19.2 and 40 KHz was required. A hierarchical WSN which is optimized for a high bandwidth and low latency is used to sent the measurements to an enterprise server which is powerful enough to carry out the analysis of the data.

The fields of complex event processing (CEP) and WSNs are closely related whereas the focus of each field is slightly different. CEP deals with the detection of complex events from a given stream of basic events. The major concern is a high throughput of a large amount of events whereas the detection of basic events is left open. Thus WSNs (especially of the third type, namely data gathering for on-line analysis) can be used to provide the input for CEP solutions.

In contrast, WSNs are used for the detection and delivery of basic events from the sensor nodes to an information sink while operating with resource constrained equipment. They allow a robust and reliable monitoring of environmental conditions in harsh environments which are inappropriate for traditional networks. Event detection in WSNs is used as an optimization method to reduce communication and energy consumption. Powerful approaches have been proposed which enable an efficient in-network evaluation of filters, aggregation and even joins.

4.3 Query evaluation in sensor networks

Of special interest in WSNs is the in-network evaluation of queries. This includes the aggregation of values and sensor fusion techniques. In contrast to naive query evaluation, which gathers all measurements in a gateway node and performs query evaluation in a centralized way, in-network query evaluation is, at least to some extend, performed in the sensor nodes. This can substantially reduce the communication of a network and thus save energy. A key issue is to reduce communication on the one hand, but on the other hand maintain robust and reliable query results in case of package loss or node failure.

Madden et al. [63] proposed to use the routing tree of a network for the stepwise application of aggregation functions (like max, avg, sum). As the data is sent from a leave of the routing tree towards the information sink it gets more and more aggregated. Since the aggregate is only a summary of multiple values the communication between nodes is substantially reduced. Similar approaches have been proposed by [102, 104, 49]. Tolle et al. [94] proposed methods for the in-network computation of median and other quantile values. Approximate aggregation techniques which reduce the number of packages sent though the network in exchange for the quality of the query result have been investigated in [76, 24, 93].

Besides in-network data aggregation a similar technique called information fusion is often applied to WSNs. Information fusion is known from digital signal processing and combines the measurements of a single or multiple sensors in order to increase the reliability and accuracy of the sensor measurements. For the efficient usage of those techniques in the field of WSNs adaptations of the algorithms for in-network computation have been developed. As a side effect of information fusion the amount of messages which is sent through the network can be reduced. A comprehensive survey of information fusion for WSNs can be found in [75].

Aggregation and information fusion is used to increase the quality of the measurements and to reduce network communication. However, these techniques can only provide limited support of sophisticated query predicates, if at all. Furthermore, the capability to evaluate general joins is required for the detection of complex events, that is events which are derived from multiple base events.

Adabi et al. [2] address the problem of very complicated filter predicates which compare the sensor data to predefined patterns. Their idea is to store filter conditions in a table and join tuples with this table in order to evaluate the filter predicates. Therefore they developed a method for in-network evaluation for joins between dynamic tuples (the measurements) and a static table (the filter conditions).

Recently the problem of in-network join computation has been addressed by the community [96, 106, 25, 14]. The efficient in-network computation of joins is quite challenging, since a sensor node can not decide locally whether a tuple has a matching join partner in the network.

Some of these approached are limited to special cases of a general join. For instance, Yang et al. [101] proposed a join method which requires one relation to contain only a few tuples and thus makes it feasible to distribute one relation among all nodes of the network. SENS-Join [96] addresses the problem of computing a general purpose join in sensor networks. The basic idea is to only sent the join attribute values to a base station in a first step in order to create a filter containing all values which find a join partner. In the second step, this filter is then distributed in the network and the complete tuples which are matching the filter are then sent to the base station where the join result is computed.

Besides the work on methods for the efficient in-network computation of aggregation, filters and joins there has been considerable work on frameworks for the evaluation of queries in a WSN [64, 102, 63]. These frameworks provide an abstraction of WSNs by regarding them as common relations and offer declarative query languages for querying the network. The propagation of queries in the network, the collection of the partial results and the computation of the query result is realized by the framework and transparent for the user. This approach has the advantage that query optimization can be applied, and that adequate and efficient methods for in-network computation can be chosen by the framework based on the given query.

4.4 Conclusion

CEP and query evaluation in WSNs are closely related. Both fields deal with measurements and observations of the real world and provide means for the efficient detection of events which occur in the underlying observations. However, the scope of the two domains is a bit different. WSNs focus on the gathering and delivery of basic events under highly resource constrained conditions. Event detection is used to reduce the amount of events which are of interest for the use and thus need to be communicated over the network. According to that, event detection is used to optimize communication in a WSN and as a consequence to increase the lifespan of the network. In contrast, CEP focuses on the detection of complex events from a given stream of basic events. Of major interest are massive streams of events (containing hundreds of thousands events per second) and efficient methods for the detection of complex events which guarantee a high throughput. It is generally assumed that the basic events of the stream are provided in a way which is appropriate for the CEP engine and the detection of those basic events is not further regarded.

Another substantial difference of both fields beside the detection of basic events is how queries, or more precisely the basic operators a query can be decomposed into, are evaluated. Due to the limited resources of the nodes of a WSN, the evaluation of even simple operators needs to be distributed among several nodes, whereas the evaluation of a single operator is typically not distributed in a network of CEP engines. Instead, the operators of a CEP query may be distributed among several nodes, but the evaluation of a single operater is never distributed among several nodes.

Summarizing, WSNs are mainly optimized for low energy consumption and heavily constrained devices, whereas CEP engines are optimized for a high throughput and quick response times.

However, WSNs which are intended for data gathering for on-line analysis, and thus power consumption and constrained devices are of lower concern, can be used (similar to a SCADA) as an input layer for a CEP engine.

5 Complex Event Processing[1]

5.1 Introduction

Event-driven information systems demand a systematic and automatic processing of events. Complex Event Processing (CEP) encompasses methods, techniques, and tools for processing events *while they occur*, i.e., in a continuous and timely fashion. CEP derives valuable higher-level knowledge from lower-level events; this knowledge takes the form of so called complex events, that is, situations that can only be recognized as combinations of several events.

The term Complex Event Processing was popularized in [62]; however, CEP has many independent roots in different research fields, including discrete event simulation, active databases, network management, and temporal reasoning. Only in recent years, CEP has emerged as a discipline of its own and as an important trend in the industry. The founding of the Event Processing Technical Society [30] in early 2008 underlines this development.

Important application areas of CEP are the following:

Business activity monitoring aims at identifying problems and opportunities in early stages by monitoring business processes and other critical resources. To this end, it summarizes events into so-called key performance indicators such as, e.g., the average run time of a process.

Sensor networks transmit measured data from the physical world to, e.g., Supervisory Control and Data Acquisition systems that are used for monitoring of industrial facilities. To minimize measurement and other errors, data of multiple sensors has to be combined frequently. Further, higher-level situations (e.g., fire) usually have to be derived from raw numerical measurements (e.g., temperature, smoke).

Market data such as stock or commodity prices can also be considered as events. They have to be analyzed in a continuous and timely fashion in order to recognize trends early and to react to them automatically, for example, in algorithmic trading.

The situations (specified as complex events) that need to be detected in these applications and the information associated with these situations are distributed over several events. Thus CEP can only derive such situations from a number of correlated (simple) events. To this end many different languages and formalisms for querying events, the so called Event Query Languages (EQLs), have been developed in the past.

There are also some surveys in the realm of CEP. For example, in [80, 79], rule-based approaches for reactive event processing are classified according to their origins. In [17], EQLs are divided into groups depending on the kind of system architecture they are used in. The survey of EQLs described in [90] distinguishes between a non-logic and logic-based view on handling the event triggered reactivity. There are also comparisons of different single CEP products, e.g., [44]. Both the multitude of EQLs and the diversity of surveys on event processing and reactivity can be attributed in part to the fact that CEP has many different roots and is only

[1]The following text will appear as a chapter in the Springer book “Reasoning in Event-based Distributed Systems” with the title “A CEP Babelfish: Languages for Complex Event Processing and Querying Surveyed”.

now recognized as an independent field.

To the best of our knowledge, there are no comprehensive surveys so far that (1) classify different EQLs into groups according to the language "style" or "flavor" and (2) compare the groups by means of the same example queries with respect to their expressivity, ease of use and readability, formal semantics, success in the industry and some other features. This chapter surveys the state of the art in CEP regarding these two points. Since CEP is a field that is very broad and without clear-cut boundaries, this chapter focuses strongly on querying events. It concentrates on EQLs that are known and specified at the outset. Other, less developed aspects of CEP such as detecting unknown events using approaches like machine learning and data mining on event streams, are not discussed here.

The contributions of this chapter are:

1. Identification and abstract description of five language styles, namely *composition operators, data stream query languages, production rules, timed state machines,* and *logic languages*
2. Illustration of each language style on a sensor network use case
3. Discussion on suitable application areas of each language style
4. Abstract description of some of the combined approaches

5.2 Terminology

Since CEP has evolved from many different research areas, a standard terminology has not yet established and found broad adoption. For example, what is called a (complex) event query might also be called a complex event type, an event profile, or an event pattern, depending on the context. We will therefore devote this section to the basic notions and our informal definitions of them.

An **event** is a message indicating that something of interest happens, or is contemplated as happening. Events can be represented in different data formats such as relational tuples, XML documents or objects of an object-oriented programming language (e.g., Java).

In this chapter, we use the following presentation of events: $event\ type\ (attribute\ name_1\ (attribute\ value_1), \ldots, attribute\ name_n\ (attribute\ value_n))$. An **event type** specifies an event structure, similar to a relational database schema specifying the structure of tuples of a relation. For example, *high_temp(area)* is an event type of an event *high_temp(area(a))* indicating high temperature in area *a*. In this event, *area* is an attribute and *a* is its value. (In the following, capital letters denote variables and small letters denote literals.) An **event attribute** is a component of the structure of an event. It can be an entry of a tuple, an XML fragment, or a field of an object, depending on the event representation. The set of attribute values of an event is called **event data**.

The formalism introduced here is by no means compelling. One could prefer to use unnamed perspective identifying attribute values by their positions or use any alternative event representation instead. Since in all languages proposed so far, events are flat or structured records or tuples, the formalism retained for this chapter is no restriction.

Since events happen at particular time which is essential for event processing, all events must have a possibly implicit attribute called event occurrence time. An **event occurrence time** is a time point or time interval indicating when this event happens. A time interval is described by two timestamps indicating its bounds. A time point is described by a single timestamp. We shall see below that using time points or time intervals has far reaching consequences for event processing.

Timing and event order are difficult issues of distributed systems. Each node (computer, device, etc.) in a distributed system has its own local clock and the clocks of different nodes are hard to be synchronized perfectly [26]. Furthermore the transmission time of messages varies depending on sender and receiver, routing, network load, and other factors. Therefore the reception order of some events may differ from their emission order [58]. These issues are ignored in this chapter for the sake of simplicity.

Another characteristical feature of events is event identification. For example, one can assign an identifier t to the event *high_temp(area(a))*, written in the following $t : \mathit{high\_temp}\ (\mathit{area}(a))$. We will see the advantages of this feature below.

Events are sent by event producers (e.g., sensors) to event consumers (e.g., Supervisory Control and Data Acquisition system) on so called **event streams**.

In order to react to an event e (e.g., turn on air conditioning in an area if an event indicating high temperature in the area arrives) or to derive a new event from another event e (e.g., derive an event indicating high temperature from an event containing a temperature measurement if the measurement is considered to be high), an event query which matches e is specified in an event query language. An **event query language (EQL)** is a high level programming language (possibly of limited expressivity) for querying events. A **simple event query** is a specification of a certain kind of *single* events by means of an event query language. A **complex event query** is a specification of a certain *combination* of events using multiple simple event queries and conditions describing the correlation of the queried events.

A **simple event** is either an event arriving on the event stream or an event derived by a simple event query (i.e., from a *single* event). A **complex event** is an event derived by a complex event query (i.e., from a certain *combination* of at least two events occurring or not occurring over time). In EPTS Glossary [30] many other kinds of events are defined, such as composite event, virtual event, derived event, raw event and some others.

Note that the occurrence time of a complex event e comprises the occurrence time of all events e it has been derived from. For example, a complex event *f:fire(area(a))* indicating fire can be derived from two simple events *s:smoke(area(a))* and *t: high_temp(area(a))* indicating smoke and high temperature respectively. f begins as soon as s or t begins and it ends as soon as both simple events are over.

The derivation of complex events is called Complex Event Processing. **Complex Event Processing (CEP)** denotes algorithmic methods for making sense of base events (low-level knowledge) by deriving complex events (high-level knowledge) from them in a timely fashion and over periods of time.

These are the most important notions in the field of event processing. In this chapter we will also need some other notions which will be informally introduced before using.

5.3 Identification of Language Styles

To bring some order into the multitude of EQLs, we try to group languages with a similar "style" or "flavor" together. We will focus on the general style of the languages and the variations within a style, rather than discussing each language and its constructs separately. It turns out most approaches for querying events fall into one of the following five categories:

1. languages based on composition operators (sometimes also called composite event algebras or event pattern languages),
2. data stream query languages (usually based on SQL),
3. production rules,
4. timed (finite) state machines, and
5. logic languages.

As we will see, the first, the second and the fifth approaches are languages explicitly developed for specifying event queries, while the third one is only a clever way to use the existing technologies of production rules to implement event queries. Similarly, the fourth approach is the use of an established technology to model event queries in a graphical way.

In Sections 5.4–5.8, we will describe each language style individually, mentioning the respective important languages from the research and industry. We will also discuss the strengths and weaknesses of each style and illustrate them on a sensor network use case which can be implemented using, e.g., TinyDB [65]. Section 5.9 summarizes the comparison by a discussion on suitable application areas of each language style. It is further worth mentioning that many industry products follow approaches where several languages of different flavors are supported or a single language combines aspects of several flavors. Section 5.10 will therefore be devoted to hybrid approaches. Section 5.11 concludes this chapter.

5.4 Composition Operators

5.4.1 General Idea

The first group of languages that we discuss builds complex event queries from simple event queries using composition operators. Historically, these languages have their roots primarily in Active Database Systems [83], though newer systems like Amit [5] run independently from a database. Some examples include: the COMPOSE language of the Ode active database [41, 42, 43], the composite event detection language of the SAMOS active database [39, 40], Snoop [23]

Composition	*fire(area(A))* = ($smoke(area(A)) \wedge high\_temp(area(A))$)$_{1\ min}$
Sequence	*fire(area(A))* = ($smoke(area(A));\ high\_temp(area(A))$)$_{1\ min}$ or *fire(area(A)) = s:smoke(area(A)); high_temp(area(A)); s+1 min*
Negation	*failure(sensor(S)) = t:temp(sensor(S)); not temp(sensor(S)); t+12 sec*
Aggregation	–

Figure 1: Example queries in pseudo code for composition operators

and its successor SnoopIB [3, 4], GEM [68], SEL [105], CEDR [11], ruleCore [92, 73], the SASE Event Language [100], the original event specification language of XChange [28, 18, 19], and the unnamed languages proposed in the following papers: [86], [71], [47], [21], [12], [88, 87].

Complex event queries are expressed by composing single events using different composition operators. Typical operators are conjunction of events (all events must happen, possibly at different times), sequence (all events happen in the specified order), and negation within a sequence (an event does not happen in the time between two other events). Consider the use of the operators in the sensor network use case below.

5.4.2 Sensor Network Use Case

Since different composition-operator-based EQLs have very different and rather unreadable syntax we formulate the example queries in pseudo code in Figure 1. The pseudo code illustrates the idea of this kind of EQLs but it does not mean that each of the queries in Figure 1 can be analogously formulated in every composition-operator-based EQL.

The first query in Figure 1 triggers fire alarm for an area when smoke and high temperature are both detected in the area within 1 minute, in other words the query derives a complex event *fire(area(A))* from the two events *smoke(area(A))* and *high_temp(area(A))*. The events *smoke(area(A))* and *high_temp(area(A))* are joined on variable *A*. Their order does not matter but it is important that both events appear within 1 minute indicated by the time window specification $(\ldots)_{1\ min}$. This is a typical example of event composition realized by the conjunction operator $\wedge$ and a time window specification.

The second example is similar to the first one but the events in the event query are connected by the sequence operator *;* denoting that the order of events is important, i.e., the event *smoke(area(A))* must appear before the event *high_temp(area(A))*. Only if the events appear within 1 minute and in the right order the complex event *fire(area(A))* is derived.

Alternatively if a composition-operator-based EQL supports event identification and relative timer events, this query can be formulated by means of the event identifier *s* for the event *smoke(area(A))* and a relative timer event *s+1 min*. In this case an EQL must decide whether the complex event $fire(area(A))$ is derived after the event $high\_temp(area(A))$ or after the event $s+1$ *min*.

Sequence operator is not as intuitive as it seems at first sight. Let A,B and C be simple event queries. Under time point semantics $(A;B);C$ is not equivalent to $A;(B;C)$, i.e., both queries do not yield the same answers. Let b,a,c be events arriving in this order and matching B, A, and C, respectively. They yield an answer for the query $A;(B;C)$ since b and c satisfy $(B;C)$ with the occurrence time (point) of c which is later than that of a. b happens before a which is not allowed by the query $(A;B);C$.

Under time interval semantics $A;(B;C)$ and $(A;B);C$ are equivalent. They both match events a,b,c arriving only in this order. $(B;C)$ matches b,c and has the occurence time interval starting as soon as b begins and ending as soon as c ends. Furthermore $A;(B;C)$ requires that a is over before b begins. This query matches a,b,c arriving exclusively in this order. Analogously $(A;B)$ matches a,b and has the time interval described by two time points, namely the begin of a and the end of b. Consequently $(A;B);C$ requires that c begins after b is over. This query can also match only the events a,b,c arriving in this order. Hence, using time points or time intervals has far reaching consequences [37].

The third example in Figure 1 shows how negation can be expressed by means of composition operators. The query uses event identification and relative timer events. It demonstrates the necessity for event identification if two events of the same type are used within one query and it has to be distinguished between them.

Assume all sensors of our network send temperature measurements every 12 seconds. The third query detects a failure of a sensor when its measurement is missing, i.e., the query derives a complex event *failure(sensor(S))* when there is an event *temp(sensor(S))* which is not followed by another event *temp(sensor(S))* within 12 seconds.

Another feature which must be supported by an EQL is aggregation. Aggregation means collection of data satisfying certain conditions, analysis of the data and construction of new data containing the result of the analysis. An example of aggregation in our use case is the computation of the average temperature reported by a sensor during the last minute every time a temperature measurement from the sensor arrives. Such a query is unfortunately not expressible by means of composition operators (compare Figure 1).

Nesting of expressions makes it possible to specify more complicated queries but we restrict ourselves to simple examples which should illustrate the main ideas of the language styles without embracing their whole expressivity.

5.4.3 Summary

Many composition-operator-based EQLs support restrictions on which events should be considered for the composition of a complex event. Event instance selection, for example, allows selection of only the first or last event of a particular type [107, 5, 47]. Event instance consumption prevents the reuse of an event for further complex events if it has already been used in another, earlier complex event [40, 107].

Composition operators offer a compact and intuitive way to specify complex events. Particularly temporal relationships and negation are well-supported. Event instance selection and consumption are features that are not present in the other approaches. Yet, there are hidden problems with the intuitive understanding of operators sometimes, e.g., several variants of the interpretation of a sequence (amongst others, interleaved with other events or not). Further, event data (i.e., access to the attribute values of an event) is often neglected in languages of this style, in particular regarding composition and aggregation.

Currently only very few CEP products are based on composition operators, among them IBM Active Middleware Technology (Amit) [5] and ruleCore [92, 73].

5.5 Data Stream Query Languages

5.5.1 General Idea

The second style of languages has been developed in the context of relational data stream management systems. Data stream management systems are targeted at situations where loading data into a traditional database management system would consume too much time. They are particularly targeted at nearly real-time applications where a reaction to the incoming data would already be useless after the time it takes to store it in a database. A typical example of data stream query languages is the Continuous Query Language (CQL) that is used in the STREAM systems [8]. The general ideas behind CQL apply to a number of open-source and commercial languages and systems including Esper [31], the CEP and CQL component of the Oracle Fusion Middleware [77], and Coral8 [72]. See also [52, 60] for the recent research in the field of data stream query languages.

Data stream query languages are based on the database query language SQL and the following general idea: Data streams carry events represented as tuples. Each data stream corresponds to exactly one event type. The streams are converted into relations which essentially contain (parts of) the tuples received so far. On these relations a (almost) regular SQL query is evaluated. The result (another relation) is then converted back into a data stream. Conceptually, this process is done at every point of time. Note that this implies a discrete time axis. (See however [50] for variations.)

For the conversion of streams into relations, stream-to-relation operators like time windows such as “all events of the last hour” or “the last 10 events” are used. For the conversion of the result relation back into a stream there are three options: “Istream” stands for “insert stream” and contains the tuples that have been added to the relation compared to the previous state of the relation, “Dstream” stands for “delete stream” and contains the tuples that have been removed from the relation compared to its previous state, or “Rstream” stands for “relation stream” and contains simply every tuple of the relation. In the following we only use “Istream”.

5.5.2 Sensor Network Use Case

Figure 2 shows equivalent example queries as Figure 1 but in Continuous Query Language (CQL). A CQL query is very similar to an SQL query. The FROM part of a CQL query is a cross product of relations, the optional WHERE part defines selection conditions, and the SELECT part is a usual projection.

For example, the FROM part of the first query in Figure 2 joins two relations *smoke* and *high_temp* which were generated out of event streams of type *smoke* and *high_temp* respectively by means of time windows. Generally there are several types of time windows. For the sake of brevity only two of them are explained here.

The first one is a simple sliding window. The resulting relation contains all stream tuples of a particular type between *now–d* and *now* where *now* is the current time point and *d* is a duration such as “1 Minute” or “12 Seconds”. The syntax for a sliding window of duration *d* is *T [Range d]* where *T* is an event type and the name of the resulting relation. For example, the notation *smoke [Range 1 Minute]* produces the relation *smoke* containing tuple representations of all events of type *smoke* which happened in the last minute.

The second time window that we explain here is a now window. The resulting relation contains only the stream tuples of a particular type with the occurrence time *now* where *now* denotes the current time point. The syntax for this window is *T [Now]* where *T* is the event type and the name of the resulting relation. For example, the result of the expression *high_temp [Now]* is the relation *high_temp* containing tuple representations of all events of type *high_temp* which happened at the current moment. Note that *T [Range 0 Minutes]* is equivalent to *T [Now]*.

Remember that the first query triggers fire alarm for an area when smoke and high temperature were both detected in the area within one minute. This temporal condition can be intuitively formulated by means of 1 minute-long simple sliding windows restricting the *smoke* and the *high_temp* streams. The join condition is specified in the WHERE block of the query. Consider the first example in Figure 2.

When the order of queried events is important the same query becomes less intuitive. Consider the second example in the figure. The query triggers fire alarm for an area when high temperature is being measured in the area now and smoke has been detected in the same area during the last minute.

The definition of correct time windows is essential as it has semantic consequences such as differentiation between an unordered composition and a sequence. Observe that a sequence of more than two events can only be expressed by means of rule chaining. E.g., the sequence of three events e_1, e_2, e_3 can be expressed in the following way: The first query guarantees that e_1 happens before e_2 and generates a complex event e as an intermediate result. The second rule queries events e and e_3 in this order and derives the resulting events.

Negation is hard to express in CQL (as well as in SQL) because the negated tuples have to be queried by an auxiliary query which is nested in the WHERE block of the main query and must be empty to let the main query produce an answer. For example, the third rule in the figure

Composition

```
SELECT Istream s.area
FROM smoke [Range 1 Minute] s,
     high_temp [Range 1 Minute] t
WHERE s.area = t.area
```

Sequence

```
SELECT Istream s.area
FROM smoke [Range 1 Minute] s,
     high_temp [Now] t
WHERE s.area = t.area
```

Negation

```
SELECT Istream t1.sensor
FROM temp [Now] t1
WHERE NOT EXISTS ( SELECT *
                   FROM temp [Range 12 Seconds] t2
                   WHERE t1.sensor = t2.sensor )
```

Aggregation

```
SELECT Istream t1.sensor, avg(t1.value)
FROM temp [Range 1 Minute] t1,
     temp [Now] t2
WHERE t1.sensor = t2.sensor
```

Figure 2: Example queries in Continuous Query Language

reports a failure of a sensor when it does not send a temperature measurement every 12 seconds.

Aggregation is well supported by the language as shown by the last example in Figure 2. Every time a temperature measurement from a sensor arrives the query computes the average temperature reported by the sensor during the last minute.

5.5.3 Summary

Data stream query languages are very suitable for aggregation of event data, as particularly necessary for market data, and offer a good integration with databases. Expressing negation and temporal relationships, on the other hand, is often cumbersome. The conversion from streams to relations and back may be considered somewhat unnatural and as may the prerequisite of a discrete time axis.

SQL-based data stream query languages are currently the most successful approach commercially and are supported in several efficient and scalable industry products. The better known ones are Oracle CEP, Coral8, StreamBase, Aleri and the open-source project Esper. However, there are big differences between the various projects and there also exist important extensions that go beyond the general idea that has been discussed here.

5.6 Production Rules

5.6.1 General Idea

Production rules are not an event query language as such, however they offer a fairly convenient and very flexible way of implementing event queries. The first successful production rule engine has been OPS [33], in particular in the incarnation OPS5 [32]. Since then, many others have been developed in the research and industry, including systems like Drools (also called JBoss Rules) [51], ILOG JRules [48], and Jess [89]. While the general ideas of production rules will be explained here, we refer the reader to [13] for a deeper introduction.

Production rules, which nowadays are mainly used in business rule management systems like Drools or ILOG JRules, are not EQLs in the narrower sense. The rules are usually tightly coupled with a host programming language (e.g., Java) and specify actions to be executed when certain states are entered [13]. The states are expressed as conditions over objects in the so-called working memory. These objects are also called facts.

Besides their use in business rule management systems that are not focused on events, production rules are also an integral part of the CEP product TIBCO Business Events, which also offers more CEP-specific features such as support for temporal aspects or modelling of event types and data.

The incremental evaluation (e.g., with Rete [34]) of production rules makes them also suitable for CEP. Whenever an event occurs, a corresponding fact must be created. Event queries are then expressed as conditions over these facts. In doing so, the programmer has much freedom but little guideline.

5.6.2 Sensor Network Use Case

Figure 3 contains our four example queries in the open source production rule system Drools. In Drools all events are represented as Java objects. Every time an event arrives some Java method has to convert it into an object, insert the object into the working memory, and call the rule engine to perform the rule evaluation (more precisely, fire all rules until no rule can fire). Note that in CEP-tailored systems such as TIBCO Business Events this happens automatically. If a complex event is derived by a rule it is also saved as an object in the working memory. We assume that in this case the *insert*-method sets the occurrence time of a complex event.

The occurrence time is a usual attribute of an object. This is actually a problem because every method can change every occurrence time inadvertently. This in turn leads to incorrect answers.

For the sake of simplicity we use time point semantics, assume that timestamps are given in seconds since the epoch (i.e., since the midnight of January 1, 1970) and we do not perform any garbage collection (i.e., deletion of events). These assumptions are not suitable for real-life applications but they help to keep the examples simple. Under the above assumptions we can express the temporal relations between events as simple comparisons of numbers. In real-life applications temporal relations would have to be programmed as Java methods that are called in Drools rules.

A Drools rule consists of two parts. The WHEN part is an event query, it specifies both, the types of queried events and conditions on the events. The THEN part derives an object representing the complex event, sets its occurrence time, and saves the object into the working memory. This newly asserted object can then also activate further rules.

Remember that the first rule detects fire in an area when smoke and high temperature are both detected in this area within one minute (consider the first rule in Figure 3). These conditions are coded into the specification of a *High_temp* object. Its attribute values are compared with the respective attribute values of a *Smoke* object *s*. In particular a *High_temp* event may happen at most one minute before or after a *Smoke* event.

In the second rule of the figure the order of the queried events is relevant. Smoke appears before high temperature is measured in the area. This is expressed by changing one of the conditions on the occurrence time of a *High_temp* object.

Negation is supported in Drools as shown by the third query. Recall that the query reports a failure of a sensor when the sensor does not send a temperature measurement every 12 seconds.

Aggregation of events is also supported. Consider the last rule in Figure 3. Every time a sensor sends a temperature measurement the query computes the average temperature reported by the sensor during the last minute. As this example illustrates aggregation is hard to express in Drools because the result of aggregation must be represented as an object in the WHEN part of a rule (an *Avg()* object in this case) to be used as a parameter of an object representing the complex event in the THEN part of a rule (an *Avg_temp()* object in this case).

Composition

```
when s: Smoke()
     High_temp(area == s.area &&
               timestamp >= (s.timestamp - 60) &&
               timestamp <= (s.timestamp + 60))
then insert(new Fire(s.area));
```

Sequence

```
when s: Smoke()
     High_temp(area == s.area &&
               timestamp > s.timestamp &&
               timestamp <= (s.timestamp + 60))
then insert(new Fire(s.area));
```

Negation

```
when t: Temp()
     not(exists(Temp(sensor == t.sensor &&
                     timestamp >= t.timestamp &&
                     timestamp <= (t.timestamp + 12))))
then insert(new Failure(t.sensor));
```

Aggregation

```
when t: Temp()
     a: Avg() from accumulate(
                    Temp(sensor == t.sensor &&
                         timestamp >= (t.timestamp - 60) &&
                         timestamp <= t.timestamp &&
                         v: value),
                    average(v))
then insert(new Avg_temp(t.sensor, a));
```

Figure 3: Example queries in Drools

As the examples show all relations between events must be programmed manually and even simple temporal conditions (already in our strongly simplified time model) require low-level code which is hard to read.

5.6.3 Summary

CEP with production rules is very flexible and well integrated with existing programming languages. However, it entails working on a low abstraction level that is — since it is primarily state and not event oriented — somewhat different from other EQLs. Especially aggregation and negation are therefore hard to express. Garbage collection, i.e., the removal of events from the working memory, has to be programmed manually. (See however [98] for work towards an automatic garbage collection.) Production rules are considered to be less efficient than data stream query languages; this is however tied to the flexibility they add in terms of combining queries (in rule conditions) and reactions (in rule actions).

5.7 Timed State Machines

5.7.1 General Idea

State machines are usually used to model the behavior of a stateful system that reacts to events. The system is modelled as a directed graph. The nodes of the graph represent the possible states of the system. Directed edges are labeled with events and temporal conditions on them. The edges specify the transitions between states that occur in reaction to in-coming events.

State machines are founded formally on deterministic or non-deterministic finite automata (DFAs or NFAs). Since states in a state machine are reached by particular sequences of multiple events occurring over time, they implicitly define complex events. Timed Büchi Automata (TBA) [7] were the first attempt to extend automata to temporal aspects for modelling real-time systems. In a TBA each transition between states depends not only on the type of arriving events but also on their occurrence time. For this, temporal conditions are added to transitions. Other examples of this kind of EQLs are UML state diagrams and regular real-time languages [46]. Many representatives of this language style were developed to achieve a particular task or solve a problem of real-time distributed systems, examples are Timed abstract state machine language for real-time system engineering [78], Timed automata approach to real time distributed system verification [56], Timed-constrained automata for reasoning about time in concurrent systems [69].

5.7.2 Sensor Network Use Case

In this chapter we do not describe different kinds of real-time automata but explain their common principle. Figure 4 contains our example queries in a pseudo code for timed state machines.

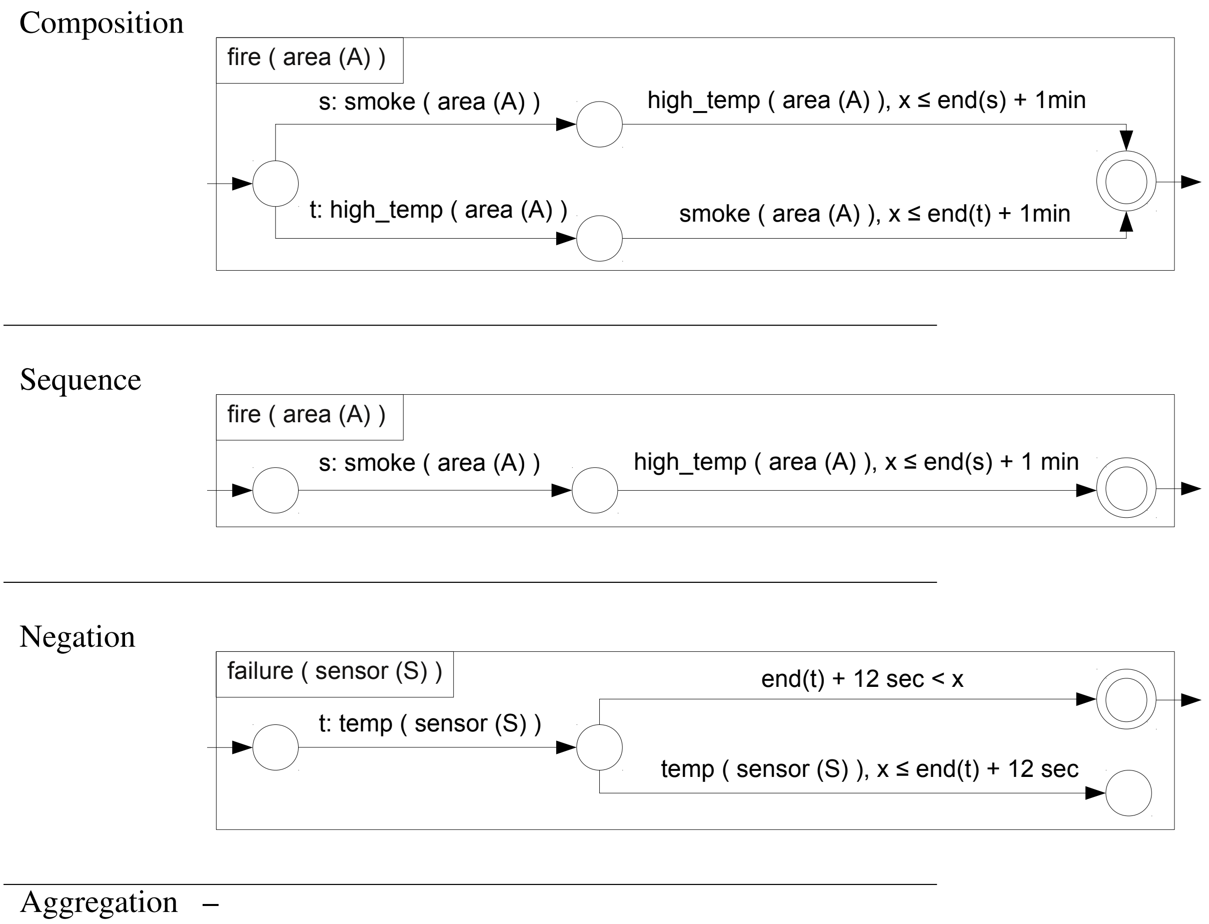
Composition
fire (area (A))
s: smoke (area (A))
high_temp (area (A)), x ≤ end(s) + 1min
t: high_temp (area (A))
smoke (area (A)), x ≤ end(t) + 1min
Sequence
fire (area (A))
s: smoke (area (A))
high_temp (area (A)), x ≤ end(s) + 1 min
Negation
failure (sensor (S))
end(t) + 12 sec < x
t: temp (sensor (S))
temp (sensor (S)), x ≤ end(t) + 12 sec
Aggregation –

Figure 4: Example queries in pseudo code for timed state machines

The pseudo code is an extension of Timed Büchi Automata [7]. The first extension is the consideration of event data. The second extension is the representation of complex events as automata in such a way that only if the end state of an automaton is reached the respective complex event is derived. A complex event can determinate a transition between states of another automaton so that arbitrary levels of abstraction can be achieved.

Remember that our first example derives a complex event *fire(area(A))* out of two events *smoke(area(A))* and *high_temp(area(A))* if these events happen within one minute. Their order does not matter. Since an automaton implicitly describes an ordered sequence we have to specify both acceptable orders of queried events. Consider the first query in Figure 4. The longer the composition of events the more acceptable orders (all possible permutations of events) must be considered by the machine, i.e., a simple composition query provokes a complicated automaton (exponential blow-up).

The events *smoke(area(A))* and *high_temp(area(A))* must happen within one minute. This condition is expressed using event identifiers, an auxiliary function *end(i)* which returns the end timestamp of event *i* and a global clock *x*. (As mentioned above, we do not consider such problems as clock synchronization in this chapter and refer the reader to [59].) Note that both events *smoke(area(A))* and *high_temp(area(A))* are joined upon the value of attribute *area*. If the end state of the state machine is reached the complex event *fire(area(A))* is derived.

The second query describes the sequence of events *smoke(area(A))* and *high_temp (area(A))*. The latter must happen at most one minute after the former to let the automaton reach its end state, i.e., to derive the complex event *fire(area(A))*. This is a very intuitive presentation.

Aggregation is not supported by timed state machines. Negation is not supported also but can be simulated by a failure state without outgoing edges and with an incoming edge which is labeled by a temporal condition and an event which should not arrive for the query to return an answer. For example, the third machine in Figure 4 detects a failure of a sensor when it does not send a temperature measurement every 12 seconds. If a temperature measurement comes within 12 seconds after the last measurement the state machine goes into the failure state, meaning that the end state is unreachable and the complex event *failure(sensor(S))* cannot be derived anymore. If 12 seconds since the last temperature measurement are over (consider the temporal condition of the incomimg edge of the end state) and no new measurement has arrived during this time, the state machine goes into the end state and derives the complex event *failure(sensor(S))*.

5.7.3 Summary

Though timed state machines provide intuitive visualization of complex events their expressivity is limited. They do not support aggregation. Negation and even composition of events are cumbersome. Conditions on the event data which are more complex than equi-joins (e.g., an attribute value must grow) cannot be expressed.

To overcome deficits of the theoretical automata, state machines are usually combined with languages of other styles. An example of this is the combination of state machines with production rules in TIBCO Business Events. There, a transition between two states is specified with a production rule. The condition of the production rule expresses when the transition is activated. Frequently reactions to the complex events that are implicit in a state machine are desirable. These can be specified for a transition (in the action part of the production rule) as well as for the entry or exit of states.

5.8 Logic Languages

5.8.1 General Idea

Logic languages express event queries in logic-style formulas. An early representative of this language style is the event calculus [54]. While event calculus is not an event query language per se, it has been used to model event querying and reasoning tasks in logic programming languages such as Prolog or Prova [55]. The latter combines the benefits of declarative and object-oriented programming by merging the syntaxes of Prolog and Java. Prova is used as a rule-based backbone for distributed Web applications in biomedical data integration. One of the key advantages of Prova is its separation of logic, data access, and computation.

XChange^{EQ} [15, 29] also adopts some ideas from event calculus-like approaches, but extends and tailors them to the needs of an expressive high-level event query language. XChange^{EQ} identifies and supports the following four complementary dimensions (or aspects) of event queries: data extraction, event composition, temporal (and other) relationships between events, and event accumulation. Its language design enforces a separation of the four querying dimensions.

A further example of this language style is Reaction RuleML [82, 81] combining derivation rules, reaction rules and other rule types such as integrity constraints into the general framework of logic programming.

5.8.2 Sensor Network Use Case

Figure 5 contains our four example queries in XChange^{EQ}. An XChange^{EQ} rule consists of two parts. The ON part, i.e., the rule body, is a complex event query which is a conjunction or disjunction of simple or complex event queries and an optional WHERE block containing temporal and other conditions on the queried events. The DETECT part, i.e., the rule head, is a construction of a complex event using the variable bindings returned by the respective event query.

Note that events are neither converted to relational tuples nor to objects of an object-oriented programming language. Furthermore, it is not possible to manipulate event timestamps neither consciously nor unwittingly. Finally, relative timer events are supported by XChange^{EQ}.

Event query specifications are very intuitive and flexible in XChange^{EQ}. There are four types of event queries charaterized by different kinds of brackets. Single brackets denote a complete event query, i.e., the query matches only those events which do not have attributes other than the ones specified in the query. In contrast double brackets denote an incolmplete event query, i.e., events matched by the query may have additional attributes. Curly brackets denote an unordered query, i.e., the order of attributes does not matter. Square brackets denote an ordered event query. Hence, there are four possible combinations of brackets, i.e., four types of event queries (ordered complete, unordered complete and so on).

Consider the first rule in Figure 5. Its complex event query is a conjunction of two simple incomplete and unordered event queries *event s: smoke*{{ *area*{{ *var A* }} }} and *event t: high_temp*{{ *area*{{ *var A* }} }} where variable *A* is bound to the value of attribute *area*. Since the same variable is used in both queries the queried events are joined on the value of this variable.

The WHERE block of the first rule in Figure 5 contains the additional temporal condition that both events, i.e., smoke and high temperature, appear within one minute. Note the use of event identifiers *s* and *t*. Note also that the temporal conditions (like *before* and *within*) are built-in into the language and must not be manually programmed.

Composition

```
DETECT   fire { area { var A } }
ON and { event s: smoke {{ area {{ var A }} }},
         event t: high_temp {{ area {{ var A }} }}
   } where { {s,t} within 1 min }
END
```

Sequence

```
DETECT   fire { area{ var A } }
ON and { event s: smoke {{ area {{ var A }} }},
         event t: high_temp {{ area {{ var A }} }}
   } where { s before t, {s,t} within 1 min }
END
```

Negation

```
DETECT   failure { sensor { var S } }
ON and { event t: temp {{ sensor {{ var S }} }},
         event i: timer:from-end [ event t, 12 sec ],
         while i: not temp {{ sensor {{ var S }} }} }
END
```

Aggregation

```
DETECT   avg_temp { sensor{ var S }, value { avg(all var T) } }
ON and { event t: temp {{ sensor {{ var S }} }},
         event i: timer:from-start-backward [ event t, 1 min ],
         while i: collect temp {{ sensor {{ var S }},
                                  value {{ var T }} }} }
END
```

Figure 5: Example queries in XChange^{EQ}

The second query contains the additional temporal condition that the smoke event must appear before the high temperature event. The effect that the *additional* temporal condition is mapped to an *additional* statement in the query is an outstanding feature of XChange$^{\mathrm{EQ}}$.

Negation and aggregation of events are supported as shown by the last two examples in Figure 5. Both negation and aggregation are restricted to finite time intervals. In the examples, the time intervals are given by relative timer events which are defined as follows:

- *timer:from-end[event e, d]* the relative timer *t* extends over the length of duration *d* starting at the end of *e*, i.e., *begin(t):=end(e), end(t):=end(e)+d*
- *timer:from-start-backward[event e, d]* the relative timer *t* extends over the length of duration *d* ending at the start of *e*, i.e., *begin(t):=begin(e)–d, end(t):=begin(e)*

In the above we write *begin(t)* and *end(t)* to denote the beginning and the end of event *t* respectively. There are of course many other relative timer events which are not discussed here, see [29].

Recall that the third example detects a failure of a sensor when it does not send a temperature measurement every 12 seconds, i.e., the query derives a complex event *failure*{ *sensor*{ *var S* } } when there is an event *temp*{{ *sensor*{{ *var S* }} }} which is not followed by another *temp*{{ *sensor*{{ *var S* }} }} event within 12 seconds.

The last query of the figure computes average temperature reported by a sensor during the last minute every time the sensor sends a temperature measurement. More precisely, every time an *event t: temp*{{ *sensor*{{ *var S* }} }} arrives, a relative timer event *i* denoting the time interval of one minute before *t*, is defined, all events happening during *i* and matched by the query *temp*{{ *sensor*{{ *var S* }}, *value*{{ *var T* }} }} are collected and a complex event *avg_temp*{ *sensor*{ *var S* }, *value*{ *avg(all var T)* } } containing the average temperature from the sensor *S*, is derived.

5.8.3 Summary

As the simple examples above demonstrate, logic languages offer a natural and convenient way to specify event queries. The main advantage of logic languages is their strong formal foundation, an issue which is neglected by many languages of other styles. (Chapter "Two Semantics for CEP, no Double Talk", in this volume describes a general, easily transferable approach for defining both, the declarative and operational semantics of an EQL). Thanks to the separation of different dimensions of event processing, logic languages are highly expressive, extensible and easy to learn and use. Some languages of this style, e.g., XChange$^{\mathrm{EQ}}$ supports an automatic garbage collection of events [16].

5.9 Application Areas of the Language Styles

Having described the strengths and weaknesses of the five language styles, we summarize the comparison by a discussion on suitable application areas of each language style.

Composition operators allow an intuitive specification of event patterns. This makes them attractive in scenarios, where business users should be allowed to define event patters such as real-time promotions and upselling (e.g., send three text messages within one hour to receive a free ringtone).

Data stream query languages are very suitable for aggregation of event data, as particularly necessary for applications involving market data (e.g., average price over 21 day sliding window) such as algorithmic trading. They also usually offer a good integration with databases, sharing in particular the common basis of SQL.

Production rules are very flexible and well integrated with existing programming languages. Since they allow the specification of actions to be executed when certain states are reached, they are particularly useful for applications involving tracking of stateful objects such as track and trace in logistics (maintain and react upon changes of the state of packages, containers, etc.) or monitoring of business processes and objects (also called Business Activity Monitoring). Due to their wide-spread use in business rules management systems, production rules often offer some support for exposing part of the logic to business users such as decision tables or trees.

Timed state machines also offer an easy and convenient way to maintain the current state. However they are limited to a finite set of states (e.g., "shipped", "delivered"). This makes them suitable, e.g., for monitoring of processes (which typically have a well-defined, finite number of states), but not suitable for applications involving infinite state spaces (e.g., a temperature control system where the temperature is a numeric value).

Logic languages have strong formal foundations, allow an intuitive specification of complex temporal conditions and account for event data. They could be successfully used in medical applications or emergency management in critical infrastructures.

Combination of different language styles in one approach allows to benefit from their strengths. This is the main reason why hybrid approaches are most successful in the industry. The next section is devoted to the combined approaches.

5.10 Combination of Different Language Styles

A comparison of the different language styles shows that so far there is no one-fits-all approach to querying events. Hence particularly industry products trend towards hybrid approaches, where several languages of different styles are supported or aspects of different styles are combined within one language. Hybrid approaches include the introduction of pattern matching into data stream query languages as in Oracle CEP [77], Esper [31], and some CQL dialects like the one used in [91], the use of composition operators on top of data stream queries [38, 22], the addition of composition operators to production rules [98], the combination of production rules and state machines, e.g., in TIBCO Business Events (see Section 5.7), the decoupled use of different languages (and possibly evaluation engines) that communicate only by means of exchanging events (derived as answers to queries).

5.11 Conclusion

CEP is an industrial growth market as well as an important research area that is emerging from coalescing branches of other research fields.

Even though the prevalent event query languages can be categorized roughly into five families as done in this document, there are significant differences between the individual languages of a family. Whether a convergence to a single, dominant query language for CEP is possible and advisable is currently in no way agreed upon.

Efforts towards a standard for a SQL-based data stream query language are on the way [50], but not yet within an official standardization body. A standardized XML syntax for production rules is being developed by the W3C as part of the Rule Interchange Format (RIF); however, the special requirements of CEP are not considered there yet. The same applies to the Production Rule Representation (PRR) by the OMG.

Activities of the Event Processing Technical Society (EPTS) [30] aim at a coordination and harmonization with the work on a glossary of CEP notions, the interoperability analysis of Event Processing systems from different vendors, a common reference architecture or framework of architectures, that handles current and envisioned Event Processing architectures, the analysis of the application areas of CEP, and the creation of a business value for a user in order to increase the adoption of Event Processing in the business and industry. The EPTS has also a working group for the analysis of EQLs.

An event query language for EMILI From our point of view, the logic style languages are most relevant with respect to the needs of EMILI, since languages of this category are the most mature event query languages currently available. For instance, the language XChange$^{\text{EQ}}$ has a clear formal semantic which includes an operational and even a declarative semantic which is unique for EQLs. Beyond that declarative languages are perfectly suited for query optimization which therefore can easily be added to an CEP engine for improving the efficiency of query evaluation. Furthermore another requirement of an EQL for EMILI is the clear separation of the four query dimensions which leads to a complete covering of all these dimensions.

Since XChange$^{\text{EQ}}$ fulfills all these requirements it is a good basis for the development of an EQL which is specifically designed and adopted to the needs of EMILI. Although XChange$^{\text{EQ}}$ forms a good basis for an EQL for EMILI, it has been developed for CEP in general and has not been optimized for applications in the domain of emergency management. Therefore the requirements which have been (and will further be) extracted from the use cases will have an significant influence on the design of the upcoming CEP language for EMILI.

Based on the description of the use cases in Deliverable D3.1 (Use Case Requirement Analysis and Specification) and its annexes two major characteristics have been identified so far which are relevant for the EQL but are not addressed in the current language definition of XChange$^{\text{EQ}}$ (some more are described in the deliverable D4.2).

First, the use cases require an integration of physical models into the CEP language. As already discussed in section 2, the cleansing of noisy sensor data would clearly benefit from such an integration. Moreover, these models can be used to support decision making by providing predictions of the likely development of a situation in the future. This information can be used to compare the effectiveness of different possible reactions to a detected situation.

Second, current EQLs offer no or only limited means for dealing with states, e.g. of the supervised infrastructure. For instance, the state of the entrance hall of a metro station may reflect whether the hall is save (i.e. there is no fire or smoke) and how many people are currently located there. Only production rule based EQLs have some notion of states. However, due to their lack of an explicit representation of events, these kind of languages are not well suited for complex event processing. Thus, their ideas for dealing with states may be adopted for the EQL of EMILI to some extend, but due to their design flaws the language itself cannot be used as a basis for an appropriate EQL.

Summarizing, an EQL for EMILI should

- have a clear operational and declarative semantic,
- make a clear separation between the four query dimensions,
- integrate physical models and
- provide a notion of states.

There are few EQLs available which fulfill the requirements for the first two points. But currently there are no approaches which combine the requirements of all four characteristics into a single and homogeneous EQL. Therefore a new EQL based on XChange$^{\text{EQ}}$ will be developed within the EMILI project which addresses all those requirements in a homogeneous language.

References

[1] Data cleaning material collection. `http://paul.rutgers.edu/~weiz/readinglist.html`.

[2] D. J. Abadi, S. Madden, and W. Lindner. Reed: robust, efficient filtering and event detection in sensor networks. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 769–780. VLDB Endowment, 2005.

[3] R. Adaikkalavan and S. Chakravarthy. Formalization and detection of events using interval-based semantics. In *Proc. Int. Conf. on Management of Data (COMAD)*, pages 58–69. Computer Society of India, 2005.

[4] R. Adaikkalavan and S. Chakravarthy. SnoopIB: Interval-based event specification and detection for active databases. *Data and Knowledge Engineering*, 1(59):139–165, 2006.

[5] A. Adi and O. Etzion. Amit — the situation manager. *The VLDB Journal*, 13(2):177–203, 2004.

[6] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393 – 422, 2002.

[7] R. Alur and D. Dill. Automata for modeling real-time systems. In *Proc. Int. Colloquium on Automata, Languages and Programming*, volume 443 of *LNCS*, pages 322–335. Springer, 1990.

[8] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[9] D. Bailey and E. Wright. *Practical SCADA for Industry*. Newnes, 2003.

[10] C. R. Baker, K. Armijo, S. Belka, M. Benhabib, V. Bhargava, N. Burkhart, A. D. Minassians, G. Dervisoglu, L. Gutnik, M. B. Haick, C. Ho, M. Koplow, J. Mangold, S. Robinson, M. Rosa, M. Schwartz, C. Sims, H. Stoffregen, A. Waterbury, E. S. Leland, T. Pering, and P. K. Wright. Wireless sensor networks for home health care. In *AINAW '07: Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops*, pages 832–837, Washington, DC, USA, 2007. IEEE Computer Society.

[11] R. S. Barga and H. Caituiro-Monge. Event correlation and pattern detection in CEDR. In *Proc. Int. Workshop Reactivity on the Web*, volume 4254 of *LNCS*, pages 919–930. Springer, 2006.

[12] M. Bernauer, G. Kappel, and G. Kramler. Composite events for XML. In *Proc. Int. Conf. on World Wide Web*, pages 175–183. ACM, 2004.

[13] B. Berstel, P. Bonnard, F. Bry, M. Eckert, and P.-L. Pătrânjan. Reactive rules on the Web. In *Reasoning Web, Int. Summer School*, volume 4636 of *LNCS*, pages 183–239. Springer, 2007.

[14] B. J. Bonfils and P. Bonnet. Adaptive and decentralized operator placement for in-

network query processing. *Telecommunication Systems*, 26(2-4):389–409, June 2004.

[15] F. Bry and M. Eckert. Rule-Based Composite Event Queries: The Language XChange$^{\mathrm{EQ}}$ and its Semantics. In *Proc. Int. Conf. on Web Reasoning and Rule Systems*, volume 4524 of *LNCS*, pages 16–30. Springer, 2007.

[16] F. Bry and M. Eckert. On static determination of temporal relevance for incremental evaluation of complex event queries. In *Proc. Int. Conf. on Distributed Event-Based Systems*, pages 289–300. ACM, 2008.

[17] F. Bry, M. Eckert, O. Etzion, A. Paschke, and J. Riecke. Event processing language tutorial. In *3rd ACM Int. Conf. on Distributed Event-Based Systems*. ACM, 2009.

[18] F. Bry, M. Eckert, and P.-L. Pătrânjan. Querying composite events for reactivity on the Web. In *Proc. Int. Workshop on XML Research and Applications*, volume 3842 of *LNCS*, pages 38–47. Springer, 2006.

[19] F. Bry, M. Eckert, and P.-L. Pătrânjan. Reactivity on the Web: Paradigms and applications of the language XChange. *J. of Web Engineering*, 5(1):3–24, 2006.

[20] F. Bry, B. Lorenz, H. J. Ohlbach, M. Roeder, and M. Weinberger. The facility control markup language FCML. *International Conference on the Digital Society*, 0:117–122, 2008.

[21] J. Carlson and B. Lisper. An event detection algebra for reactive systems. In *Proc. ACM Int. Conf. On Embedded Software*, pages 147–154. ACM, 2004.

[22] S. Chakravarthy and R. Adaikkalavan. Events and streams: Harnessing and unleashing their synergy! In *Proc. Int. Conf. on Distributed Event-Based Systems*, pages 1–12. ACM, 2008.

[23] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proc. Int. Conf. on Very Large Data Bases*, pages 606–617. Morgan Kaufmann, 1994.

[24] H. Chan, A. Perrig, and D. Song. Secure hierarchical in-network aggregation in sensor networks. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 278–287, New York, NY, USA, 2006. ACM.

[25] A. Coman, M. A. Nascimento, and J. Sander. On join location in sensor networks. In *MDM '07: Proceedings of the 2007 International Conference on Mobile Data Management*, pages 190–197, Washington, DC, USA, 2007. IEEE Computer Society.

[26] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, third edition, 2001.

[27] A. Daneels and W. Salter. What is SCADA? In *Proc. of Int. Conf. on Accelerator and Large Experimental Physics Control Systems*, pages 339–343, 1999.

[28] M. Eckert. Reactivity on the Web: Event Queries and Composite Event Detection in XChange. Master's thesis (Diplomarbeit), Institute for Informatics, University of Munich, 2005.

[29] M. Eckert. *Complex Event Processing with XChange^{EQ}: Language Design, Formal Semantics and Incremental Evaluation for Querying Events*. PhD thesis, Institute for Informatics, University of Munich, 2008.

[30] Event Processing Technical Society (EPTS). `http://www.ep-ts.com`.

[31] EsperTech Inc. Event stream intelligence: Esper & NEsper. `http://esper.codehaus.org`.

[32] C. Forgy. OPS5 user's manual. Technical Report CMU-CS-81-135, Carnegie Mellon University, 1981.

[33] C. Forgy and J. P. McDermott. OPS, a domain-independent production system language. In *Proc. Int. Joint Conf. on Artificial Intelligence*, pages 933–939. William Kaufmann, 1977.

[34] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[35] H. Galhardas. *Data Cleaning: Model, Language and Algoritmes*. PhD thesis, University of Versailles, 2001.

[36] H. Galhardas, D. Florescu, and D. Shasha. Declarative data cleaning: Language, model, and algorithms. In *In VLDB*, pages 371–380, 2001.

[37] A. Galton and J. C. Augusto. Two approaches to event definition. In *Proc. Int. Conf. on Database and Expert Systems Applications*, volume 2453 of *LNCS*, pages 547–556. Springer, 2002.

[38] V. Garg, R. Adaikkalavan, and S. Chakravarthy. Extensions to stream processing architecture for supporting event processing. In *Proc. Int. Conf. on Database and Expert Systems Applications*, volume 4080 of *LNCS*, pages 945–955. Springer, 2006.

[39] S. Gatziu and K. R. Dittrich. Events in an active object-oriented database system. In *Proc. Int. Workshop on Rules in Database Systems*, pages 23–39. Springer, 1993.

[40] S. Gatziu and K. R. Dittrich. Detecting composite events in active database systems using petri nets. In *Proc. Int. Workshop on Research Issues in Data Engineering: Active Database Systems*, pages 2–9. IEEE, 1994.

[41] N. H. Gehani, H. Jagadish, and O. Shmueli. Event specification in an active object-oriented database. In *Proc. Int. ACM Conf. on Management of Data (SIGMOD)*, pages 81–90. ACM, 1992.

[42] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model & implementation. In *Proc. Int. Conf. on Very Large Data Bases*, pages 327–338. Morgan Kaufmann, 1992.

[43] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Compose: A system for composite specification and detection. In *Advanced Database Systems*, LNCS, pages 3–15. Springer, 1993.

[44] M. Gualtieri and J. R. Rymer. The Forrester Wave^{TM}: Complex Event Procecessing (CEP) Platforms. `http://www.forrester.com/rb/Research/wave%26trade%3B_complex_event_processing_cep_platforms%2C_q3/q/id/48084/t/2`, 2009.

[45] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kauffman, 2001.

[46] T. A. Henzinger, J.-F. Raskin, and P.-Y. Schobbens. The regular real-time languages. In *In Proc. 25th Int. Coll. Automata, Languages, and Programming (ICALP'98*, pages 580–591. Springer, 1998.

[47] A. Hinze and A. Voisard. A parameterized algebra for event notification services. In *Proc. Int. Symp. on Temporal Representation and Reasoning*, pages 61–65. IEEE, 2002.

[48] ILOG. ILOG JRules. `http://www.ilog.com/products/jrules`.

[49] C. Intanagonwiwat, D. Estrin, R. Govindan, and J. Heidemann. Impact of network density on data aggregation in wireless sensor networks. In *ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 457, Washington, DC, USA, 2002. IEEE Computer Society.

[50] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Çetintemel, M. Cherniack, R. Tibbetts, and S. Zdonik. Towards a streaming SQL standard. In *Proc. Int. Conf. on Very Large Data Bases*, volume 1, pages 1379–1390. VLDB Endowment, 2008.

[51] JBoss.org. Drools. `http://www.jboss.org/drools`.

[52] M. Kersten, E. Liarou, and R. Goncalves. A query language for a data refinery cell. In *Proc. Int. Workshop on Event-Driven Architecture, Processing and Systems*, 2007.

[53] E. M. Knorr. *Outliers and Data Mining: Finding Exceptions in Data*. PhD thesis, University of British Columbia, 2002.

[54] R. A. Kowalski and M. J. Sergot. A logic-based calculus of events. *New Generation Compututing*, 4(1):67–95, 1986.

[55] A. Kozlenkov, R. Penaloza, V. Nigam, L. Royer, G. Dawelbait, and M. Schroeder. Prova: Rule-based Java scripting for distributed web applications: A case study in bioinformatics. In *Current Trends in Database Technology (EDBT)*, volume 4254 of *LNCS*, pages 899–908. Springer, 2006.

[56] J. Krákora, L. Waszniowski, and Z. Hanzálek. Timed automata approach to real time distributed system verification. In *In Proc. of IEEE Int. Workshop on Factory Communication Systems (WFCS)*, pages 407–410, 2004.

[57] L. Krishnamurthy, R. Adler, P. Buonadonna, J. Chhabra, M. Flanigan, N. Kushalnagar, L. Nachman, and M. Yarvis. Design and deployment of industrial sensor networks: experiences from a semiconductor plant and the north sea. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 64–

75, New York, NY, USA, 2005. ACM.

[58] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[59] Q. Li and D. Rus. Global clock synchronization in sensor networks. *IEEE Transactions on Computers*, 55(2):214–226, 2006.

[60] E. Liarou, R. Goncalves, and S. Idreos. Exploiting the power of relational databases for efficient stream processing. In *Int. Conf. on Extending Database Technology (EDBT)*, volume 360, pages 323–334. ACM, 2009.

[61] B. G. Lipták. *Instrument Engineers' Handbook: Process Software and Digital Networks*, volume 3. CRC Press, 2002.

[62] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002.

[63] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. In *OSDI*, 2002.

[64] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[65] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: An acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems*, 30(1):122–173, 2005.

[66] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, New York, NY, USA, 2002. ACM.

[67] G. Mallapragada, Y. Wen, S. Phoha, D. Bein, and A. Ray. Tracking mobile targets using wireless sensor networks. In S. Latifi, editor, *ITNG*, pages 873–878. IEEE Computer Society, 2010.

[68] M. Mansouri-Samani and M. Sloman. GEM: A generalized event monitoring language for distributed systems. *Distributed Systems Engineering*, 4(2):96–108, 1997.

[69] M. Merritt, F. Modugno, and M. R. Tuttle. Time-constrained automata. In *CONCUR '91: 2nd Int. Conf. on Concurrency Theory*, volume 527 of *LNCS*, pages 408–423. Springer, 1991.

[70] A. E. Monge. *Adaptive detection of approximately duplicate database records and the database integration approach to information discovery*. PhD thesis, University of California, San Diego, 1997.

[71] D. Moreto and M. Endler. Evaluating composite events using shared trees. *IEE Proceedings — Software*, 148(1):1–10, 2001.

[72] J. Morrell and S. D. Vidich. Complex Event Processing with Coral8. White Paper. `http://www.coral8.com/system/files/assets/pdf/Complex_Event_Processing_with_Coral8.pdf`, 2007.

[73] MS Analog Software. ruleCore(R) Complex Event Processing (CEP) Server. `http://www.rulecore.com`.

[74] H. Müller and J.-C. Freytag. Problems, methods, and challenges in comprehensive data cleansing. *HUB-IB-164, Humboldt University Berlin*, 2003.

[75] E. F. Nakamura, A. A. F. Loureiro, and A. C. Frery. Information fusion for wireless sensor networks: Methods, models, and classifications. *ACM Comput. Surv.*, 39(3):9, 2007.

[76] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 563–574, New York, NY, USA, 2003. ACM.

[77] Oracle Inc. Complex Event Processing in the real world. White Paper. `http://www.oracle.com/technologies/soa/docs/oracle-complex-event-processing.pdf`.

[78] M. Ouimet and K. Lundqvist. The timed abstract state machine language: Abstract state machines for real-time system engineering. *Journal of Universal Computer Science*, 14(12):2007–2033, 2008.

[79] A. Paschke and H. Boley. Rules capturing events and reactivity. In *Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches*, pages 215–252. IGI Global, 2009.

[80] A. Paschke and A. Kozlenkov. Rule-based event processing and reaction rules. In *Rule Interchange and Applications*, volume 5858 of *LNCS*, pages 53–66. Springer, 2009.

[81] A. Paschke, A. Kozlenkov, and H. Boley. A homogenous reaction rule language for Complex Event Processing. In *In Proc. 2nd Int. Workshop on Event Drive Architecture and Event Processing Systems*, 2007.

[82] A. Paschke, A. Kozlenkov, H. Boley, S. Tabet, M. Kifer, and M. Dean. Reaction RuleML. `http://ibis.in.tum.de/research/ReactionRuleML/`, 2007.

[83] N. W. Paton, editor. *Active Rules in Database Systems*. Springer, 1998.

[84] Rahm and Do. Data cleaning: Problems and current approaches. *IEEE Bulletin 23(4)*, 2000.

[85] K. Römer and F. Mattern. The design space of wireless sensor networks. *IEEE Wireless Communications*, 11(6):54–61, 2004.

[86] C. Roncancio. Toward duration-based, constrained and dynamic event types. In *Proc. Int. Workshop on Active, Real-Time, and Temporal Database Systems*, volume 1553 of *LNCS*, pages 176–193. Springer, 1997.

[87] C. Sánchez, S. Sankaranarayanan, H. Sipma, T. Zhang, D. L. Dill, and Z. Manna. Event correlation: Language and semantics. In *Proc. Int. Conf. on Embedded Software*, volume 2855 of *LNCS*, pages 323–339. Springer, 2003.

[88] C. Sánchez, M. Slanina, H. B. Sipma, and Z. Manna. Expressive completeness of an event-pattern reactive programming language. In *Int. Conf. on Formal Techniques for Networked and Distributed Systems*, volume 3731 of *LNCS*, pages 529–532. Springer, 2005.

[89] Sandia National Laboratories. Jess, the rule engine for the Java(TM) platform. `http://herzberg.ca.sandia.gov/`.

[90] K.-U. Schmidt, D. Anicic, and R. Stühmer. Event-driven reactivity: A survey and requirements analysis. In *SBPM2008: 3rd Int. Workshop on Semantic Business Process Management in Conjunction with the 5th European Semantic Web Conf. (ESWC'08)*. CEUR Workshop Proceedings, 2008.

[91] B. Seeger. Kontinuierliche kontrolle. *IX: Magazin für Professionelle Informationstechnik*, 2, 2010.

[92] M. Seiriö and M. Berndtsson. Design and implementation of an ECA rule markup language. In *Proc. Int. Conf. on Rules and Rule Markup Languages for the Semantic Web*, volume 3791 of *LNCS*, pages 98–112. Springer, 2005.

[93] A. Sharaf, J. Beaver, A. Labrinidis, and K. Chrysanthis. Balancing energy efficiency and quality of aggregate data in sensor networks. *The VLDB Journal*, 13(4):384–403, 2004.

[94] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri. Medians and beyond: new aggregation techniques for sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 239–249, New York, NY, USA, 2004. ACM.

[95] G. Simon, M. Maróti, A. Lédeczi, G. Balogh, B. Kusy, A. Nádas, G. Pap, J. Sallai, and K. Frampton. Sensor network-based countersniper system. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 1–12, New York, NY, USA, 2004. ACM.

[96] M. Stern, E. Buchmann, and K. Böhm. Towards efficient processing of general-purpose joins in sensor networks. In *ICDE '09: Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 126–137, Washington, DC, USA, 2009. IEEE Computer Society.

[97] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay, and W. Hong. A macroscope in the redwoods. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 51–63, New York, NY, USA, 2005. ACM.

[98] K. Walzer, T. Breddin, and M. Groch. Relative temporal constraints in the Rete algorithm for complex event detection. In *Proc. Int. Conf. on Distributed Event-Based Systems*, pages 147–155. ACM, 2008.

[99] G. Werner-Allen, K. Lorincz, M. Welsh, O. Marcillo, J. Johnson, M. Ruiz, and J. Lees. Deploying a wireless sensor network on an active volcano. *IEEE Internet Computing*, 10(2):18–25, 2006.

[100] E. Wu, Y. Diao, and S. Rizvi. High-performance Complex Event Processing over streams. In *Proc. Int. ACM Conf. on Management of Data (SIGMOD)*, pages 407–418. ACM, 2006.

[101] X. Yang, H. B. Lim, T. M. Özsu, and K. L. Tan. In-network execution of monitoring queries in sensor networks. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 521–532, New York, NY, USA, 2007. ACM.

[102] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31(3):9–18, 2002.

[103] J. Yick, B. Mukherjee, and D. Ghosal. Wireless sensor network survey. *Computer Networks*, 52(12):2292 – 2330, 2008.

[104] J. Zhao, R. Govindan, and D. Estrin. Computing aggregates for monitoring wireless sensor networks. In *Proceedings of the First IEEE International Workshop on Sensor Network Protocols and Applications*, pages 139–148, 2003.

[105] D. Zhu and A. S. Sethi. SEL, a new event pattern specification language for event correlation. In *Proc. Int. Conf. on Computer Communications and Networks*, pages 586–589. IEEE, 2001.

[106] X. Zhu, H. Gupta, and B. Tang. Join of multiple data streams in sensor networks. *IEEE Trans. on Knowl. and Data Eng.*, 21(12):1722–1736, 2009.

[107] D. Zimmer and R. Unland. On the semantics of complex events in active database management systems. In *Proc. Int. Conf. on Data Engineering*, pages 392–399. IEEE, 1999.