Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit Nr. 42

Situation Recognition Based on Complex Event Processing

Ana Cristina Franco da Silva

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Ing. habil. Bernhard Mitschang
Supervisor:	Dipl.-Inf. Pascal Hirmer
Commenced:	June 1, 2015
Completed:	December 1, 2015
CR-Classification:	I.5, I.5.1, H.3.4, I.2.9

Abstract

In the Internet of Things, physical objects – the *things* – are connected through a network and actively exchange information about themselves and their surroundings. This paradigm enables the existence of so called smart environments, in which numerous context-aware applications can be deployed. Such applications can have a significant impact in the every-day life (e.g., smart homes, smart cities, etc.). Context-awareness allows applications to recognize situations of interest and properly react to them when necessary. However, deriving the large amount of raw, low-level sensor data into higher-level knowledge is a challenging task. In the last years, Complex Event Processing (CEP) has emerged as an important trend in applications that recognize situations in real or near real time. CEP can be employed to process sensor data in a continuous and timely fashion, in order to recognize situations as soon as they occur. Within the scope of this master thesis, a *Situation Recognition System* based on sensor data is developed using a CEP engine. This system can be used to monitor many situations in parallel based on the perceived surroundings of things that send context information, i.e. sensor values, to the system through the Internet. The recognition of situations is based on a non-executable model called Situation Template, which offers a means to easily describe the conditions for the occurring situations. Furthermore, this master thesis presents a sensor push approach so that sensor data is available to the Situation Recognition System as soon as possible. Moreover, this work analyzes three different CEP engines and motivates the choice of a CEP engine that copes with the powerfulness of Situation Templates. To execute the situation recognition using CEP, this work implements mappings from Situation Templates onto executable representations, i.e., CEP queries, to be deployed into the chosen CEP engine. Finally, a prototypical implementation of the Situation Recognition System is presented and evaluated via runtime measurements.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

In the Internet of Things (IoT), things are connected through a network and take an active part in the Internet by exchanging information about themselves and their surroundings [4]. These things (e.g., mobile phones, smart watches, smart bands) are equipped with sensor technologies and network connectivity, what permits them to gather and exchange data. Environments containing things that perceive their surroundings and communicate with each other are called smart environments. They enable the deployment of numerous applications, which can have a significant impact on many aspects of the every-day life [5]. A smart environment can be a home, an office, a factory, or even a city equipped with things, in which for example, incidents can be prevented by proper monitoring and alarm systems.

If something happens in a smart environment that might require a reaction, i.e., a situation occurs [2], a situation recognition system can recognize this occurrence and send notifications of it to situation-aware applications. This enables such applications to react to occurring situations. However, situation recognition systems need to deal with a challenging aspect in the IoT, namely the management of the data within the IoT context. When considering a set of things interconnected and constantly exchanging all types of information, the volume of the generated data and how it is handled become critical [6]. Therefore, the complexity of deriving knowledge from a large amount of sensor data demands the use of sophisticated techniques to process this data in a continuous and timely fashion, so that situations can be recognized as soon as they occur [6].

Deriving higher-level knowledge from raw, low-level data has been approached using different technologies from many independent research fields (discrete event simulation, active databases, network management, temporal reasoning, etc.), and in different application fields (business activity monitoring, market data analysis, sensor networks, etc.) [6]. In the last years, Complex Event Processing (CEP) has emerged as an important trend in industry applications that need to detect situations in real or near real time [6]. CEP contributes to the improvement of operational situational awareness in many business scenarios, from network management to business optimization, resulting in enhanced situational knowledge and the ability to sense, detect and respond to situations more accurately [7].

The main goal of this master thesis is the development of a system that uses Complex Event Processing to recognize situations based on sensor data. Situations are specified using a model called Situation Template [3] containing all essential information that the Situation Recognition System needs for their recognition, i.e., the conditions that have to apply for their occurrence. Section 1.1 describes the problem this work aims to solve and the principal objectives. Section 1.2 describes three motivating scenarios that are used throughout this work to explain the approach.

1.1 Problem Definition and Objectives

The cloud-based situation recognition service SitRS [3] has been developed within the scope of the DFG[1] project SitOPT [8] at the University of Stuttgart. This service can be used to integrate things into the internet by deriving their situational state based on sensor data. Situations are modeled as Situation Templates, which are mapped onto event- and flow-based representations (executable Situation Templates). These representations are then deployed to an execution engine, for example Node-RED[2], in order to recognize the occurrence of the modeled situations. However, the prototypical implementation of this service using Node-RED revealed some limitations during the recognition of situations, mainly in respect to scalability and parallelism. By executing Situation Templates inside a single runtime environment, it has been shown that the runtime highly increases with the number of situations being monitored in parallel. Moreover, the current Situation Template schema only allows to express situations that compare sensor data with fixed predefined values. Conditions based on time are not yet supported. Finally, intermediate data is stored in a data cache from which the relevant data for the situation recognition is pulled in predefined time intervals, which leads to stale sensor data [3]. Therefore, this work aims to further improve the existing service by:

- **Increasing the powerfulness of Situation Templates.**

  Modifications of Situation Templates are required in order to support time-based conditions. In this way, conditions such as "temperature greater than 90 degrees for 10 seconds" can be defined. Further modifications are required to enable data from different sensors to be compared, for example "temperature of sensor A greater than temperature of sensor B". Moreover, Situation Templates should

[1]http://www.dfg.de/
[2]http://www.nodered.org/

also support conditions that aggregate the data from different sensors, such as "average temperature of sensors A and B greater than 90 degrees".

- **Employing a direct sensor push approach.**

  Making sensor data available to the Situation Recognition System through a direct approach, i.e., without caching sensor values, can lead to improvements regarding efficiency and correctness for the recognition of occurring situations.

- **Implementing mappings from Situation Templates to CEP queries.**

  In this work, situations are defined as Situation Templates, which can be mapped onto representations to be deployed into execution environments. The mapping from Situation Templates to CEP queries is necessary in order to support the situation recognition using CEP execution engines. By using mappings, an abstraction is provided that enables the modeling of situations in an easy way. This relieves the user from modeling situations directly as CEP queries, which can be long and complicated depending on the situation to be modeled.

- **Executing the situation recognition using a CEP engine.**

  Finally, the execution of the situation recognition using CEP technologies enables enhancements in respect to efficiency, scalability and parallelism. The analysis and choice of a suitable CEP engine are an essential part of this work.

1.2 Motivating Scenarios

This section describes motivating scenarios for the Situation Recognition System resulting from this master thesis and are used throughout this thesis to explain the approach. It exclusively presents scenarios related to situations in manufacturing environments, however, the results of this work can be applied to many other fields as well.

- **Monitoring objects on a conveyor belt.**

  The goal of this scenario is to monitor several distance sensors strategically positioned near a conveyor belt. It aims to recognize when (i) an object on the conveyor belt is upside down (Figure 1.1), or (ii) an object is not positioned correctly on the conveyor belt (Figure 1.2).

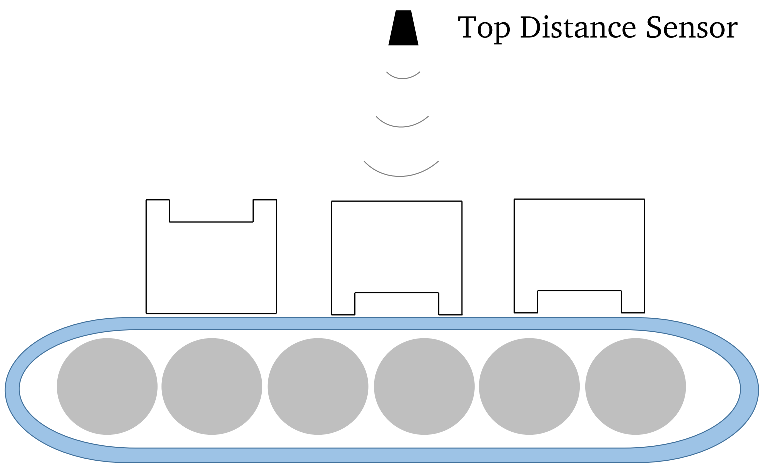


Figure 1.1: Distance sensor placed above a conveyor belt to detect if an object is upside-down

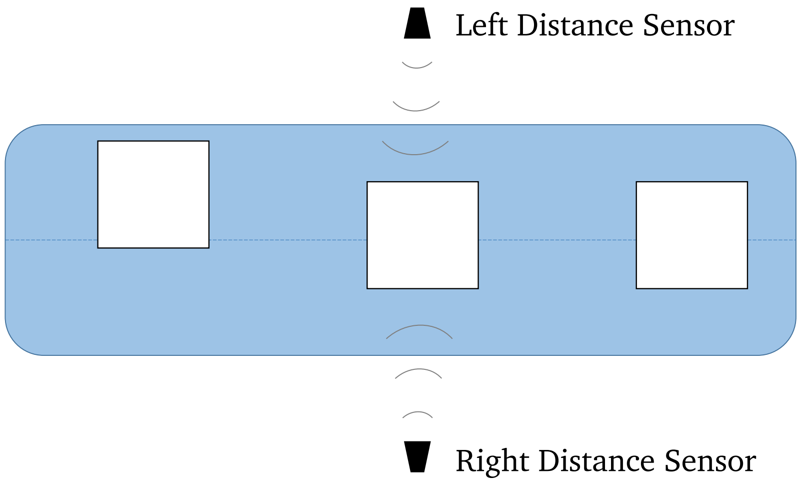


Figure 1.2: Distance sensors placed at a conveyor belt to detect if objects are wrongly positioned

- **Monitoring level of containers.**

  The goal of this scenario is to monitor distance sensors positioned above different types of material containers (Figure 1.3). It aims to recognize when (i) the produced goods in a container have reached a maximum level, which means this container must be emptied, or (ii) the materials in a container have reached a minimum level, which means this container must be refilled.

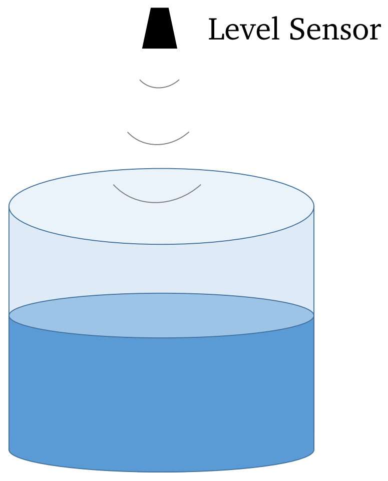


Figure 1.3: Distance sensor placed above a container

- **Monitoring machine status.**

  The goal of this scenario is to monitor temperature sensors of a machine and distance sensors positioned above containers that supply this machine with material. It aims to recognize when (i) the machine is overheated, or (ii) a material container has reached a minimum level, which leads the machine to be in a blocked state.

For each of the previously described scenarios, the Situation Recognition System developed in this work can additionally recognize when a situation stopped occurring because some reactions took place. The situation-aware applications interested in the situations can register themselves to the recognition system in order to receive notifications when a situation is recognized and possibly react to it properly. The sensor data used to recognize the situations is provided through a push approach, which will be explained, along with how these scenarios were realized, in Chapter *4 – Situation Recognition based on CEP*.

1.3 Structure of the Thesis

The remainder of this master thesis is structured as follows: Chapter 2 gives an overview of the Internet of Things, Complex Event Processing and Situation Templates, which are essential topics to comprehend this master thesis. Chapter 3 presents the analysis of three CEP engines and motivates the choice of the CEP engine used in this work. Furthermore, how the Situation Recognition System was realized using the chosen CEP engine is explained in Chapter 4. Chapter 5 evaluates the prototypical implementation of the Situation Recognition System. The related work of this master thesis is presented in Chapter 6. Finally, Chapter 7 summarizes this work and describes future work.

2 Basic Concepts

This chapter covers important topics necessary to comprehend the concepts of this master thesis. An overview of the Internet of Things, Complex Event Processing, and Situation Templates is given in the following sections.

2.1 Internet of Things

The Internet of Things (IoT) paradigm envisions the pervasive presence of an assortment of things, interconnected through network connections and having unique addressing schemes [6]. These things are then able to interact and cooperate with each other to create new applications and reach common goals. The IoT contribution is the value of information generated by the various interconnected things and the consequent derivation of this information into knowledge that can be used to provide better quality of life to the society. The goal of the IoT is *"to enable things to be connected anytime, anyplace, with anything and anyone ideally using any path/network and any service"* [6]. Furthermore, it aims to integrate real world information into networks, services and applications by using technologies such as Wireless Sensor Network (WSN) and Radio Frequency Identification (RFID). Advances in wireless networking technology and standardization of communication protocols make it possible to collect data from wireless identifiable devices almost anywhere at any time [9, 6]. The main goals of IoT are the creation of smart environments and self-aware things (e.g., smart buildings, smart cities, smart transport, smart homes, etc.), which facilitates the development of applications for many different sectors of our daily live (climate, food, energy, mobility, digital society, health, manufacturing, etc.) [9].

2.2 Complex Event Processing

Etzion and Niblett [2] define an *event* as something that has happened in the real world, within a particular system or domain. The word *event* is also used to imply a programming entity or object that represents such an occurrence in a computing system. Events, whether simple or complex, are common in our everyday live, for example the arrival of an email, the missing of a flight, which causes the missing of a connection, a broken machine at the production line, etc. The main reason for event recognition is to have the opportunity to react to them. The reaction might be as simple as responding to an email or something more complicated as choosing among alternatives, for example, if a machine is malfunctioning in a production environment, possible reactions are, depending on the currently available budget, to repair the machine or to replace it. Events that might require a reaction are called *situations* [2]. Detecting and reporting them so that they can be reacted to is one of the main topics in *event processing*, which can be simply defined as *"computing that performs operations in events"* [2]. These operations include filtering certain events and examining a collection of events to find a particular pattern [2].

The ability to detect a *pattern of events*, i.e., relationships among events, is a very powerful feature in event processing [10]. Events tend to arrive in patterns, mixed up with unrelated events, where one event alone might carry a piece of information that only makes sense together with other related events [11]. By analyzing the several events of a pattern, we gain knowledge about what is happening or going to happen, where it is happening, and why. A pattern might contain only one significant event at a given moment or contain hundreds of events, arriving during a millisecond or maybe spread out over days. Because of that, it may not be possible to know in advance how long the pattern will take to match, but event processing technologies still enable the detection of interesting patterns [11].

Furthermore, event processing encompasses the following characteristics that open up to a rich range of possibilities while developing highly scalable and dynamic systems:

- *Abstraction.* The event processing logic can be separated from the application logic. This allows the event processing logic to be modified without having to change applications [2].
- *Decoupling.* The events produced by one application can be consumed by completely different applications [2]. There is no need for producing and consuming applications to know about each other [10].
- *Real-world focus. "Event processing frequently deals with events that occur, or could occur, in the real world"* [2].

In the 1990s, a set of principles of event processing had emerged [2]. These principles, called *Complex Event Processing (CEP)*, encompasses methods, techniques and tools for processing events while they occur, i.e., in a continuous and timely fashion [1]. Lower-level events are derived into valuable higher-level knowledge, known as complex events. A *complex event* is a situation that can only occur if several other events that are related to each other occurred [12]. CEP has many independent roots in different research fields, including discrete event simulation, active databases, network management, and temporal reasoning. But in recent years, CEP has emerged as an important trend in industry applications [1, 6]. CEP provides a natural decoupling of basic events with a strong relationship to the semantics of the underlying technology (e.g., sensor readings) and complex events closer to the semantics of the application. In this way, it enables information systems to perform independent reconfigurations on the technical and application level [10]. Moreover, the stepwise correlation of events reduces the data load and thus contributes towards a highly scalable information system [10].

There are two types of Complex Event Processing [1]: at the first type, complex events are specified as a-priori known patterns. Such patterns are defined using event query languages, which offer means to detect complex events efficiently. At the second type, complex events should be detected from previously unknown patterns. In this case, machine learning and data mining methods are used. This work focuses on the first type of CEP, in which situations are well-known and defined as complex events with help of event query languages. In contrast to database queries, which are executed on a finite set of data, event queries are evaluated while the events occur, continuously on a (conceptually) infinite stream of events [1]. Figure 2.1 shows the difference between database and event queries, in which database queries are executed once on a finite set of data and event queries are continuously executed on an infinite stream of events.

2.2.1 Example Applications

This section shows a few examples of how Complex Event Processing can be used for today's applications [2].

- A financial system to detect frauds: it collects events from banking systems and analyzes them. Certain patterns of activity might suggest that a person is possibly (but not necessarily) in the process of committing a fraud. In this system, event processing is used to detect evolving phenomena.
- A manufacturing management system to diagnose mechanical failures based on observable symptoms: in this case the events are symptoms that describe things not working properly. The main purpose of the event processing is to find the root cause of these symptoms. This system demonstrates the use of event processing for problem determination and resolution.

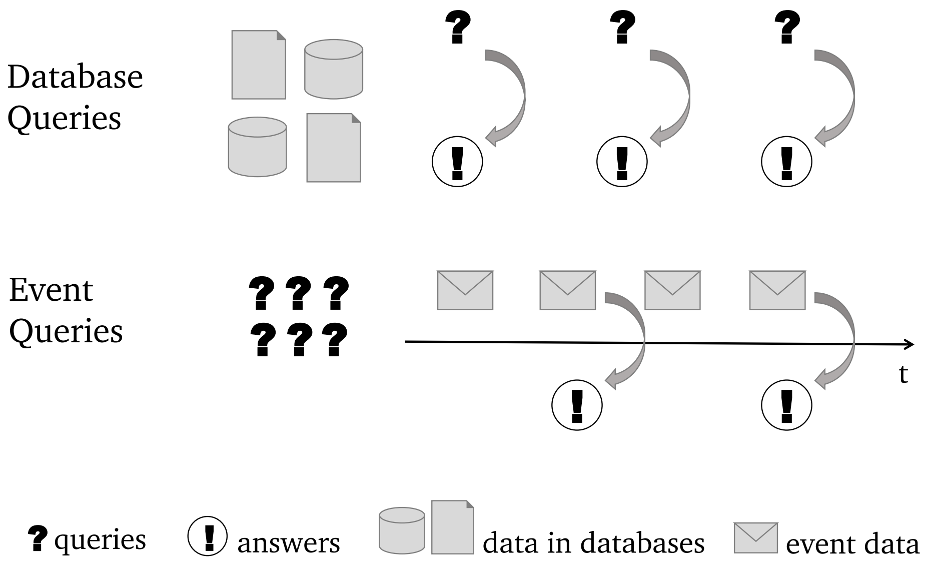


Figure 2.1: Difference between database and event queries based on [1]

These applications are different from one another, but they follow the same methodology: (i) events are reported, sometimes by multiple event producers, (ii) some processing of the events is done, and (iii) additional events are created and consumed either by humans or by automated processes. Many event processing applications keep the event processing logic separate from the event producers and consumers, as depicted in Figure 2.2. By adopting this pattern, the event processing logic can be done on a dedicated event processing platform. For examples of available platforms, cf. Chapter *3 – Analysis – Complex Event Processing Engines*. Such event processing platforms provide at least: a language for expressing event processing logic, a runtime to execute event processing logic and an event distribution mechanism.

2.3 Situation Templates

Hirmer et al. [3] argue that, to facilitate situation-awareness for the Internet of Things, different levels of data processing are needed. The reason for this is that sensors provide extensive amounts of low-level data, which are not easy to handle. Therefore, sensor data needs to be interpreted in order to derive high-level situations, which can be understood and further processed easier than low-level data.

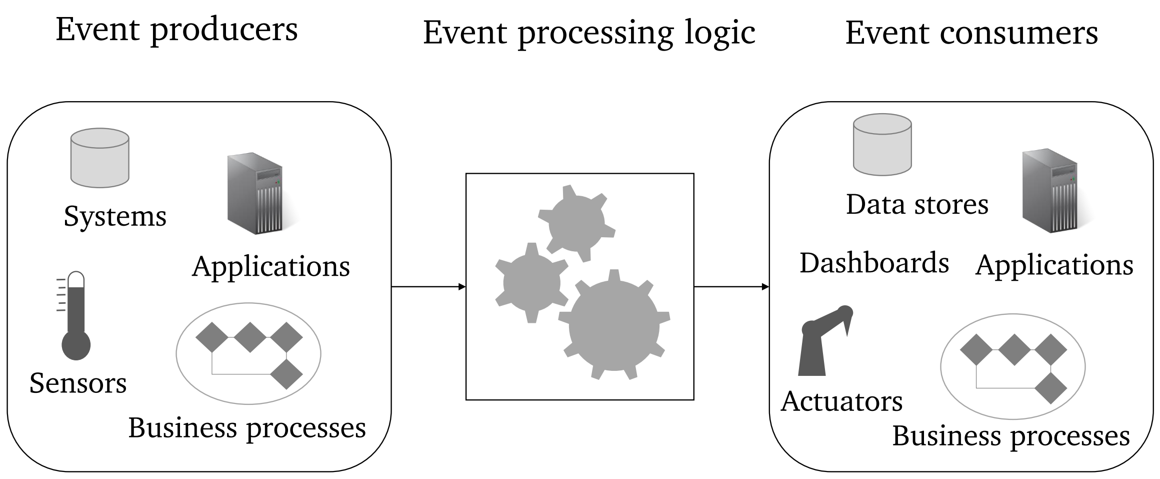


Figure 2.2: Structure of an event processing application based on [2]

Figure 2.3 shows an overview of the different processing levels presented in [3]. The first level, called *data level*, contains sensor devices and only raw sensor data is available to it. This data is not easy to process and the context information (e.g., the data type, the relation of the data to a thing) is not available at this level. Because of that, sensors push their data to the next level, called *information level*. At this level, sensor data (e.g., temperature) are enhanced with information about their relations to real world things (such as machines in a smart factory), turning in this way into information about the environment. Based on this information, situations are then derived from sensor data, which leads to knowledge about the smart environment. This knowledge facilitates producing situation-aware applications, since it can be processed on a higher-level of abstraction.

To be able to recognize situations based on sensor data, it is necessary to define them with all crucial information for their recognition. Hirmer et al. [3] provide a model, called *Situation Template (ST)*, that contains the monitored sensors and the conditions that have to match for a certain situation to be recognized. This model is based on Situation-Aggregation-Trees (SAT), which are directed graphs forming a tree structure. The branches of a SAT are aggregated bottom-up, which leads to a single root node that represents the situation [13].

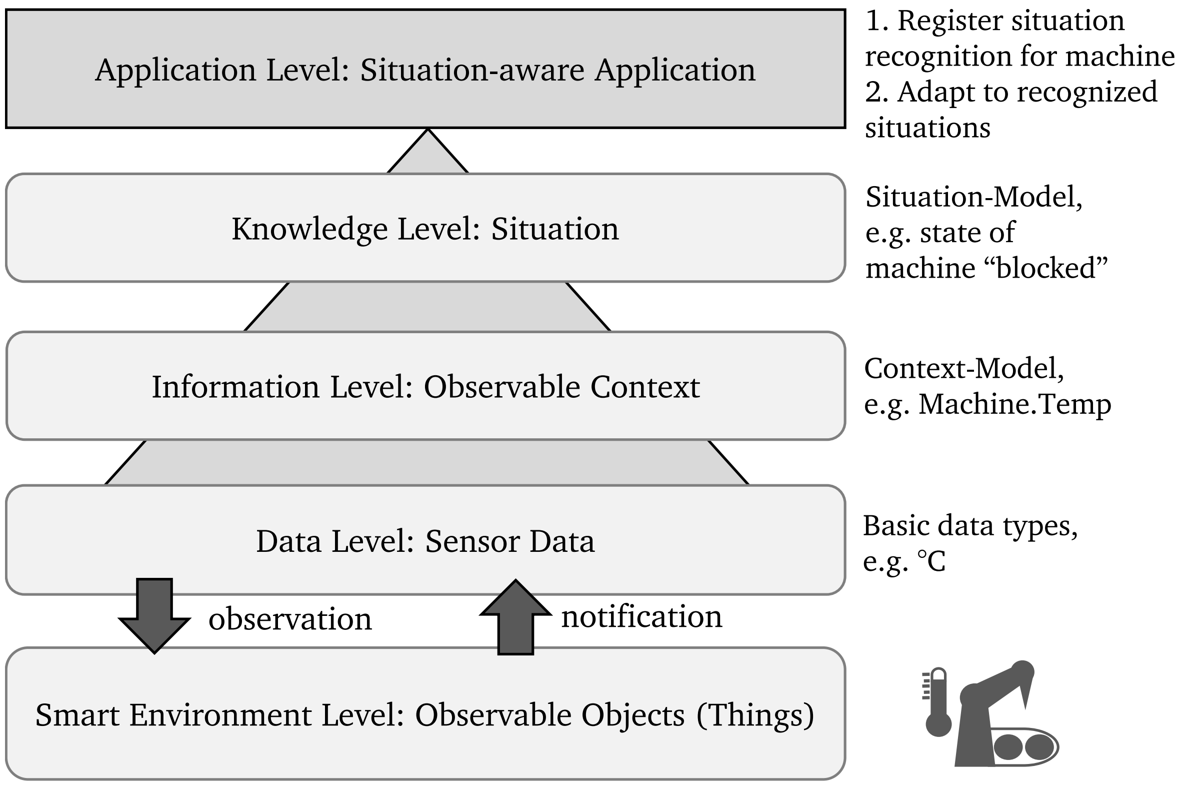
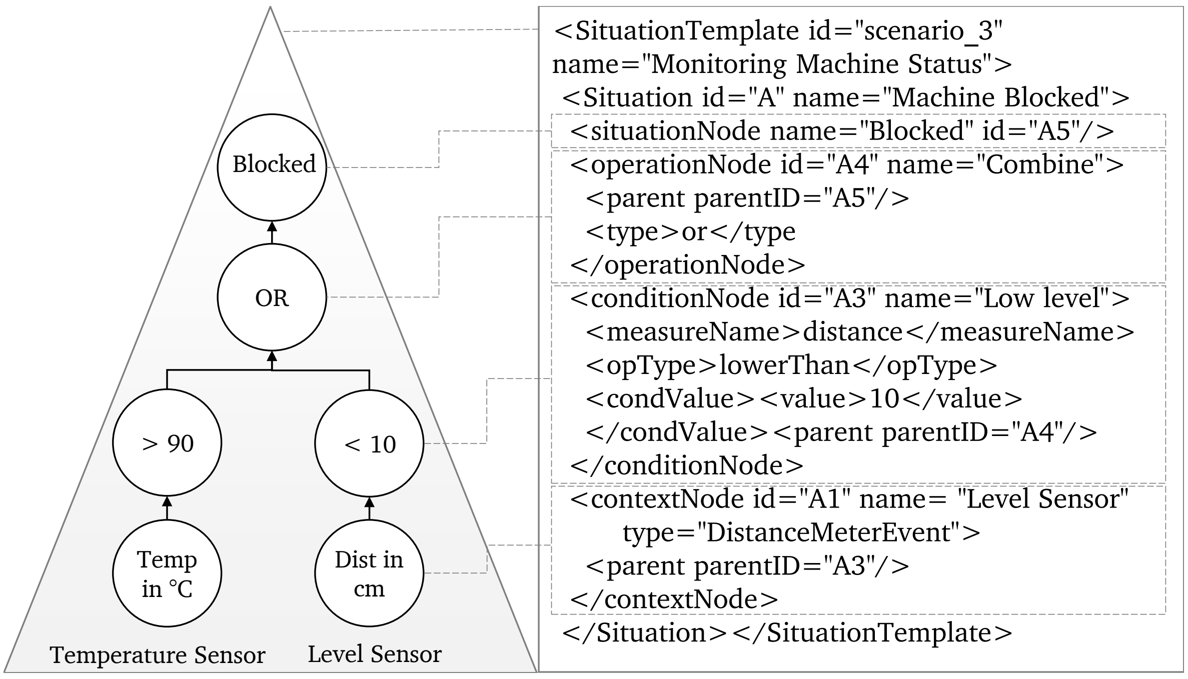

Application Level: Situation-aware Application
1. Register situation recognition for machine
2. Adapt to recognized situations
Knowledge Level: Situation
Situation-Model, e.g. state of machine “blocked”
Information Level: Observable Context
Context-Model, e.g. Machine.Temp
Data Level: Sensor Data
Basic data types, e.g. °C
observation
notification
Smart Environment Level: Observable Objects (Things)

Figure 2.3: Process levels based on [3]

Hirmer et al. [3] have proposed a XML-based schema for representing SATs. An example of a Situation Template modeled in XML for the scenario “monitoring machine status” is depicted in Figure 2.4, which is basically composed by various nodes from different types reflecting the aforementioned processing levels: *context nodes* are leaf nodes representing the monitored sensors of a certain thing. Such nodes correspond to the data level (sensor devices) and information level (sensors’ relations to things). Context nodes are connected to *condition nodes*, which filter sensor data based on defined conditions. Condition nodes can be aggregated by *operation nodes* using logical operations until the root node, i.e., the situation node, is reached. Finally, the *situation node* represents the situation to be recognized. The combination of condition, operation and situation nodes corresponds to the knowledge level, where sensor data is aggregated, interpreted and derived to situations.

Once situations are defined as Situation Templates, they can be mapped onto representations that can be deployed into execution environments, such as Node-RED, Esper, etc. The mapping from a Situation Template to an executable representation



```
<SituationTemplate id="scenario_3"
name="Monitoring Machine Status">
 <Situation id="A" name="Machine Blocked">
  <situationNode name="Blocked" id="A5"/>
  <operationNode id="A4" name="Combine">
    <parent parentID="A5"/>
    <type>or</type
  </operationNode>
  <conditionNode id="A3" name="Low level">
    <measureName>distance</measureName>
    <opType>lowerThan</opType>
    <condValue><value>10</value>
    </condValue><parent parentID="A4"/>
  </conditionNode>
  <contextNode id="A1" name= "Level Sensor"
        type="DistanceMeterEvent">
    <parent parentID="A3"/>
  </contextNode>
 </Situation></SituationTemplate>
```

Figure 2.4: Situation Template modeled in XML based on [3]

is necessary in order to enable support of different execution engines. In this way, a situation recognition system that uses Situation Templates can avoid being dependent on a specific engine [3].

3 Analysis – Complex Event Processing Engines

An essential task of this master thesis is the analysis and consequently the choice of a suitable Complex Event Processing engine, which will process the execution of the situation recognition.

Situations are modeled as Situation Templates, therefore the implementation of mappings from Situation Templates to corresponding CEP queries that recognize the modeled situations is required. It is crucial that the CEP engine provides an event query language capable to cope with the powerfulness of Situation Templates. Furthermore, it is also expected from the CEP engine to execute event queries in parallel, to be scalable towards a large number of queries and to detect and notify situations in real or near real time.

Several complex event and data stream processing engines were considered to be used in this master thesis, namely the open-source engines *Esper* [14], *Siddhi* [15], and *Odysseus* [16]. Sections 3.1, 3.2 and 3.3 give an overview of the engines and Section 3.4 evaluates them in order to motivate the engine choice for this work. These are some important aspects taken into account during the analysis of the engines:

Access to events participating in situations. Such events contain the relevant data to decide whether and how to react to situations [1]. When a situation is recognized, the CEP engine must provide means to access the events involved in this situation, in this way allowing the notified applications to react to it properly. For example, if a machine is malfunctioning, events can provide the specific location or identifier of the machine, so that an application can initiate its repair or replacement. Furthermore, the CEP engine must provide straightforward ways to push events into the engine, in order to facilitate the employment of a sensor data push approach.

Composition of events. Events tend to arrive in patterns, where one event alone might carry a piece of information that only makes sense together with other events [2]. The CEP engine must provide an event query language that allows to combine several individual events, whether from the same source or different sources, so that their combined occurrences result in a complex event [1]. This will enable the recognition of situations based on data from different types of sensors, e.g., the scenario “monitoring machine status” described in 1.2, which uses temperature and distance sensors. It will also enable data coming from different sensors of the same type to be compared and aggregated. Furthermore, some situations need to be expressed as specific events that occur within a particular time interval or in a specific order. Therefore, it must be possible to create queries that involve temporal conditions (correlation of events) and that match patterns. For example, if a SmartHome application receives an *alarm-clock* event and motion is detected in the corridor within 5 minutes, these events are correlated and could mean the user got up and the application should start the coffee machine. The application could also power off some devices (radio, television, etc.) when it matches a pattern such as the user closed the door after turning off the lights.

Accumulation of events. Queries involving aggregation of events or the absence of a particular event are not applicable to infinite streams [1]. Consequently, the event query language must allow to formulate queries that run on finite extracts of event streams, known as windows. They can be sliding windows, which are moving windows of past events based on a time interval or a number of events, or tumbling windows, which buffer events and release them once on every specified time interval or number of events. Especially important for this work is the possibility to run queries to recognize situations happening for a specific period of time, so that conditions such as “distance lower than 10 cm for 10 seconds” can be supported. Such a condition can be formulated as the absence of an event measuring a distance greater than or equal to 10 cm in a time window of 10 seconds.

3.1 Esper

This section is mostly based on [14, 17].

Esper is an open-source Java component for Complex Event Processing and Event Series Analysis, which has been developed to address the requirements of applications that need to process a large volume of incoming events and respond to situations of interest.

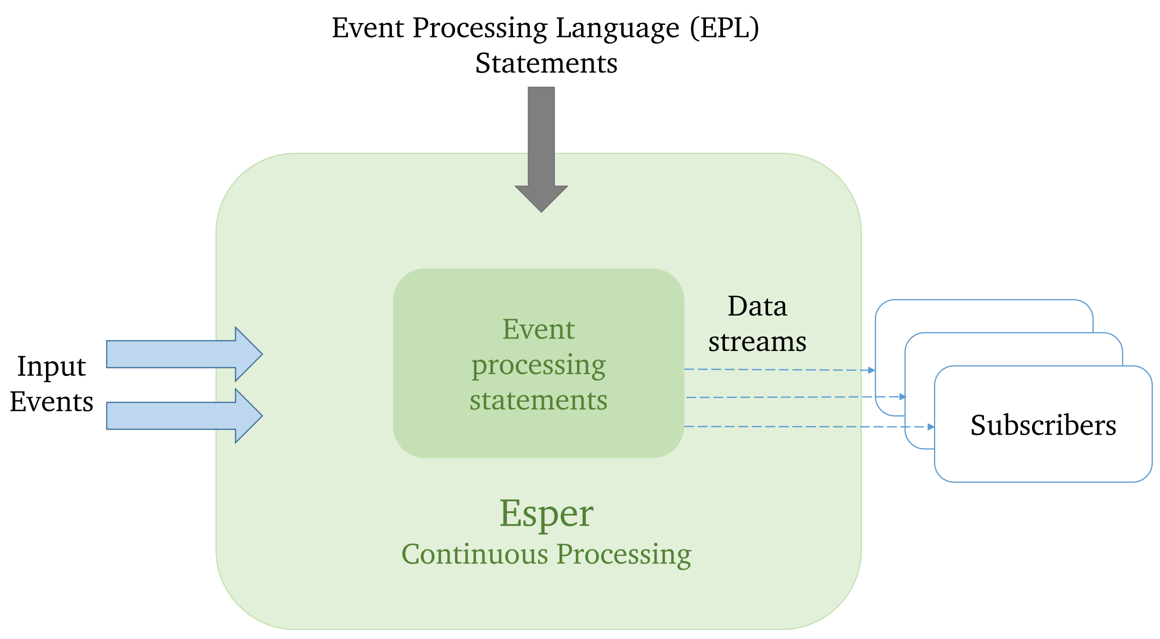


Figure 3.1: The Esper engine

An overview of the Esper engine and its continuous processing is given in Figure 3.1. Esper provides a runtime API to send input events to the engine. Events can be represented in different formats, e.g., POJO, Map, Object-array, or DOM Node. Continuous queries, called statements in the context of Esper, specify how input events should be processed and can be added, started, and stopped while the Esper engine is running. Listing 3.1 shows how to use the Esper API to obtain an engine instance, register statements, and send events to the engine. The example query returns the average temperature over all *TemperatureEvent* events that arrived in the last 30 seconds.

Listeners can be attached to statements, which are invoked by the engine in response to events that change the result of a statement. An example of a listener, which needs to implement the *UpdateListener* interface, and how it is attached to a statement is depicted in Listing 3.2.

Queries are formulated in a SQL-based language, which is called Event Processing Language (EPL). In respect to the composition of events, the EPL enables to express complex matching conditions that include temporal windows and joining two or more different event streams. Esper offers an event pattern language that can be used to specify event pattern matching based on expressions. It allows to derive complex

Listing 3.1 Adding a statement and sending events to the Esper engine

```
// obtaining engine instance
EPServiceProvider epService = EPServiceProviderManager.getDefaultProvider();

// registering a continuous query
String expression = "select avg(temperature)
                     from org.myapp.event.TemperatureEvent.win:time(30 sec)";
EPStatement statement = epService.getEPAdministrator().createEPL(expression);

// sending events
TemperatureEvent event = new TemperatureEvent("thing", 25.0);
epService.getEPRuntime().sendEvent(event);
```

Listing 3.2 Adding listener to an EPL statement

```
public class MyListener implements UpdateListener {
   public void update(EventBean[] newEvents, EventBean[] oldEvents) {
      EventBean event = newEvents[0];
      System.out.println("avg=" + event.get("avg(temperature)"));
   }
}

//adding listener to statement
MyListener listener = new MyListener();
statement.addListener(listener);
```

events from simple events by defining patterns that match sequences of events, absence of events or combinations of events. Pattern expressions can consist of filter expressions combined with the following pattern operators: operators controlling the pattern finder creation and termination (every), logical operators (and, or, not), event order operators (followed-by), guards filtering out events and causing termination of the pattern finder (such as timer:within), and observers for time events (such as timer:interval, timer:at). A sample pattern expression is shown in Listing 3.3. This pattern matches if the temperature of a machine increases by 5 degrees within 5 min. In the example, the tags e1 and e2 were assigned to the events in the pattern. Tags are important because only tagged events are placed into the output events that listeners will receive when the pattern matches. Moreover, Esper allows to filter events using user-defined static methods that return a Boolean value (e.g., *PositionEvent(MyLib.isInRange(x, y))*) and it is possible to do pattern matching based on regular expressions.

In respect to the accumulation of events, Esper allows the use of *data windows* (time, length, etc.), which inform the engine about how long to retain relevant events or under what conditions events can be discarded. Data windows can be *sliding* or

Listing 3.3 Example of an event pattern in Esper

```
every e1=TemperatureEvent -> e2=TemperatureEvent(e1.machine_id = machine_id, temp >=
    e1.temp + 5) where timer:within(5 min)
```

tumbling. Additionally, Esper provides *named windows*, which are globally visible data windows that share sets of events between queries efficiently, removing the need to keep the same events in multiple places.

Furthermore, Esper provides scalability in the face of large numbers of continuous queries and a high degree of parallelization while processing the same or multiple queries [14]. According to the Esper documentation, it is not limited to be executed on a single machine, has no dependencies on external services and does not require any external storage. It offers high throughput by processing between 1,000 to 100k events per second and low latency for reacting to situations (from a few milliseconds to a few seconds) [14].

3.2 WSO2 Siddhi

Siddhi is an open-source, lightweight, easy-to-use Complex Event Processing engine. It can process many data streams and notify complex events according to the event queries specified by the user. The development of Siddhi started within a research project at the University of Moratuwa and has been further developed by WSO2 Inc.[1] [18].

An overview of the Siddhi engine and its relations to events and queries is given in Figure 3.2. Siddhi is the standalone version of the WSO2 Complex Event Processor (WSO2 CEP)[2] and can be easily embedded within a Java program. However, by using WSO2 CEP running as a server, it is possible to send events to the system using, for example, Web Services or emails. Siddhi receives events from the different event sources through input adapters. Input events can be represented in different formats (e.g., POJO, XML) but they are converted to tuples before they are passed to the Siddhi engine. Then, Siddhi processes the incoming events according to the active queries and generates output events as soon as queries match. Those events are either consumed internally or sent to external subscribers through output adapters [19]. Listing 3.4 shows how to programmatically define event streams and send data to the Siddhi

[1]http://www.wso2.com
[2]http://www.wso2.com/products/complex-event-processor/

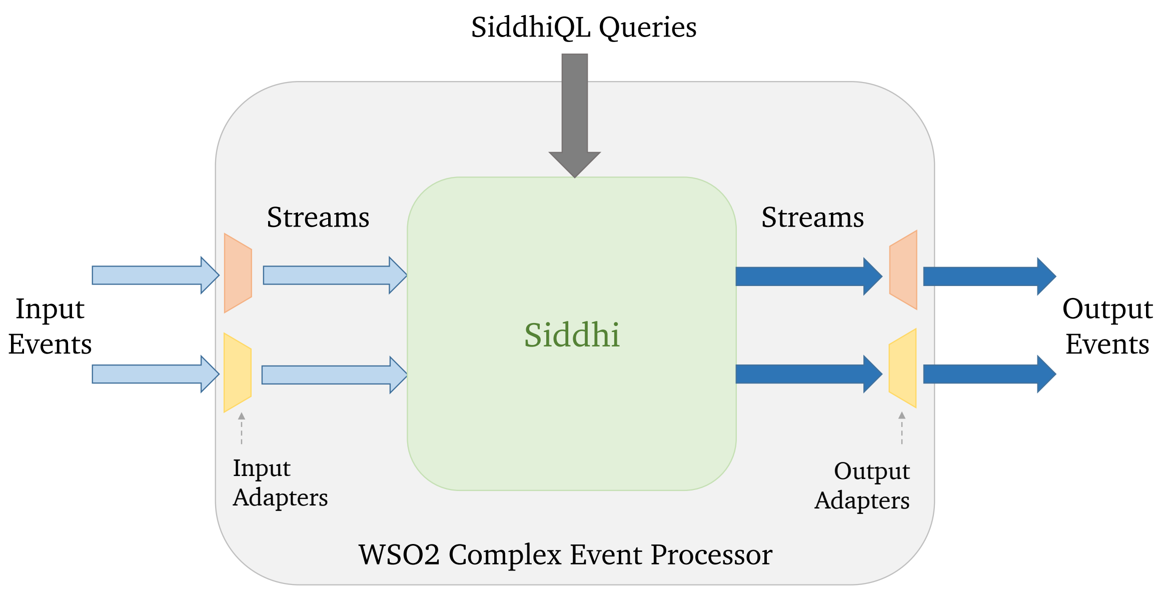


Figure 3.2: The Siddhi system

Listing 3.4 Defining streams and sending data to the Siddhi engine

```
// Create Siddhi Manager
SiddhiManager siddhiManager = new SiddhiManager();

// Define stream
siddhiManager.defineStream("define stream distanceStream ( id int, name string, value
    float )");

// Pushing data
InputHandler distanceStream = siddhiManager.getInputHandler("distanceStream");
distanceStream.send(new Object[]{1, "distance", 20});
```

engine. In this example, the defined event stream *distanceStream* receives events representing measurements of distance sensors. The event is represented as a tuple containing the sensor identifier, the measure name and the measured value. An event is sent to the Siddhi engine by pushing it into the defined stream.

Siddhi provides a SQL-like query language (SiddhiQL) designed to process the event streams, which are logical series of events ordered in time, and to identify complex event occurrences. This language facilitates temporal correlations between event streams with historic data, which enables the detection of complex event patterns

Listing 3.5 Example of a SiddhiQL query and callbacks

```
// Creating query
String queryReference = siddhiManager.addQuery("from distanceStream [ value <= 10 ] " +
                                   "select id, value " +
                                   "insert into outStream;");

siddhiManager.addCallback(queryReference, new QueryCallback() {
      @Override
      public void receive(long timeStamp, Event[] inEvents, Event[] removeEvents) {
                  EventPrinter.print(timeStamp, inEvents, removeEvents);
      }
});

siddhiManager.addCallback("outStream", new StreamCallback() {
      @Override
      public void receive(Event[] events) {
                  EventPrinter.print(events);
      }
});
```

[15]. Queries formulated in SiddhiQL are then submitted to Siddhi, converted to a runtime representation and deployed into the Siddhi engine core. Queries can be added or removed while the Siddhi engine is running. Callbacks can be added to the Siddhi engine to get notified when a specific query matches. Furthermore, callbacks to a specific event stream can also be added. Listing 3.5 shows a simple SiddhiQL query, which filters events with a distance value lower than 10 cm, and furthermore shows how to add it to the Siddhi engine, as well how to register callbacks. Besides formulating queries directly in SiddhiQL, it is also possible to build query objects using a Java API.

In respect to the composition of events, joins can be used to merge up to two streams based on a given condition, where each stream must be associated with a window. During a join, each incoming event on each stream is matched against all events of the other input event stream window. An output event is generated for all matching event pairs. Moreover, SiddhiQL allows pattern processing, where event streams can be correlated over time and event patterns can be detected based on the order of arrival, however, there can be other events in between the events matching the pattern condition. Events can be correlated over one or multiple input streams and each matched event needs to be referenced in order to be available for further processing and output generation. Listing 3.6 shows a pattern that matches if the temperature of a machine increases by 5 degrees within 5 min. The *every* keyword makes sure that the pattern matching does not stop after the first match.

Listing 3.6 Example of an event pattern in Siddhi

```
define stream TempStream(machineID long, temp double);

from every( e1=TempStream ) -> e2=TempStream[e1.machineID==machineID and (e1.temp + 5)
    <= temp ]
within 5 min
select e1.machineID, e1.temp as initialTemp, e2.temp as finalTemp
insert into AlertStream;
```

SiddhiQL also provides means for sequence processing, where event streams can be correlated over time and event sequences are detected based on their arrival order, with the difference that there cannot be other events in between the events matching the sequence condition [20]. Furthermore, Siddhi does not support negation and handling out of order events [19].

In respect to the accumulation of events, windows are provided in order to capture a subset of events from input event streams. Each input stream can have at most one window. Siddhi supports sliding and batch (tumbling) windows, which can be further divided into *time-based* and *length-based* [19].

Furthermore, on a PC with a 4 Core processor and 4 GB of memory, Siddhi can evaluate per second up to 6M events generated from the same JVM, and about 100.000 events per second over network connection [15].

3.3 Odysseus

This section is mostly based on [16].

Odysseus is an open-source in-memory data stream management system (DSMS) designed for the real time processing of large volumes of data. It is a platform that provides different data processing steps (also called operators), such as filter and correlate, so that complex events and high-level information can be derived from simple events. Odysseus is a research prototype that has been continuously developed since 2007 by the University of Oldenburg and partners[3].

Cugola and Margara [21] distinguish between the data stream processing model and the complex event processing model as follows. The first model processes data streams coming from different sources in order to produce new data streams as output,

[3]https://www.uni-oldenburg.de/informatik/is/forschung/projekte/odysseus/

where this processing model can be considered as an evolution of the traditional data processing of database management systems (DBMS) [21]. On the other hand, the complex event processing model interprets the flowing information as notifications of events happening in the external world, which have to be filtered and combined to understand what is happening [21]. It focuses on detecting particular patterns of events whose occurrences have to be notified to the interested applications [21].

Odysseus implements a client-server architecture and is composed of the Odysseus Server and the Odysseus Studio. The *Odysseus Server* is the core of the system and provides the data processing operations, the query interface, and the user management. The *Odysseus Studio* is an user interface developed for the administration of the Odysseus Server and for the creation of queries, where it contains an integrated editor to formulate them. Alternatively to the Odysseus Studio, the Odysseus Server can also be administrated via a Web Service interface.

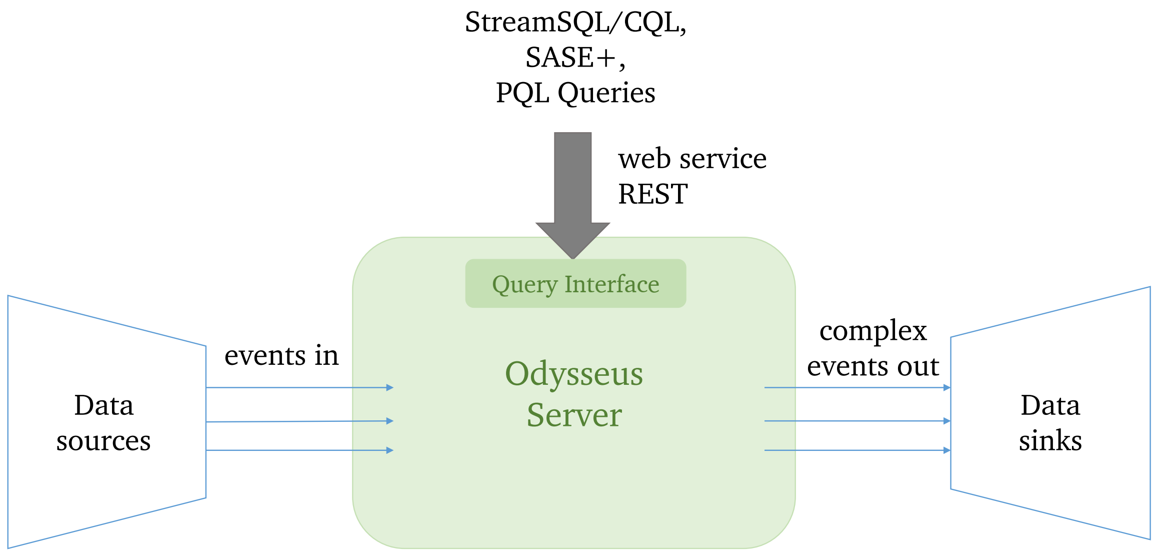


Figure 3.3: The Odysseus Server

An overview of the Odysseus Server and its relations to events and queries can be seen in Figure 3.3. Data sources send their data actively so that the processing is done reactively when necessary. Then, Odysseus processes the incoming events according to the active queries and forwards the results to applications through data sinks. A flexible adapter interface for connecting data sources and data sinks is provided. Listing 3.7 shows a query to create an incoming connection for a data source (outgoing

Listing 3.7 Example of a query to connect a data source to Odysseus

```
CREATE STREAM distanceStream (id INTEGER, name STRING, value FLOAT)
   WRAPPER 'GenericPush'
   PROTOCOL 'CSV'
   TRANSPORT 'TCPClient'
   DATAHANDLER 'Tuple'
   OPTIONS ('port' '123', 'host' 'example.com')
```

Listing 3.8 Example of a CQL query in Odysseus

```
SELECT id,value FROM distanceStream WHERE value <= 10
```

connections can be created by using CREATE SINK instead of CREATE STREAM). In this example, a TCP connection to the specified host on the given port is opened, through which the data is pushed to the Odysseus system. The incoming CSV data is translated into a tuple of the form *(integer, string, float)*. Odysseus allows various event data formats, such as XML, CSV, HTML, JSON, or byte-based formats. It also provides some protocols and transport mechanisms (TCP connections, files, serial interfaces) but adding new ones is possible through interfaces and the component-based architecture.

When data sources and data sinks are connected to Odysseus, queries can be defined to tell Odysseus how the data should be processed. They are passed to the Odysseus server through a query interface, which is provided as a Web Service and as REST interface. They can be written in StreamSQL/CQL, a SQL-based language, or in Procedural Query Language (PQL). Listing 3.8 shows an example of a simple CQL query that filters events from the data stream *distanceStream* where *value* is lower than or equal to 10.

In respect to the composition of events, the join operator can be used to combine events from two different sources, but it is only possible if the time interval of the two streams overlap and the join predicate evaluates to true. Moreover, for detecting complex patterns the SASE+ language[4] can be used. For that, the additional plugin *CEP and Pattern Feature* needs to be installed in order to enable Odysseus to perform Complex Event Processing.

The SASE+ language focuses on temporal event patterns where they can specify a sequence of events occurring in a specific order (sequencing) or the non-occurrences of events (negation). A pattern can also be used to extract a finite number of events

[4]http://avid.cs.umass.edu/sase/index.php?page=language

Listing 3.9 Data source definition in PQL

```
Arduino := ACCESS({ SOURCE = 'Arduino',
TRANSPORT = 'RS232',
DATAHANDLER = 'Tuple',
WRAPPER = 'GenericPush',
PROTOCOL = 'CSV',
SCHEMA = [['temperature', 'DOUBLE'], ['distance', 'DOUBLE']],
OPTIONS = [['port', '/dev/ttyACM3'], ['baud', '9600']] })
```

Listing 3.10 Example of an event pattern in Odysseus

```
match = SASE({schema=[['temperature','Double']],
type='Result',
query='PATTERN SEQ(Arduino+ a[]) WHERE skip_till_any_match(a[]){
  a[1].temperature >= 0.8 * a[a.LEN].temperature
 } WITHIN 60 seconds RETURN a[a.LEN].temperature'
}, Arduino)
```

with a particular property (kleene closure). Furthermore, different events can be compared via value-based and temporal constraints (parameterized predicates) and sliding windows can be defined for patterns that need to occur within a specified period of time [22].

Listing 3.10 shows a query written in PQL to detect a pattern in the stream of the defined source in Listing 3.9. The SASE operator takes three parameters: the output schema, the type name of the output, and the query, which receives a pattern description written in the SASE pattern language. The pattern should match when the temperature drops by more than 20% compared to some value during the last 60 seconds, for example, when someone opened a window in the room. When Odysseus detects such a situation, it is possible to send the result of the SASE operator to an application, for example, through a HTTP Request (see Listing 3.11).

Listing 3.11 Generating an Odysseus notification in PQL

```
out = SENDER({ SINK = 'Sink',
TRANSPORT = 'HTTP',
DATAHANDLER = 'Tuple',
WRAPPER = 'GenericPush',
PROTOCOL = 'CSV',
OPTIONS = [['uri', 'http://www.example.com'], ['method', 'post']]
}, match)
```

In respect to the accumulation of events, Odysseus provides windows, which are subsets of elements that should be treated together, for example aggregated. It provides sliding windows and tumbling windows, which can be time-based (e.g., the last 10 seconds) or element-based (e.g., the last 10 elements).

Furthermore, on a PC with Intel Core i5 2.5 GHz processor and 8 GB of memory, Odysseus can evaluate above 100.000 events per second and execute over 30,000 queries with more than 1,000,000 processing steps in parallel [16]. However, the processing speed strongly depends on the defined data sources and queries.

3.4 Evaluation

It has been shown that all three options are good equipped with powerful mechanisms to execute situation recognition by using Complex Event Processing (CEP) features. While Esper and Siddhi engines are lightweight Java components and can be easily embedded into any Java application, Odysseus offers a highly customizable OSGi-based framework for creating event stream management systems. In order to simply use the Odysseus Server, which does the actual needed processing, applications can integrate a provided Java web service client into themselves, to be able to define data sources, add, remove and run queries. Additionally, since Odysseus is primarily a data management engine, in order to use pattern matching, the *CEP Feature* plug-in needs to be installed to allow it to use event pattern matching. This is done via the user interface Odysseus Studio.

For all engines, events can be represented in different formats so that the application can choose which data format suits better. Input events can be easily made available to the execution engine through push mechanisms. Esper and Siddhi provide APIs to actively send events to them, while in Odysseus events will be made available to the engine through specified data sources, for example through a TCP connection or serial port. Esper and Siddhi allow to register listeners to queries, where they are informed about the result of queries, including the events involved in the recognized situations. In Odysseus, results are sent to the registered data sinks, for example, through a HTTP request.

All three engines can express event queries on SQL-based languages, where those queries describe how to process and combine event streams, as well how to create new event streams. With respect to the composition of events, Siddhi and Odysseus can join only up to two streams at a time, while Esper is able to join two or more streams. For all three engines, it is possible to create queries able to use temporal conditions and to match complex patterns. Esper provides a pattern language that can be integrated into EPL statements, Odysseus uses the SASE+ language to define patterns, and Siddhi is equipped with SiddhiQL, which directly allows the formulation of patterns into its queries.

SiddhiQL limits pattern processing and sequence processing to at most two different streams at a time when logical operators (e.g., and, or) are used. It does not support patterns involving the absence of events, while both Esper and Odysseus can detect such patterns. SASE+ focuses on temporal event patterns and can involve events from different streams but only for patterns based on a sequence of events. However, correlation of events and complex patterns can still be expressed based on the values of event attributes. Esper can handle both pattern and sequence processing involving multiple streams and additionally allows the utilization of user-defined functions on the definition of patterns.

With respect to the accumulation of events, all engines support sliding and tumbling windows, both based on time and length (i.e., number of events). Esper distinguishes further between data windows (time, length, etc.) and named windows, which are globally visible data windows that share event sets between queries efficiently.

Furthermore, Suhothayan et al. [19] compared the performance of Siddhi and Esper by running different types of queries (simple filtering, filtering with time windows and pattern matching). Siddhi had better performance results but both engines showed high throughput and stable behaviors.

Esper is an established CEP engine and has been used as the core of many other CEP engines. It is a lightweight engine that can be easily integrated into Java applications. On the other hand, Odysseus, which is a powerful data management engine and built for extensions, requires some more effort to be integrated due to its complex nature. It provides various plug-ins for additional features, different event languages can be used for different purposes. It can also be extended with new languages, operators, data source connectors, and sink connectors. Moreover, Esper provides an event language capable to cope with the current powerfulness of Situation Templates and to the expected extensions to be executed by this work. Esper can join events from two or more streams, supports pattern and sequence processing involving multiple streams, can detect the absence of events, and allows patterns to employ user-defined functions, while Siddhi and Odysseus presented some limitations in some of these aspects. Ultimately, Esper provides comprehensive documentation that facilitates its employment while formulating sophisticated event patterns for many different situations of interest. Because of the above mentioned reasons, the Esper engine was chosen for the recognition of situations in this master thesis.

4 Situation Recognition based on CEP

This chapter explains all steps necessary to realize the Situation Recognition System using the Esper engine, which was described in *Section 3.1 – Esper*. These steps can be divided into four categories (cf. *Section 1.1 – Problem Definition and Objectives*): (i) the adaptation of the Situation Template schema, in order to increase its powerfulness, (ii) the sensor data provisioning so that data can be actively pushed to the Situation Recognition System, (iii) the transformation of Situation Templates to Event Process Language (EPL) statements to be executed by the Esper engine, and (iv) the actual execution of the situation recognition.

4.1 Situation Template Adaptation

The Situation Template schema proposed in [3] was extended in this master thesis to support the features described in the following. These features are: (i) time-based conditions, (ii) comparison of data from different sensors, and (iii) aggregation of data from different sensors. Listings 4.1 and 4.2 depict the extended schema definition for a condition node, i.e., the elements that a condition node contains and its attributes. The modifications in the schema are marked with the color blue.

Time-based conditions. A time-based condition in the context of this work means a condition that is valid for a specific period of time, such as "temperature greater than 90 degrees for 10 seconds". This enhancement enables the modeling of situations that depend on time to be actually recognized. For example, if an object should not stay more than one minute in some production processing step, a situation can be modeled to recognize when a distance sensor detects a constant distance to the object for more than one minute. Furthermore, *time-based conditions* avoid erroneous sensor readings to be misinterpreted as a situation. For example, considering that a machine should be turned off when its temperature is greater than 90 degrees. If the temperature sensor reads a single value greater than 90 degrees but the following sensor values are lower than that, this will be recognized as a situation and the machine will be

Listing 4.1 XSD extensions at condition node elements

```
1  <xs:complexType name="tConditionNode">
2    <xs:sequence>
3      <xs:element name="measureName" type="xs:string" />
4      <xs:element name="opType" maxOccurs="1" minOccurs="1">
5        <xs:simpleType>
6          <xs:restriction base="xs:string">
7            <xs:pattern value="lowerThan|greaterThan|equals|notEquals|between" />
8          </xs:restriction>
9        </xs:simpleType>
10     </xs:element>
11     <xs:element name="condValue">
12       <xs:complexType>
13         <xs:sequence>
14           <xs:element name="value" type="xs:string" maxOccurs="2" />
15         </xs:sequence>
16       </xs:complexType>
17     </xs:element>
18     <!-- element for comparing variable values -->
19     <xs:element name="condVariable" minOccurs="0">
20       <xs:complexType>
21         <xs:sequence>
22           <xs:element name="variable" type="tVariableInput" minOccurs="2"
23             maxOccurs="2" />
24         </xs:sequence>
25       </xs:complexType>
26     </xs:element>
27     <!-- Time interval in milliseconds -->
28     <xs:element name="timeInterval" type="xs:integer" minOccurs="0" maxOccurs="1" />
29     <xs:element name="parent" type="tParent" minOccurs="0" maxOccurs="unbounded" />
30   </xs:sequence>
31   ...
32 </xs:complexType>
```

turned off. This can be avoided by checking if the temperature stays greater than 90 degrees for a specific period of time. To realize *time-based conditions*, the condition node was extended with the element *timeInterval*, where an integer value representing a period of time in milliseconds can be defined (line 28 of Listing 4.1). An example of a time-based condition modeled in XML is depicted in Listing 4.3. This condition checks if the temperature of a machine is greater than 90 degrees for 10 seconds.

Comparison of data from different sensors. By comparing sensor values at the situation recognition level, applications do not have to take care of such a processing

Listing 4.2 XSD extensions at condition node attributes

```
<xs:complexType name="tConditionNode">
  ...
  <xs:attribute name="id" type="xs:ID" use="required" />
  <xs:attribute name="name" type="xs:string" />
  <!-- Condition node type: condValue (comparison to predefined value) or
    condVariable (comparison of variables) -->
  <xs:attribute name="type">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:pattern value="condValue|condVariable" />
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <!-- can be used together only with condValue -->
  <xs:attribute name="aggregation">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:pattern value="avg|min|max" />
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
</xs:complexType>
```

Listing 4.3 Situation Template extract of a time-based condition

```
<conditionNode id="A2" name="Overheated machine">
        <measureName>temperature</measureName>
        <opType>greaterThan</opType>
        <condValue>
                <value>90</value>
        </condValue>
        <timeInterval>10000</timeInterval>
        <parent parentID="A4"/>
</conditionNode>
```

step themselves and they can thereby focus on the reaction to be executed. For example, consider an application that should keep the temperature of a room uniform and the room is equipped with two temperature sensors and two heating systems. To keep the temperature uniform, the values of the two temperature sensors are periodically compared in order to find out the room area with lower temperature. Then, the better suitable heating system, i.e., the one closer to the room area, is activated. With the enhancement realized in this master thesis, a situation can be modeled to recognize when the room temperature is not uniform anymore, i.e., the temperature sensor values are different, and also which room area has the lower temperature. In this case, the application is notified when the situation occurs and it just needs to activate the appropriate heating system.

As mentioned in *Section 2.3 – Situation Templates*, context nodes represent the monitored sensors and are connected to condition nodes. Originally, sensor data was always compared with values defined in the element "condValue" (cf. line 11 of Listing 4.1). To enable the comparison of two sensor values, the situation node was extended with the attribute *type* (cf. line 7 of Listing 4.2). This attribute can assume the values "condValue" or "condVariable". The type "condVariable" defines that the data from two different sensors will be compared with each other. The context nodes representing the sensors of the comparison are explicitly referenced at the *condVariable* element of the condition node (see line 22 of Listing 4.1). The provided comparison operators for this type of condition node are: *lower than*, *greater than*, *equal to* and *not equal to*.

Listing 4.4 shows a condition node example in which the values of different sensors are compared. Considering the scenario depicted in *Figure 1.2 – Distance sensors placed at a conveyor belt to detect if objects are wrongly positioned*, the two distance sensors placed at the conveyor belt can be represented by the context nodes *A0* and *A1*. If the measured distance value from sensor *A0* is lower than the value from sensor *A1* for 10 seconds, it means that the object is positioned more to the left of the conveyor belt.

Listing 4.4 Situation Template extract of a condition comparing sensor data of different sensors

```
<conditionNode id="A3" name="Package not centered" type="condVariable">
        <measureName>distance</measureName>
        <opType>lowerThan</opType>
        <condVariable>
                <variable contextNodeID="A0"/>
                <variable contextNodeID="A1"/>
        </condVariable>
        <timeInterval>10000</timeInterval>
        <parent parentID="A4"/>
</conditionNode>
```

Aggregation of data from different sensors on the context level. For some scenarios, it might be more reasonable to process a value representing the readings of a group of sensors than to process the readings of each sensor. For example, if there is a machine equipped with many temperature sensors, and an application needs to be notified when at least one sensor reads a temperature greater than 90 degrees. Instead of checking if the measured value of each sensor is greater than 90 degrees, this can be simplified by first calculating the maximum temperature value, and then checking if this resulting value is greater than 90 degrees.

Aggregating sensor values already on the context level, reduces the effort and time that would be needed to model and process condition nodes for each monitored sensor. Therefore, the condition node was enhanced with the attribute *aggregation* (see line 15 of Listing 4.2). This attribute enables data from different sensors to be aggregated using various functions. The resulting value can be compared with predefined values at condition nodes. If a condition node has this attribute, all sensor data coming from the context nodes connected to this condition node will be aggregated prior to the comparison. The provided aggregation functions are: *average*, *minimum value* and *maximum value*.

Considering the scenario depicted in *Figure 1.2*, two distance sensors are placed on a conveyor belt and can detect the position of objects on the conveyor belt together. If the minimum distance value of the two sensors is lower than 10 cm, it means that the object is wrongly positioned either left or right on the conveyor belt, what represents a collision risk. Listing 4.5 shows an example of a condition node in which it checks if the resulting minimum value of two measured distances is lower than 10 cm.

Listing 4.5 Situation Template extract of a condition that aggregates sensor values on the context level

```
<conditionNode id="A2" name="Collision risk" type="condValue" aggregation="min">
        <measureName>distance</measureName>
        <opType>lowerThan</opType>
        <condValue>
                <value>10</value>
        </condValue>
        <parent parentID="A4"/>
</conditionNode>
```

4.2 Sensor Data Provisioning

This section explains how the collected data by sensors of a thing can be transferred over the Internet to the Situation Recognition System developed in this master thesis. The chosen approach uses the publish/subscribe (pub/sub) communication model [23]. This model consists of three component types: subscriber, publisher and broker. Subscribers are the components interested in consuming certain information, they usually have to contact the broker explicitly to register their interest. Publishers produce certain information by publishing them. The broker ensures that the data published is received by the subscribers [24]. The interested reader is referred to [23] for further information regarding pub/sub systems.

This work uses pub/sub clients and a broker that are based on the MQTT (Message Queue Telemetry Transport)[1] protocol. MQTT is a lightweight, topic-based pub/sub protocol, which was designed to minimize bandwidth and assure the delivery of messages at different Quality of Service (QoS) [25] levels. It is ideal to be used in the IoT context where the connected things have low bandwidth and limited battery power [26]. MQTT is a connection-oriented protocol, i.e., it requires the clients to set up connections with the broker before they can publish or subscribe to topics [24].

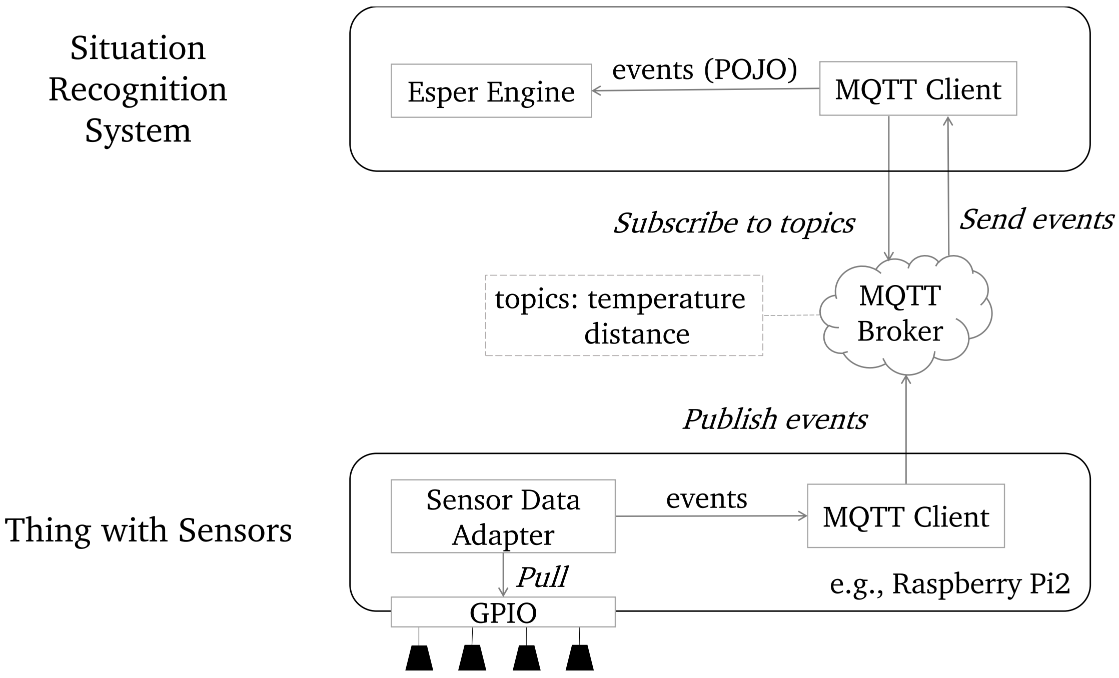


Figure 4.1: Provisioning of sensor data to the Situation Recognition System using MQTT

Figure 4.1 illustrates the setup used to collect sensor data and to send this data to the Situation Recognition System. A Raspberry Pi[2] was used to read data from different sensors. A measured sensor value, its timestamp and the sensor identifier build an event, which is formatted as a JSON string and published in the topic corresponding to the type of the sensor (e.g., temperature, distance). The Raspberry Pi publishes

[1]http://www.mqtt.org/
[2]https://www.raspberrypi.org/products/raspberry-pi-2-model-b/

events using the open-source MQTT Client Paho[3]. The Situation Recognition System is also equipped with a Paho MQTT Client, which is used to subscribe to all the topics and to automatically receive the events of the subscribed topics. The received events, which are JSON formatted strings, are converted to Plain Old Java Objects (POJOs) and are forwarded to the Esper engine. The IBM Bluemix IoT Service[4] is used as the MQTT Broker for this work, but open-source MQTT Brokers (e.g., Mosquitto[5], Moquette[6], etc.) could be used as well. In order to use the IBM Bluemix IoT Service, some configuration steps need to be executed. Please refer to the IBM developerWorks homepage[7] for more information regarding setting up the Bluemix IoT Service and registering things.

4.2.1 Collecting Sensor Data

The Raspberry Pi is a very tiny general-purpose computer. It can be used to build systems using sensors, actuators and microcontrollers by using its General Purpose Input/Output (GPIO) port in order to communicate with external hardware [27]. Through the GPIO port, the Raspberry Pi receives sensor data, controls actuators and communicates to other computing devices using different protocols, including Serial Peripheral Interface (SPI) [27].

To simulate the scenarios presented in *Section 1.2*, the Raspberry Pi collects data from ultrasonic distance sensors and a temperature sensor. Listing 4.6 shows the Python code for setting up the GPIO port for the communication with an ultrasonic distance sensor and for reading data from it. Such a sensor calculates the time taken to reflect ultrasound waves between the sensor and a solid object. In line 14, a signal is sent to the sensor through the GPIO port, which triggers the sensor to emit an ultrasonic pulse. The sensor detects the reflected waves and measures the time between the trigger and the returned pulse. Through the GPIO port, the sensor then sends a *high* signal to the Raspberry Pi (see line 19). This signal has the duration of the time interval measured by the sensor. The Python code must then calculate the distance based on the duration of this *high* signal (see line 21).

Listing 4.7 contains the Python code for reading values from the temperature sensor. To read and convert the analog output of the temperature sensor, the linker base

[3]http://www.eclipse.org/paho/
[4]http://www.ibm.com/Bluemix
[5]http://mosquitto.org/
[6]https://github.com/andsel/moquette
[7]https://www.ibm.com/developerworks/cloud/library/cl-mqtt-bluemix-iot-node-red-app/

for Raspberry Pi[8], which has the analog-to-digital converter (ADC) chip MCP3004 on-board, was used. The ADC chip communicates to the Raspberry Pi using a SPI interface.

Listing 4.6 Python code to read the distance from an ultrasonic sensor

```python
import RPi.GPIO as GPIO
import time
class ultraDist(object):
        def __init__(self, ptrig, pecho):
                self.ptrig = ptrig
                self.pecho = pecho
                GPIO.setmode(GPIO.BCM)
                GPIO.setup(pecho, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
                GPIO.setup(ptrig, GPIO.OUT)
                GPIO.output(ptrig, 0)
        def getValue(self): # in cm
                GPIO.output(self.ptrig, 0)
                time.sleep(0.1)
                GPIO.output(self.ptrig, 1)
                time.sleep(0.00001)
                GPIO.output(self.ptrig, 0)
                while(0 == GPIO.input(self.pecho)):
                        start = time.time()
                while(1 == GPIO.input(self.pecho)):
                        None
                delay = (time.time() - start) * 1000 * 1000
                time.sleep(0.1)
                return (delay / 58.0)
```

4.2.2 Sending Events to the Situation Recognition System

Once the Bluemix IoT Service is running and the Raspberry Pi is registered on it, the Python code from Listing 4.8 can be used to set up the connection with the IBM IoT MQTT Broker. Line 8 creates a MQTT client instance, which is used to connect to the MQTT Broker in line 12. The MQTT broker is accessible at the host "<mq_org>.messaging.internetofthings.ibmcloud.com", where *mq_org* is created automatically when the IoT service is configured.

After connecting to the MQTT Broker, the method *sendMessage()* in line 18 of Listing 4.9 is used to to publish the collected sensor data to a specific topic. Line 23 builds a

[8]http://www.linksprite.com/

Listing 4.7 Python code to read data from a temperature sensor

```python
import spidev
class analogInputReader(object):
        def __init__(self):
                self.spi = spidev.SpiDev()
                self.spi.open(0,0)
        def readadc (self, adPin): # read SPI data from MCP3004 chip, 4 adc’s (0 thru 3)
                if ((adPin > 3) or (adPin < 0)):
                        return -1
                r = self.spi.xfer2([1,8+adPin <<4,0])
                adcout = ((r[1] &3) <<8)+r[2]
                return adcout
        def getLevel (self, adPin):
                value = self.readadc(adPin)
                volts = (value*3.3)/1024
                return (volts, value)
        def getTemperature (self):
                v0 = self.getLevel(0)
                temp = (((v0[0] * 1000) - 500)/10) # celsius
                return temp
```

Listing 4.8 Python code to set up a connection to the IBM IoT MQTT broker

```python
import paho.mqtt.client as mqtt
...
# data for ibm internet of things service
mq_org = "##"
mq_clientId = "d:" + mq_org + ":" + mq_type + ":" + mq_id

# create MQTT client, set user name and password, set mqtt client callbacks...
client = mqtt.Client(client_id=mq_clientId, clean_session=True, userdata=None,
    protocol=mqtt.MQTTv311)
...
# connects to IBM IoT MQTT Broker
mq_host = mq_org + ".messaging.internetofthings.ibmcloud.com"
client.connect(mq_host, 1883, 60)
```

JSON object containing the *sensor identifier*, the *timestamp* of the current measurement, and the read *distance* from the sensor. This object is published as a string in the topic *distance*. The MQTT Broker will then ensure that the Situation Recognition System, which acts as a subscriber, receives each published event.

Listing 4.9 Python code to publish data on a topic

```python
# init GPIO ports for the distance sensors
distA = ultrasonicDistance.ultraDist(17, 18) # ptrig, pecho
topic_distance = "iot-2/evt/%s/fmt/json" % ("distance")

# publishes message to MQTT broker
def sendMessage(topic, msg):
        client.publish(topic=topic, payload=msg, qos=0, retain=False)

# send message, topic: distance
t = datetime.utcnow().strftime('%Y-%m-%d %H:%M:%S.%f')[:-3]
msg_sensor_0 = { "sensorID": "A0","timestamp": t, "distance": "%.1f" %
    (distA.getValue())}
sendMessage (topic_distance, json.dumps(msg_sensor_0))
# send message, topic: temperature ...
```

4.2.3 Receiving Events at the Situation Recognition System

The Situation Recognition System uses the Paho Java Client to connect to the MQTT Broker and to subscribe to topics. How the Situation Recognition System connects to the IBM IoT MQTT Broker is shown in Listing 4.10. Line 18 shows how to subscribe to the interesting topics. The Situation Recognition System needs to implement the interface *MqttCallback*, in order to get notified when an event for the subscribed topics arrives at the MQTT Client. The callback function *messageArrived()* is then executed when an event arrives. The implementation of this callback function is depicted in Listing 4.11. The received events, which are JSON formatted strings, are mapped to POJOs and are then forwarded to the Esper engine for the recognition of situations. *Section 4.4 – Situation Recognition* explains how the situation recognition is executed using the Esper engine.

4.3 Situation Template Transformation

In order to use the Esper engine to execute the situation recognition, Situation Templates need to be mapped to continuous queries, called EPL statements in the Esper context. This mapping has been implemented in Java and uses the Java Architecture for XML Binding (JAXB) to parse the Situation Template to Java objects. As mentioned in *Section 3.1*, Esper allows event pattern matching, where a Situation Template defines a set of conditions that combined can be seen as a complex event pattern in the context of Esper. Therefore, the conditions of a Situation Template

Listing 4.10 Java code to connect to the IBM IoT MQTT broker and subscribe to topics

```java
public class SitRSMqttClient implements MqttCallback {
    private static final String MQ_EVENT_DISTANCE    = "/evt/distance";
    private static final String MQ_EVENT_TEMPERATURE = "/evt/temperature";
    MqttClient client;
    ...
    // connects to MQTT Broker and subscribe to topics
    private void initialize() {
        // data for ibm internet of things service ...
        try {
            // creates mqtt client, sets username, password, mqtt client callbacks
            client = new MqttClient(mq_url, mq_clientId);
            MqttConnectOptions options = new MqttConnectOptions();
            ...
            // connects to Mqtt broker
            client.connect(options);

            // subscribes to topics
            client.subscribe("iot-2/type/+/id/+" + MQ_EVENT_DISTANCE + "/fmt/json");
            client.subscribe("iot-2/type/+/id/+" + MQ_EVENT_TEMPERATURE + "/fmt/json");
        }
        catch (MqttException e) {...}
    }
}
```

need to be formulated as an event pattern that will match when the modeled situation occurs. Patterns alone just detect if a situation occurred. To have access to the pattern match results, it is necessary to combine the pattern matching with event stream analysis. In this way, it is possible to know exactly which events caused the situation. Each situation to be recognized can then be represented as the EPL statement `select * from pattern [ "corresponding_situation_template_pattern" ]`. To exemplify how a Situation Template can be transformed to an event pattern, the scenarios "Monitoring machine status" and "Monitoring objects on a conveyor belt" from *Section 1.2* are used.

Figure 4.2 depicts the scenario "Monitoring machine status" modeled in XML and in a corresponding tree representation. The situation "Machine is blocked" occurs if at least one of the following conditions evaluate to true: (i) the temperature of the machine is greater than 90 degrees, or (ii) the level of the material container of the machine is lower than 10 cm, i.e., the material runs low.

To build a pattern expression for the first condition, we extract the monitored sensor identifier (*A0*) from the context node *A0*, and the event type generated by the sensor

Listing 4.11 Java code to receive events from the MQTT Broker

```java
SituationHandler sitHandler;
ObjectMapper     mapper = new ObjectMapper();
...
@Override
public void messageArrived(String topic, MqttMessage message) throws Exception {
    int beginIndex = topic.lastIndexOf("/evt/"); // parse topic
    int lastIndex = topic.lastIndexOf("/fmt/json");

    if (beginIndex != -1 && lastIndex != -1) {
        String eventName = topic.substring(beginIndex, lastIndex);

        if (MQ_EVENT_DISTANCE.equals(eventName)) {
            DistanceMeterEvent eventFromJSON = mapper.readValue(new String(
                    message.getPayload()), DistanceMeterEvent.class);
            sitHandler.sendEvent(eventFromJSON);

        } else if (MQ_EVENT_TEMPERATURE.equals(eventName)) {...}
    }
}
```

(*TemperatureEvent*). From the condition node, we extract the measure variable name (*temperature*), the comparison operator (*greater than*), and the predefined value for the comparison (*90*). With all this information, we can then build a pattern that matches on every *TemperatureEvent* event that was generated by the sensor *A0* and has a measured temperature greater than 90 degrees. This pattern is depicted in Listing 4.12.

Listing 4.12 Pattern expression for a simple condition

```
every A0_stream=TemperatureEvent(sensorID='A0', temperature > 90)
```

We then create a pattern expression for the second condition as well, and combine both expressions with the logical operator *or*. Finally, this results in an EPL statement that recognizes the situation “Machine is blocked”. This statement is depicted in Listing 4.13. *A0_stream* in line 2 and *A1_stream* in line 4 are called tags and they are important because only tagged events are available to listeners when the pattern matches.

Section 4.1 – Situation Template Adaptation explained, which changes were made in the Situation Template schema in order to increase its powerfulness with (i) time-based conditions, (ii) comparison of data from different sensors, and (iii) aggregation of data from different sensors on the context level. The following paragraphs describe

```
<SituationTemplate id="scenario_3" name="Machine Status">
 <Situation id="A" name="Machine Blocked">
  <situationNode id="A5" name="Machine Blocked"/>
  <operationNode id="A4" name="Combine">
    <parent parentID="A5"/><type>or</type>
  </operationNode>
  <conditionNode id="A3" name="Low level" type="condValue">
    <measureName>distance</measureName>
    <opType>lowerThan</opType>
    <condValue> <value>10</value> </condValue>
    <parent parentID="A4"/></conditionNode>
  <conditionNode id="A2" name="High temp" type="condValue">
    <measureName>temperature</measureName>
    <opType>greaterThan</opType>
    <condValue><value>90</value></condValue>
    <parent parentID="A4"/></conditionNode>
  <contextNode id="A1" name="Level Sensor" type="DistanceEvent">
    <parent parentID="A3"/></contextNode>
  <contextNode id="A0" name= "Temp Sensor" type="TempEvent">
    <parent parentID="A2"/></contextNode>
 </Situation></SituationTemplate>
```

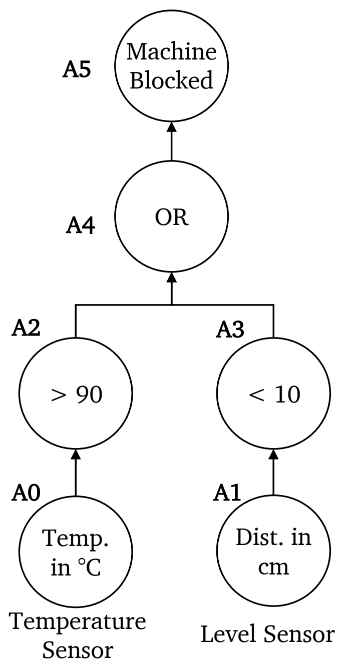


Figure 4.2: Situation Template for monitoring a machine status

Listing 4.13 EPL statement to recognize the situation from "Monitoring machine status"

```
select * from pattern [
 ( every A0_stream=TemperatureEvent(sensorID='A0', temperature > 90))
 or
 ( every A1_stream=DistanceMeterEvent(sensorID='A1', distance < 10))
]
```

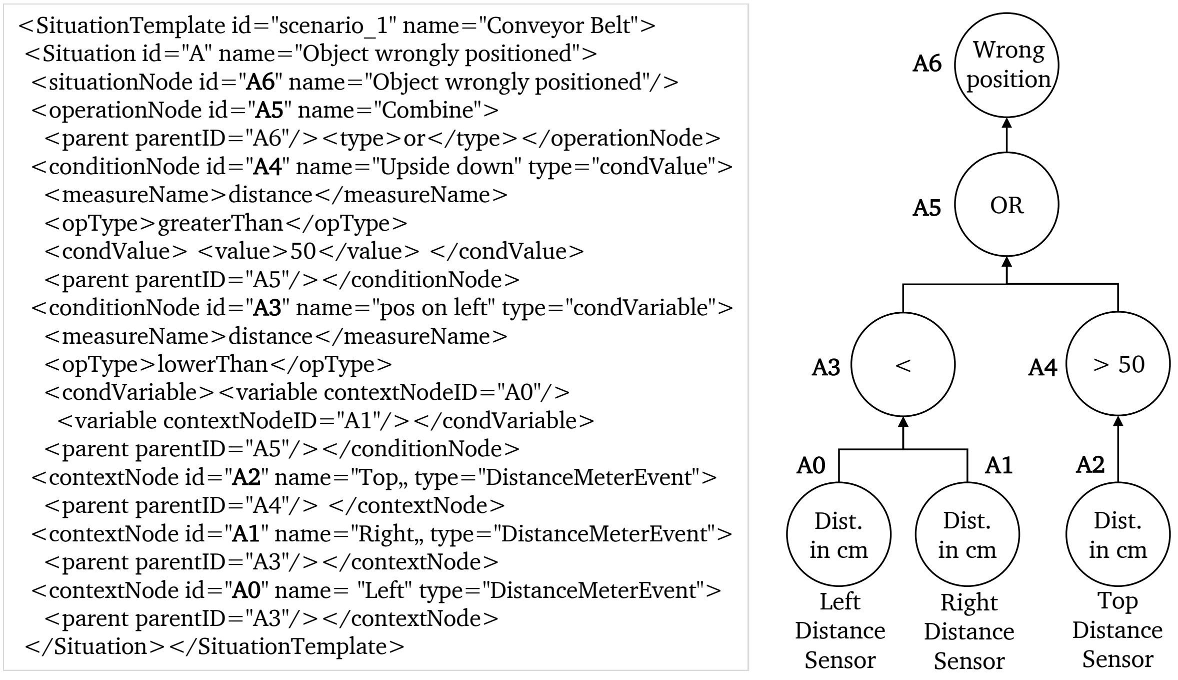
how conditions involving such features can be mapped to corresponding pattern expressions. Additionally, it shows how a pattern expression looks like for recognizing when a situation stopped occurring.

Time-based conditions. The definition of a time-based condition was given in Section 4.1. To explain how to formulate a pattern expression for a time-based condition, the following example is used: "the temperature of the machine is greater than 90 degrees for 5000 milliseconds". The pattern for such a condition first looks for an event, in which the temperature is greater than 90 degrees. Then, it checks for the next 5000 milliseconds if the temperature does not become lower than or equal to 90 degrees, i.e., the temperature stays greater than 90 degrees. This pattern is depicted in

Listing 4.14 where it checks the absence of an event where "temperature lower than or equal to 90 degrees". To detect the absence of an event, it is recommended to use the *timer:interval* observer together with *and not* operators. The *followed by (->)* operator in line 2 means that first the left hand expression must occur and only then the right hand expression is evaluated.

Listing 4.14 Pattern expression for a time-based condition

```
1 every A0_stream=TemperatureEvent(sensorID='A0', temperature > 90)
2 ->
3 (timer:interval (5sec) and not TemperatureEvent(sensorID='A0', temperature <= 90))
```



```
<SituationTemplate id="scenario_1" name="Conveyor Belt">
 <Situation id="A" name="Object wrongly positioned">
  <situationNode id="A6" name="Object wrongly positioned"/>
  <operationNode id="A5" name="Combine">
    <parent parentID="A6"/><type>or</type></operationNode>
  <conditionNode id="A4" name="Upside down" type="condValue">
    <measureName>distance</measureName>
    <opType>greaterThan</opType>
    <condValue> <value>50</value> </condValue>
    <parent parentID="A5"/></conditionNode>
  <conditionNode id="A3" name="pos on left" type="condVariable">
    <measureName>distance</measureName>
    <opType>lowerThan</opType>
    <condVariable><variable contextNodeID="A0"/>
      <variable contextNodeID="A1"/></condVariable>
    <parent parentID="A5"/></conditionNode>
  <contextNode id="A2" name="Top„ type="DistanceMeterEvent">
    <parent parentID="A4"/> </contextNode>
  <contextNode id="A1" name="Right„ type="DistanceMeterEvent">
    <parent parentID="A3"/></contextNode>
  <contextNode id="A0" name= "Left" type="DistanceMeterEvent">
    <parent parentID="A3"/></contextNode>
 </Situation></SituationTemplate>
```

Figure 4.3: Situation Template for monitoring objects on a conveyor belt

Comparison of data from different sensors. This feature was previously explained in Section 4.1. To exemplify how to create an event pattern expression for it, the scenario "Monitoring objects on a conveyor belt" from *Section 1.2 – Motivating Scenarios* is used. Figure 4.3 depicts this scenario modeled in XML and in a corresponding tree representation. The situation "object is wrongly positioned" occurs if at least one of the following conditions is true: (i) an object on the conveyor belt is positioned left, (ii) an object on the conveyor belt is positioned right, or (iii) an object is upside down.

To determine if the first condition is true for the scenario depicted in *Figure 1.2*, the measured distances from the *left sensor* and the *right sensor* are compared with each other. The pattern for such a condition looks for an event from the *left sensor*, which is followed by an event from the *right sensor*. Then, the measured distances from both events is compared. Finally, the pattern matches if the measured distance of the *left sensor* is lower than the distance of the *right sensor*. This pattern is depicted in Listing 4.15, where the *left sensor* corresponds to the sensor with identifier *A0* and the *right sensor* to the sensor with identifier *A1*. The *not* operator in line 4 makes sure that always the last value of the *left sensor* is used when an event from the *right sensor* arrives. This pattern considers that both sensors generate events continuously. In this way, ordering the events with the *followed by* operator does not affect the evaluation of the condition "distance from the *left sensor* is lower than the distance from the *right sensor*". Listing 4.16 shows an alternative pattern expression, where the order of the events is not taken into consideration.

Listing 4.15 Pattern expression for comparing data from different sensors

```
1 every A0_stream=DistanceMeterEvent(sensorID = 'A0')
2 ->
3 (A1_stream=DistanceMeterEvent(sensorID = 'A1', A0_stream.distance < distance)
4  and not DistanceMeterEvent(sensorID = 'A0'))
```

Listing 4.16 Alternative pattern expression for comparing data from different sensors

```
every (A0_stream=DistanceMeterEvent(sensorID = 'A0')
 and
 A1_stream=DistanceMeterEvent(sensorID = 'A1'))
while (A0_stream.distance < A1_stream.distance)
```

Aggregation of data from different sensors. This feature was previously explained in Section 4.1. To exemplify how to create a pattern expression for such a feature, the scenario "Monitoring objects on a conveyor belt" is used (cf. *Figure 1.2*). In this scenario, two distance sensors are placed at a conveyor belt and together they can detect the position of objects on the conveyor belt. For example, they can be used to detect if the position of an object on the conveyor belt represents a lateral collision risk. For this situation, we can use a pattern expression that checks if the minimum distance value of the two sensors is lower than 10 cm, which means the object is wrongly positioned either left or right on the conveyor belt, and therefore represents a collision risk. This pattern is depicted in Listing 4.17, which shows a pattern expression for calculating the minimum value of two distance values and comparing the result with a predefined value. Esper allows to invoke user-defined static methods that return a Boolean value. In line 4 of Listing 4.17, the user-defined method *Aggregation.check_min* computes the minimum value of all measured distances and returns true if this value is lower than 10 cm.

Listing 4.17 Pattern expression for aggregating data from different sensors

```
1 every A0_stream=DistanceMeterEvent(sensorID = 'A0')
2  and
3  A1_stream=DistanceMeterEvent(sensorID = 'A1')
4 while (Aggregation.check_min('<', 1, {10, A0_stream.distance, A1_stream.distance}))
```

Recognizing when a situation stops occurring. Besides recognizing when a situation occurred, it is also necessary to recognize if this situation stopped occurring. To exemplify how to create a pattern expression to recognize when a situation stops occurring, the scenario “Monitoring level of containers” (cf. *Figure 1.3*) is used. In this scenario, the distance sensor positioned above the container is used to measure the current material level in the container. One important situation to detect is that the material in the container runs low. When this situation is handled, i.e., the material container is refilled, the situation stops occurring.

To recognize that the material runs low, we can use a pattern expression that matches when it receives an event where the level of the material is lower than 10 cm. To recognize that the container was refilled, this pattern expression is extended to recognize that such event was followed by an event where the measured level became greater than or equal to 10 cm. This pattern is depicted in Listing 4.18. It recognizes that the sensor *A1* measured a distance lower than 10 cm, and after that it measured a distance greater than or equal to 10 cm.

Listing 4.18 Pattern expression for recognizing if a situation stopped occurring

```
every A1_stream=DistanceMeterEvent(sensorID='A1', distance < 10)
->
A1_stream_sitStopped=DistanceMeterEvent(sensorID='A1', distance >= 10)
```

This section explained how Situation Templates are mapped onto EPL statements, which are used by the Esper engine to recognize situations. Moreover, this work increases the powerfulness of Situation Templates and implements corresponding mappings based on the enhanced Situation Template schema. This permits the modeling of sophisticated situations involving: (i) time-based conditions, (ii) comparison of data from different sensors, or (iii) aggregation of data from different sensors on the context level. Table 4.1 shows examples of conditions using these features and how they can be combined with each other.

	simple condition	time-based condition
Comparison of sensor data to predefined value	distance lower than 10 cm	distance lower than 10 cm for 10 seconds
Comparison of aggregated sensor data to predefined value	the minimum distance of all sensors lower than 10 cm	the minimum distance of all sensors lower than 10 cm for 10 seconds
Comparison of data from different sensors	distance of sensor 1 lower than distance of sensor 2	distance of sensor 1 lower than distance of sensor 2 for 10 seconds

Table 4.1: Examples of supported conditions by the Situation Template

4.4 Situation Recognition

The architecture of the prototypical implementation of the Situation Recognition System is depicted in Figure 4.4. The Situation Recognition System provides an API (*Situation Registration API*) to register situations for starting its recognition, where the thing identifier and the Situation Template identifier are provided. A registration identifier is returned, which can be used to deregister the situation. The registration process is taken care of at the *Query Handler*. If an EPL statement for the given situation is not already active, the *Query Handler* triggers the mapping of the Situation Template to EPL statements and adds it to the Esper engine. The implementation of the mapping was previously explained in *Section 4.3 – Situation Template Transformation*. How the Situation Recognition System receives events for processing was explained in *Section 4.2 – Sensor Data Provisioning*.

In order to receive the results from the Esper engine when an EPL statement matches, the *Query Handler* adds subscribers to the active queries. Listing 4.19 shows a code snippet from the *Query Handler* that triggers the mapping of a Situation Template to an EPL statement, which is added to the Esper engine to execute the situation recognition. A subscriber object is then added to the EPL statement in line 10.

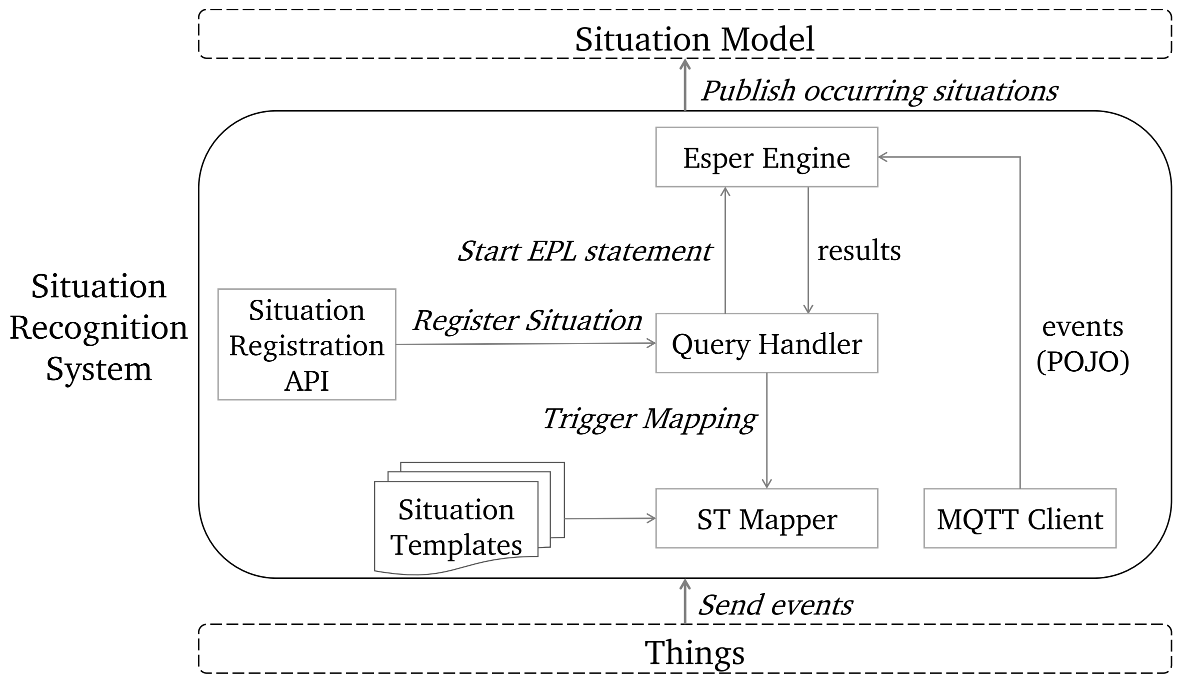


Figure 4.4: Situation Recognition System Architecture

Listing 4.19 Java code snippet for adding an EPL statement to the Esper engine and adding a subscriber to the statement

```java
SituationMapper mapper = new SituationMapper(file);
String situationTemplateID = mapper.getSituationTemplateID();
String mappedStatement_sitOccurred = mapper.getQuerySituationOccurred();

// creates and starts running the query
EPStatement epStatement =
    epService.getEPAdministrator().createEPL(mappedStatement_sitOccurred);

// sets subscriber for the query
EPLQuerySubscriber subscriber = new EPLQuerySubscriber(this, situationTemplateID);
epStatement.setSubscriber(subscriber);
```

In this prototype, the occurring situations are published to the *situation-aware workflow management system* called *SitOPT* [8] by using its situation model. *SitOPT*'s situation model management manages all the occurring situations and notifies the registered systems about the situation changes [8]. When a registered system is notified, it can invoke a *Situation Handler* in order to handle the corresponding situation properly. When the *SitOPT* system receives a notification from the situation model management, it decides whether to invoke its *Situation Handler* for adapting the workflow so that the notified situation can be handled [8].

Listing 4.20 shows the subscriber implementation, the class *EPLQuerySubscriber*. The subscriber class needs to provide the method *update()* as shown in line 6, where the results from the Esper engine for the EPL statement will be received. The events that cause the situation to occur are then extracted from the results, formatted using the *situation model* from [8] and published to *SitOPT* (see line 40).

Furthermore, the developed prototype uses the MQTT Client to publish occurring situations to the IBM IoT MQTT Broker as well. This enables the Raspberry Pi to act as a subscriber, where it receives notifications when situations occur. The Raspberry Pi can then invoke a *Situation Handler* that reacts to occurring situations by controlling the actuators connected to it, for example, by starting an audible alarm signal using a buzzer.

Listing 4.20 The subscriber implementation

```
public class EPLQuerySubscriber {
    private String           situationTemplateID;
    private SituationHandler parent;
    ...
    // Notification from Esper are received here in this method.
    public void update(Map<String, Object> row) {
        if (row.size() > 0) {
            String timestamp = new SimpleDateFormat("yyyy-MM-dd h:mm:ss.SSS").
            format(new Date());

            List<SensorValue> values = new ArrayList<SensorValue>();

            for (Object event : row.values()) {
                if (event instanceof EventBean) {
                    EventBean eventBean = (EventBean) event;

                    if (DistanceMeterEvent.class.getSimpleName().
                        equals(eventBean.getEventType().getName())) {

                      DistanceMeterEvent eventObj=(DistanceMeterEvent) eventBean.
                      getUnderlying();
                      SensorValue sensorValue = new SensorValue();
                      sensorValue.setQuality(0);
                      sensorValue.setValue(eventObj.getDistance());
                      sensorValue.setTimestamp(eventObj.getTimestamp());
                      sensorValue.setSensor(eventObj.getSensorID());
                      values.add(sensorValue);

                    } else if (TemperatureEvent.class.getSimpleName().
                                equals(eventBean.getEventType().getName())) {
                      // ...
                    }
                }
            }
            // build matched situation obj
            MatchedSituation matchedSituation = new MatchedSituation();
            matchedSituation.setSituationtemplate(situationTemplateID);
            matchedSituation.setThing(thingId);
            matchedSituation.setTimestamp(timestamp);
            matchedSituation.setSensorvalues(values);

            // notify system SitOPT
            parent.onSituation(matchedSituation);
        }
    }
```

5 Evaluation

This chapter contains the evaluation of the Situation Recognition System developed within this master thesis. It presents the runtime measurements and a load test based on the prototypical implementation. To conduct the measurements, a machine running the Windows 8 operating system with 8 GB of RAM and an Intel(R) Core(TM) i5-4300U CPU @ 1,90GHz 2,49GHz processor was used.

The Situation Template used for the measurements monitors objects on a conveyor belt (cf. Section 1.2), modeled as shown in Figure 4.3. The following was measured: (i) the time taken to transform the Situation Template to an EPL statement, and (ii) the time taken to recognize that the modeled situation occurs. The Situation Template contains altogether 7 nodes to be transformed into an EPL statement, which detects the modeled situation. To measure the time that the Esper engine requires to recognize the modeled situation, this situation was induced by sending events locally to the Esper engine that satisfy one condition of the used Situation Template, i.e., events that satisfy the condition "distance measured by the *top sensor* is greater than 50 cm".

Table 5.1 shows the measurement results for the transformation of a single Situation Template. It also shows the measurement results for the situation recognition, where only one EPL statement is active in the Esper engine. These measurements do not include the time it would have taken to send events to the Esper engine over a network. The measurements show that both the transformation step and the situation recognition are executed in a reasonable time.

Furthermore, a load test was also conducted in order to find out how many Situation Templates can be executed in parallel inside a single runtime environment using the same Situation Template as above. The results are shown in Table 5.2. This load test shows that the runtime slightly increases with the number of situations to be monitored in parallel. It shows that the runtime increases only to 64,55 ms for each situation when the Esper engine executes 100 EPL statements in parallel.

In conclusion, executing the situation recognition using the Esper engine overcomes the limitations revealed by the prototypical implementation of SitRS [3], which uses the Node-RED execution environment (cf. Section Evaluation in [3]). Hirmer et al.

Measurement	**ST Transformation**	**Situation Recognition**
1	142 ms	3 ms
2	168 ms	2 ms
3	147 ms	4 ms
4	136 ms	3 ms
5	135 ms	3 ms
∅	145,6 ms	3 ms

Table 5.1: Runtime measurements of the prototype

# ST	**Recognition Time / Situation ∅**
1	3 ms
5	6,96 ms
10	12,78 ms
20	21,60 ms
50	33,30 ms
100	64,55 ms

Table 5.2: Load test of the prototype

show that the runtime highly increases with the number of monitored situations, what happens due to the overhead produced by Node-RED while processing parallelized flows [3].

6 Related Work

This chapter presents several works that realize situation recognition. Their approaches to the recognition of situations are compared with the approach introduced in this master thesis.

Because context is poorly used in our computing environments, we have a lack of understanding what context is and how it can be used [28]. This thesis uses the definition of context provided in [28, 29]:

> "Context is any information that can be used to characterize the situation of an entity, which can be a person, place or object".

Abowd et al. [29] discuss how to efficiently provide context information to applications and how to make these applications aware and responsive. Their approach to the development of context-aware applications is to collect contextual information through automated means (e.g., using sensors instead of user input) and make it easily accessible to the application's runtime environment [28]. This context information can be used to determine when relevant entities are in a particular state, i.e., when a situation occurs [28]. This master thesis also adopts the approach of using context information collected by sensors to identify situations. This is done by using Situation Templates [3], a model that describes: (i) the relevant context information (i.e., the monitored sensors), and (ii) how this information should be interpreted and processed to recognize the occurrence of a situation. Situation Templates were previously explained in *Section 2.3*.

This master thesis focuses on modeling situations as Situation Templates and using them for the situation recognition, where Complex Event Processing (CEP) is employed to execute the situation recognition. Several situation recognition systems using different approaches have been proposed. Attard et al. [30] present an ontology-based solution for the recognition of personal recurring situations, where context information is generated by the devices a person owns. It employs a context matching technique that periodically compares a person's live context to previously identified situations and also suggests matched situations, which are gradually characterized by the feedback of the user.

Dargie et al. [31] use an ontology to describe servers and a service management system, which manages a video platform focusing on energy efficiency. Description Logic (DL) reasoning is then employed to identify situations of interest. Dargie et al. [31] argue that it cannot be assured that the available non-commercial reasoners are fast enough to recognize complex situations for applications that require fast response times.

Kokar et al. [32] claim that ontology-based situation recognition approaches have the advantage that once facts about the world are stated, other facts can be inferred. However, there is a missing commonality in the concepts used in the analysis and synthesis of situation aware processing. For example the interpretation of situation is different across different works even if they are within the same area [32, 33]. The approach followed in this thesis, i.e., using Situation Templates, is not limited to recognize only particular situation types. Any kind of situation can be modeled based on the available context model [3].

Further situation recognition approaches based on machine learning can be found in [33, 13]. This master thesis considers that situations are well-known and all necessary information for the recognition of a situation can be modeled as Situation Templates. Therefore, the Situation Recognition System developed within this work does not employ any learning techniques (e.g., machine learning) and neither relies on user feedback to adapt the situation recognition.

Several situation recognition systems using CEP were proposed in [34, 35, 36]. Taylor and Leidinger [34] propose the use of ontologies to specify and recognize complex events, whose occurrence can be detected in digital messages streamed from multiple sensor networks. The developed ontology is accessed through an user interface, where the user specifies the events of interest. The specification of an event of interest is then processed in order to generate configuration commands for a CEP system. The CEP system monitors the specified data streams and generates notifications, which can be delivered to clients when the event occurs [34]. In this master thesis, complex events are not directly specified, but rather abstracted as situations. The approach in this work also realizes transformations of the event specifications (i.e., the modeled situations) into commands for a CEP system. The difference is that we use Situation Templates to model situations of interest instead of an ontology. This enables not only the employment of CEP systems but also other technologies for situation recognition. An user interface for the modeling of situations of interest is part of future work (cf. Section 7).

Hasan et al. [35] propose to use Complex Event Processing along with a dynamic enrichment of sensor data in order to realize situation awareness. In this approach, the situations of interest are directly defined in the CEP engine, i.e., the user formulates the situations of interest using CEP query languages [35]. A dynamic enrichment component processes and enriches the sensor data before the CEP engine evaluates them against the situations of interest. This approach and the one in this master thesis differ in following: No complex dynamic enrichment of the sensor data is done in this master thesis. The necessary information about the sensor for the situation recognition (e.g., the sensor identification) is kept at a minimum. This information together with the sensor reading are made available directly to the CEP engine, dispensing any further processing step of the sensor data. Furthermore, instead of defining situations directly as CEP queries, which can be long and complicated depending of the situation, this work defines the situations of interest as Situation Templates. The abstraction provided through Situation Templates enables the employment of CEP engines as well as the use of other technologies for the situation recognition. Besides that, the use of Situation Templates also facilitates the modeling step of situations for the user, so that the user does not have to deal with the complexity of formulating CEP queries. The formulation of CEP queries is taken care of by mappings, which automatically create the necessary CEP queries for a given Situation Template.

Glombiewski et al. [36] present the implementation of an event processing system on top of JDBC[1] that enables standard database systems to provide event processing functionality. Their approach for an event processing abstraction layer, called Java Event Processing Connectivity (JEPC) [37], is currently implemented on top of different event processing systems. JEPC provides an easy-to-use API and a powerful query language, which are independent of the used event processing system [37]. It also provides JEPC bridges to the different event processing systems, where the JEPC API is mapped to the specific API of the underlying event processing system [37]. This master thesis also provides an API, which is used to register and deregister situations to be monitored. Though, this work models situations as Situation Templates. This avoids dependency of any execution engine and therefore also avoids the direct use of a query language. Similar to JEPC, we also realize mappings for the underlying event processing system (i.e., Esper [14]) where currently only one event processing system is supported for the recognition of situations. The utilization of others event processing systems (e.g., Siddhi [15], Odysseus [16], etc.) to execute the recognition of situations is part of future work (cf. Section 7).

Finally, as explained in *Section 1.1 – Problem Definition and Objectives*, this works builds upon the situation recognition service *SitRS* presented in [3], where relevant context information collected by sensors is used to identify situations. Situation Templates contain the sensors being monitored as well as the conditions that have to match

[1]http://www.oracle.com/technetwork/java/javase/jdbc/index.html

for a certain situation. This work mainly (i) increases the powerfulness of Situation Templates, (ii) provides a sensor data push approach, (iii) implements mappings from Situation Templates to CEP queries, and (iv) uses a CEP engine to execute the situation recognition (cf. Section 4).

7 Summary and Outlook

This master thesis presents a Situation Recognition System that employs Complex Event Processing (CEP) to recognize situations of interest based on sensor data. Because this system employs CEP for the situation recognition, it can be used to recognize many situations in parallel and is able to process a large amount of sensor data in a continuous and timely fashion. The used approach specifies situations of interest as *Situation Templates*. A Situation Template contains all necessary information to recognize the modeled situation, where it contains: (i) the monitored sensors, and (ii) the instructions of how sensor data should be processed to recognize the occurrence of the situation, i.e., the conditions that should match for the situation to occur.

Within the scope of this master thesis, the powerfulness of Situation Templates is increased in order to enable the modeling of more sophisticated situations. Besides expressing conditions that compare sensor data with fixed predefined values, Situation Templates are enhanced to (i) express conditions that should hold for a specific time interval, (ii) model conditions that compare the data of two different sensors, and (iii) formulate conditions that first aggregate the data of many sensors before the resulting value is compared with any predefined value. Furthermore, this work presents a sensor push approach so that sensor data is sent to the Situation Recognition System as soon as this data is available. This approach avoids any pull requests from the Situation Recognition System and avoids any caching of sensor data as well. Moreover, this work analyzes three different CEP engines and chooses a CEP engine that can cope with the enhanced powerfulness of Situation Templates. In order to finally execute the situation recognition using CEP, this work develops mappings from Situation Templates onto executable representations to be deployed into the chosen CEP engine. These representations, called CEP queries, evaluate the sensor data sent to the CEP engine and generate notifications when a CEP query matches, i.e., when a situation occurs.

The developed Situation Recognition System provides an API to register situations for starting their recognition and to stop their monitoring as well. Upon the registration of a situation, the corresponding Situation Template is automatically mapped onto a CEP query, which is handed to the CEP engine and executed in order to recognize the occurrence of the situation. In the developed prototype, the recognized situations are published to the *situation-aware workflow management system* SitOPT [8] by using its situation model, which notifies the registered systems about occurring situations and their changes. Furthermore, this work shows a possibility to use the Situation Recognition System to send notifications to actuators, which can then properly react when a situation occurs. The results of the Situation Recognition System show that the employment of CEP for the situation recognition allows the recognition of situations in a reasonable time. Furthermore, it is shown that the system is capable of monitoring many situations in parallel inside a single runtime environment.

As future work, the Situation Recognition System should be provided as a cloud-based service. A step in this direction is already taken by the proposed sensor data push approach, which uses the IBM Bluemix IoT Service[1] to integrate things into the internet and to make sensor data available to the Situation Recognition System automatically.

This work does not take into account the quality of sensor measurements, i.e., it does not support the uncertainty in measurements that sensors generally have. This aspect plays an important role in the quality of the recognized situations. Therefore, a future work is to increase the quality of the recognized situations, by enabling conditions to have a tolerance degree while evaluating the sensor values.

Currently, situations are modeled directly in the XML language. To facilitate the modeling of situations, an user interface should be provided, where the user can create a graphical representation of the situation. The Situation Template in XML can then be extracted from this graphical representation. Such an user interface also facilitates the understanding of modeled situations, since a visual representation is better to comprehend than a textual representation. Furthermore, other CEP engines (e.g., Siddhi [15], Odysseus [16], etc.) should be supported besides the Esper engine.

[1]http://www.ibm.com/Bluemix

Bibliography

[1] Michael Eckert and François Bry. Complex event processing (cep). *Informatik-Spektrum*, 32(2):163–167, 2009. (Cited on pages 6, 16, 17, 18, 23 and 24)

[2] Opher Etzion and Peter Niblett. *Event processing in action*. Manning Publications Co., 2010. (Cited on pages 6, 9, 15, 16, 17, 19 and 24)

[3] Pascal Hirmer, Matthias Wieland, Holger Schwarz, Uwe Breitenbücher, and Frank Leymann. SitRS - A Situation Recognition Service Based on Modeling and Executing Situation Templates . In *Proceedings of the 9th Symposium and Summer School On Service-Oriented Computing (SummerSOC)*, 2015. (Cited on pages 6, 10, 18, 19, 20, 21, 37, 57, 58, 59, 60 and 61)

[4] Commission of The European Communities (2008). Future networks and the internet. early challenges regarding the internet of things. COM (2008) 594. SEC (2008) 2507, 2008. (Cited on page 9)

[5] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010. (Cited on page 9)

[6] Ovidiu Vermesan and Peter Friess. *Internet of things: converging technologies for smart environments and integrated ecosystems*. River Publishers, 2013. (Cited on pages 9, 15 and 17)

[7] The complex event processing blog. `http://www.thecepblog.com/`. (Cited on page 9)

[8] Matthias Wieland, Holger Schwarz, Uwe Breitenbücher, and Frank Leymann. Towards Situation-Aware Adaptive Workflows. In *Proceedings of the 11th Workshop on Context and Activity Modeling and Recognition (COMOREA) IEEE Conference on Pervasive Computing (PerCom)*, 2015. (Cited on pages 10, 54, 55 and 64)

[9] Ovidiu Vermesan, Peter Friess, Patrick Guillemin, Sergio Gusmeroli, Harald Sundmaeker, Alessandro Bassi, Ignacio Soler Jubert, Margaretha Mazura, Mark Harrison, M Eisenhauer, et al. Internet of things strategic research roadmap. *O.*

Vermesan, P. Friess, P. Guillemin, S. Gusmeroli, H. Sundmaeker, A. Bassi, et al., Internet of Things: Global Technological and Societal Trends, 1:9–52, 2011. (Cited on page 15)

[10] Alejandro Buchmann and Boris Koldehofe. Complex event processing. *it-Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik*, 51(5):241–242, 2009. (Cited on pages 16 and 17)

[11] David C Luckham. *Event processing for business: organizing the real-time enterprise*. John Wiley & Sons, 2011. (Cited on page 16)

[12] David Luckham. *The power of events*, volume 204. Addison-Wesley Reading, 2002. (Cited on page 17)

[13] Oliver Zweigle, Kai Häussermann, Uwe-Philipp Käppeler, and Paul Levi. Supervised learning algorithm for automatic adaption of situation templates using uncertain data. In *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*, pages 197–200. ACM, 2009. (Cited on pages 19 and 60)

[14] EsperTech. Event processing with esper and nesper. http://www.espertech.com/esper/, . (Cited on pages 23, 24, 27 and 61)

[15] WSO2. Wso2 complex event processor. http://wso2.com/products/complex-event-processor/, . (Cited on pages 23, 29, 30, 61 and 64)

[16] The Odysseus Team. Odysseus - the event processing system. http://odysseus.informatik.uni-oldenburg.de/. (Cited on pages 23, 30, 34, 61 and 64)

[17] EsperTech. Esper reference. http://www.espertech.com/esper/release-5.3.0/esper-reference/html/index.html, . (Cited on page 24)

[18] WSO2. Siddhi complex event processing engine. https://github.com/wso2/siddhi, . (Cited on page 27)

[19] Sriskandarajah Suhothayan, Kasun Gajasinghe, Isuru Loku Narangoda, Subash Chaturanga, Srinath Perera, and Vishaka Nanayakkara. Siddhi: A second look at complex event processing architectures. In *Proceedings of the 2011 ACM workshop on Gateway computing environments*, pages 43–50. ACM, 2011. (Cited on pages 27, 30 and 35)

[20] WSO2. Siddhiql guide 3.0. https://docs.wso2.com/display/CEP400/SiddhiQL+Guide+3.0, . (Cited on page 30)

[21] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):15, 2012. (Cited on pages 30 and 31)

[22] Sase - language. http://avid.cs.umass.edu/sase/index.php?page=language. (Cited on page 33)

[23] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2): 114–131, 2003. (Cited on pages 41 and 42)

[24] Urs Hunkeler, Hong Linh Truong, and Andy Stanford-Clark. Mqtt-s - a publish/subscribe protocol for wireless sensor networks. In *Communication systems software and middleware and workshops, 2008. comsware 2008. 3rd international conference on*, pages 791–798. IEEE, 2008. (Cited on page 42)

[25] Daniel Menasce et al. Qos issues in web services. *Internet Computing, IEEE*, 6(6): 72–75, 2002. (Cited on page 42)

[26] Mqtt. http://mqtt.org/. (Cited on page 42)

[27] Eben Upton and Gareth Halfacree. *Raspberry Pi user guide*. John Wiley & Sons, 2014. (Cited on page 43)

[28] Anind K Dey. Understanding and using context. *Personal and ubiquitous computing*, 5(1):4–7, 2001. (Cited on page 59)

[29] Gregory D Abowd, Anind K Dey, Peter J Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In *Handheld and ubiquitous computing*, pages 304–307. Springer, 1999. (Cited on page 59)

[30] Judie Attard, Simon Scerri, Ismael Rivera, and Siegfried Handschuh. Ontology-based situation recognition for context-aware systems. In *Proceedings of the 9th International Conference on Semantic Systems*, pages 113–120. ACM, 2013. (Cited on page 59)

[31] Waltenegus Dargie, J Mendez, C Mobius, K Rybina, V Thost, A-Y Turhan, et al. Situation recognition for service management systems using owl 2 reasoners. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2013 IEEE International Conference on*, pages 31–36. IEEE, 2013. (Cited on pages 59 and 60)

[32] Mieczyslaw M Kokar, Christopher J Matheus, and Kenneth Baclawski. Ontology-based situation awareness. *Information fusion*, 10(1):83–98, 2009. (Cited on page 60)

[33] Vivek K Singh, Mingyan Gao, and Ramesh Jain. Situation recognition: an evolving problem for heterogeneous dynamic big multimedia data. In *Proceedings of the 20th ACM international conference on Multimedia*, pages 1209–1218. ACM, 2012. (Cited on page 60)

[34] Kerry Taylor and Lucas Leidinger. Ontology-driven complex event processing in heterogeneous sensor networks. In *The Semanic Web: Research and Applications*, pages 285–299. Springer, 2011. (Cited on page 60)

[35] Souleiman Hasan, Edward Curry, Mauricio Banduk, and Seán O'Riain. Toward situation awareness for the semantic sensor web: Complex event processing with dynamic linked data enrichment. *SSN*, 839:69–81, 2011. (Cited on pages 60 and 61)

[36] Nikolaus Glombiewski, Bastian Hoßbach, Andreas Morgen, Franz Ritter, and Bernhard Seeger. Event processing on your own database. In *BTW workshops*, pages 33–42, 2013. (Cited on pages 60 and 61)

[37] University of Marburg Database Research Group. Jepc - java event processing connectivity. http://dbs.mathematik.uni-marburg.de/research/projects/jepc/. (Cited on page 61)

All links were last followed on November 26, 2015.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature