Methods and Tools for Management of Distributed Event Processing Applications

Zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
der Fakultät für Wirtschaftswissenschaften
des Karlsruher Institut für Technologie (KIT)

genehmigte
Dissertation
von

Dipl-Inform.Wirt. Dominik Riemer

Tag der mündlichen Prüfung:	04. August 2016
Referent:	Prof. Dr. Rudi Studer
Korreferent:	Prof. Dr. Thomas Setzer

Abstract

Gathering and processing events from cyber-physical systems provides users with the opportunity to continuously be aware of current performance indicators and potentially upcoming issues (situational awareness) as well as to optimize maintenance processes based on the current condition of machines or other equipment (condition-based maintenance). Due to the large volume and variety of data and, in addition, the demand for real-time analysis, these scenarios require appropriate technologies. In this context, *Event Processing* has become an established technology to process event streams in real-time while providing capabilities to detect event patterns based on spatial, temporal or causal relationships.

In contrast, today's available systems still suffer from high technical complexity in terms of their underlying declarative languages. On the one hand, such systems require deep technical knowledge of event processing systems, making the development of real-time applications a time-consuming task due to slow development cycles. On the other hand, event processing applications are often highly dynamic in regard to oftentimes changing requirements of observed situations as well as frequent syntactic and semantic changes of incoming sensor data.

The main contribution of this thesis enables application specialists to define, modify and execute event processing applications in a self-service manner by abstracting from underlying technical details in form of so-called real-time processing pipelines. The contributions of this thesis are summarized as follows:

1. A methodology supporting the development of real-time applications under special consideration of extensibility and accessibility for application specialists.
2. Models to semantically describe characteristics of event producers, event processing agents and event consumers.
3. A system to execute processing pipelines consisting of geographically distributed event processing components.
4. A software artifact that supports graphical modeling of processing pipelines and automatic pipeline execution.

These contributions are introduced, applied and evaluated based on multiple scenarios from two application domains, manufacturing and logistics.

Contents

Figures

Tables

Listings

Part I

Introduction

1 Introduction

The term Internet of Things (IoT) defines a vision of a global infrastructure of physical and virtual devices which communicate based on standard web protocols [Vermesan and Friess 2014]. Typical IoT applications are often built on top of platforms providing capabilities to integrate a large variety of hardware- or software-based sensors and actuators in order to provide insights, derive situations and to trigger actions.

In industrial settings, the Industrial Internet of Things (IIOT) is an emerging paradigm focusing on integration of data continuously gathered from industrial processes, their correlation and analysis in order to (automatically) react on expected or unexpected situations. A good example to describe benefits of IIOT applications is the manufacturing domain. Production processes often involve a large number of different production machines, human workers and transportation systems as intermediaries between machines performing parts of the production process. Common goals in these use cases include optimized usage of resources and to achieve minimal production downtimes by avoiding unexpected failures. Recently, continuous monitoring of machine data in real-time has led to novel methods such as *Condition-based Maintenance (CBM)* [Jardine et al. 2006]. In CBM, machine maintenance is not longer scheduled at fixed intervals, but triggered in case of predicted breakdowns calculated through continuous analysis of any kind of sensor data which is known to have potential influence on machine performance. While in the past Manufacturing Execution Systems (MES) were often operated in an isolated way disconnected from other machines involved in the same production process, IIOT applications are able to further improve maintenance intervals by also taking into account dependencies between systems which are not directly connected. For instance, planned maintenance of two machines can be aligned in order to reduce downtime.

From a technology perspective, systems supporting such use cases need to gather and integrate data from multiple sources in a continuous manner, and perform operations (e.g., transformations and analyses) on these data streams in order to detect *situations of interest*, e.g., specific states in the system indicating a real-world problem which could affect the observed object of interest.

Event Processing Systems (EPS) have established themselves as technology drivers for processing of real-time data by applying declarative rules (e.g., in form of event

patterns) on a continuous stream of incoming events in order to detect situations with minimal delay. In this context, events are defined as occurrences within a particular system [Etzion and Niblett 2010]. Event processing is based on the Event-driven architecture (EDA) paradigm. EDA is a software architecture style defined by the following five principles: *Event reporting* (notifications of particular occurrences within a domain of interest), *push notifications* (events are pushed to interested *consumers* instead of requesting them), *immediate responsiveness* (consumers automatically react upon the occurrence of an event), *one-way communication* (events are sent in a fire-and-forget style) and *freedom of commands* (events do not contain information on operations that should be performed on it) [Chandy and Schulte 2009].
Event Processing provides methods to transform input event streams to output streams by implementing an event processing network which defines a set of filtering or transformation components called *Event Processing Agents* (EPA) and the routing between EPAs through the network. One of the main advantages of EP is its capability for event-at-a-time operations which enables systems to detect sequences of events as well as co-occurrences between events in a continuous fashion. Moreover, implementations of event processing networks are not hard-coded in a typical software engineering process, but make use of declarative (e.g., SQL-based) languages which contain the application logic. This logic is then translated into an executable event processing network specific to the actual technology used.
As EDA and, in particular, event processing systems work well in geographically distributed systems, they are a good fit as technology foundations for (I)IoT applications that demand for real-time data processing and immediate situation detection. However, a particular drawback of today's existing solutions for event processing is its accessibility for application specialists: While requirements for specific event processing applications are generated by domain experts such as business analysts, their implementation still remains a rather technical process requiring for development skills and deep knowledge of event-based applications. This circumstance not only slows down development processes of real-time applications, it also hinders event processing technologies from wider adoption in emerging application areas such as IIOT applications. Moreover, it can be shown that many use cases requiring for real-time processing do not rely on static programs which can be left untouched for a long time, but need to be adapted frequently in order to reflect new requirements and business needs.
This problem is further strengthened due to a lack of standards for event processing languages and representations for events leading to heterogeneous data formats, communication protocols and run-time implementations which cannot be easily connected. Especially for distributed application scenarios building upon concepts such as marketplaces where *application specialists* are able to create event processing applications in a self-service manner by combining data sources, transformations and consumers (also described as *sinks*) offered by different providers, this problem remains a challenge.

While similar problems have already been investigated in areas following the request/response interaction style like Service-Oriented Architectures (SOA) resulting in solutions for (semi-) automatic web service composition (e.g., Semantic Web Services) and Workflow Management, solutions for (complex) event processing following the EDA paradigm are still missing.
This thesis develops a novel methodology targeting the development process of event processing applications applicable to distributed application scenarios such as the IoT. Furthermore, we provide models to semantically describe event producers, event processing logic and event consumers in a higher-level way independent from their specific run-time implementation. These models are used to provide an integration layer which leverages business analysts to create event processing pipelines out of heterogeneous systems without the need for deep technical knowledge of event processing technology and by using a graphical modeling interface.

1.1 Research Questions

Our research aims to provide a novel methodology which allows application specialists such as business analysts with little programming experience to quickly build event processing *pipelines*, i.e., event processing applications which integrate distributed, heterogeneous real-time processing logic, event producers and event consumers. The main problem, the inability for application specialists to develop event processing applications along with the need to provide ways for such users to be able to quickly observe and get insights on ongoing problems and opportunities within their domain of interest leads to the following principal research question:

How can we enable application specialists to define and modify distributed event processing applications?

This research question poses three terms which require for further explanations: *Application specialists*, *event processing applications* and *distributed systems*. First of all, we target application specialists as the main role we aim to support in our research as an audience which is not familiar with technical details of specific characteristics of event-driven applications or even their technical implementation. Although this role might often be confused with business analysts as experts within a domain of interest, we use the term application specialists to strengthen the main purpose of our research which tries to shift development-oriented tasks to more specialist-oriented tasks. Roles and assigned tasks in our methodology are further detailed in chapter 6 where we introduce our methodology.
Second, event processing applications are systems following the event-driven architecture paradigm, which has a strong focus on push-based communication between data producers and consumers in a very loosely coupled fashion. In section 2, we

show that event processing applications usually implement an *event processing network* consisting of a set of data transformation components. In section 3, we argue why access for application specialists to stream processing applications is important due to a strong focus on the generation of real-time insights, which also implies short intervals from requirements elicitation to a deployed and running application.
Finally, support for distributed systems is a mandatory requirement for solutions targeting these problems. Opposed to event processing systems operating as *single-host systems*, an open research question here is how stream processing applications in geographically distributed settings can be built.
The principal research question itself can be broken down into four sub-questions, whereas each question targets a different view or aspect of the main research question.

Research Question 1 (Development Process)**.** *How can we improve the development process of event processing applications?*

The first research question deals with the development process of event processing applications. Answering this question requires an investigation of today's development process for event processing and the roles involved. Therefore, we analyze typical tool support available today ranging from pure programming-oriented models to graphical editors. Observations gathered from these tools are used to collect requirements for a novel methodology. The main challenge of this research question is to keep a high level of flexibility and expressivity provided by existing technologies, while at the same time usability for non-development oriented roles such as application specialists must be guaranteed.
This question is motivated in section 3.3.1 and answered in chapter 6.

Research Question 2 (Model)**.** *How can we model event processing blocks independent from their specific run-time implementation?*

Event processing blocks refer to the three main building blocks of event processing networks, namely event producers, event processing agents and event consumers. On the one hand, this question targets the development of a vocabulary providing a high-level abstraction from event processing operators in order to achieve accessibility for application specialists. On the other hand, technical details have to be included into the vocabulary to ensure proper execution of event processing networks. Moreover, as we specifically target scenarios such as the IoT, these models must include fine-grained specifications of sensor properties (e.g., quality aspects) and be independent from specific implementation details. The main challenge is to provide reusable models which are specific enough to support event processing operators as well as custom on-line analytics methods and ensuring extensibility, while at the same time general enough to allow business analysts to instantiate them without developer support.
This question is motivated in section 3.3.1 and answered in chapter 7.

Research Question 3 (Domain Knowledge)**.** *How can we separate domain knowledge from the technical specification of event processing languages?*

Domain knowledge refers to non-technical details which are part of the specification of an event processing network. For instance, event pattern representations include both *technical details*, such as event schema descriptions, as well as *domain knowledge* such as threshold values. This makes even those modifications, which are only related to the business logic of event processing networks, a time-consuming task. Therefore, an approach is required to separate the definition of domain knowledge from the specific technical implementation. The goal is to store and maintain knowledge apart from technical details and automatically adapt affected event processing applications once the knowledge base has been modified.
We further motivate this question in section 3.3.2 and answer it in chapter 7.

Research Question 4 (Execution)**.** *How can we author and execute event processing pipelines consisting of heterogeneous processing blocks?*

This question includes two aspects: The first part is related to the authoring of event processing pipelines. This part aims at providing ways to build pipelines consisting of multiple building blocks described using implementation-independent models in a way which is acceptable for application specialists. The second part of this question is how to execute such pipelines even though they are composed of heterogeneous and geographically distributed run-time implementations.
This question is motivated in section 3.3.3 and answered in chapter 8.

1.2 Research Methodology

The research methodology of this thesis is based upon the design science paradigm. Design science refers to a problem-solving process which aims to create and evaluate IT artifacts intended to solve identified organizational problems [Hevner et al. 2004]. Artifacts in this sense may be software like implemented prototypes demonstrating the solution the research was targeting at, but may also be models or formal logic. Design science research relies on the principle that "knowledge and understanding of a design problem are acquired in the building and application of an artifact". Thus, Henver et. al. have proposed guidelines which should be addressed in research relying on the design science paradigm, namely *Design as an Artifact* ***(G1)***, *Problem Relevance* ***(G2)***, *Design Evaluation* ***(G3)***, *Research Contributions* ***(G4)***, *Research Rigor* ***(G5)***, *Design as a Search Process* ***(G6)*** and *Communication of Research* ***(G7)***.
The main artifact developed within the course of this thesis is a novel methodology for the development and management of stream processing applications. This artifact has also been instantiated in a software toolkit called *StreamPipes* [Riemer et al. 2015] which covers all phases and tasks our methodology is built upon **(G1)**. This

methodology is built upon a specific organizational problem that has been identified as the non-accessibility of stream processing applications for application specialists such as business analysts. In chapter 3, we focus on problem relevance **(G2)** by further elaborating on the specific problem, and, even more important, by showing the need to solve this problem. Design evaluation **(G3)** is done in the end of this thesis by presenting multiple evaluations we performed to gain insights on the performance of our solution, model completeness and consistency as well as usability. Therefore, the main research contributions **(G4)** of this thesis are the methodology and a prototype system we developed as the artifact itself. Guidelines **G5** and **G6** have been implemented by presenting foundations presented in chapter 2, a requirements collection and a discussion of related work in chapters 4 and 5 as well as the presentation of evaluation results in chapter 9. The main contributions of this research have already been communicated to technical-oriented audiences at various outlets presented in more detail later in this section **(G7)**.

1.3 Contributions

This research has led to several contributions along the research questions identified and detailed in section 1.1.

- **(C1) Methodology**. We present a novel methodology supporting the development of stream processing applications. It consists of two phases, the first one named setup phase targeted at software developers and the second one named execution phase targeted at business analysts and pattern engineers (further detailed in section 6.4). The goal of the setup phase is the development of *reusable* stream processing logic along with a semantic description specifying input requirements, generated output and required static data such as user input or required external knowledge. This description is used within the execution phase to assist application specialists in defining event processing pipelines out of the main event processing building blocks event producers, event processing agents and event consumers. This contribution is related to research question 1.
- **(C2) Models**. Besides the methodology, we have developed vocabularies and models to describe event producers, event streams, event processing agents and background knowledge. These models are designed to support characteristics of heterogeneous data sources (e.g., text-based streams and sensor data streams), but specifically focuses on fine-granular description of sensor-based streams, e.g., by providing sensor-specific capabilities to define quality aspects such as frequency and accuracy. Models additionally support the description of event processing logic as a higher-level abstraction based on input requirements, abstract output type and required static data. The models have been defined in RDF

ensuring interoperability between heterogeneous run-time implementations. This contribution is related to research questions 2 and 3.
- **(C3) System**. We provide a system to define and execute stream processing pipelines in a distributed manner. This contribution targets distributed application scenarios such as IoT platforms and marketplaces and includes a reference architecture to build stream processing pipelines for users with little programming skills. This contribution is mainly related to research question 4.
- **(C4) Software Artifact**. The conceptual contributions C1-C3 have been implemented as a software artifact. To demonstrate feasibility and applicability of our approach for evaluation purposes, we provide software tooling covering all phases and tasks of our methodologies. This includes run-time wrappers abstracting from data models, formats and protocols for different event processing engines demonstrating the ability of our approach to combine heterogeneous run-time implementations, and an application programming interface (API) to describe stream processing logic according to our models without required knowledge of semantic web technologies. In addition, we provide a web-based authoring tool for the definition of event processing pipelines, a model editor and code generation module supporting the extension of our system at runtime and an integration layer for semantics-based matching of stream requirements and capabilities between event processing building blocks and execution management.

1.4 Research Projects and Publications

The outcome of the research presented in this thesis is the result of our work in several European and German-funded research projects. Our methodology and its design have been presented in various publications, while software demonstrations of our prototype have been given at several conferences.

ALERT (Active Support and Real-Time Coordination based on Event Processing in FLOSS development, 08/2011-05/2013, EU-STREP)

The goal of the ALERT project was to provide a platform which improves the collaboration of software developers in medium and large-scale software development projects. ALERT followed the idea of integrating different tools supporting the whole software development life cycle such as issue trackers (e.g., Jira), source code management systems (e.g., Subversion), wikis (e.g., Mediawiki). On top of real-time sensors that were deployed in those systems, we developed an event processing infrastructure which allows users to define a complex event pattern that matches their interest by using a graphical modeling tool. For instance, developers could create notification if a bug they were assigned to was reopened within a specific time period after it was

closed together with frequent activity in an online forum covering technical aspects of this bug.
In ALERT, we discovered the need for an approach as described in this thesis and performed first experiments with a higher-level language for event processing systems. We further analyzed the need of non-technical users to define situations of interest in an ad-hoc manner. In addition, we developed the conceptual model for a transformation module that is able to translate event patterns modeled in a graphical editor into a logic-based event processing engine (ETALIS).
Publications

- Dominik Riemer, Ljiljana Stojanovic, Nenad Stojanovic. **Using Complex Event Processing for Modeling Semantic Requests in Real-Time Social Media Monitoring.** Sixth International AAAI Conference on Weblogs and Social Media (ICWSM). 2012, Dublin, Ireland. (see [Riemer et al. 2012]).
- Ljiljana Stojanovic, Sinan Sen, Jun Ma, Dominik Riemer. **ALERT: Semantic Event-Driven Collaborative Platform for Software Development.** Proceedings of the 6th International Conference on Distributed Event-Based Systems. 2012, Berlin, Germany. (see [Stojanovic et al. 2012]).
- Dominik Riemer, Ljiljana Stojanovic, Nenad Stojanovic. **Demo: ALERT: Real-Time Coordination in Open Source Software Development.** Proceedings of the 7th International Conference on Distributed Event-Based Systems (DEBS). 2013, Arlington, Texas, USA. (see [Riemer et al. 2013a]).

Reflex (Reinforcing FLEXibility of SMEs by Dynamic Business Process Management, 08/2011 - 03/2013, EU-SME)

In Reflex, we were working on an architecture for adaptive business process management in the logistics domain. Despite further collecting requirements for application specialist-oriented access to event processing applications, our work in Reflex has also influenced the development of our event model related to the representation of geographically-enriched sensor data.
Publications

- Babis Magoutas, Dominik Riemer, Dimitris Apostolou, Jun Ma, Gregoris Mentzas, Nenad Stojanovic. **An Event-Driven System for Business Awareness Management in the Logistics Domain.** Business Process Management Workshops, Springer Berlin Heidelberg, 2013. (see [Magoutas et al. 2013]).

ProaSense (The Proactive Sensing Enterprise, 11/2013-01/2017, EU-STREP)

The main parts of this thesis have been developed within the ProaSense project. The overall goal of ProaSense was to develop a system for proactive monitoring and detection of failures within industrial manufacturing processes. In ProaSense, the methodology presented in this thesis has been conceptually designed, implemented and evaluated.

Publications

- Dominik Riemer, Nenad Stojanovic, Ljiljana Stojanovic. **A Methodology for Designing Events and Patterns in Fast Data Processing.** Proceedings of the 25th Conference on Advanced Information Systems Engineering (CAiSE). 2013, Valencia, Spain. (see [Riemer et al. 2013b]).
- Dominik Riemer, Ljiljana Stojanovic, Nenad Stojanovic. **SEPP: Semantics-based Management of Fast Data Streams.** Proceedings of the 7th IEEE International Conference on Service-Oriented Computing and Applications (SOCA). 2014, Matsue, Japan. (see [Riemer et al. 2014]).
- Dominik Riemer, Florian Kaulfersch, Robin Hutmacher, Ljiljana Stojanovic. **StreamPipes: Solving the DEBS Challenge with Semantic Stream Processing Pipelines.** Proceedings of the 9th International Conference on Distributed Event-Based Systems (DEBS). 2015, Oslo, Norway. (see [Riemer et al. 2015]).

1.5 Structure of the Thesis

This thesis is structured as illustrated in figure 1.1. First, we describe basic foundations in the field of event processing by discussing relevant terms, architectures and technologies. Section 3 motivates our research by presenting two motivating scenarios which illustrate challenges this thesis targets. These scenarios are then used to formulate a problem statement. Section 4 discusses previous work on design-time issues of event processing systems as well as other related areas with similar problems such as workflow management. Motivation, identified problems and related work are then used to perform a requirements elicitation process in 5. Afterwards, in section 6, we introduce our methodology targeted at providing access for application specialists to event processing applications. This methodology, consisting of two phases, is further detailed in section 7 by presenting a semantics-based model for the representation of streams, event processing agents and event consumers, followed by a model for the definition and execution of stream processing pipelines in section 8. Section 9 presents evaluation results, which is followed by a conclusion and an outlook for future work.

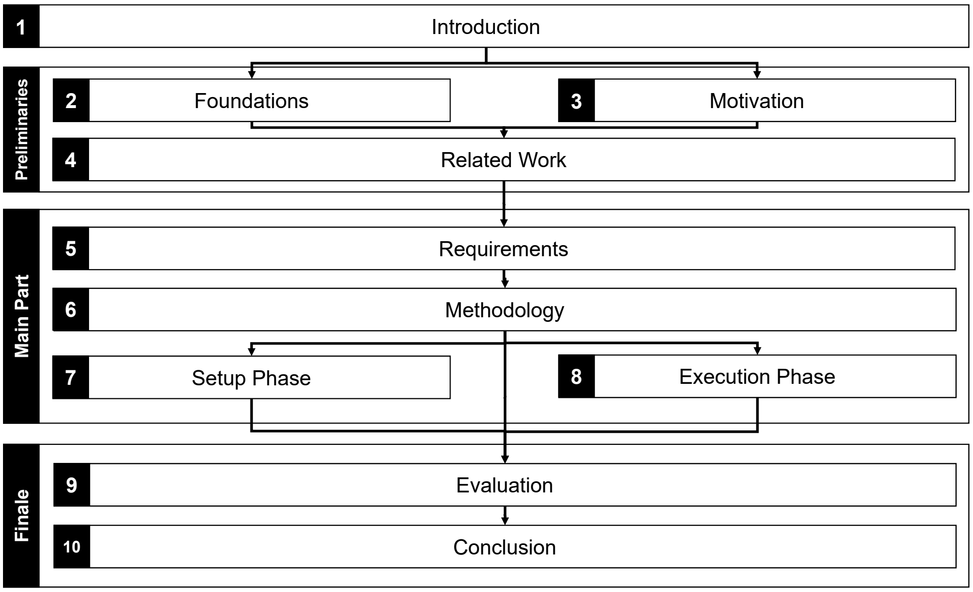


Figure 1.1 Thesis Structure

Part II

Preliminaries

2 Foundations

This chapter introduces the foundations of this thesis. We start with explaining basics in the area of *Information Flow Processing*. We further elaborate on the notion of *Events* and *Event Processing* as the main field this thesis is built upon.
The main content of this thesis is based in the field of Real-Time Data Processing. Real-time data processing refers to systems that are able to process data immediately as it arrives. In this context, it is important to define the scope of the notion of real-time. In contrast to hard real-time systems where data must be processed within the scope of a given time guarantee, near real-time (or also business real-time) systems are designed to process data as soon as possible with small latency, but without certain processing guarantees [Luckham 2011].

2.1 Information Flow Processing

Systems for Real-Time Data Processing originate from the area of *Information Flow Processing (IFP)*. Such systems have their origin in early systems for discrete event simulation. In general, IFP subsumes tools which are capable of timely processing of large amounts of information as it flows through the system [Cugola and Margara 2012]. The two main building blocks of IFPs mentioned here are *timeliness* and *flow processing*. Timeliness refers to the notion of near real-time and expects an IFP system to immediately detect any information that is being observed by the system immediately after occurrence. The term flow processing characterizes IFPs by the continuous analysis of data in contrast to other systems like databases where queries are triggered upon the request of a user.
IFPs can be further categorized into a set of different technologies, namely *Active Databases*, *Data Stream Management Systems*, *Publish/Subscribe* and *Complex Event Processing* [Andrade et al. 2014].
One of the first approaches to bring real-time awareness to business-level software were active databases [McCarthy and Dayal 1989]. Such systems are usually built as extensions on top of traditional databases and provide capabilities to react on specified conditions of database states. Usually, such systems implement the *Event-Condition-*

Action (ECA) paradigm [Paton and Diaz 1999] by capturing events from a database, checking for a condition that has been specified in advance and executing a certain action. While such systems were already able to detect Situations of Interest (SoI) in a timely manner, they do not target use cases that include very high volumes of real-time data as in today's IoT scenarios and were not primarily designed to perform stateful operations, e.g., detection of sequences on event streams by applying sliding windows.

In contrast to active databases, Data Stream Management Systems (DSMSs) go one step further and operate on unbounded event streams. Streams are considered an unbounded sequence of form of so-called *events*, usually ordered by time. A DSMS expects a query as an input and runs this query in a continuous manner until it is (manually) stopped [Babcock et al. 2002]. Incoming data is matched against such a Continuous Query (CQ). In general, DSMS work like a database from an upside-down view: While databases perform rather dynamic queries against a more static dataset, DSMS run static queries against dynamic event streams [Cugola and Margara 2012]. Typical DSMS scenarios often include monitoring tasks where users are actively notified (also in form of event streams) once the situation specified in the CQ can be found in the incoming data stream.

The next technology area in the field of IFPs are publish/subscribe systems. Such systems implement a loosely coupled, asynchronous architectural design which relies on the following components: A message publisher, a message-oriented middleware (typically a message broker) and a message subscriber. Publishers push data asynchronous in a fire-and-forget manner to the middleware. Depending on the architecture, subscribers can express their interest in a specific event by means of either an event channel, where streams are published to a specific *topic* that usually groups events of a similar type (topic-based publish/subscribe) or by means of a condition such as filter rules which are directly applied on each incoming event (content-based publish/subscribe) [Eugster et al. 2003]. Many publish/subscribe systems additionally support topic hierarchies enabling systems to subscribe to a more general topic and receive all subtopics that are leafs of the topic hierarchy.

Finally, the (Complex) Event Processing (CEP) paradigm has become a popular technology within the last years. CEP relies on many foundations from DSMS and publish/subscribe systems and extend these with more advanced operators to define *event patterns* in form of detecting sequences or correlations between single events in order to create derived events [Andrade et al. 2014]. CEP systems also make a strong focus on the occurrence time of events by providing mechanisms to correlate events based on sliding data windows.

In the following sections, we introduce the term *Event Processing* in more detail and elaborate on characteristics and implementation details of event processing systems.

2.2 Events

In real-time data processing, we often face the notion of an *event*. The term event in the context of real-time data processing was first introduced by Luckham who defines an event as being the record of an activity in a system [Luckham 2002]. In this sense, we consider an event as something that occurs after an activity in any kind of system has happened. We will later see that a system in this definition can be anything that is being able to generate an event, e.g. a software system that receives orders from an web shop or a hardware sensor that measures air humidity.
Another definition of the term event is given by Etzion who defines an event as follows [Etzion and Niblett 2010]:

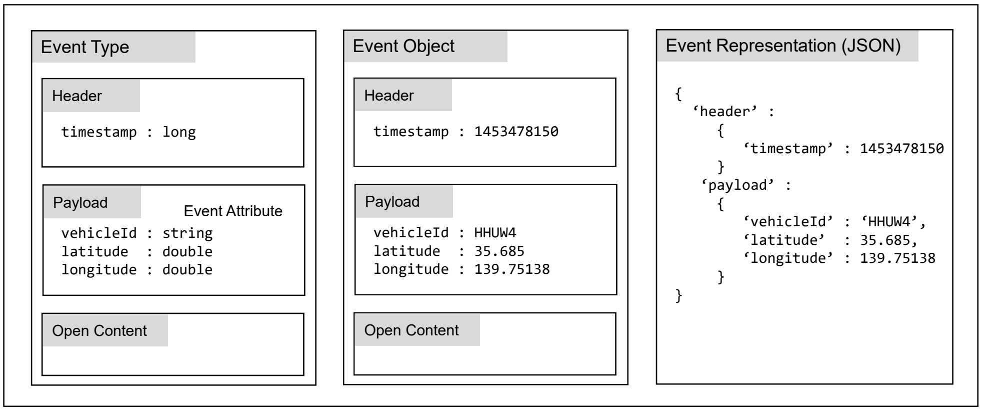
> An event is an occurrence within a particular system or domain; it is something that has happened or is contemplated as having happened in that domain.

The main difference to Luckham's event definition is a slightly broader view on an event occurrence. Etzion does not only consider anything that has happened within a system (which is closely related to Luckham's *record of an activity*) as an event, but also includes things that are contemplated as having happened in his definition. This allows to cover the practical fact that most event processing systems may also detect events which must later be recognized as false positives, which Etzion explains by the means of a fraud detection system producing irrelevant fraud events [Etzion and Niblett 2010].
Now as we have defined the general *notion* of an event, we elaborate on the *characteristics* of events. According to Luckham, events are mainly defined by three aspects [Luckham 2002]:

- **Form** requires that an event contains at least a set of data components.
- **Significance** requires that an event signifies the activity it is related to
- **Relativity** requires an event to be related (e.g., by time or causality) to another event of a system.

As an example, we consider an event that is produced by a Global Positioning Service (GPS) sensor to submit current vehicle positions. The form of such an event typically includes data components like the actual encoded geospatial coordinates and a vehicle identifier. Its significance is determined by the activity of a new coordinate that the sensor has acquired. Finally, the relativity of such an event is often characterized by the time of this event which assigns the coordinates to a specific time at which the value has been measured.
These requirements also clearly distinguish an event from a simple *message*, which also may have a form similar to an event but may be missing the two other characteristics, *significance* and *relativity* [Luckham 2002].
Finally, we elaborate on the *representation* of events. As we assume that most events relevant in the scope of event processing systems have to be processed by an IT system

to gather any new insights from the raw event, events need to be represented in a machine-readable form. This includes the structure of an event as well as the actual data format. In order to describe the structure, the main building blocks of an event are *event objects*, *event types* and *event attributes* [Etzion and Niblett 2010].
First, *Event objects* are considered the representation of an event occurrence. A set of event objects with the same structure and semantics specify an *event type*. Each event object has a set of data components which include the data that is part of the event payload. These data components are called *event attributes*.
In this context, we may often also find the term *event instance* as a synonym for *event object* and *event property* as a synonym for *event attribute*.
Many event processing systems available today allow to define an event instance based on the following structure, as also described by Etzion: An event instance consists of a *header*, a *payload* and eventually some additional *open content*.
Figure 2.1 illustrates an example event representation. Header attributes are commonly referred to as specifications that contain meta-information about the event. An important header attribute is usually the occurrence time of an event, but may include other attributes such as event certainty, a type specification or further information about the source that produces the event instance. Event payload focuses on the data components which signify the corresponding activity. Typically, the event type specifies a fixed set of event attributes for all event instances of this type. Payload definitions usually include the name of an event attribute as well as the data type (which in most cases must at least contain a primitive data type specification). Last, the open content part may contain more information on the event instance in a potentially arbitrary format.



```
{
  'header' :
     {
        'timestamp' : 1453478150
     }
   'payload' :
     {
        'vehicleId' : 'HHUW4',
        'latitude'  : 35.685,
        'longitude' : 139.75138
     }
}
```

Figure 2.1 Event Representation: Example

In terms of the data format, the representation of an event depends on the actual implementation that is used for event processing. In many cases, events are represented in standard data formats such as programming language-specific (e.g., Java's Plain Old Java Objects (POJOs)) or interchangeable formats (e.g., Extensible Markup Language (XML)[1] or JavaScript Object Notation (JSON)[2]) .

2.3 Event Processing

Event processing is about performing operations on events [Luckham 2002]. Operations in event processing might include reading, creating, transforming and deleting events [Etzion and Niblett 2010]. It is noteworthy that events in event processing are typically considered as immutable [Luckham and Schulte 2011]. According to our definition of the term event above, where an event is defined as something that has happened and therefore cannot being made unhappened, deleting events is often referred to removing events from a stream in form of a filtering operation. During the last years, event processing has gained considerable attraction in both research and practice. This has led to quite a large variety of different terms, which, besides event processing are mainly *Stream Processing*, *Complex Event Processing* or *Event Stream Processing*. For instance, some authors distinguish between *Stream Processing* and *Complex Event Processing* in the form that the first is focused more on stateless operations (one event at a time is transformed without taking into account other events) on event streams with very high throughput and the latter setting a focus on fine-grained, stateful operations (other events are kept in memory) on streams in order to find (time and causal) correlations [Bruns and Dunkel 2010]. On the other hand, conceptual models such as the *Event Processing Network* which is described in more detail below, define a complete set of abstract processing agents subsuming stateless and stateful operators. Therefore, in this thesis we will consequently use the term *Event Processing* to subsume all technologies that are able to a) focus on the processing of unbounded data streams, b) provide a high-level API or Domain-Specific Language (DSL) which implement one or more specific event processing agents and c) provide a runtime implementation which performs operations on events as defined by Etzion [Etzion and Niblett 2010].
In the following section, we further explain structure and scope of event processing networks.

2.3.1 Event Processing Network

The architecture of event processing systems can be described as an Event Processing Network (EPN). An EPN consists of three main components, namely *Event Producer*

[1] https://www.w3.org/XML/
[2] http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

(EP), *Event Processing Agent (EPA)* and *Event Consumer (EC)* as well as a fourth component *Event Channel*, which acts as an intermediary element between some of the three main components. An EPN defines an event processing application by describing the flow of an event stream from an event producer to an event consumer as well as any intermediate processing tasks which transform the input event stream to an output event stream. An EPN might consist of an arbitrary number of event producers, event channels, event processing agents and event consumers and can contain feedback loops, so that the output of an EPA can be the input of a preceding EPA. An EPN can be formally defined as a graph [Sharon and Etzion 2007]. Event processing systems typically implement EPNs in form of rule-based languages, imperative programming languages or stream-oriented languages similar to database query languages [Etzion and Niblett 2010].
Figure 2.2 illustrates a simple EPN which is further explained in the following sections.

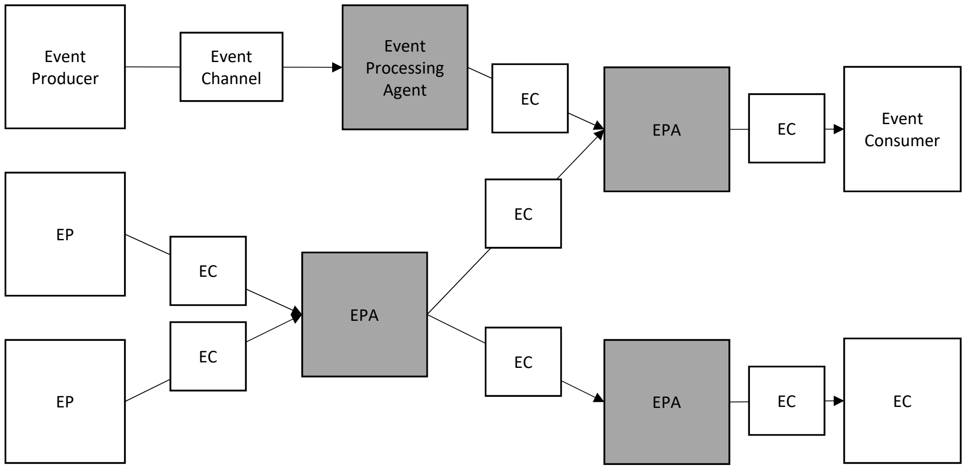


Figure 2.2 Event Processing Network

Event Producer (EP)

An EP acts as a source of an event processing system and therefore pushes data into the system. It is often related to some real-world entity such as a hardware sensor and can produce one or more event streams. Producers observe the system they belong to in an non-invasive manner so that the system itself is not being changed but can be observed for the occurrence of an event of interest [Luckham 2002]. Additionally, producers adapt the observed events and transform them into a machine-readable format which can be processed by the event channels and processing agents.

Examples for event producers include software applications (e.g., order management systems), processes (e.g., workflow management systems) or sensors (e.g., mobile phone sensors). Etzion categorizes different types of event producers by hardware, human interaction and software [Etzion and Niblett 2010]. In this categorization, human interaction refers to events that are manually generated by humans, such as mobile terminals in factories which expect scrap information to be entered by factory workers or security systems which expect biometric data for access control.

Event Channel

The next building block of event processing networks are event channels. The main purpose of event channels is to receive events from an EP or EPA, to make routing decisions and to forward events to another main component of the EPN.
At a first glimpse, it may appear non-intuitive that channels are modeled as blocks in an EPN architecture (and not as an arrow as the name *channel* may suggest). However, channels often have their own logic: Quality of Service (QoS) attributes such as security considerations (which roles are allowed to access a specific topic), encryption of streams, filters (content-based publish/subscribe) or performance restrictions therefore justify the modeling of channels itself as blocks.
In real-world systems, channels are usually implemented by using a message-oriented middleware which supports the publish/subscribe pattern. In this case, events are published to the middleware, where they are forwarded to the succeeding components of the EPN. In some settings it might be feasible that all event channels of an EPN use a single event broker. However, especially in highly distributed application scenarios such as the IoT where components have to communicate via different publish/subscribe protocols, event channels can rely on heterogeneous channel implementations.

Event Processing Agent (EPA)

EPAs are the central concept of an EPN. Processing agents perform the actual real-time processing logic by transforming input event streams to output streams according to their specific type. In this section, we introduce the conceptual model and a hierarchy of event processing agents from the literature which is illustrated in figure 2.3. The section thereafter shows how this model can be implemented in different event processing systems.
In general, EPAs perform operations which can either be stateless (as only a single event is used to perform the operation) or stateful (as more than one event is used in order to produce the output) [Etzion and Niblett 2010].
The most basic EPA type is the **Filter EPA**. Filtering does not transform an event (as the event type of the output is equal to the input event type), but outputs a subset of the incoming event stream. According to Etzion, a Filter EPA may have several *output terminals*, e.g., in order to output a result stream which contains the events filtered

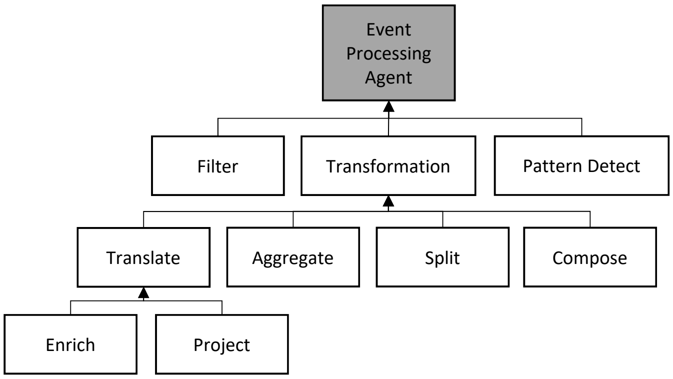


Figure 2.3 Hierarchy of Event Processing Agents

by the filter expression, another output stream which includes all events that did not match the filter expression and a third output stream providing events where the filter expression could not be applied upon. A valid specification of a filter EPA does not require to offer all output terminals.
Transformation EPAs denote a class of processing agents which, in contrast to pure filtering, perform an operation that also changes the type of the input event stream. Generally speaking, a transformation *derives* an output event from one or more input events. This class subsumes the EPAs as follows:

- **Translate EPAs** expect a single input event and derive an output event by applying a derivation formula. Therefore, the output event type differs from the input event type. For instance, an EPA which transforms a vehicle position event that provides the geospatial vehicle position as latitude/longitude coordinates to another position event which represents the same position using another coordinate system is considered a transformation EPA. Two special cases of translate EPAs exist: First, an **Enrich EPA** extends an input event with one or more additional event attributes (e.g., extending the vehicle position event with an additional attribute *city* which is being derived by a reverse geocoding function). Additional properties are often derived from *global state elements*, which perform a (e.g., database) function on static data sources by providing an attribute from the input event as a function parameter (e.g., an event property that identifies a machine which is used to enrich the event with additional machine parameters). **Project EPAs** operate the other way around and therefore remove an event attribute from the input event stream, so that the output event contains only a subset of the input event properties.

- **Aggregate EPAs** operate in a stateful fashion. The input is a set of events, which are temporarily stored in the EPA, often in form of sliding windows. A function is applied in order to calculate aggregated values based on the input events. As an example, we consider an event which measures the oil temperature of a machine. A component which measures the average oil temperature during the last 5 minutes is considered an Aggregate EPA.
- **Split EPA**. Sometimes it is required to split an event stream in order to route events to multiple successors which perform different operations. In this case, a single input event is being duplicated by a Split EPA so that a collection of output events is produced. Output events are then routed to different destinations. An example use case is an EPN which observes and aggregates oil temperature values as explained above. Usually, such monitoring EPNs are used for two purposes: One the one hand, the aggregated values are filtered against a threshold value which could trigger a notification in case that high temperature values occur in a machine. On the other hand, such data often needs to be stored in storage components such as databases in order to prepare data for off-line analysis. Such use cases can be implemented with the help of Split EPAs.
- Finally, **Compose EPAs** require a collection of events as input and produce a single derived event as an output. Such an operation is similar to join operations known from database query languages. Compose EPAs can either compose a new output event based on the last event from each input stream, or they can provide a matching operator which allows to define restrictions on the input events that should be combined.

Pattern Detect EPAs provide operators to detect more advanced *situations* in event streams. We already mentioned that occurrence time is an important concept in event processing systems. Pattern Detection makes strong use of event time by taking into account the sequence of an event in the stream and is seen as one of the most important benefits of event processing. In order to explain the function of a Pattern Detect EPA, we start with defining the notion of pattern. For a complete specification of event patterns, we refer to the literature [Etzion and Niblett 2010].

In general, a pattern in event processing is defined as a template which specifies a combination of events. At runtime, this template is matched against the event stream and outputs an event once the template satisfies the event sequence.

According to Etzion, Pattern Detect EPAs can be divided into basic patterns and dimensional patterns. **Basic patterns** implement basic operations on event types or on a set of event types. This category includes *logical patterns*, *threshold patterns*, *subset selection patterns* and *modal patterns*. Logical patterns define a template where a specific set of event must occur together, e.g., within a certain time. Threshold patterns additionally include an aggregation of event occurrences, which can be used, for instance, to detect at least a minimum count of events of the same type within a certain time. Subset patterns define a class of patterns where the template specifies

a set of output events which must be inferred from the input stream (e.g., the top-k highest temperature values of a machine during one hour) . Finally, modal patterns require that a given assertion is valid for all events in the event set of interest. The *always pattern* which detects a set of events where all events in the set satisfy a given assertion is an example from this group.
In contrast, **dimensional patterns** focus on the occurrence time or other dimensions of events. These include, besides temporal aspects, spatial aspects or a combination of both.
A frequently used example pattern which makes strong use of temporal aspects is the *sequence pattern*. This pattern detects a sequence of events, which can be defined in a fine-granular manner by not only taking into account the order of event occurrences, but also the exact time which elapses between two or more events in the participant set. Examples for spatial patterns include geospatial operations on location-based events. For instance, geospatial patterns can be used to perform advanced geofencing operations in order to detect when a location-based event arrives within a given geographical area.
Although this list of pattern detection EPAs cannot be considered complete and is likely to be extended with other dimensions or subtypes, an event processing operation can usually be assigned to one of the top-level EPAs as defined by Sharon and Etzion [Sharon and Etzion 2007].

Event Consumer (EC)

ECs represent sinks in event processing networks. ECs receive events from its predecessors in the event processing network and include application logic which decides about the intended actions that should be invoked after an event has been received. An EC can perform a wide variety of different subsequent operations, whereas a high-level categorization can be given in the same way as event producers: Hardware event consumers often implement actuators which directly invoke a specific action of a hardware device. For instance, a situation detected by the event processing network (e.g., high oil pressure values observed over some time period) might require to automatically open a valve in the corresponding machine. Human interaction consumers represent a class of event consumers which require any human interaction, this category mainly involves the generation of alerts, notifications or visualizations in form of dashboard tools, which, depending on the consumer implementation, might also include semi-automatic decision-making support. Finally, software consumers interact with and manipulate software systems. A typical use case in this category is the interaction with business processes, so that event consumers might automatically invoke web services in order to trigger business processes.

2.3.2 Event Processing Systems

Now after having presented the conceptual model behind the event processing paradigm, we introduce some characteristics of existing Event Processing Systems (EPSs) from a more implementation-oriented point of view. Although many EPSs originate from financial application domains being the main technology driver for real-time situation detection tasks (e.g., fraud detection and algorithmic trading), event processing has already been applied in heterogeneous application scenarios such as logistics [Metzke et al. 2013], manufacturing [Bousdekis et al. 2015], social media monitoring and marketing [Riemer et al. 2012], early news and topic detection [Osborne et al. 2014] or even earthquake prediction [Olson et al. 2011]. Therefore, many existing systems today are generic enough to support a wide variety of use cases and major software companies like Oracle[3], SAP[4], Microsoft[5] or Software AG[6] have brought event processing products to the market[7].
Besides industrial solutions, a number of open-source event processing systems exist, for instance, Esper [8], Etalis [Anicic et al. 2012] and the WSO2 Complex Event Processor[9].
Recently, the demand for fast processing of event streams with very high throughput has led to the emergence of another class of EPS under the label *Distributed Stream Processing*. These systems do not provide the same level of developer support in terms of the programming model, but are able to provide implicit parallelism so that processing tasks are automatically distributed around a cluster of computing nodes [Andrade et al. 2014]. The first system in this category is Apache Storm[10], which was open-sourced by Twitter where Storm was acquired and later extended in order to provide real-time search results on the Twitter stream. Besides Storm, Apache Spark Streaming[11] and Apache Flink[12] also offer scalable open-source solutions for large-scale event processing. Although sometimes EPSs are distinguished from stream processing systems in terms of their processing semantics by means of support for event-at-a-time operators (e.g., event sequences) in contrast to a set-at-a-time focus for stream processing systems, we can recently observe that the boundaries between both

[3] Oracle Event Processing, see http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/index.html
[4] SAP Event Stream Processor, see http://go.sap.com/product/data-mgmt/complex-event-processing.html
[5] Microsoft StreamInsight, see https://technet.microsoft.com/en-us/library/ee362541(v=sql.111).aspx
[6] Apama, see http://www.softwareag.com/corporate/products/apama_webmethods/analytics/overview/
[7] see http://www.complexevents.com/2015/05/10/cep-tools-market-survey-1q-2015 for a market survey of EP applications
[8] http://www.espertech.com, also available with commercial support
[9] http://wso2.com/products/complex-event-processor/
[10] http://storm.apache.org
[11] http://spark.apache.org
[12] http://flink.apache.org

categories blur as they converge into a single class of systems. For instance, Flink now also provides a CEP library which implements support for event-at-a-time operators such as sequences and sliding windows.
In the next section, we give a short overview of implementation aspects of these systems. First, we focus on design-time issues by analyzing the programming model of some systems. Second, we discuss run-time aspects by discussing some typical commonalities and differences of the systems introduced in this section.

Design-Time Aspects

Design-time aspects mainly cover issues around the programming model of EPS. The programming model defines the event processing network by means of required event schemas, the processing logic, the data flow between the individual components of the network and consumers. Although recently there has been some research in the area of pattern mining (e.g., [Margara et al. 2014]), i.e., by deriving relevant situations of interest directly from the event stream, the definition of the intended behavior of an EPN is usually done in an expert-driven manner.

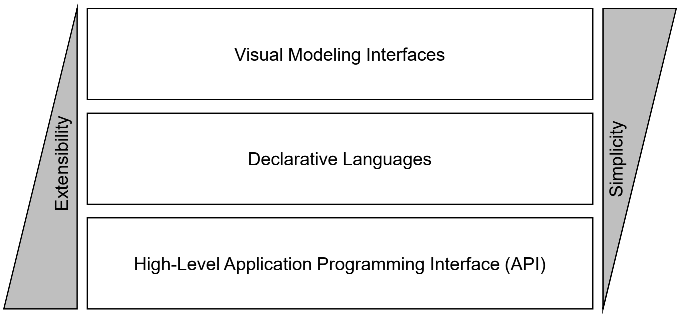

Visual Modeling Interfaces
Declarative Languages
High-Level Application Programming Interface (API)
Extensibility
Simplicity

Figure 2.4 Programming Models of Event Processing Systems

In general, the programming model can be divided into different abstraction layers as illustrated in figure 2.4. The most basic category are high-level APIs. Such APIs are often used as a programming model for state-of-the-art distributed stream processing systems. Although these models often provide high-level support for specific operators frequently used in event processing, they require the developer of the event processing logic to be familiar with the programming language the API relies on. The complete application logic is therefore written in code and must be compiled and deployed in order to execute the event processing network. Therefore, runtime changes of the event processing network (e.g., the modification of event schemes) are not possible or hard to achieve in many cases. Although this category offers a high

level of extensibility, development of such logic is often more time-consuming than other programming models.
Due to that reason, declarative or domain-specific languages (DSL) have been developed in order to assist developers in writing event processing logic. This category comprises *stream-oriented languages* and *rule-oriented languages* [Etzion and Niblett 2010].
Stream-oriented languages are based on data flow programming and basically implement an event processing network as a directed graph of nodes. The syntax of such languages is often defined as an extension of existing Structured Query Language (SQL)-based languages known from relational databases. Extensions comprise operators specific to event processing, especially data windows and dimensional patterns. An example stream-oriented language is Esper's Event Processing Language (EPL) or Oracle's Continuous Query Language (CQL).
Rule-oriented languages subsume *production rules*, *active rules* and *logic programming rules*. Production rules detect state changes in a forward-chaining way by firing an action once a condition has been fulfilled in the event processing network. Active rules implement the event-condition-action paradigm by waiting for a specific event to occur, evaluating this for a given condition and firing an action once it is fulfilled.
Finally, visual modeling interfaces provide graphical tool support to define an event processing network. This allows developers to implement the processing logic in a drag-and-drop style by connecting streams to EPAs and consumers. Although these systems usually require less programming, they are often provided as plug-ins for integrated developer environments and therefore target developers and not business analysts as they still require knowledge on implementation details of the system. For instance, StreamBase Studio [13] (illustrated in figure 2.5) is based on the Eclipse Modeling Framework (EMF). Although these systems are extensible by providing high-level APIs for the definition of new processing agents, the modeling toolkits themselves are bound to the operators available in the system.

Run-Time Aspects

Run-time aspects focus on the behavior of an EPS once it has been configured. The programming model described in the previous section is translated into an executable program depending on the implementation of the event processing system.
One important run-time aspect of EPSs is scalability. Scalability relates to a system's performance and refers to its capability to accommodate increasing loads by adding new resources (often in form of additional computing nodes). Scalability in event processing systems is an important issue as the throughput of an event stream might change very fast without prior warnings. For instance, in 2013 Twitter's tweet count

[13] http://www.streambase.com/

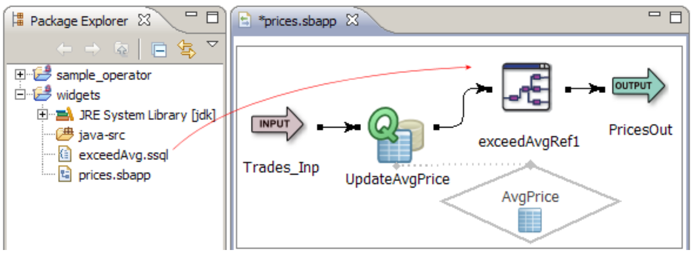


Figure 2.5 Visual Modeling of Event Processing Networks in StreamBase Studio [*StreamBase Visual Editor* 2016]

jumped from 5000 tweets per second to over 140,000 tweets per second within an hour due to a public event in Japan[14].

However, EPSs supporting the full set of EPAs (from the category sometimes called *Complex Event Processing Systems*) are often implemented as single-host or multi-host systems. In contrast to most distributed event processing systems, where streams can be repartitioned at run-time in order to adapt to increased loads, single-host systems perform the whole processing in a single processing node limiting the overall throughput of those systems. In contrast, multi-host systems are able to distribute the work load to different processing nodes, but require manual effort (e.g., re-compiling) to modify nodes in the computing cluster. This is due to the fact that more complex event processing networks often make strong use of time windows and event-at-a-time operators that are hard to distribute due to extensive state management requirements [Andrade et al. 2014].

Another important run-time issue of event processing systems is fault-tolerance which refers to the capability of a system to recover from failures at run-time. Especially in highly distributed systems, failures can frequently happen due to hardware problems such as network connectivity issues or unavailable processing nodes or software problems such as unexpected crashes. Therefore, fault-tolerance must be achieved throughout the whole system by monitoring the nodes of an event processing network and by providing strategies to replay data through the data flow graph in case any incomplete processing has occurred. Recently, architectures such as the Kappa Architecture [Erb and Kargl 2015] have been proposed which describes a system comprised of Kafka [Kreps et al. 2011] as a highly distributed, fault-tolerant message broker and Apache Flink [15] providing a scalable, fault-tolerant event processing system. For instance, Flink uses a checkpointing mechanism based on distributed snapshots to establish fault-tolerance by still maintaining exactly-once processing guarantees.

[14] https://blog.twitter.com/2013/new-tweets-per-second-record-and-how
[15] http://flink.apache.org

3 Motivation

In this chapter, we motivate our research by presenting two scenarios from the Industrial Internet of Things (IIOT) domain. These scenarios are used to define basic needs having influence on event-driven application design. Furthermore, we analyze problems these applications face today that are related to the main research questions this thesis targets.

3.1 Industrial Internet of Things

In general, the Internet of Things (IoT) relies on the idea of a world-wide network where uniquely addressable objects communicate with each other based on standard communication protocols [Infso 2008]. Therefore, objects in this sense can represent anything which is able to communicate with other objects, ranging from hardware like sensors, actuators or mobile phones to software systems like enterprise applications. The main vision of the IoT is to embed things which have previously operated independently from the outside world into a common layer which allows for communication between things itself and autonomous decisions [Atzori et al. 2010]. Some possible IoT-related use cases include the *Smart Factory*, which focuses on connecting machines, devices and sensors in production facilities or *Smart Supply Chains* focusing on the integration of supply chain networks.
The application of the IoT paradigm to industrial use cases is often summarized as the *Industrial Internet of Things*. The IIoT vision is closely related to the term *Industrie 4.0* [Kagermann et al. 2008] often used in European countries as a synonym for the fourth industrial revolution. One of the main assumptions of the IIoT is the digitalization of industrial processes. While today's production systems often already employ a high degree of automation, they are often operated in silos where data is kept inside specialized systems. However, by using standardized communication protocols, this data can be used to foster *intelligent production systems and processes* which are able to perceive information, derive findings from it and to change the behavior of machines and plants [Kagermann et al. 2008].

From an event processing perspective, IoT applications in general are challenging application scenarios as they demand for the main pillars event processing is built upon as discussed in chapter two. Therefore, event processing systems can be seen as a critical success factor to further implementing the vision of an IIOT.
In the following sections, we present two IIOT scenarios we will use as examples throughout the thesis in order to motivate our research contributions.

3.1.1 Digital Factory

Our first use case is related to the *Digital Factory*. We consider a company providing services for the oil drilling business. Oil drilling is a very challenging business that requires companies to continuously monitor drilling equipment in order to be able to quickly react on potential problems which occur during an ongoing drilling operation. For this reason, many safety- and operational-critical devices are already equipped with sensors which continuously measure a number of parameters in real-time.
In our scenario, we envision a drilling machine which produces events from two sensors: The first one measures the swivel oil temperature, while the second one measures swivel vibration. An example event schema is illustrated in figure 3.1.

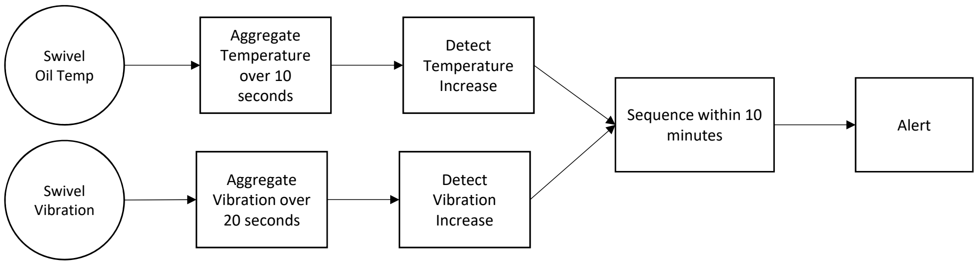

Swivel Oil Temp
Aggregate Temperature over 10 seconds
Detect Temperature Increase
Swivel Vibration
Aggregate Vibration over 20 seconds
Detect Vibration Increase
Sequence within 10 minutes
Alert

Figure 3.1 Example EPN: Digital Factory

Now we consider the following situation that might be relevant to detect a potential failure in a drilling machine: If a significant temperature increase is being detect in the swivel oil temperature sensor followed by an increase in the vibration of the swivel's vibration sensor within a certain time period, this might be an indication for a sudden failure of its function.
Such a situation is a pretty basic use case for CEP systems. A simple event processing network which would be able to detect such a situation is illustrated in 3.1. In this example, we first use an Aggregation EPA to aggregate temperature and vibration events by computing the average within a small time window. Afterwards, we detect abnormal temperature and vibration values by computing the relative increase of

these parameters. Finally, a Pattern Detect EPA is applied on both streams to detect a sequence of both increase events within a specified time.
We will use this example throughout the thesis, but it is important to stress the fact that such situations might change very often depending on additional sensors that are installed in existing machines, new machines that are deployed at different oil fields, changed requirements from the business perspective or other circumstances which require operational staff to deploy and modify such situations frequently.

3.1.2 Connected Logistics

Our second use case is based in the logistics and transportation domain. In this application area, the basic needs for event processing originates from the fact that production companies today rely on just-in-time delivery of production goods due to minimized local stocks. However, it is often the case that goods are delivered late which can be caused by a number of exceptions during ongoing transportation processes. For instance, heavy traffic may cause delays to transportation vehicles, third party manufacturers of ordered goods might not be ready to load products on time, goods could get damaged in the course of delivery. For these reasons, continuous monitoring and detection of potential threats during the execution of transportation processes has become mandatory.
In one of our projects we have developed a flexible solution to enhance transparency and disruption management capabilities to inbound transport processes of automotive companies. In this solution, drivers use a mobile application which continuously gathers both human-generated information (e.g., acknowledgments that specific products have been loaded onto the vehicle) as well as sensor readings such as location or acceleration. This information is then forwarded to a component performing event processing in order to detect (potential) disruptions and to generate recommended actions to mitigate the impact of problems.
One example pattern in this case is the computation of the average time a driver spends at the premises of a supplier. Long waiting times in front of a loading ramp at a specific supplier might be a trigger for delays in the ongoing transportation process, but could also easily violate an existing Service Level Agreement (SLA) between supplier and receiver. Such a performance indicator can already be computed in a continuous manner with standard event processing tooling and would result in an event processing network similar to figure 3.2. In this example, we span a geofence around the supplier’s premises and define an EPA which detects if a vehicle enters the defined geographical area. In the same way, we detect vehicles which leave the specified area. By combining both events and subtracting the timestamps of both events, we can calculate the length of the per-vehicle stay of a transportation vehicle. In addition, we could detect the absence of a departure event within a maximum

time period to immediately initiate countermeasures such as re-planning of other tour stops.

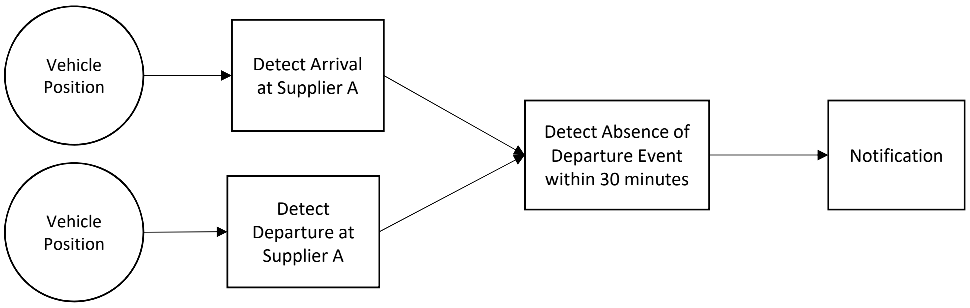

Vehicle Position
Detect Arrival at Supplier A
Vehicle Position
Detect Departure at Supplier A
Detect Absence of Departure Event within 30 minutes
Notification

Figure 3.2 Example EPN: Logistics

In this use case, it is important to stress that even medium-sized companies might easily have a rather large network of suppliers that are involved in daily inbound logistics processes (so-called milk runs). This quickly leads to the need for a large number of very similar event patterns from a structural point of view (i.e., all of them detecting the time of a vehicle spent at the supplier's premises), with the main difference that the background knowledge (in our example the geographical location of the geofences and the allowed duration of a stay) differs between the pattern definitions.

3.2 Needs

Although both scenarios have slightly different characteristics, we can observe some similar needs of today's existing or emerging requirements for event-driven applications from a business-level perspective. In this section, we outline needs related to these use cases by means of the development process and execution of event processing networks. Afterwards, we identify the gap between these requirements and existing solutions in order to formulate a problem statement.
In general, it is worthwhile to categorize the purpose of real-time IIOT applications which demand for the application of event processing techniques:

- **Integrated Monitoring**. Although many devices (e.g., machines) in IIOT applications provide interfaces to monitor a system's current state, integrated solutions which deliver a single user interface in form of real-time cockpits are still rare in real production settings. Integrated monitoring solutions need to combine machine data coming from a large variety of machines with other data such as production plans in order to detect not only failures in a single system, but by also observing *interdependencies* between different heterogeneous systems affecting potential production failures. Therefore, integrated monitoring solutions are a typical use case for event-driven applications as they must reflect the current state of the whole production with little delay. Such systems also need to be configurable so that experts are able to generate on-demand insights depending on different contexts (e.g., produced products or shift changes) and different granularity levels.

- **Continuous Data Harmonization**. With a rising number of sensors that are used to monitor production processes, data harmonization becomes an important problem. Data harmonization refers to continuous transformation of events from multiple input sources to a another output source. For instance, sensor data should often be transferred in a continuous manner to external systems in order to construct data bases for off-line analyses (e.g., to transfer data to a distributed file system such as Hadoop File System (HDFS)[1]) or search engines such as Elasticsearch[2]). Such an approach is similar to existing Extract, Transform, Load (ETL) processes known from data warehouses [Vassiliadis 2009], but transforms data directly once it is generated. An important problem in this context is *data drift* which describes changes in sensor data based on syntax (e.g., structural changes in the data produced by machines such as additional parameters) or semantics (e.g., change of measurement units). Therefore, systems supporting continuous data harmonization need to be flexible in order to detect data drift and react quickly by adding and modifying input sources, transformations and data sinks.
- **Situational Awareness**. Besides monitoring, *situational awareness* is an important point in order to not only observe the state of ongoing production processes, but also to immediately identify situations and potential failures immediately. Situational awareness refers to an understanding of what is currently going on in a specific subject of interest and sets the basis for decision making. Being aware of situations often requires for more complex transformation processes such as on-line analytics and correlations between different sensors in contrast to pure monitoring solutions, for instance, in order to enable condition-based maintenance scenarios. In addition, situations to be detected might change quickly and continuously due to the availability of new sensors, new algorithms or changed business requirements. An example from the manufacturing domain is the integration of pattern detection algorithms in order to correlate sequences of events (e.g., from oil temperature and current flow rate) in order to get early indications on potentially harmful situations.

[1] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[2] https://www.elastic.co/

It is noteworthy that all these categories highlight the need for frequent changes of event processing networks, which strengthens the requirement for techniques which abstract from hard-coded, rigid applications to more high-level languages.

3.2.1 Access for Application Specialists

By looking at these three general application scenarios presented above, it becomes clear that event processing networks have to be able to adapt quickly, to support a large variety of protocols and data formats and to be able to be extended in terms of new input sources or transformation tasks. As such changes might occur unexpectedly as new requirements arise, the time needed from requirements specification to the execution of the underlying event processing network at run-time needs to be minimized. Therefore, tool support is mandatory which reduces time for creating and modifying event processing networks.
This raises the question which user roles should create the application logic. As an Event Processing Network (EPN) typically represents some form of a business need, it has early been recognized that the application logic can be best defined by business analysts. Surveys have shown that more than 80% of companies implementing real-time applications expect business analysts or specialists to write the application logic, whereas only 16 percent would assign such a task to software developers [Etzion and Halle 2013]. Similar observations have previously also been made in related areas such as business rules systems, where lots of research has been done in order to enable business users to create rules [Witt 2012].
For distributed (I)IoT applications, this issue has become even more important due to a rather large number of sensors originating from potentially many different sources and therefore large *pattern bases* which need to be managed. Access for non-programmers in these scenarios imposes not only the need to abstract from existing graphical editors as they still rely on a programming-oriented modeling style, but also ways to provide support for *managing* pattern bases, e.g., by reducing time and effort needed to modify existing patterns [Sen 2013].
Therefore, leveraging non-programmers to write event processing logic ad-hoc without necessary developer consultancy is an essential requirement in order to support real-time IIOT applications.

3.2.2 Distributed Event Marketplaces

A second need is related to the specific characteristics of IIOT applications. In general, one major characteristic of IoT applications is a high degree of geographically distributed sensors, processing logic and actors which communicate over standard web protocols. In this context, the vision of event marketplaces has been proposed (albeit with the assumption of a central processing logic as part of the marketplace) [Stühmer

et al. 2012a]. Such marketplaces aim at developing elastic and reliable architectures for complex event-driven interaction in highly distributed and heterogeneous service systems. From a more practical point of view, the overall goal of event marketplaces are platforms which serve as an endpoint for heterogeneous event-producing sensors and applications, vendor-independent (real-time) processing services for analytics or situation detection and event-consuming applications. This is illustrated in figure 3.3. Suppliers of input data, data transformations and consumer logic offer processing capabilities to a central registry in the marketplace. Users can express their interest in form of situations or data harmonization rules by defining processing pipelines. Such pipelines are then deployed as distributed event processing networks with the marketplace offering non-functional services such as pricing mechanisms and security and acting as a mediator between providers and consumers of processing services.
As an example, a logistics service provider could be interested in a processing pipeline which correlates vehicle positions, weather and traffic data in order to detect incidents leading to delayed deliveries. In this case, the corresponding pipeline would consume traffic and weather data from a third-party provider and vehicle positions directly from internal systems which receive GPS coordinate from the vehicles. Another marketplace participant could offer a real-time processing service to continuously calculate the estimated time of arrival based on position, weather and traffic data. This is an example for a generic real-time service which can be bound to any input stream matching the input criteria of the service provider.
In IIOT applications, such marketplaces allow for the development of distributed event processing pipelines by providing mechanisms such as unified authoring interfaces that integrate decentralized processing logic and provide application specialists with opportunities to define applications in a self-service manner without the need for developer consultancy.

3.3 Problem Statement

The two needs gathered in the previous section are now used to discuss problems of existing solutions. First, we outline and analyze the current development process of event processing applications. It is shown that non-programmer access to developing such kind of applications is prevented by a rather technical abstraction of event processing logic, and that this assumption is valid for declarative languages, high-level developer APIs and even graphical modeling interfaces.
Afterwards, we analyze current problems in terms of the applicability of existing solutions for pipeline authoring and execution in more geographically distributed settings.

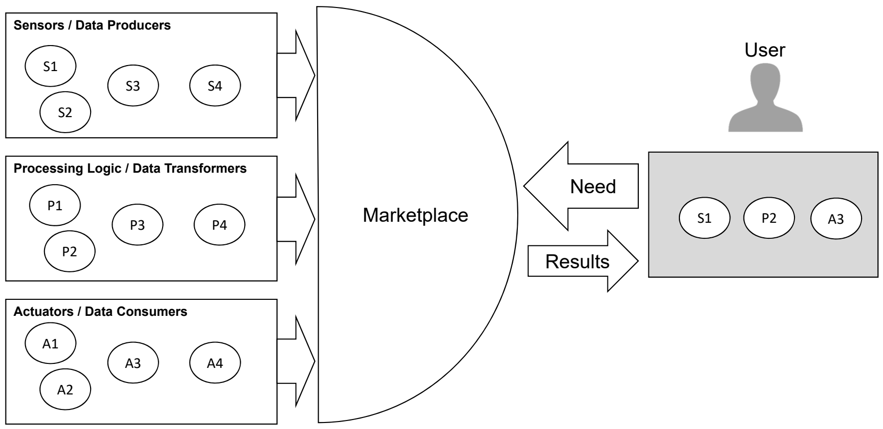


Figure 3.3 Event Marketplace

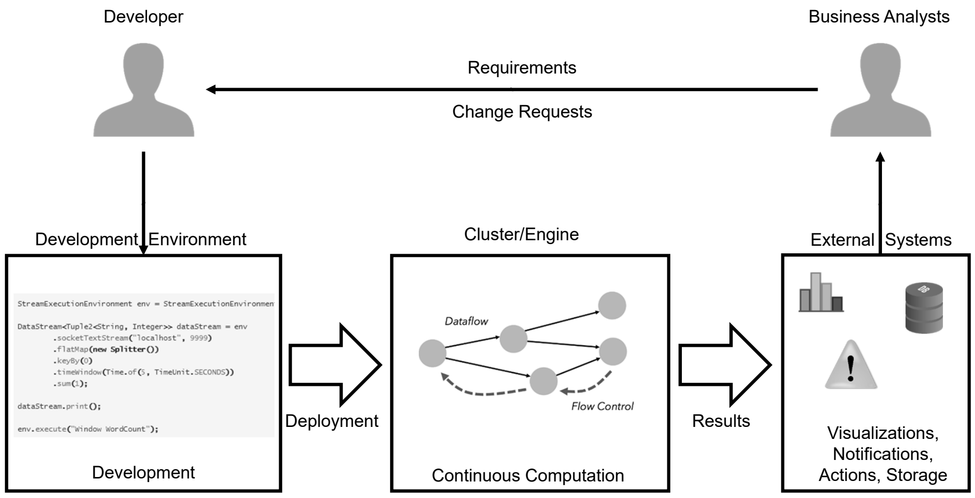
3.3.1 Development Process

A typical development process of event processing applications is illustrated in figure 3.4. Such a process consists of at least two roles, business analysts and software developers[3].

Business analysts initially specify requirements for event processing systems. Usually, such requirements are driven by the need to get informed about a situation of interest and can usually be assigned to one of the following categories:

- **KPI tracking.** A Key Performance Indicator (KPI) is a metric which quantifies "the current efficiency and effectiveness of past actions through acquisition, collation, sorting, analysis, interpretation and dissemination of appropriate data" [Neely 2002]. In this sense, a KPI just measures a defined metric (e.g., the current scrap rate of a machine). Event processing use cases usually focus on rather short-term KPIs in order to reflect the current state of an object of interest.
- **Goal fulfillment.** Goals or targets in business contexts are referred to an objective which is defined by a strategy and pursued during subsequent business operations [Jiang et al. 2011]. Goals are assigned to a specific KPI and specify whether the KPI currently fulfills a defined threshold value. In event processing, applications tracking goal fulfillments often trigger notifications once a goal is satisfied or missed.
- **Situation detection.** According to the Business Intelligence Model (BIM), a situation is defined as a partial state of the world [Barone et al. 2010]. In business

[3] Although there might be easily more roles involved in the development of event processing applications such as testing personnel similarly to standard software engineering processes, we focus on a minimal set of roles in order to foreground the arguments made.



```
StreamExecutionEnvironment env = StreamExecutionEnvironmen

DataStream<Tuple2<String, Integer>> dataStream = env
        .socketTextStream("localhost", 9999)
        .flatMap(new Splitter())
        .keyBy(0)
        .timeWindow(Time.of(5, TimeUnit.SECONDS))
        .sum(1);

dataStream.print();

env.execute("Window WordCount");
```

Figure 3.4 Development Process of EP Applications

contexts, situations often refer to an opportunity the company aims for or a threat which should be avoided. Situation detection is a frequent use case for event processing applications, as these often result in immediate actions.

- **Data preparation.** Data preparation tasks serve to provide business analysts with data which is further being used for off-line analyses or other analytics purposes. In this case, event processing applications are used in order to harmonize data streams, e.g., by filtering out events that are irrelevant for the business analyst.

Such requirements are subsequently given to developers in order to implement the application as an event processing network. Typically, this process includes the following steps:

- **Development of input adapters**. The system which produces the required input data needs to be connected to the event processing application. If data is not yet published to a messaging middleware, input adapters must be explicitly written in order to implement push-style event notifications.
- **Event schema definition**. Afterwards, the input event schema needs to be defined. A minimal schema definition contains information about event property names as well as their (primitive) types. Depending on the specific implementation, some event processing engines support definition of schemes using the provided declarative language. In other cases, schema definitions need to be done in a programmatic way.

- **Development of the event processing logic**. The main task is the development of the specific run-time logic. If the system provides a declarative language to define the event processing logic, an event pattern can be written and directly deployed to the engine. In other cases, the logic needs to run through the whole software development cycle, i.e., testing, packaging and deployment.
- **Development of output adapters**. Finally, output adapters must be written according to the desired action. This includes connection to storage systems, dashboards or notification modules which are either developed manually or make use of existing application programming interfaces or web services.

Once the event processing logic has been deployed in the engine, results are continuously computed until it is manually stopped. Events are sent to external systems where business analysts are able to consume them as requested initially. As the development process as described here is in general manual and developer-focused, the whole life cycle must be re-initiated whenever change requests or new requirements arise.

We will now take a closer look at the specific technical implementation of event processing applications based on various solutions currently available systems offer.

Event Pattern Languages

First, we analyse the definition of an event processing application using a declarative language. Listing 3.1 shows an example SQL-based language written in the Esper Event Processing Language (EPL)[4]. Similar languages are available for other tools, e.g., Apama, Oracle CEP or WSO2. The listing shows a rather simple application consisting of two event statements, whereas the first statement calculates a moving average of oil temperature values over a time window of 10 seconds per machine. The second statement receives this stream as an input and detects an increase of the oil temperature by 50 per cent within 1 minute.

In these languages, streams are defined in a similar way to relations in databases. Our example defines two complex events by using an *insert into* directive. The payload of a derived event is defined by the *select* keyword of the language, and an input stream is defined using the *from* keyword. An input stream can be partitioned by adding a *group by* clause. Data windows can be directly applied on an input stream by providing the window type and its length. The second statement shows a sequence pattern, which defines a sequence of one event occurrence, followed by a time interval of 1 minute and another event occurrence of the same type.

Although such event patterns are conceptually easy to define, there are two drawbacks in terms of usability and expressivity.

In terms of usability, it is worthwhile to note that, due to an increased complexity of such languages compared to database query languages, technical skills are required

[4] http://www.espertech.com

Listing 3.1 Example Esper Pattern to Detect Oil Temperature Increase

```
insert into AggTemperature
  select machineId, avg(temperature) from SwivelOilTemperature.win:time(10
       seconds)
  group by machineId

insert into OilTempIncrease
  select a1.temperature, a2.temperature from pattern
    [every (a1=AggTemperature -> timer:interval(1 minute) -> a2=AggTemperature)
         ]
   where
    a1.machineId=a2.machineId and
    a2.temperature > a1.temperature*1.5
```

and training is needed. Implementation skills are also required in order to connect results of event patterns to data sinks. Moreover, as shown in listing 3.1, resulting event patterns are hardly reusable, e.g., when applying the same business logic to a different input stream. For instance, the business logic defined in our example implements a rather generic task, detection of an increase of a numerical value within a time period, which could be applied to other input streams requiring for similar processing. However, as the implementation is bound specifically to an event type and its properties, the transfer of event processing business logic to support different situations and input event streams is hard to achieve.
Concerning expressivity, use cases often require more complex analytics-oriented event processing tasks (e.g., on-line recognition of drilling activities) that cannot be expressed with standard event processing operators. Although most event processing languages can be extended with custom application logic, it further slows down development processes.

High-Level Programming APIs

Recent frameworks for distributed, scalable event processing such as Apache Flink or Apache Spark Streaming[5] provide high-level application programming interfaces to define event processing logic. This software must be written by developers, packaged, tested and deployed into a cluster. Listing 3.2 illustrates the same use case as described above using Flink's *DataStream API*. The API provides a powerful abstraction from the internal complexities of Flink's runtime (implemented as a distributed streaming dataflow) by omitting details about scheduling and distribution of processing nodes, and also provides high-level access to event-at-a-time operators. However, writing such code requires for extensive knowledge of a programming language as well as

[5] http://spark.apache.org

Listing 3.2 Development of an event processing application in Apache Flink

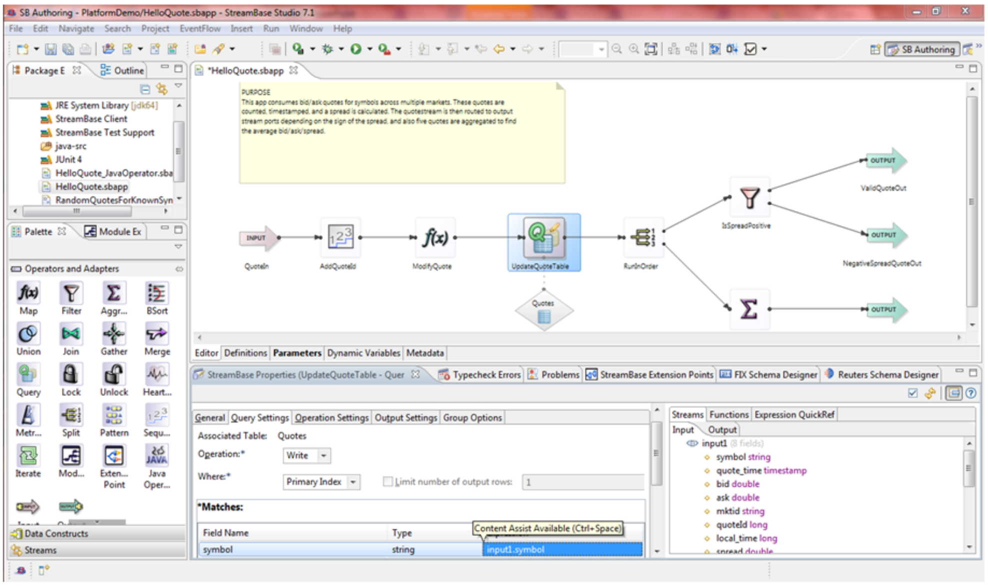
```
DataStream<Tuple2<String, Integer>> stream = inputStream
.keyBy("machineId"),
.window(TumblingTimeWindows.of(Time.of(10,  TimeUnit.Seconds)))
.avg(1);

DataStream<Tuple2<String, Integer>> stream = oilTemp
.windowAll(TumblingTimeWindows.of(Time.of(1,  TimeUnit.MINUTES))
.apply(new AllWindowFunction<Tuple2<String, Integer>, Tuple2<Integer, Integer>,
     Window() {
public void apply(Window window, Iterable<Tuple2<String, Integer>> values,
    Collector<Integer, Integer> out) {
// Custom application logic
}
}
```

knowledge of the specific APIs making development of event processing applications for scalable system hard to achieve for business analysts.
It is worth mentioning that, although this approach seems to be a step backwards compared to declarative languages as introduced above, scalable systems for event processing have a higher complexity compared to single-host systems and therefore often require for additional configuration (e.g., proper stream partitioning). Nevertheless, declarative languages suitable for scalable event processing engines are likely to be introduced in the near future.

Graphical Modeling Tools

In order to improve accessibility of event processing systems in terms of faster development cycles, graphical modeling tools have been proposed. From commercial systems, StreamBase represents one of the first event processing systems providing a graphical user interface to model event processing networks.
StreamBase (as shown in figure 3.5) provides a Graphical User Interface (GUI) to develop an event processing application using a blocks-and-arrows style, whereas blocks represent sources, EPAs and sinks and arrows define the flow of events through the network. Although resulting graphs in StreamBase are intuitive to read, definition of these graphs remains a technical task targeted at developers.
First, while input adapters can be chosen from pre-defined libraries leveraging access to event streams originating from frequently used event sources such as message brokers, software systems or file systems, event schemas must be defined by providing data types and exact field names. Additionally, provided event processing operators are low-level, e.g., aggregation or pattern detect operators. This requires general understanding of event-driven systems.


SB Authoring - PlatformDemo/HelloQuote.sbapp - StreamBase Studio 7.1
File Edit Navigate Search Project EventFlow Insert Run Window Help
*HelloQuote.sbapp
QuoteIn
AddQuoteId
ModifyQuote
UpdateQuoteTable
RunInOrder
IsSpreadPositive
ValidQuoteOut
NegativeSpreadQuoteOut
Quotes
Map Filter Aggr... BSort Union Join Gather Merge Query Lock Unlock Heart... Metr... Split Pattern Sequ... Iterate Mod... Exten... Point Java Oper...
Editor Definitions Parameters Dynamic Variables Metadata
StreamBase Properties (UpdateQuoteTable - Quer
Typecheck Errors Problems StreamBase Extension Points FIX Schema Designer Reuters Schema Designer
General Query Settings Operation Settings Output Settings Group Options
Streams Functions Expression QuickRef
Input Output
Content Assist Available (Ctrl+Space)

PURPOSE
This app consumes bid/ask quotes for symbols across multiple markets. These quotes are counted, timestamped, and a spread is calculated. The quotestream is then routed to output stream ports depending on the sign of the spread, and also five quotes are aggregated to find the average bid/ask/spread.

Associated Table: Quotes
Operation:* Write
Where:* Primary Index
Limit number of output rows: 1

*Matches:

Field Name	Type	
symbol	string	input1.symbol

input1 (8 fields)
- symbol string
- quote_time timestamp
- bid double
- ask double
- mktid string
- quoteId long
- local_time long
- spread double

Figure 3.5 Visual Modeling of Event Processing Applications with StreamBase [*StreamBase Studio* 2016]

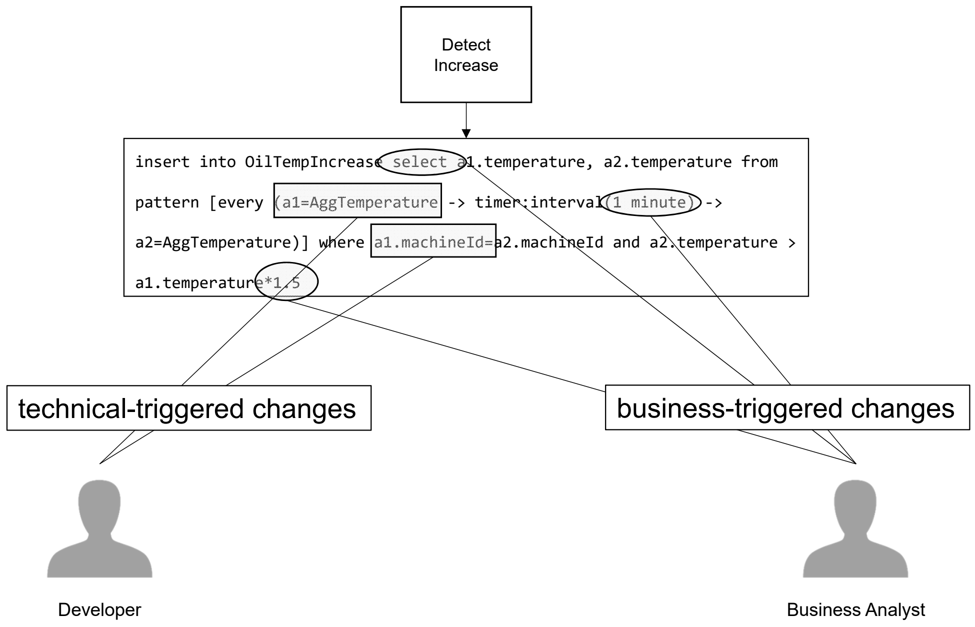
Finally, tools like StreamBase are often implemented as extensions of integrated development environments such as Eclipse[6] and therefore require at least knowledge of standard development life cycles and tooling (such as debugging, testing and packaging). Consequently, the goal of StreamBase is more to reduce development effort of event processing applications from the software development perspective.
We show in chapter 4 that existing efforts from both industry and academia to provide better accessibility for the definition of event processing networks can be compared to solutions like StreamBase. In summary, although existing graphical authoring tools are able to reduce development efforts with better tool support, these solutions are not capable of shifting definition of event processing networks from developer roles to business-oriented roles.

3.3.2 Mixture of event processing logic and knowledge

A second problem is related to mixture of event processing logic and background knowledge. In general, an event processing application consists of some form of business logic (defining *what* should be done) and its parameterization with specific

[6] http://www.eclipse.org

parameters coming from business requirements. An example related to the aforementioned use case is given in figure 3.6. Technical-oriented parts concern the definition of schema-related aspects such as input streams and property definitions the logic should be applied upon. Additionally, the pattern also includes domain knowledge describing the length of time windows, the specific output needed and the increase itself.



```
insert into OilTempIncrease select a1.temperature, a2.temperature from
pattern [every (a1=AggTemperature -> timer:interval(1 minute) ->
a2=AggTemperature)] where a1.machineId=a2.machineId and a2.temperature >
a1.temperature*1.5
```

Figure 3.6 Event Pattern: Background Knowledge

Mixing both domain knowledge and technical specification in a single pattern representation leads to problems concerned with modifications. In case of change requests, even if the logic of a pattern remains unchanged, it needs to be modified once business requirements change. Such behavior could be avoided if a knowledge base is kept apart from the technical specification allowing business analysts to change business-related parts of the pattern.
Admittedly, besides complete recompiling which would be necessary in the case of most scalable event processing systems, knowledge can also be separated into external configuration files. However, allowing business analysts to adapt such files results in custom-built instead of generic solutions.

Altogether, event processing applications are hard to maintain and lack re-usability due to their languages which mix domain knowledge and technical details, and they are bound to specific event streams.

3.3.3 Heterogeneity and Interoperability

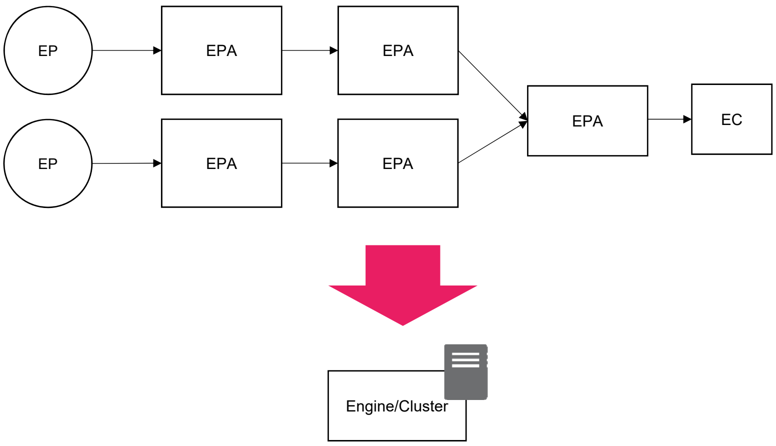


Figure 3.7 Event Processing Network implemented in a single system

A possible solution to the problems mentioned above would be the design of a high-level language which enables non-programmers to write event processing applications, e.g., a graphical editor abstracting from technical details. This is a suitable approach for use cases as illustrated in figure 3.7, where an EPN is deployed in a single engine or a scalable event processing system. The advantage of such a solution is that such a language is only required to cover the features supported by the system itself.
However, we explicitly target geographically distributed environments suitable for IoT use cases, for instance, to support aforementioned marketplace scenarios. This results in architectures as highlighted in figure 3.8.
First, geographical distribution of the EPN requires for the integration of independent event processing agents communicating over web protocols. Second, EPAs within an EPN can be implemented using heterogeneous implementations such as scalable event processing systems, single-host engines or custom algorithms. Third, communication between nodes is not standardized: Multiple protocols (e.g., MQTT[7], AMQP[8],

[7] http://mqtt.org/
[8] https://www.amqp.org/

Websocket[9]) might be supported within a single EPN , but also support for data formats may vary between the individual processing elements. The to-be situation is illustrated in the lower-right corner: Authoring systems supporting non-programmer access to the definition of event processing applications should support execution of event processing networks in distributed environments by abstracting from the technical heterogeneity mentioned above.

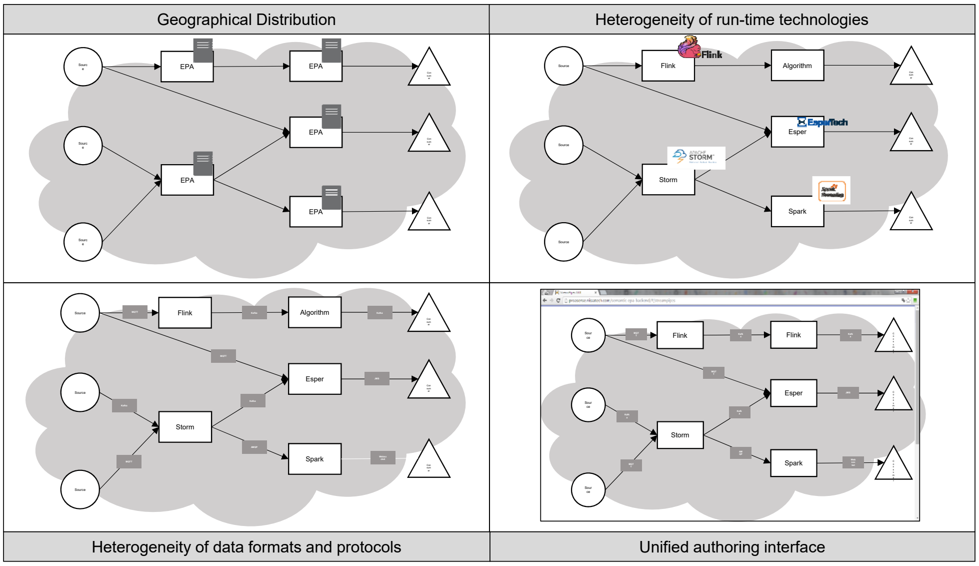

Geographical Distribution
Heterogeneity of run-time technologies
EPA
Flink
Algorithm
Esper
Storm
Spark
Source
Heterogeneity of data formats and protocols
Unified authoring interface

Figure 3.8 Challenge: Authoring of geographically distributed EPNs

3.3.4 Conclusion

In this chapter, the motivation behind our research has been presented. We defined two use cases taken from the Industrial Internet of Things domain. Based on the use cases, we identified business needs requiring for technical support. Furthermore, based on these needs, we analyzed currently available systems and approaches in order to provide a problem statement. Finally, we identified non-programmer access to event processing systems in terms of both usability and adaptivity and knowledge-agnostic pattern representations under the constraint to support definition of geographically distributed system as the main research problems targeted within the course of this thesis.

[9] https://www.w3.org/TR/2009/WD-websockets-20091222/

4 Related Work

This chapter analyzes relevant state of the art in the area of event processing. In addition, related work from other domains tackling similar problems is investigated and compared to specific requirements for event processing applications.

4.1 Related work on Event Processing

This section encompasses related work in the area of event processing. Our focus in on research on development methodologies, i.e. tackling the whole development life cycle of event processing applications and approaches focusing modeling support for Event Processing System (EPS). Furthermore, we discuss work related to interoperability issues and semantic approaches to event processing. In general, accessibility for non-programmers to EP could be improved by more advanced support for expert-driven definition of EP applications in a top-down manner, or by applying bottom-up approaches such as learning of event processing networks as proposed recently in [Margara et al. 2014]. Although self-learning approaches can be useful in order to assist users in defining event processing applications (e.g., to detect unexpected situations), this thesis focuses on expert-driven definition as many of the use cases presented in section 3.3.1 require business analysts to know in advance the specific purpose of an event processing application.

4.1.1 Development methodologies

An approach for user-oriented rule management has been proposed in [Obweger et al. 2011a; Obweger et al. 2011b]. Aiming at providing business users to create event processing applications, they distinguish between two types of rules. Infrastructural rules are defined by system operators and serve as input for higher-level sense-and-response-rules created by business operators. Pattern definitions are provided by defining a set of input parameters, output parameters and a decision graph which describes the event processing logic itself. Rules are deployed as executor nodes in the commercial platform *UC4 Decision*. Although we make use of a similar approach

by dividing technical-oriented tasks from business logic-oriented tasks, we differentiate from this approach mainly by the usage of semantic models to describe event processing blocks and an abstraction from a vendor-specific event processing logic.
[Sen and Stojanovic 2010; Sen 2013] introduced GRuVe, a methodology supporting the management of event patterns. The authors propose a life cycle covering generation, execution and evolution of event patterns and provide an RDFS-based model to describe event patterns. In addition, an authoring tool for graphical pattern definition is introduced, however, it is restricted to the event processing operations filter and pattern detection as well as support for time windows. Compared to our approach, we target a higher-level abstraction for non-programmers, while the engine-specific evolution approaches (c.f., white box event processing) can be re-used in our methodology.
A business-oriented methodology for complex event processing has been developed by [Vidačković et al. 2010; Vidačkovič and Weisbecker 2011]. The methodology is based on the Business Motivation Model (BMM) which provides methods to specify business-related KPIs, goals and objectives. Based on these definitions, the authors identify event patterns monitoring these metrics. An executable program suitable for the Esper event processing engine is generated out of manually defined *KPI derivation rules*. The main purpose of this approach is to provide better support for process monitoring. The main difference to our approach is that we target other use cases besides KPI calculation, e.g., continuous ingestion of events to third party systems. In addition, our methodology does not focus on supporting a single event processing language, but to integrate heterogeneous event processing run-time implementations.

4.1.2 Modeling support

Model4CEP [Boubeta-Puig et al. 2015; Boubeta-Puig et al. 2014] is a domain-specific modeling language for event processing. Its goal is to provide a modeling language characterized by high expressiveness and flexibility, independence from specific implementation code and suitability for business users. Model4CEP consists of a *CEP domain meta model* to define events, event properties and their types and a modeling language to define event patterns. The capability of the pattern language comprises unary and binary operators, temporal operators, (filter) conditions and arithmetic operations. These elements can be connected using a graphical editor based on the Eclipse framework. Although the system claims to be extensible, the provided operators implement rather low-level event processing logic requiring users to be familiar with event-driven thinking. In contrast to our approach, Model4CEP focuses on standard CEP operators (especially Pattern Detect EPAs and Filtering), while other EPAs requiring custom processing logic such as enrichment of events or transformations are not the main focus of this work.

SpChains [Bonino and Corno 2012; Bonino and De Russis 2012; Bonino et al. 2013] is a framework supporting modeling of event processing applications independent from a specific run-time implementation. SpChains defines an (extensible) set of *stream processing blocks* which can be connected in order to create *stream processing chains*. Each block parametrizes a specific event processing logic and blocks can be combined using a filter-and-pipe pattern. Blocks consist of input and output ports handling a specific event type and can be instantiated by a provided XML-based language. Although SpChains abstracts from stream processing logic to input/output descriptions of operators similar to our approach, a main difference is that we represent streams and event processing logic using a semantics-based model allowing for more sophisticated matching between multiple processing blocks.
[Karampiperis et al. 2014] present a graphical authoring tool called *Event Recognition Designer Toolkit (ERDT)*. The main design goals of ERDT were simplicity and user-friendliness, support for multiple event processing languages and cross-platform implementation. ERDT is extensible and provides out-of-the-box support for temporal operators and logical operators. Graphically defined rules can be compiled to executable languages like SQL and Event Calculus. The main difference to our work is that ERDT compiles a graphical model into a specific target language, e.g., a graphical pattern definition is executed in a single-host system, whereas our approach aims to integrate heterogeneous run-time implementations by a common description layer on top of the implementation logic.

4.1.3 Interoperability

We discuss related work concerned with interoperability of event processing systems. [Hoßbach et al. 2013] present a middleware for Event Producer (EP) systems which aims at overcoming heterogeneity of EP systems by providing an abstract model based on common EP functionality named Java Event Processing Connectivity (JEPC). JEPC targets Java-based implementations of EP systems and supports operators widely used in event processing such as aggregations, patterns and time windows. EP systems can be connected by implementing *bridges* to specific implementations. As a result, JEPC supports switching of EP systems without the need to modify existing event processing logic. This approach has been further detailed in [Hoßbach 2015]. Drawbacks of this approach are limited support for non-standard event processing operators and the limitation to a specific programming language.
Another approach aiming at increasing interoperability are Event Stream Processing Units (SPUs) as introduced by [Appel et al. 2013]. SPUs also encapsulate event processing logic by using an abstraction mechanism called *Eventlets*. Eventlets, being higher-level containers for event processing logic, consist of meta data which describes preconditions required for an eventlet to listen for. If an event stream matches this filter condition, the SPUs business logic is executed. Eventlets listen for streams pub-

lished through a Java Message Service (JMS)-enabled middleware. In contrast to our approach, once eventlets are implemented and deployed, they listen for events which match the preconditions specified in the eventlet description, but they still rely on a pre-defined implementation. Our methodology aims at instantiating event processing logic based on pipelines which have been defined by non-developers.
Potocnik and Juric [Potocnik and Juric 2014] introduce the concept of complex event-aware services as part of SOA. The authors extend the Web Service Definition Language (WSDL) with CEP-specific elements, however, the usage of WSDL limits the description of Event Processing Agents (EPAs) to technical aspects omitting the semantics of EPAs.
In order to achieve better interoperability, semantics-based approaches to event processing have been developed. For instance, a semantics-based event model is introduced in [Stühmer et al. 2012b; Stühmer 2015], where the authors develop a lightweight model to represent events in RDF. The advantage is that events can be exchanged between multiple consumers using standard web protocols, however, it relies on the usage of RDF as common event format. On the pattern definition side, in [Anicic et al. 2011] EP-SPARQL has been proposed as an extension of SPARQL. EP-SPARQL supports most operators frequently used in event processing and is translated to ELE rules supported by the logic-based event processing engine ETALIS. Similar approaches in this area are C-SPARQL [Barbieri et al. 2009] and SPARQLStream [Calbimonte et al. 2010].
These approaches focus on the extension of event processing systems at *run-time* by providing native RDF processing and often also stream reasoning capabilities. However, although interoperability is improved by making use of standard web languages and data models, these approaches do not differ from existing event processing languages in terms of non-programmer access and, in addition, strictly rely on events represented in RDF. In our work, we do not explicitly expect semantics-based event formats at run-time, but use semantic descriptions at *design-time* to model the characteristics of events including event schemas and their technical representation. This enables us to use more lightweight event formats at run-time which potentially improves event processing performances.

4.1.4 Event processing and background knowledge

Other approaches make use of semantics to assist users at design-time by expanding or replacing event patterns with background knowledge represented in ontologies. Work in this area primarily focuses on separating the business logic of event processing applications from historical as well as static data.
For instance, [Binnewies and Stantic 2011; Binnewies and Stantic 2012] introduced the OECEP (ontology-enhanced complex event processing) framework. OECEP relies on the core concepts *rule rewriting*, *input rewriting* and *output rewriting* in order to separate background knowledge available in an RDF-based knowledge base from the pattern

definition. As an example, queries can be expressed by referring to a concept in the ontology, whereas these concepts are replaced with its instances prior to deployment. A similar concept has also been proposed in our earlier work [Riemer et al. 2012] with the introduction of *semantic requests*, extensions of standard event processing languages that can be used to automatically expand event patterns with ontological background knowledge. However, both approaches are solutions targeting implementation-specific event processing languages and do not consider a generic approach to integrate multiple implementations in a single event processing application. [Teymourian et al. 2012; Teymourian and Paschke 2010] introduced knowledge-based event processing, also tackling the problem of separating knowledge from more technical representations. The authors apply a concept named *Event Query Pre-Processing*, where event patterns are preprocessed, and rewritten according to static data fetched from a knowledge base. Patterns are executed in the rule engine *Prova*.
The main advantage of these methods is that inferencing, which usually implies significance fall-offs in terms of performance (see [Stühmer 2015] for a discussion), are shifted from run-time to event pattern design-time, where performance is much less an issue in applications processing high throughputs of events. However, these approaches are engine-specific solutions that do not tackle the integration of distributed event processing logic which is, for instance, needed for application scenarios such as the event marketplace.

4.2 Related work on Semantic Web Services

Similar problems, integration of heterogeneous and geographically distributed computing resources, as well as modeling issues from a non-programmer perspective, have also been investigated in other domains, especially by the web services community.
Web services are typically defined as functionality of information systems exposed through standard web technologies [Alonso et al. 2004] or, more precisely, "self-contained, modular business applications that have open, Internet-oriented, standards-based interfaces" [Walsh 2002]. Web services therefore abstract from specific implementation details by providing a well-defined public interface which can be invoked by consumers if requested. In this context, the service-oriented architecture (SOA) paradigm has become the architectural approach to composition of web services in order to enable automatic execution of business processes in *workflow management systems*.
In general, approaches targeting semantic web services which rely on request / response-based architectures cannot be directly applied to our approach which focuses on publish/subscribe architectures. However, solutions regarding the description of requirements and capabilities of web services can partially be re-used and extended

in our approach. Therefore, sections 4.2.1 and 4.2.2 present several approaches, while differences to our approach are discussed in section 4.2.3.

4.2.1 Semantic Web Services

One of the first approaches to provide semantic annotations of web services is OWL-S [Martin et al. 2004]. OWL-S describes a service by means of a *Service Profile*, a *Service Model* and a *Service Grounding*. Service profiles specify *what a service does*, e.g., goals a service can accomplish. The service model describes how a services can be used, e.g., by providing information about the structure of valid requests, service invocation and also possible service compositions. Finally, service groundings describe technical details on service invocation, such as specific protocols and message formats.
The Web Service Modeling Ontology (WSMO) is another approach to semantic web services [Feier et al. 2005; Roman et al. 2005]. WSMO specifies an ontology with the top notions *Ontologies* formalizing background knowledge, *Goals* specifying client objectives, *Service Descriptions* describing functional, non-functional and behavioral aspects and Mediators as enabler for interoperability. In contrast to OWL-S, WSMO does not rely on a web standard such as OWL, but implements its own formalism. One of the main purposes of WSMO is the notion of goals which should enable automatic service discovery and invocation. WSMO's reference implementation *WSMX* also includes an editor for specifying WSMO-compliant ontologies and a service registry.
[Gessler et al. 2009] proposed the Simple Semantic Web Architecture and Protocol for Semantic Web Services (SSWAP). The goal of SSWAP is to offer a more lightweight alternative to SWS compared to OWL-S and WSMO. The authors developed a common syntax, shared semantics and a mechanism supporting semantic discovery based on an architecture similar to REST. In SSWAP, participants exchange data via graphs represented in OWL-DL. A service provider publishes a *Resource Description Graph (RDG)* to offer a service capability at a specific URL. A consumer can invoke such a service by sending a HTTP POST request containing a *Resource Invocation Graph (RIG)* to the service's URL. SSWAP's reference implementation also includes a graphical tool to create service pipelines in a drag-and-drop fashion. In our approach, a similar communication model is being used to exchange descriptions of event processing elements by using *description graphs* and exchange details on the specific configuration of event processing logic by using *invocation graphs*.
Multiple web services can be combined in order to form composite applications, so that capabilities of individual web services can be used to achieve more complex goals. A major challenge when combining web services is the matching of individual capabilities of two services in order to determine their compatibility. In this context, several approaches have been discussed in the web services community to discover web services based on *service requests* and *service advertisements*. [Paolucci et al. 2002] present a semantic web service matching approach based on the service description

language DAML-S and propose four different degrees of matching: An *exact* match is given if an advertisement and a request are equivalent, a *plug-in* match indicates that an advertisement subsumes a requirement, a *subsumes* match describes a partial fulfillment of the request and *fail* a non-match between advertisement and request. Our approach also aims to match an offered event stream against a stream requirement provided by an event processing agent or an event consumer based on their semantics, however, in contrast to semantic web service approaches which aim to enable (semi-) automatic web service composition, our approach aims at supporting users while defining processing pipelines out of individual pipeline elements. Therefore, in our approach we seek to find exact matches between pipeline elements based on a set of element characteristics (schema, grounding and quality) targeted at stream processing applications as described in chapter 7.

4.2.2 Linked Services

Due to several reasons, most semantic web service approaches did not gain much popularity. On the one hand, this is due to the fact that also the corresponding web services standards (namely WSDL/SOAP/UDDI) were not broadly adapted due to their high complexity[1]. Most first-generation semantic web services suffered from the same problem together with further increased complexity and inadequate tool support. Therefore, another generation of web services based on RESTful architectures has gained attraction (for a complete survey, see [Verborgh et al. 2014]).
Linked USDL [Pedrinaci et al. 2014], an improved version of the Unified Service Description Language (USDL) [Kona et al. 2009] provides an RDFS-based vocabulary to describe web services. Linked USDL relies on existing vocabularies such as the Minimal Service Model (MSM) [Pedrinaci and Domingue 2010] and the GoodRelations ontology [Hepp 2008] and defines top-level concepts to describe *ServiceOfferings* offering a set of services, *Services* itself and *ServiceModels* as families of services with common characteristics, but also defines concepts for service binding, e.g., *CommunicationChannels*. Tool support for creating and exporting services based on Linked USDL is available.
Hydra [Lanthaler and Gütl 2013] is another approach to lightweight semantic web services. It provides a vocabulary to create and describe Web APIs based on RESTful architectures and JSON-LD as a message interchange format. Due to this focus, the advantage of Hydra is that it re-uses technologies that are already widely used in the web and therefore lowers the barrier for web developers to annotate their existing applications with semantics. Hydra defines web services based on an *ApiDocumentation* which describes, among others, input requirements in form of *supportedClasses* and

[1] For instance, the unofficial web services technology stack known as WS-* comprises more than 50 standards for purposes such as security, reliability, policies, trust and notifications, among others

their required data model as *supportedProperties*. *Operations* the web service supports rely on the basic HTTP operations GET, POST, PUT and DELETE.

4.2.3 Discussion

Although semantic web services are similar to our approach in terms of the objective (providing a higher-level abstraction on the actual implementation to provide better accessibility and ensure interoperability), different underlying architectural paradigms prevent direct re-use of web service approaches due to the following reasons:
First, almost all web services operate on a request/response architecture. In such architectures, requests are sent from a service consumer to a *service provider*, where a result is computed and sent back to the consumer. In contrast, event-driven architectures operate on a publish/subscribe architecture. A producer of events does not know in advance of its consumers, and there could be more than one consumer for a specific event type. As an important difference, a message used for web service invocation includes specifications on the operation that should be executed as well as the input data the web service should consume. In event-driven architectures, events do not provide any commands on its processing need (see the definition of the term event in section 2.2), the routing of the events is defined in the EPN specification itself.
Next, event-driven services rely on different execution model. A workflow management usually creates a new process instance per event, executes services as defined in the process model and destroys the instance once the process has finished. In event processing, a single instance is created at deployment time of the event processing network. The network then listens for events to arrive, processes them according to the EPN specification and runs potentially forever until it is stopped manually. Therefore, event processing applications have a much stronger focus on stateful processing.
Finally, while service-oriented architectures usually perform operations based on the input data to calculate an output, event-driven applications have a focus on the *manipulation* of events, e.g., transforming, enriching or filtering events. This has important implications related to the modeling of the output of an event processing operation, most importantly, the output cannot be defined at design-time as the specific event input is not known at the time the EPN is designed.
However, despite these differences, useful lessons can be learned from approaches to semantic web services. Therefore, we can re-use and adapt ideas from semantic web service approaches such as input requirements and capability definitions.

Part III

Main Part

5 Requirements

This chapter derives requirements as a basis for the design of our approach. Requirements are collected based on the needs gathered from the motivating scenarios in chapter 3 and related work in chapter 4. We start by discussing the requirements elicitation process, in more detail, how requirements have been derived based upon the research questions identified in chapter 1. The following sections present various requirements related to the methodology, the underlying model and the system itself.

5.1 Requirements Elicitation

Requirements elicitation is a process that aims to seek, uncover, acquire and elaborate requirements for computer-based systems [Loucopoulos and Karakostas 1995]. Therefore, collection of requirements implies a process that involves several activities. Such activities typically involve *understanding of the application domain*, *identification of sources of requirements*, *analyzing stakeholders*, *selection of techniques for requirements elicitation* and finally the *elicitation of requirements from stakeholders itself* [Zowghi and Coulin 2005].

Many of these activities have already been presented and discussed in detail within this thesis. Concerning the **application domain**, our work targets application domains which require for frequent changes of event processing logic and therefore need approaches leveraging fast development of Event Processing Systems (EPSs), as described in section 3.2.

Sources of requirements mainly include (potential) stakeholders and relevant documents such as reports, research papers and existing systems. This activity was mainly performed within chapter 4, where we discussed related academic approaches as well as industry systems. Additionally, some of our requirements also originate from our experience in various research projects (as mentioned in section 1.4) and interviews with stakeholders involved in the development and application of event processing systems.

In our work, **analysis of stakeholders** can be best performed based on the intended users of our approach, where we mainly target software developers and business analysts. On the one hand, business analysts need to be supported for fast development

of event processing applications. On the other hand, development of event processing logic which is re-usable and independent from specific implementations requires for methodological and tool support for software developers.
Techniques, approaches and tools provide different ways to gather requirements. As research work generally relies on existing approaches, introspection (i.e., deep analysis of something) played a major role in our requirements elicitation process. Additionally, interviews and a domain analysis of existing real-world systems contributed to requirements collection.
Upon these activities, we can elicit requirements by identifying an initial classification along the identified research questions. Research question 1, dealing with the reduction of the development effort of event processing applications, leads to a class of general requirements relevant for the whole development methodology. Research question 2, dealing with the modeling of event processing logic independent from implementation details, as well as question 3, concerned with the separation of background knowledge from event processing logic, can be assigned to a class of model-related requirements. Finally, the actual execution of distributed event processing pipelines leads to a set of system-related requirements.
Figure 5.1 illustrated the specific categories of requirements related to our identified research questions.

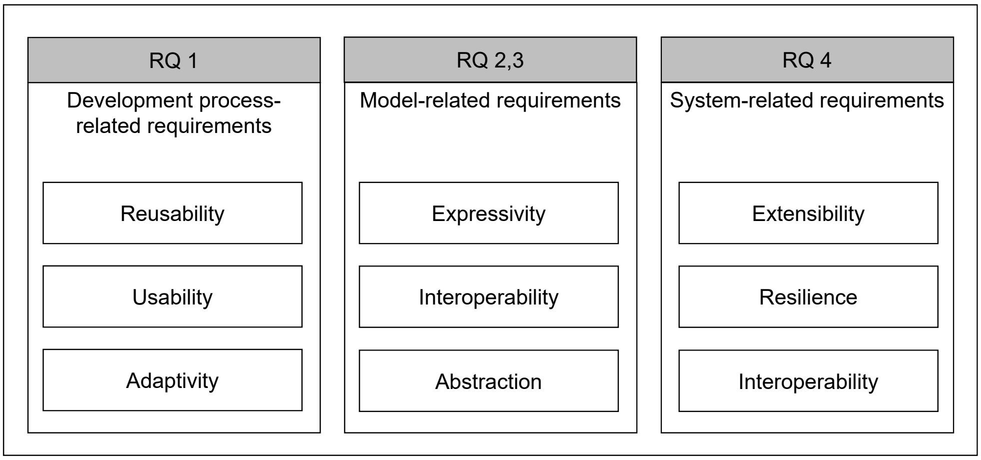


Figure 5.1 Requirements related to research questions

5.2 Development process-related requirements

We start with presenting requirements related to the general development process of event processing applications.
The first requirement deals with re-usability. In section 3.3.1, we have already shown that many pattern languages are hard to maintain, as input and output streams are directly bound to the event pattern description. Therefore, event processing logic should be provided in a more generic way, allowing for more intuitive parameterization of the specific functionality without having to touch the actual application logic. This results in the following requirement:

Requirement R1: Generic event processing logic
Event processing logic is defined in a generic way independent from specific input and output streams.

Besides generic processing logic, use cases often require domain-specific processing of event streams. For instance, in the oil drilling scenario presented in section 3.1.1, specialized algorithms need to be part of the application in order to detect drilling operations in a data-driven way. Such requirements generally restrict a possible solution of having a *fixed* set of generic, reusable components, but stress the need to provide support for the integration of domain-specific event processing logic at any time, resulting in requirement 2:

Requirement R2: Domain-specific event processing logic
Domain-specific event processing logic must be supported.

As described in section 2.3.1, event processing applications are represented by a network connecting event producers, processing agents and consumers. In section 2.3.1, we have shown that several ways (domain-specific languages, high-level APIs and visual editors) exist to build an EPN. As we target business analysts to build event processing applications without developer support, this approach requires for graphical modeling languages which abstract from technical details.

Requirement R3: Graphical Modeling Language
Event processing applications can be defined using a graphical modeling language.

Integration of custom domain-specific event processing in graphical modeling languages implies the need for individual development. For instance, specific algorithms need to be developed programmatically before they can be made available as high-level building blocks in graphical notations. Therefore, tool support for developers is required to speed up the development of event processing logic, leading to requirement 4:

Requirement R4: Developer Support
Developer support to create reusable event processing logic is available.

In section 3.3.2, the need to separate knowledge from the technical specification of event processing applications was motivated. Knowledge implies static data which is needed to instantiate an event processing agent, as well as historic data. This requires tools to assist in the definition of knowledge models which can be re-used in the graphical modeling language.

Requirement R5: Knowledge Modeling
Knowledge can be modeled separated from the event processing logic.

The next requirement deals with the ability of the system to modify existing logic. Once requirements change, users should be able to adapt an EPN without the need to restart components. For instance, sensor data might change over time. In such cases, adaptation of event processing applications should be possible without the need to adapt programming code. This requires for mechanisms to support modifications of existing EPNs.

Requirement R6: Adaptivity of EP applications
Event processing applications can be modified without developer consultancy.

5.3 Model-related requirements

The next category of requirements is related to the underlying model. The purpose of this model is to provide capabilities to describe event processing logic independent from their implementation, and to abstract from technical details. In this way, the model should be expressive enough to cover the targeted use cases and general event processing capabilities and needs to contain all required technical specifications to instantiate event processing logic properly, but at the same time needs to abstract from technical details in terms of accessibility for non-programmers.
The following requirements are therefore concerned with expressivity. In section 2.3.1, we presented a hierarchy of event processing agents from the literature. This model has important implications in terms of the design of an event processing system.

Requirement R7: Event processing agents
The model supports all basic types of event processing agents.

In order to understand event streams, the event schema and its properties need to be modeled. A minimal description of an event property contains its key and its type specification. Additional schema-related properties such as value ranges further increase the understanding of an event schema.

Requirement R8: Event schema
The model supports definition of event schemas and its properties, types and value ranges.

Many event processing applications make use of events originating from sensors. Quality aspects of sensors on both stream level (e.g., frequency and latency) and event level (e.g., accuracy and resolution) are useful metrics to determine whether a specific

Event Processing Agent (EPA) is able to process a specific event. This results in the following requirement:

Requirement R9: Quality of events
The model supports definition of quality aspects related to events and processing agents.

Besides technical specifications of event, static data is often required to instantiate event processing logic. Static data involves parameters in an event processing application such as expected user input. Further examples include links to data stored in 3rd party systems such as production plans and referenced knowledge items related to business-level requirements, such as specific products or places.

Requirement R10: Static data
The model supports definition of static data.

In section 3.3.3, we have highlighted the need for event processing systems that are able to integrate heterogeneous and distributed event processing logic. A model supporting interoperability should be *format-agnostic*, i.e., the model itself should specify the event format used at run-time.

Requirement R11: Run-time representation: Event format
The model is independent from a specific run-time event format.

A similar requirement can be formulated for communication protocols. As many standards within certain application domains exist for transmitting events based on publish/subscribe protocols, the model needs to be agnostic in terms of a specific protocol. Instead, the protocols used at run-time by event streams and those which are supported by processing elements need to be described by the element itself.

Requirement R12: Communication protocol
The model is independent from a specific run-time communication protocol.

The next requirement deals with the representation of the model itself. In order to ensure interoperability, this representation should be based on existing standards. In this context, the Resource Description Framework (RDF) and the schema language RDFS[1] provide a widely-used framework to describe data models based on web technologies. RDF allows to describe a resource in form of statements, where a statement is formulated as a subject-predicate-object expression. Since resources are identified by unique, global identifiers, existing resources in RDF can be easily re-used and extended. As a result, many vocabularies for various domains already exist which allow to re-use knowledge that is represented in RDF.
In addition, the schema language RDFS provides ways to define concepts and data types. In contrast to object-oriented programming languages, concepts described in RDFS are based on the *Open World Assumption (OWA)*, which assumes, that, in contrast to the semantics of common data models, non-defined facts are not automatically considered false. An implication of this assumption is that properties are not

[1] http://www.w3.org/TR/rdf-schema/

assigned to concepts directly, but can generally be applied to any RDFS concept. This circumstance allows for extension and re-use of knowledge represented in RDF at any time.
Due to these reasons, we can benefit from re-using existing vocabularies not only in terms of our model itself, but also use it as a representation to store background knowledge which we aim to isolate from the technical EPN specification in RDF.

Requirement R13: Design-time representation: Model
RDF is used as a data model and to represent knowledge.

Finally, in order to achieve our goal to assist application specialists in creating event processing applications, the model should provide higher-level access to event processing operators. In other words, one major requirement for the models that are to be developed is their ability to be expressive enough to support the development of advanced event processing applications, but at the same time to be able to abstract from technical details. This leads to the following requirement:

Requirement R14: Abstraction
The model should abstract from low-level event processing operators.

5.4 System-related requirements

In this section, we discuss requirements related to the *system*. These requirements mainly include technical aspects which affect the execution of the event processing system.
The first requirement is related to extensibility. In distributed settings such as the IoT, event processing systems must be easily extensible with new data sources, processing logic and sinks. In marketplace scenarios as presented in section 3.2.2, participants may provide new processing capabilities at any time to other participants. Therefore, our system needs to be capable of run-time extensibility.

Requirement R15: Extensibility
Event producers, processing agents and consumers can be added at run-time.

In order to assist non-programmers in creating event processing applications based on graphical modeling, the compatibility of two arbitrary elements which should be connected in an EPN must be evaluated. Therefore, a matching mechanism is needed. A simple matching schema could work on the syntax level. In this case, a processing element would define the requirement for a specific data type an incoming stream should have. However, such mechanisms are not suitable for advanced scenarios, where many different characteristics (e.g., the measurement unit of an event property or the frequency of a stream) influence a potential matching. This leads to the following requirement:

Requirement R16: Matching
The system is able to evaluate the compatibility of two event processing blocks based on their semantics.

At run-time, failures in distributed systems might occur due to several reasons. For instance, sensors might simply fail, data being sent by sensors could change causing the event processing application to stop, or network issues could lead to the absence of events. Therefore, event processing systems need to be able to recognize failures.

Requirement R17: Failures
Failures in processing nodes can be detected.

Besides failure recognition, systems ideally should be able to recover from failures. This requirement involves two aspects: On the run-time layer, event processing systems often have their own mechanisms to ensure fault tolerance. In other cases, where failures do not depend on the system itself, but, for instance, on the hardware of sensors, fault tolerance also involves automatic replacement with components providing the same capability, e.g., backup sensors.

Requirement R18: Fault Tolerance
The system provides capabilities to recover from failures.

The next two requirements are related to the ability of the system to integrate heterogeneous event processing systems. First, event processing applications which need to integrate geographically distributed systems must be supported to cover scenarios such as the event marketplace.
Second, as our approach does not rely on the development of a specific event processing run-time implementation, our system should be independent from the event processing technology itself. Both assumptions result in the following two requirements:

Requirement R19: Distributed Execution
The system can integrate distributed event processing logic.

Requirement R20: Heterogeneous run-times
The system does not depend on a specific event processing run-time system.

6 Methodology

In order to solve the problems introduced in the first part of the thesis, this chapter introduces a methodology which aims at providing a novel way to develop event processing applications. In section 6.1, we introduce basic terms which are frequently being used in the course of this thesis. Section 6.2 briefly presents our approach in form of a walkthrough of this chapter. Afterwards, we introduce a methodology supporting the development of event processing applications in section 6.3. Our approach targets different user roles, which are defined in section 6.4. The individual phases of the methodology are further explained in sections 6.5 and 6.6. Finally, section 6.7 introduces tools we provide in order to support individual tasks of the proposed methodology.

6.1 Terms

In order to facilitate a common understanding of the terms used within the following chapters, we refine the following notions partially already used in this thesis:
Processing Pipeline. A processing pipeline describes an event processing network consisting of a set of event producers, event processing agents and event consumers and a data flow describing the communication between these elements. A processing pipeline can employ heterogeneous underlying run-time technology in a single pipeline. In section 8.2, processing pipelines are defined formally.
Pipeline Element. A pipeline element is either an event producer, an event processing agent or an event consumer.
Processing Element. We use the term processing element to subsume consuming pipeline elements in an event processing network, which includes event processing agents and event consumers.

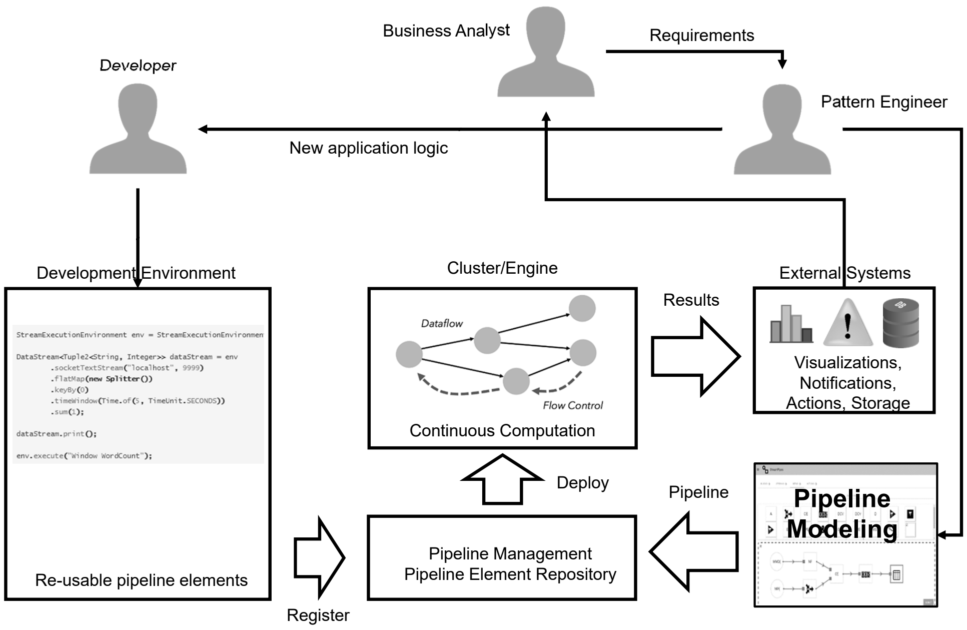
6.2 Walkthrough

Before we dive deeper into details of the methodology itself, we briefly sketch our approach. Figure 6.1 gives a high-level view of the conceptual architecture. This

model extends the development process of event processing applications as sketched in section 3.3.1.
Requirements collected by business analysts are realized by pattern engineers. Our approach aims at enabling pattern engineers to model processing pipelines consisting of event producers, processing agents and event consumers using graphical tool support. Pipelines are created based on a set of *re-usable* pipeline elements. These elements are created by software developers based on requests for new application logic formulated by pattern engineers. Re-usable pipeline elements generally consist of a specific implementation covering execution details of an element, and a semantic description which provides information on how an element can be used.
Once new application logic is required, developers first start with modeling details on a pipeline element. In case of a new requirement for an event producer, a description is generated which includes details on the producer itself and event streams published by the producer. This description contains information on the event objects, their content and technical details on the event format as well as the technical connectivity, e.g., the communication protocol used to transmit event streams. Afterwards, adapters can be implemented for event streams, e.g., connectors which consume data directly from sensors.
In a similar process, processing agents are defined. In contrast to event producers, processing agents are first described by providing information on input stream requirements the component expects in order to ensure proper execution in addition to static data (e.g., required user input) which is needed for the instantiation of the processing logic. Furthermore, developers additionally provide information on how the agent transforms an event stream to a specific output stream. Afterwards processing agents are further customized with specific application logic.
Pipeline elements are registered in a centralized repository, located in a pipeline management component. This repository provides elements that can be used by pattern engineers to create processing pipelines, which can afterwards be deployed in the engine in order to execute the event processing logic. Events are forwarded to external systems in order to support business analysts directly with the output produced by a processing pipeline.

6.3 Methodology: Overview

In general, a methodology is defined as a set of methods, rules, or ideas that are important in a science or art [Merriam-Webster 2009]. Methodologies usually define procedures which aim at solving or assist to solve a specified problem. Especially in software engineering, methodologies pursue the goal to split development processes into smaller parts, such as phases, tasks and tools. The overall objective in this thesis is to enable non-programmers to develop event processing applications.



```
StreamExecutionEnvironment env = StreamExecutionEnvironmen

DataStream<Tuple2<String, Integer>> dataStream = env
        .socketTextStream("localhost", 9999)
        .flatMap(new Splitter())
        .keyBy(0)
        .timeWindow(Time.of(5, TimeUnit.SECONDS))
        .sum(1);

dataStream.print();

env.execute("Window WordCount");
```

Figure 6.1 Conceptual model of our approach

Based on the requirements presented in the previous chapter, we developed a methodology to support this objective, which is illustrated in figure 6.2.
In general, our methodology consists of two phases:
The **setup phase** deals with the development of building blocks which are prerequisites for the definition of real-time applications by focusing on two main processes which constitute the setup phase:
The first process covers the definition of pipeline elements which should be later available for the development of event processing applications. This phase targets technical experts, i.e., software developers. The main tasks during this process are to implement the application logic of processing elements.
The second process during the setup phase is the modeling of knowledge required for specific processing pipelines (which are built in the execution phase). Obviously, this process is targeted at business analysts who are experts in the application domain of interest.
Second, the **execution phase** deals with the definition, execution and maintenance of processing pipelines. Processing pipelines run through a life cycle consisting of

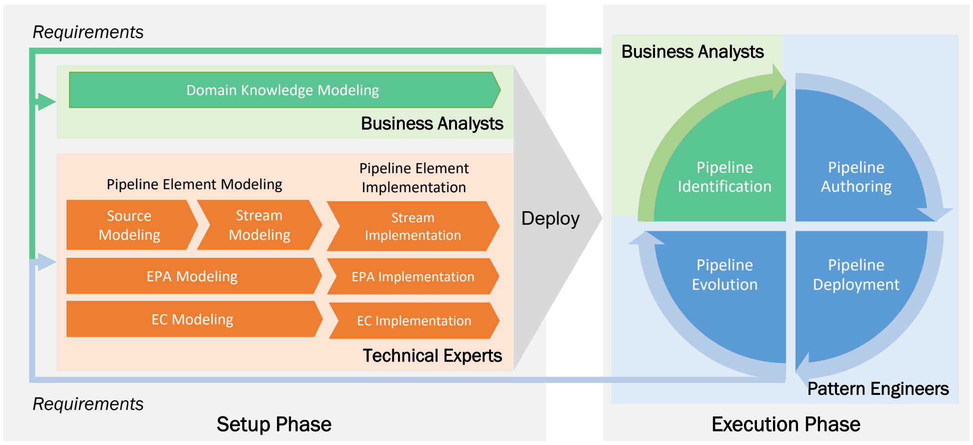


Figure 6.2 Methodology

pipeline identification, authoring, deployment and evolution. This phase is targeted at business analysts and pattern engineers.

The setup phase is, as a preparation phase, triggered by requirements coming from needs which usually originate from the execution phase. Requirements might be expert-driven, i.e., they are derived by business analysts during pipeline identification. For instance, the requirement to implement a new sensor adapter might come from the need to monitor a specific parameter the sensor produces. In addition, requirements can be evolution-driven, e.g., system-generated recommendations to adapt event processing logic based on analyzed event streams. These requirements are gathered (semi-) automatically during the execution of pipelines by applying pipeline evolution techniques [Sen 2013].

The main advantage of the approach suggested by our methodology is that reuse of pipeline elements is facilitated. Developers in the setup phase first define pipeline elements using a vocabulary which abstracts from a specific technical implementations. Furthermore, we provide tool support which enables developers to implement processing elements which do not depend on specific input streams, but are able to solely operate based on modeled input requirements and output specifications.

Before we define roles and further explain the individual processes and tasks within each phase, we give an illustrative example for a logistics-based use case.

Example. *During pipeline identification, a business analyst is interested in getting notifications once a vehicle which transports hazardous goods arrives at one of the company's warehouses. The company has already equipped drivers with a mobile application which is able*

to collect position data and information about the current load, but this data is only stored in a central database for off-line analysis.
In the setup phase, business analysts first define domain knowledge. In this example, domain knowledge contains information about warehouse locations as well as the definition of products along with their classification as being hazardous. In the meanwhile, technical experts are required to develop an adapter which forwards the events collected by the app in real-time to a message broker, an event processing agent which implements a geofencing algorithm, another event processing agent which implements a filter functionality based on products and a notification component.
Taking the example of the geofencing component, the first modeling task is to describe the expected input stream of the component. In our example, the developer defines a stream restriction which accepts any event stream that provides geographical coordinates. In addition, the model defines required static data in form of a coordinate which marks the center of the geofence. This coordinate can be linked to a concept in the knowledge base. Based on the model, provided tool support is able to generate an implementation template, which leaves developers with the task to implement the specific geofencing functionality. After deployment, the geofencing component can be used in the pipeline editor.
The specific tasks within the setup phase and execution phase are further detailed in sections 6.5 and 6.6.

6.4 Roles

We have already introduced three roles we relate to in our methodology, namely *business analysts*, *pattern engineers* and *technical experts*. In this section, we briefly define these roles and discuss the core competencies they require.

- **Business Analyst.** According to the International Institute of Business Analysis [Brennan 2009], the role of business analysts is defined as a "liaison among stakeholders in order to understand the structure, policies, and operations of an organization, and to recommend solutions that enable the organization to achieve its goals". As we focus on the development of real-time applications, our definition of this role is more specialized and targets business analysts who are concerned with tasks related to immediate situation detection or have an interest in data preparation for further analysis. In many cases, business analysts define relatively fine-grained business goals and opportunities, which should be detected at runtime as well as (potential) threats which might affect the fulfillment of goals.
- **Pattern Engineer.** Pattern engineers play a key role in our methodology as they are responsible for the development of processing pipelines. This role is closely related to the role of *Systems Analysts*, who are defined as a "person who uses analysis and design techniques to solve business problems using information

technology" [Walford 2014]. Thus, systems analysts usually build a bridge between more business-oriented analysts on the one hand, and technical experts on the other hand. This perfectly matches the role we envision for pattern engineers. The main responsibility of this role is to implement processing pipelines based on business requirements. Pattern engineers compose processing pipelines by choosing appropriate event streams, and connect them with processing elements prepared by technical experts during the setup phase. The main competencies required for the pattern engineer role are therefore to have basic knowledge of event-driven computing as well as data transformation processes and analysis.

- **Technical Experts.** Technical experts are responsible to build the technical basis needed for building processing pipelines, i.e., to build re-usable pipeline elements. The role primarily targets people responsible for software engineering tasks and requires specific knowledge on event processing systems and their implementations.

6.5 Setup Phase

In this section, we describe tasks assigned to the setup phase in more detail. Figure 6.3 details processes and tasks within this phase. In general, two main processes are defined: Knowledge modeling deals with the definition of background knowledge which is later needed in the execution phase to define domain-specific data. The event processing process is related to technical-oriented tasks in order to prepare pipeline elements for use within the execution phase.

6.5.1 Event processing tasks

Main goals of event processing-related tasks during the setup phase are to model, implement and deploy event producers, event streams and event consumers.

Pipeline Element Modeling

The first task is to define an implementation-independent model of necessary pipeline elements. While the specific details of this model are presented in detail in chapter 7, this section discusses the basic elements that need to be described for each pipeline element are introduced in this section.

Event Producers and Streams. Event producers are entities which produce a number of event streams. The ultimate goal of this modeling task is to describe the run-time behavior of streams, including the structure of events, the meaning of event properties such as measurements and their technical accessibility. Figure 6.4 shows sample properties which can be defined for event producers. It is noteworthy that we do not define the run-time message itself, but only the semantic description how events

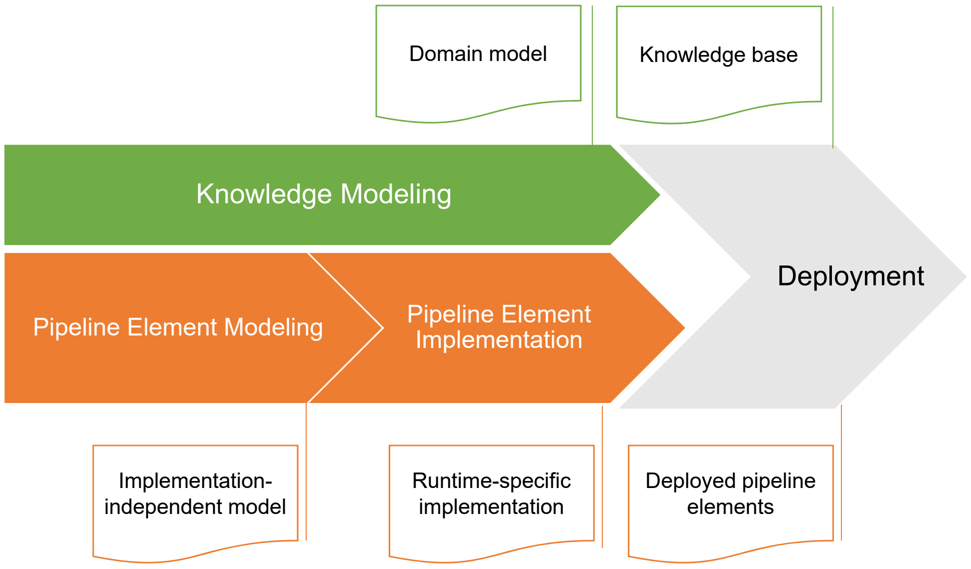
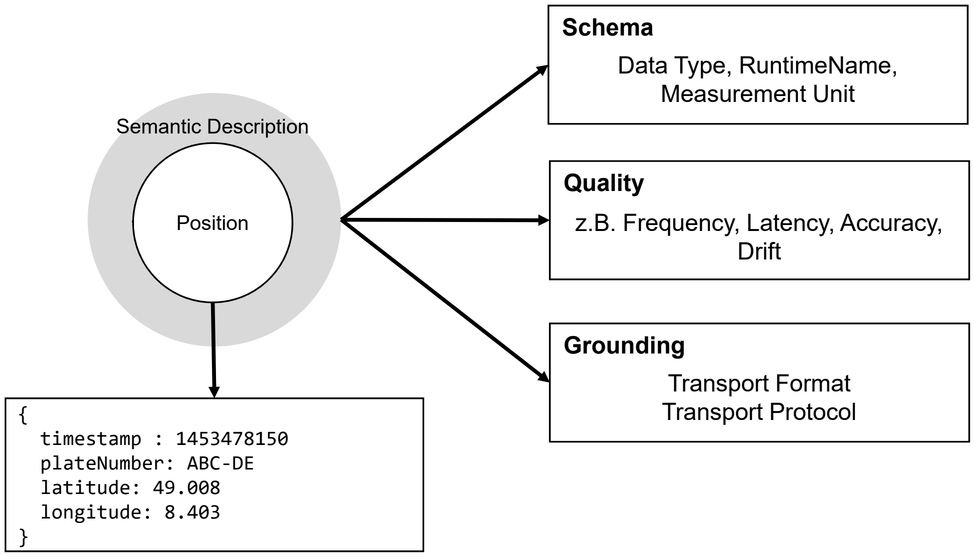

Domain model
Knowledge base
Knowledge Modeling
Deployment
Pipeline Element Modeling
Pipeline Element Implementation
Implementation-independent model
Runtime-specific implementation
Deployed pipeline elements

Figure 6.3 Setup Phase: Processes and Tasks

are expected to look like at run-time. This design decision allows to use lightweight message formats at run-time and even binary formats which are potentially faster when processing event streams with high frequency, while still being able to benefit from an advanced description of events.
First, an event producer needs to be specified. An event producer is related to a specific real-world object which produces event streams, such as a specific sensor. A single event producer can produce multiple event streams. For each stream, its specific characteristics are defined. In general, an event stream consists of an unbounded sequence of events having the same type. Thus, the event type (or, in other words, the event schema) needs to be modeled. The schema usually consists of a number of event properties which correspond to specific measurement values of the event source, for instance, a temperature sensor produces a stream which contains temperature measurement values. For each event property, several characteristics can be defined such as:

- The property type, which defines the data type of the measurement value in the technical representation (e.g., a numerical value),
- the runtime name, which defines the identifier of the measurement value in the technical representation (e.g., a variable in a JSON representation),
- the semantics of the property, which define *what* is being measured and



```
{
  timestamp : 1453478150
  plateNumber: ABC-DE
  latitude: 49.008
  longitude: 8.403
}
```

Figure 6.4 Setup Phase: Modeling of Event Producers

- the value specification, e.g., the value range of a numerical property.

Besides the schema, quality specifications can be given. Qualities can be defined either on the stream level (e.g., the frequency of events a stream produces), or on the event level (e.g., accuracy of a measurement). In addition, technical information on the event channel is required which specifies how the event stream can be consumed, including its run-time format used to represent the event and its communication protocol.
Additional properties can be assigned to a stream definition, this is further detailed in chapter 7.
Event Processing Agents. Example properties which are modeled for EPAs are illustrated in figure 6.5. EPAs are basically defined by their input requirements, their output specification and additionally required human input. The input is defined based on the event stream model, however, an EPA specifies requirements the underlying implementation needs to execute the processing logic. Example requirements include schema-level requirements (e.g., a specific data type) which specify a set of event properties an incoming stream must have. Quality requirements are concerned with minimum or maximum quality criteria an input stream needs to provide. For instance, an EPA which provides an online algorithm might require a minimum frequency of incoming events in order to produce proper results.
Besides input requirements, the output of an EPA is described by providing transformation types. These types specify the characteristics of an output stream in relation

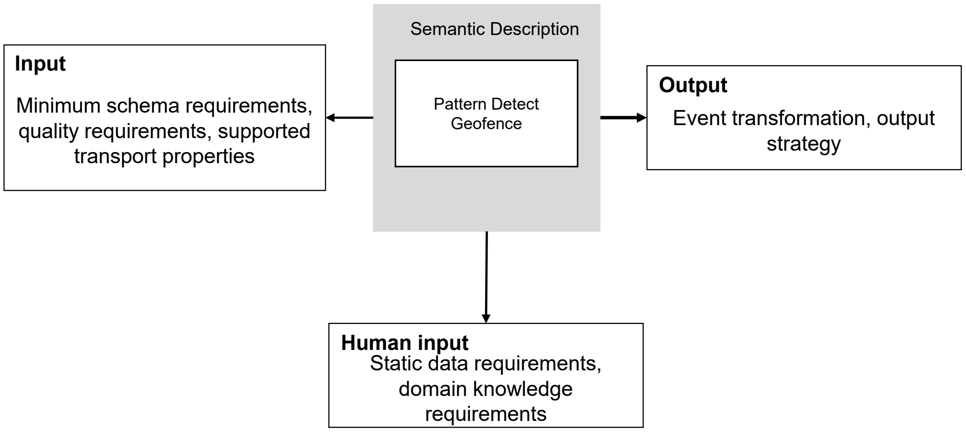

Semantic Description
Pattern Detect
Geofence
Input
Minimum schema requirements, quality requirements, supported transport properties
Output
Event transformation, output strategy
Human input
Static data requirements, domain knowledge requirements

Figure 6.5 Setup Phase: Modeling of Event Processing Agents

to the specific input stream. Transformation types depend on the type hierarchy of an EPA introduced in section 2.2. For instance, a filter EPA does not change the schema of an input event stream, but might influence the frequency of an output stream.
Finally, static data input needs to be described for an EPA. In this task, developers specify additional information which is needed to instantiate the EPA in the execution phase. This might involve the specification of supported operations (for instance, a Filter EPA could operate on multiple filter conditions), human input (e.g., additional user input required to instantiate the EPA) or required instances from the knowledge base.
Event Consumers. The third model-related task in the setup phase is modeling of event consumers. Required properties for event consumers are in large parts identical to EPAs, as they also specify input stream requirements and static input. However, consumers as sinks in an event processing network do not need to specify the output type.
The main results produced in this task are implementation-independent models which describe the intended run-time behavior of event sources, processing agents and consumers.

Pipeline Element Implementation

After pipeline elements have been modeled in an implementation- independent way, the next task is to create re-usable run-time implementation logic.
For EPAs and event consumers, this logic is not bound to specific event streams, but is solely able to operate on the input requirements and output specification defined in

the modeling task. Implementation tasks first involve the selection of an appropriate event processing system (e.g., an existing framework as introduced in section 2.3.2 or a custom implementation). Afterwards, the software engineering process consists of two parts:
The first task is not specific to the business logic, but related to the technical connectivity of the processing element. Input and output adapters need to be defined, e.g., communication channels and protocols which should be supported. Subsequently, the specific business logic (e.g., the event transformation process) of the element can be implemented. In section 7.6.1, we will show that appropriate tool support can be used to automate the generation of code needed for the first task.
For event producers, the implementation task includes the development of specific event adapters which read directly from sensors or other event sources. In addition, the technical connection to the event channel used for transmitting event data to a message broker needs to be implemented.
The main result which is produced in this task is a software artifact which contains a re-usable implementation of a pipeline element.

6.5.2 Knowledge modeling

The second process assigned to the setup phase is knowledge modeling. While the setup phase serves to identify and capture *domain knowledge*, it is mainly used in the execution phase to support pipeline authoring with domain-specific background knowledge. Domain ontologies generally capture knowledge for a particular type of domain [Studer et al. 1998]. The exact process to identify and model knowledge for this purpose is a separate topic and therefore out of scope of this thesis, but widely-used methodologies for *knowledge engineering* can be re-used [Sure et al. 2003; Sure et al. 2004; Fernández-López et al. 1997].
The resulting artifact of the knowledge modeling task is a domain model as illustrated in figure 6.3.

6.5.3 Deployment

Finally, the artifacts developed in the setup phase are made accessible to the execution phase by performing a deployment task. Captured knowledge is usually deployed in a knowledge base which only contains the domain knowledge that has been identified as relevant for building processing pipelines. Deployment also includes making the knowledge base technically accessible (e.g., in form of a SPARQL[1] endpoint) for tools used within the execution phase.
The implementations of pipeline elements are also deployed to a repository. This repository provides endpoints to access the pipeline element model, i.e., the descrip-

[1] a query language for RDF-based databases: https://www.w3.org/TR/rdf-sparql-query/

tion of an element. This description can be read by tools in the execution phase to retrieve information on available event streams, processing agents and consumers. Depending on the event processing system used for the actual implementation, additional tasks might be required prior to deployment, e.g., setting up of the required run-time infrastructure which executes the event processing logic.

6.6 Execution Phase

Now we move to the execution phase and the tasks which are performed in this phase as illustrated in figure 6.6.
In general, the execution phase starts with the identification of required pipelines by business analysts. From this point, pattern engineers are responsible for pipeline authoring, deployment and finally evolution. This process can be triggered either in an expert-driven manner, or in an evolution-driven way: By analyzing the current execution status, recommendations for new pipelines or modifications to existing pipelines can be generated. If any missing event sources or processing elements are identified in one of these tasks, the setup phase is entered in order to implement the missing components.

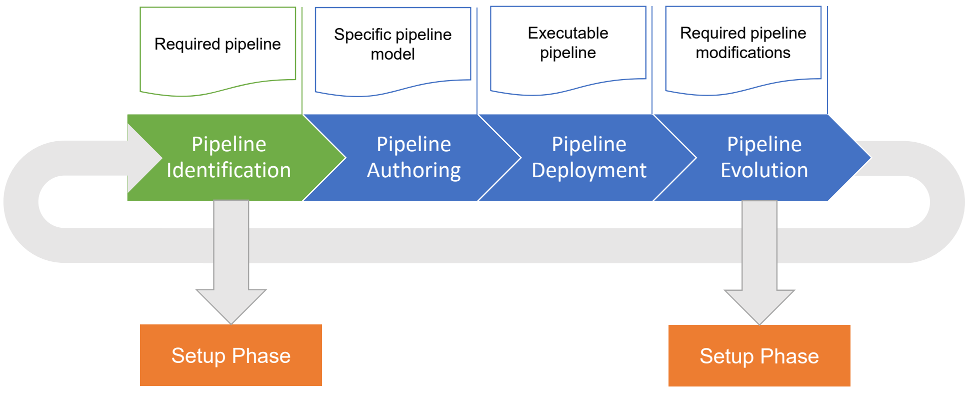


Figure 6.6 Execution Phase: Tasks

6.6.1 Pipeline Identification

Before processing pipelines can be defined by pattern engineers, the purpose of the pipeline needs to be defined. Pipeline identification is a manual task performed by business analysts. It follows a requirements engineering process which depends on the intended purpose of the processing task.

In section 3.2, we have already introduced three general application domains where real-time processing is advantageous, namely monitoring, data harmonization and situation detection.
While pipelines which are intended to support data harmonization are usually built with the goal to transfer data to external systems in order to prepare data for offline analysis, pipeline identification for this category is pretty straightforward as requirements directly arise from business needs.
Business needs targeting monitoring and situation detection are in general related to business goals. Business goals describe target values for specific performance indicators an organization aims to achieve. Tracking of current performance and goal fulfillment are the main business drivers for monitoring-related pipelines. Furthermore, enterprise architecture models such as the Business Motivation Model (BMM) [*Business Motivation Model - Version 1.1* 2010] provide methods to identify threats (as negative goal influencers) and opportunities (as positive goal influencers), which are the business drivers for situation detection tasks in event processing.
In this thesis, we do not further investigate pipeline identification as it is more related to business-related research areas, but many approaches already exist which allow to identify business goals, performance indicators and situations [Kavakli 2004].

6.6.2 Pipeline Authoring

Afterwards, the identified pipeline is being defined by pattern engineers. This task includes the selection of appropriate event streams according to the identified business need. Based on the input streams, processing elements (which have already been implemented in the setup phase) need to be selected and connected to describe the data flow in form of a processing pipeline. In this process, each element needs to be configured based on the input stream and static data requirements depending on the specific element. In addition, configuration also includes the assignment of knowledge items from the domain ontology. Supporting pipeline authoring for non-programmers is one of the main objectives of this thesis. Chapter 8 presents an in-depth description of this task and details on processing pipelines.
The main artifact produced within this task is a stream-specific pipeline model.

6.6.3 Pipeline Deployment

After pipelines have been defined, the pipeline model must be translated into an executable model. This is done by invoking the run-time implementation of each processing element which is part of the pipeline. Depending on the underlying event processing systems which contain the implementation logic, additional implementation-specific tasks might be required. For instance, CEP systems which use a declarative pattern language to reflect the event processing logic need to be parameterized with stream-specific input event types and static data provided in the authoring task. In contrast, many distributed event processing systems need to submit the instantiated implementation logic to a cluster which controls distributed execution.

6.6.4 Pipeline Evolution

Finally, pipeline evolution deals with monitoring and evolution of pipelines. Monitoring is mainly concerned with technical surveillance of the current execution status of pipelines and its processing elements. For instance, failures of sensors or individual processing elements need to be detected as they occur in order to avoid unexpected data loss. In contrast, evolution, which is also based on monitoring, primarily deals with the inspection of the data flow in a pipeline with the main goal to generate recommendations on adaptation of processing elements. Supporting adaptivity of real-time processing pipelines in a bottom-up manner by using data mining and machine learning techniques is still an active research area, some approaches can be found in [Sen et al. 2010a; Sen et al. 2010b; Lee et al. 2015; Margara et al. 2014].

6.7 Tool Support: StreamPipes

In order to demonstrate the feasibility of this methodology and for evaluation purposes, we created tool support in form of StreamPipes [Riemer et al. 2015]. StreamPipes is a software framework which provides various tools for the individual phases and tasks of our methodology. Figure 6.7 shows the tool coverage of StreamPipes in relation to the methodology. The setup phase is supported by four different tools: First, the knowledge editor (1) helps to capture and manage domain-specific knowledge. Second, a description model editor (2) provides a web-based interface to generate implementation-independent models to describe pipeline elements. In addition, a Software Development Kit (SDK, 3) is provided which offers high-level access to model-specific parameters and abstracts from event processing-specific implementation tasks which are not directly related to the business logic. Finally, we have created runtime wrappers (4) for various existing event processing systems in order to show the applicability of our approach for a wide range of existing tools which are typically used for event processing.
For the execution phase, we have created a graphical editor (5) to compose event processing pipelines out of existing, reusable pipeline elements. The integration and execution engine (6) is a back end component for the authoring tool and manages the life cycle of a processing pipeline by providing matching capabilities between processing elements at design-time (e.g., in order to determine whether two pipeline elements are compatible to each other based on their underlying description) and handles the execution management of distributed event processing logic.

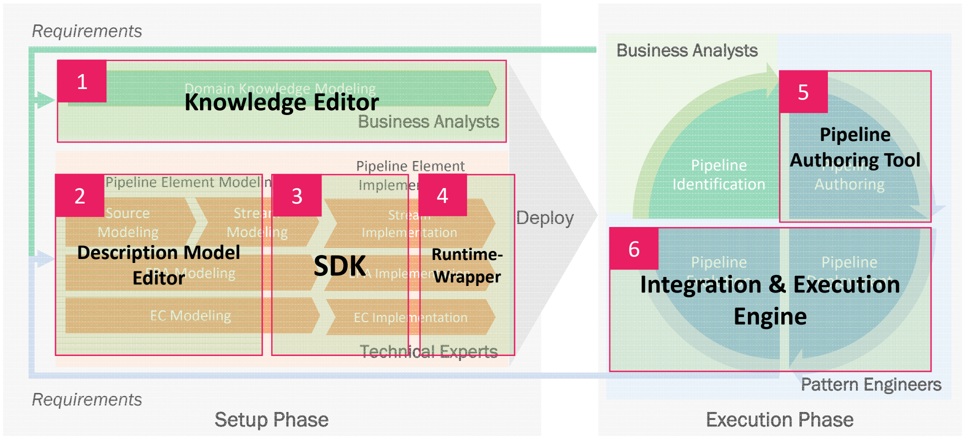


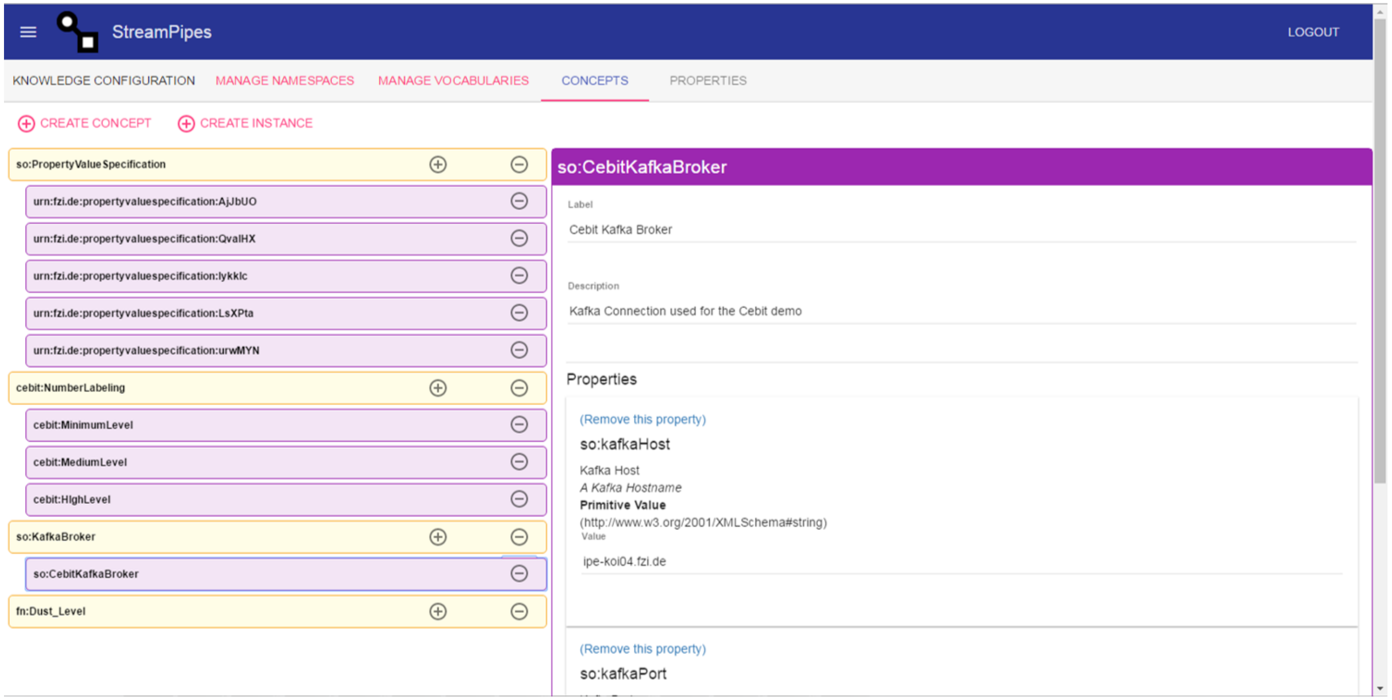
Figure 6.7 Methodology: Tool Coverage

This section only briefly introduces these tools. Details are provided within chapters 7 and 8, where we focus on technical details that are concerned with each phase.

6.7.1 Knowledge Editor

The user interface of the knowledge editor is illustrated in figure 6.8. This tool is basically an ontology editor for the management of domain knowledge. Although various tools exist to manage ontologies (see below), we decided to create a lightweight editor which is directly integrated into the framework due to the following reasons: As modeling of knowledge is a task assigned to the role of business analysts, usability is an important issue. Most ontology editors such as Protégé[2] or TopBraid Composer (TBC) [3] are more technical-oriented tools which require advanced knowledge of semantic web applications and are therefore not suited for our purpose. Furthermore, by providing our own solution we were able to provide a simplified user interface that also allows convenient access for more advanced concepts such as definition of property value specifications. Although such functionality is supported by existing ontology modeling tools (by providing the required concepts and properties), our editor provides higher-level access to such specifications.
Features of the knowledge editor include definition of concepts, properties and instances, management of namespaces, and import of existing RDF-based vocabularies.

[2] http://protege.stanford.edu/
[3] http://www.topquadrant.com/tools/ide-topbraid-composer-maestro-edition/



so:PropertyValueSpecification
urn:fzi.de:propertyvaluespecification:AjJbUO
urn:fzi.de:propertyvaluespecification:QvalHX
urn:fzi.de:propertyvaluespecification:lykklc
urn:fzi.de:propertyvaluespecification:LsXPta
urn:fzi.de:propertyvaluespecification:urwMYN
cebit:NumberLabeling
cebit:MinimumLevel
cebit:MediumLevel
cebit:HighLevel
so:KafkaBroker
so:CebitKafkaBroker
fn:Dust_Level

so:CebitKafkaBroker

Label
Cebit Kafka Broker

Description
Kafka Connection used for the Cebit demo

Properties

(Remove this property)
so:kafkaHost
Kafka Host
A Kafka Hostname
Primitive Value
(http://www.w3.org/2001/XMLSchema#string)
Value
ipe-koi04.fzi.de

(Remove this property)
so:kafkaPort

Figure 6.8 Knowledge Editor

Furthermore, data models can be specified for concepts by making use of schema.org's domainIncludes and rangeIncludes properties.

6.7.2 Description Model Editor

Figure 6.9 shows the main screen of the description model editor. This editor serves to assist technical experts in describing process elements according to our vocabulary without requiring knowledge of semantic web technologies.
Features include definition of new event producers, event processing agents and consumers. If a source has already been implemented and is publishing event streams to an existing message broker, they can be made directly accessible into the pipeline authoring tool from this interface.
The description model editor is connected to a code generation module (see below) which allows developers to generate an implementation template for supported event processing systems directly from the model.

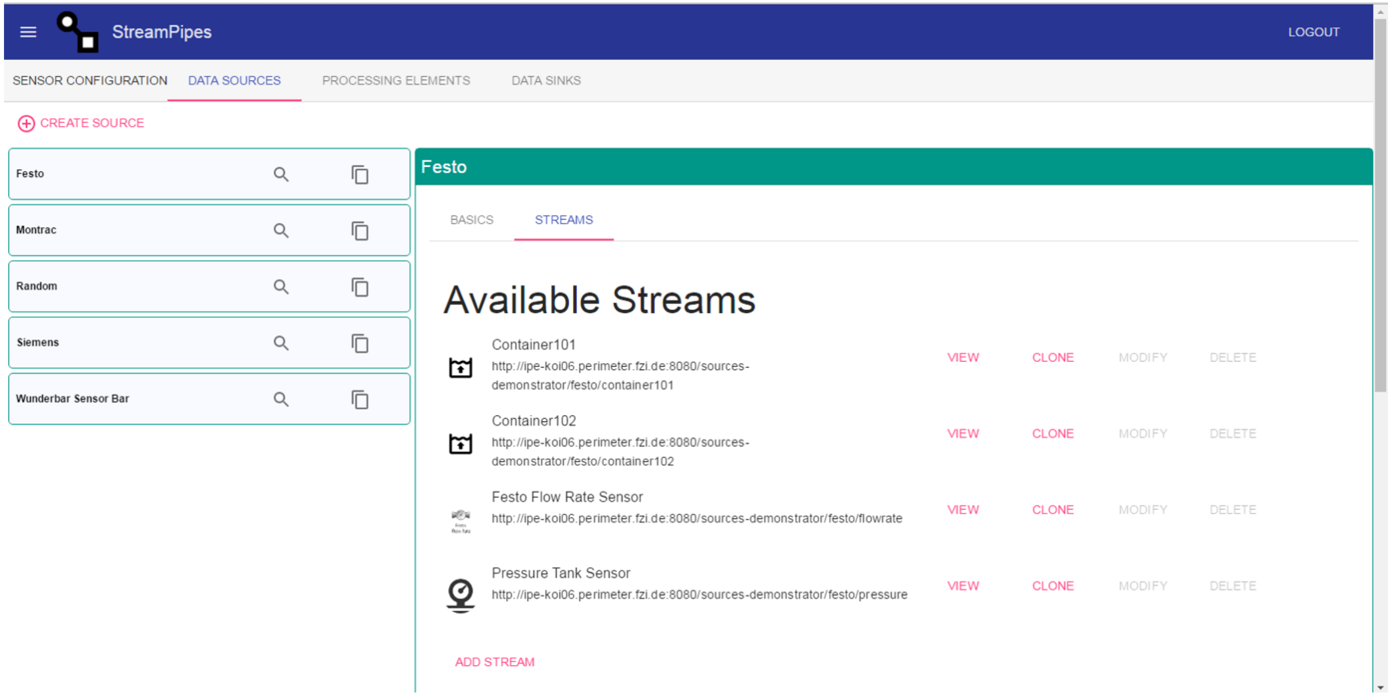
6.7.3 Software Development Kit (SDK)

The SDK is a software library implemented in Java which provides high-level programming access to our model. Instead of using the model editor, descriptions for pipeline elements can also be created by using the SDK. In addition, the SDK includes a web server implementation to make event processing logic available for invocation at a specified HTTP endpoint. Such an endpoint publishes the description of a pipeline element and receives invocation messages from the execution engine. The SDK automatically performs transformations between RDF graphs and Java-based programming models, allowing developers to completely abstract from the RDF layer.

6.7.4 Runtime-Wrapper

In order to show the ability of our approach to manage processing pipelines which are consisted of heterogeneous underlying run-time technologies, StreamPipes includes run-time wrappers for several open source event processing frameworks. These wrappers are built as a bridge between our implementation-independent model and framework-specific implementation details.
StreamPipes currently includes run-time wrappers for the CEP engine Esper[4], the distributed stream processing frameworks Apache Flink[5] and Apache Storm[6] and a wrapper for custom event processing systems that do not rely on an existing framework.

[4] http://www.espertech.com
[5] http://flink.apache.org
[6] urlhttp://storm.apache.org



Available Streams

Container101
http://ipe-koi06.perimeter.fzi.de:8080/sources-demonstrator/festo/container101
VIEW CLONE MODIFY DELETE

Container102
http://ipe-koi06.perimeter.fzi.de:8080/sources-demonstrator/festo/container102
VIEW CLONE MODIFY DELETE

Festo Flow Rate Sensor
http://ipe-koi06.perimeter.fzi.de:8080/sources-demonstrator/festo/flowrate
VIEW CLONE MODIFY DELETE

Pressure Tank Sensor
http://ipe-koi06.perimeter.fzi.de:8080/sources-demonstrator/festo/pressure
VIEW CLONE MODIFY DELETE

Figure 6.9 Description Model Editor

6.7.5 Pipeline Authoring Tool

The pipeline authoring tool (illustrated in figure 6.10) is a web-based editor for modeling processing pipelines. The editor consists of three separate views to select from available event streams, processing agents and consumers. Elements can be dragged into an *assembly area* to create pipelines. Whenever a new connection between to elements is made, the integration engine is invoked to verify that a connection is possible and to recompute the required human input which is needed for instantiation of a processing element based on the set of previously defined static data requirements. The authoring tool additionally includes functionality for users to receive recommendations for elements and to execute and modify pipelines directly from the user user interface. Chapter 8 presents the authoring tool in detail.

6.7.6 Integration and Execution Engine

Finally, StreamPipes includes an engine to support the integration of heterogeneous run-time event processing logic and to control the execution of processing pipelines. Integration is supported by matching algorithms to determine the ability of two pipeline elements to be connected within a processing pipeline. This algorithm expects two description graphs as an input, performs verification checks based on schema-level requirements, quality-level requirements and supported communication protocols and negotiates the run-time communication between two elements. In section 8.3, we further describe this matching process.
Execution includes the instantiation of distributed processing pipelines by invoking the implementation logic of each involved pipeline element. In addition, functionality to support pipeline evolution is provided, e.g., by establishing monitoring nodes for running pipelines to detect failures and create recommendations on pipeline modifications based on available pipeline elements.

6.8 Summary

In this section, we proposed a methodology supporting the development process of event processing applications. We presented the methodology itself, followed by a more detailed discussion of phases, processes and tasks. Our methodology splits the development process into two phases, the setup phase and the execution phase. While the setup phase serves to prepare re-usable event processing logic, the execution phase deals with the definition of processing pipelines by connecting re-usable elements with specific input event streams. The main advantage of this approach is, while non-programmers are provided with the possibility to define event processing applications abstracted from technical details without developer consultancy, at the same time the event processing system does not rely on a predefined, fixed set of pipeline elements.

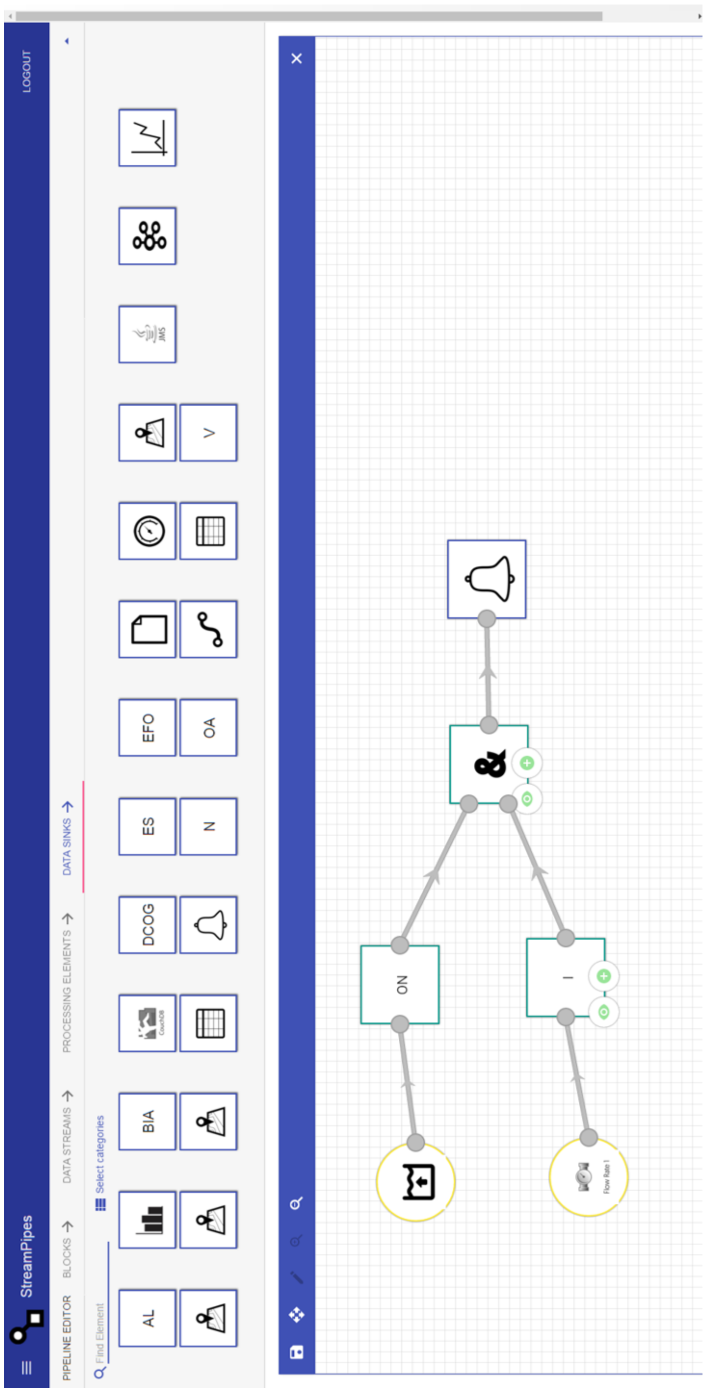


Figure 6.10 Pipeline Authoring Tool

The setup phase ensures extensibility of the system at any time once new requirements arise. Thus, our methodology combines a highly flexible system with an intuitive way to create event processing applications.
Additionally, we introduced tool support in form of StreamPipes, a reference implementation providing end-to-end tool coverage of our methodology. Details on specific tools that are part of StreamPipes are further explained in the upcoming sections.
In chapter 7, we build upon this methodology by introducing a vocabulary supporting the setup phase. Chapter 8 targets the execution phase in more detail by presenting details on pipeline authoring and deployment.

7 Setup Phase

In this chapter, we focus on the setup phase by developing models which are needed to describe event producers, processing agents and sinks independent from their underlying implementation. In addition, we discuss how this description can be made accessible to other elements and define the execution flow of events at run-time. In section 7.1, the general approach is briefly described. Section 7.2 introduces some existing vocabularies we re-use and extend as a basis for a vocabulary which defines a semantic model for pipeline elements. This vocabulary is presented in detail in sections 7.3, 7.4 and 7.5. Finally, tools we have developed to support the setup phase are presented in section 7.6.

7.1 Walkthrough

As in the previous chapter, we briefly sketch the approach which is being introduced within this chapter.
Figure 7.1 illustrates a conceptual view on the *pipeline element modeling* task of the setup phase. The goal of this task is to define a specification of event producers, processing agents and consumers, meaning a description of the run-time event streams produced by an event source and requirements as well as output specifications for Event Processing Agents (EPAs) and consumers.
As pipeline elements should be used as part of processing pipelines, this description needs to be made available in order to allow other components to retrieve information on the element. Our approach realizes this by providing a *description graph* which contains the complete specification of a pipeline element. This graph is based on the RDF model and represented as a JSON-LD[1] document. This document is attached to a RESTful[2] interface, so that information on a pipeline element can be retrieved globally

[1] JSON-LD is a JSON-based format to serialize RDF data. In contrast to RDF/XML, JSON-LD has a syntax which is easier to read and, in contrast to human-readable RDF serializations such as Turtle, JSON-LD is based on standards used in the web.

[2] Representational State Transfer (REST) is an architectural style for web services based on HTTP operations. REST handles manipulation of resources (e.g., web documents) by use of standard HTTP operations GET, POST, PUT and DELETE.

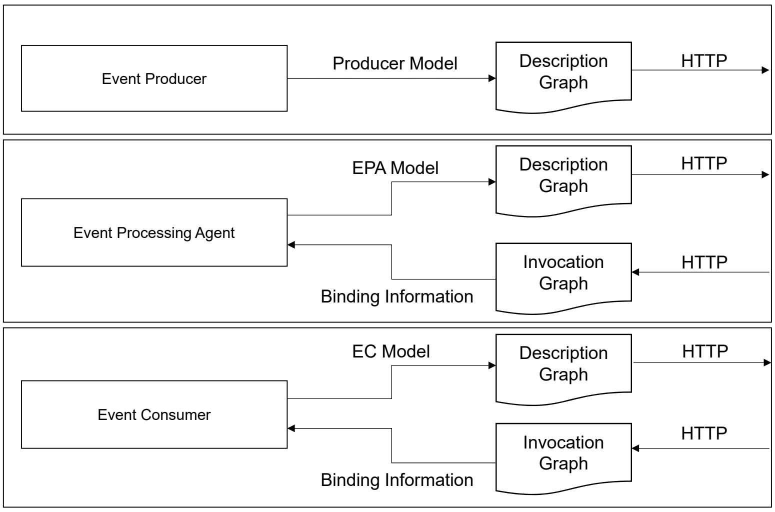


Figure 7.1 Setup Phase: Approach

from the web. Section 7.2 introduces the vocabulary used to generate description graphs.
Event producer descriptions are passive, i.e., they only provide information on event streams. That is, the way streams are published to a message broker and details on the events that are sent. In contrast, processing elements are active elements, i.e., they need to be instantiated with *binding information*, i.e., a configuration containing the specific input streams and static data needed in order to execute the run-time logic. The instantiation of a processing element is called invocation. In our architecture, a processing element is being invoked by sending an *invocation graph* to a REST interface of the element. In section 8.2, we define a vocabulary used to create invocation graphs.

7.2 Ontology re-use

One of the advantages of using RDF as a data model is drastically increased re-usability of shared conceptualizations. Ontology re-use enables interoperability of systems by merging different ontologies into a single one and by extending, specializing or adapting existing ontologies to support new objectives [Pinto and Martins 2000].
In concern to our approach, re-use of existing ontologies is beneficial for our provided vocabulary to describe pipeline elements, as concepts from other ontologies can be extended and specialized. In addition, re-use facilitates knowledge modeling. By

re-using standard external vocabularies which follow the linked data principles[3], interoperability is increased as vocabularies can be shared among multiple stakeholders, which is a main requirement for distributed application scenarios such as event marketplaces.

In general, our ontology re-uses several existing vocabularies:

- **Dublin Core**. Dublin Core (DC) is a vocabulary which provides terms that can be used to describe resources [Weibel et al. 1998]. The current DC vocabulary provides an element set called *Dublin Core Metadata Initiative (DCMI) Metadata Terms*[4], which defines a set of *terms* to describe metadata. The DC vocabulary is available as an RDF model.
- **Semantic Sensor Network Ontology (SSN)**. The SSN is an ontology to describe capabilities and properties of sensors and their observations [Compton et al. 2012]. The core vocabulary allows to model capabilities of sensors (e.g., measurement capabilities), observation values, operating restrictions of sensors and deployment-related properties. The SSN model is defined as an Web Ontology Language (OWL) ontology using the OWL 2 standard.
- **Schema.org**. Schema.org has been developed by major search engines Google, Bing and Yahoo! as a common vocabulary aiming to support data markup on web pages [Guha 2011]. It is worth to note that Schema.org does not directly depend on RDF as a data model, but defined its own model with some semantic differences to the RDF(S) standard. For instance, while RDFS provides the properties *rdfs:domain* and *rdfs:range* to assign an instance to a specific concept, Schema.org provides the properties *domainIncludes* and *rangeIncludes*, which define a set of allowed concepts for a given property. This approach has the advantage that it adheres more to the intuitive understanding of most web developers as it helps to define specific properties a concept might have, which is practically not possible in RDFS. The Schema.org representation is available in RDFa[5] and therefore can be reused in RDF-based vocabularies [Brickley 2011].

In this chapter, we use the following namespaces to identify these vocabularies:

- **DC** to identify the Dublin Core vocabulary,
- **SSN** to identify the SSN vocabulary,
- **SO** to identify the Schema.org vocabulary,
- **RDF** to identify the RDF vocabulary,
- **RDFS** to identify the RDF Schema vocabulary, and
- **EPA** to identify the vocabulary developed within this chapter.

[3] Four main rules are typically defined as *Linked Data Principles*: (1) Use of URIs as names for things, (2) use of HTTP URIs, (3), make things dereferencable, (4) include links to other URIs. (https://w3.org/DesignIssues/LinkedData.html)

[4] http://dublincore.org/documents/dcmi-terms/

[5] Resource Description Framework in Attributes (RDFa) is a W3C standard to augment HTML document with RDF-based metadata.

As follows, we present details on the vocabulary which allows to create description graphs for event producers, event processing agents and event consumers. For each of these elements, we describe the main purpose of defined concepts and properties including examples.

7.3 Semantic Event Producers

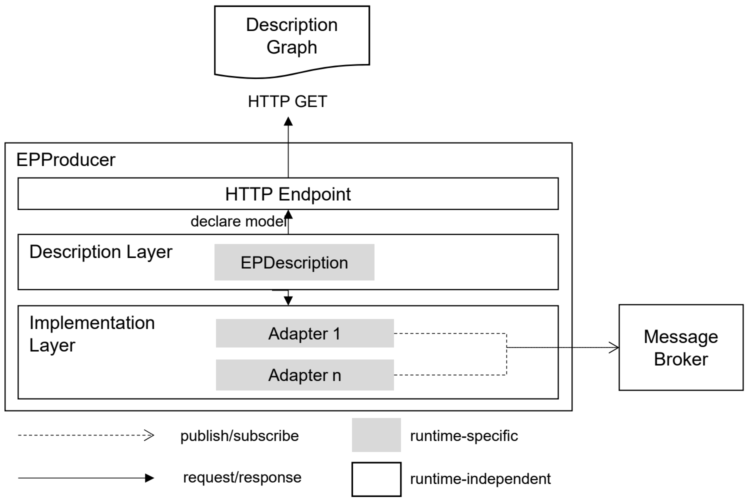


Figure 7.2 Semantic Event Producers: Conceptual View

In this section, a conceptual architecture and a vocabulary to describe event producers is introduced. Figure 7.2 illustrates this architecture. In general, an event producer consists of a description layer and an implementation layer. The description layer specifies the properties of a producer, most importantly, the stream it provides and their characteristics. These properties are represented in a description graph containing all necessary information required to consume the event streams belonging to a producer and are subsequently made available to the web. A description graph therefore has a unique URL and can be retrieved by performing a HTTP GET request on the URL, which returns the graph serialized as a JSON-LD document.
At run-time, an EP produces one or more event streams, and events are continuously pushed to a message broker using a publish/subscribe protocol. Details on the specific protocol, such as the URL of the broker or the target topic are derived directly from the description.

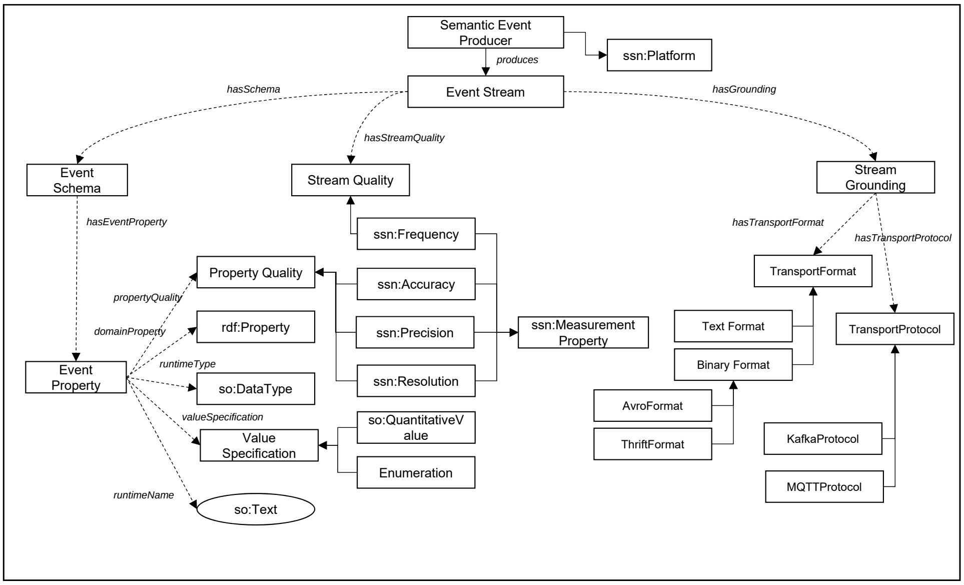


Figure 7.3 Semantic Event Producers: Vocabulary

As follows, we call an event producer which publishes a semantic description of its specific properties according to our model a *Semantic Event Producer (S-EP)*.
We will now describe the vocabulary used to specify an S-EP in more detail. Figure 7.3 outlines the main terms and their relations. The goal of this figure is not to show a precise model, but is intended to give a first view on the general way our model works.
The concepts illustrated in this figure are further explained within this section by using the following approach: The vocabulary is defined in a top-down manner starting with the definition of an S-EP itself. Related concepts of S-EPs are explained in subsections, where we start with explaining the general purpose of each concept. Afterwards, the vocabulary is introduced in detail. Finally, the real-world usage is shown based on illustrative examples.
The top-level concept is the term SemanticEventProducer, which is a direct subclass of ssn:Platform. The ssn:Platform concept allows to define metadata for the producer itself, e.g., its location and or name if needed. A single S-EP can produce one or more event streams. Human-readable information which indicates the purpose of the S-EP can be assigned to a producer definition.

Listing 7.1 Example Event: Vehicle Position

```
{
  "timestamp" : 1234567,
  "plateNumber" : "AB-CDE",
  "latitude" : 48.23,
  "longitude" : 7.46,
}
```

Vocabulary

Tables 7.1 and 7.2 explain top-level concepts and properties of an S-EP.

Table 7.1 Concepts: Semantic Event Producers

Concept	rdfs:subClassOf	Description
epa:SemanticEventProducer	ssn:Platform	An entity which produces event streams
epa:EventStream	rdfs:Class	Defines an event stream

Table 7.2 Properties: Semantic Event Producers

Property	so:domainIncludes	so:rangeIncludes	Description
dc:title	epa:SemanticEventProducer epa:EventStream	so:Text	Assigns a human-readable title to a SemanticEventProducer or an EventStream
dc:description	epa:SemanticEventProducer epa:EventStream	so:Text	Assigns a human-readable description to an SemanticEventProducer or an EventStream
epa:produces	epa:SemanticEventProducer	epa:EventStream	Assigns an EventStream to a SemanticEventProducer

Example

We illustrate the usage of our vocabulary based on the following example: We imagine a mobile phone application used in the logistics domain which continuously gathers real-time data on the current location of transportation vehicles. These parameters are published in a single event stream. The run-time event format we aim to describe in this example is a JSON message consisting of four event properties as defined in listing 7.1:

Listing 7.2 Example: Event Producer Definition

```
:App rdf:type epa:SemanticEventProducer .
:App dc:title "Mobile application producer" .
:App dc:description "Publishes events produced by the mobile application" .
:App epa:produces :PositionStream .
```

Listing 7.2 shows the definition of an event producer. For better readability, these examples are given in Turtle[6] notation, which is a W3C recommendation to represent RDF triples.

7.3.1 Event Stream

An event stream defines a continuous stream of events produced by an S-EP. It can also be described with a title and a description and is further related to three concepts (event schema, stream quality, stream grounding, further explained below) which detail the characteristics of the stream itself. The model and an example instantiation of an event stream, further described in section 7.3.1 are illustrated in figure 7.4. An event stream publishes an unbounded sequence of events of the same type. The event type defines the *payload* of an event, i.e., its structure. Such specifications can be attached to an event stream by relating it to an instance of the EventSchema concept. Besides information about the structure of an event, characteristics of the event stream itself can be relevant. This mainly includes non-functional properties, such as the frequency or throughput the event stream provides. For this purpose, a stream can be related to a StreamQuality. Finally, information about the connectivity to an event stream must be given. As event streams are possibly consumed by other processing elements, information on the communication channel and the run-time event format needs to be known in advance in order to subscribe to the stream. Modeling technical information on an event stream can be done by defining a specific StreamGrounding.

Vocabulary

Concepts and properties related to streams are summarized in the following tables 7.3 and 7.4.

[6] https://www.w3.org/TR/turtle/

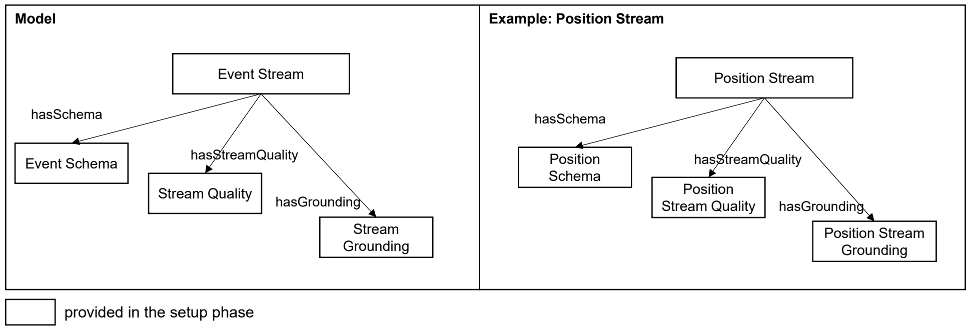


Figure 7.4 Event Stream: Model and Example

Table 7.3 Concepts: Event Streams

Concept	**rdfs:subClassOf**	**Description**
epa:EventSchema	rdfs:Class	An entity specifying the event type
epa:StreamQuality	epa:Quality	An entity specifying quality aspects of an event stream
epa:StreamGrounding	rdfs:Class	Specifies the technical grounding of an event stream

Table 7.4 Properties: Event Streams

Property	**so:domainIncludes** / **so:rangeIncludes**	**Description**
epa:hasSchema	epa:EventStream / epa:EventSchema	Assigns an event schema to an event stream
epa:hasStreamQuality	epa:EventStream / epa:StreamQuality	Assigns a stream quality to an event stream
epa:hasGrounding	epa:EventStream / epa:StreamGrounding	Assigns a stream grounding to an event stream

Example

Listing 7.3 creates an example *PositionStream*. It contains a title and a description to describe the stream in a human-readable manner and, in addition, relates this stream to a specific schema, a stream quality and a stream grounding. Examples of these specific instances are described in subsequent sections.

Listing 7.3 Example: Event Stream Definition

```
:PositionStream rdf:type epa:EventStream .
:PositionStream dc:title "Vehicle Position Stream" .
:PositionStream dc:description "Stream that publishes data gathered by several
    sensors" .
:PositionStream epa:hasSchema :PositionSchema .
:PositionStream epa:hasStreamQuality :PositionStreamQuality .
:PositionStream epa:hasGrounding :PositionStreamGrounding .
```

7.3.2 Event Schema

An event schema defines the type of an event. In event processing applications, an event consist of a set of event properties (also called event attributes) which specify the payload of an event. Figure 7.5 illustrates the vocabulary provided to define an event schema. Event properties might be of one of the following types:

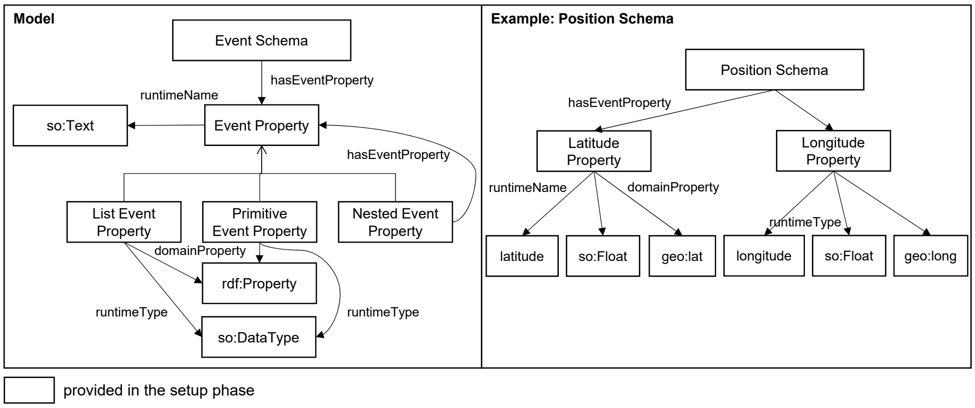


Figure 7.5 Event Schema: Model and Example

PrimitiveEventProperty defines a type of a simple key-value based event property, where the key specifies the identifier of a specific measurement. A primitive property has a runtimeName indicating this key in the resulting run-time event format, a runtimeType indicating its data type and a domainProperty. Run-time types are modeled based on schema.org's DataType definition. This concept specifies several subtypes to define basic data types such as numbers and strings similar to the data type definition from XML Schema[7] often used in RDF vocabularies, but provides a type hierarchy for numerical values.

[7] https://www.w3.org/TR/swbp-xsch-datatypes/

The domainProperty relation allows us to add a semantic annotation to the event property specification. The description of an event property on a semantic level instead of pure data type level provides benefits during the pipeline authoring task, as it can be used as an additional information source to determine the matching between two processing elements. The range of a DOMAINPROPERTY includes any sub property of RDF:PROPERTY. Although it may sound counterintuitive to define an RDF:PROPERTY as the range of a property itself, we made this design decision due to the circumstance that existing RDF vocabularies can be better re-used in our approach, which is clarified in the example at the end of this section.
Primitive event properties are rather simple event types. In many cases, event types can have a more complex structure consisting of list-based properties or nested structures. To support such use cases, we have defined a concepts to represent lists containing multiple values of the same type (LISTEVENTPROPERTY) and nested properties. Nested event properties consist of a RUNTIMENAME and an additional definition of underlying event properties and therefore allow to define *event hierarchies*.

Vocabulary

The complete vocabulary to describe an event schema is explained in tables 7.5 and 7.6.

Table 7.5 Concepts: Schema Definitions

Concept	rdfs:subClassOf	Description
epa:EventProperty	rdfs:Class	Represents a single event property of an event schema
epa:PrimitiveEventProperty	epa:EventProperty	An entity which represents a key-value-based event property
epa:ListEventProperty	epa:EventProperty	Represents a list-based event property
epa:NestedEventProperty	epa:EventProperty	Represents a nested event property

Table 7.6 Properties: Schema Definitions

Property	so:domainIncludes	Description
	so:rangeIncludes	
epa:hasEventProperty	epa:EventSchema	Assigns an event property to an event schema
	epa:EventProperty	
epa:runtimeName	epa:EventProperty	Assigns the name of the event property at run-time to an event property
	so:Text	

Continued on next page

Property	so:domainIncludes / so:rangeIncludes	Description
		Continued from last page
epa:runtimeType	epa:PrimitiveEventProperty epa:ListEventProperty / xsd:string	Indicates the primitive type of an event property at run-time
epa:domainProperty	epa:PrimitiveEventProperty epa:ListEventProperty / rdf:Property	Indicates the semantics of an event property

Example

The following listing 7.4 shows a schema definition for our example. It defines four primitive event properties *timestamp*, *plateNumber*, *latitude* and *longitude* with different run-time types. The example also shows the usage of domain properties. For instance, the GEO:LAT property from the W3C Geo Ontology[8] is assigned to the LATITUDEPROPERTY in order to indicate this property measures a geospatial coordinate.

Listing 7.4 Example: Event Schema Definition

```
:PositionSchema rdf:type epa:EventSchema .
:PositionSchema epa:hasEventProperty :TimestampProperty .
:PositionSchema epa:hasEventProperty :PlateNumberProperty .
:PositionSchema epa:hasEventProperty :LatitudeProperty .
:PositionSchema epa:hasEventProperty :LongitudeProperty .

:TimestampProperty rdf:type :PrimitiveEventProperty .
:TimestampProperty epa:runtimeName "timestamp" .
:TimestampProperty epa:runtimeType so:Integer .
:TimestampProperty epa:domainProperty epa:timestamp .

:PlateNumberProperty rdf:type :PrimitiveEventProperty .
:PlateNumberProperty epa:runtimeName "plateNumber" .
:PlateNumberProperty epa:runtimeType so:Text .
:PlateNumberProperty epa:domainProperty :licensePlate .

:LatitudeProperty rdf:type :PrimitiveEventProperty .
:LatitudeProperty epa:runtimeName "latitude" .
:LatitudeProperty epa:runtimeType so:Float .
:LatitudeProperty epa:domainProperty geo:lat .

:LongitudeProperty rdf:type :PrimitiveEventProperty .
:LongitudeProperty epa:runtimeName "longitude" .
:LongitudeProperty epa:runtimeType so:Float .
:LongitudeProperty epa:domainProperty geo:long .
```

[8] https://www.w3.org/2003/01/geo/

7.3.3 Quality

Besides the pure event schema, quality attributes of event streams can be relevant. For instance, processing elements often require a minimum throughput of an incoming event stream in order to produce viable results (e.g., an activity detection algorithm based on accelerometer sensor data). In this case, the binding of an event stream that doesn't match this throughput requirement should be omitted at design-time. In addition, more information on the run-time behavior of an event property can be useful: For instance, if the measurement range of a sensor-based property is already known at design-time, this information can be used to assist users in selecting appropriate filter expressions. On the other hand, by assigning an accuracy value to an event property, we can prohibit the connection to processing elements which require for a finer-grained measurement. As an example, an event processing agent which performs a geofencing operation usually requires a minimum accuracy of GPS signals in order to produce useful results.
In order to be able to consider such use cases, our model supports the definition of quality attributes as illustrated in figure 7.6. Qualities can be defined on two levels: StreamQuality defines quality definitions on the stream level, PropertyQualities relate quality definitions to event properties. Both concepts are subclasses of the ssn:MeasurementProperty concept.
While specific concepts to define stream- and property-based qualities are out of scope of this thesis, the ssn:Frequency concept is an example subconcept of a StreamQuality. Examples for property qualities are ssn:Accuracy (defining the deviation of the observed measurement and the real value of the measurement) and ssn:Latency (defined as the time between a request to receive sensor observations and the provision of the result), among others.
Both quality-related concepts can be instantiated by specifying a quantityValue expressing the quantitative value of the quality, and an additional so:unitCode which indicates the measurement unit. The range of this property is a UN/CEFACT Common Code[9] specifying the unit in a standard format.
In addition, *value specifications* can be assigned to event properties. A value specification provides information about the expected values of an event property at run-time. The value specification depends on the type of an event property. In case of sensors, sensor observations are often provided as numerical values. To specify the value range of numerical values, we relate the concept so:QuantitativeValue to a property quality. This type is provided in the schema.org vocabulary and defines properties to define a minimum value, a maximum value and a unit code. In order to support text-based event properties which have a restricted set of expected values, we have defined the concept epa:Enumeration, which is a subconcept of so:Enumeration with an additional property to specify a single runtime value for each member of the enu-

[9] http://www.unece.org/cefact/codesfortrade/codes_index.html

meration. Enumerated values are frequently used in event processing, especially for *header properties* which often define identifiers to distinguish events of the same type coming from different sources (e.g., a vehicle identifier).

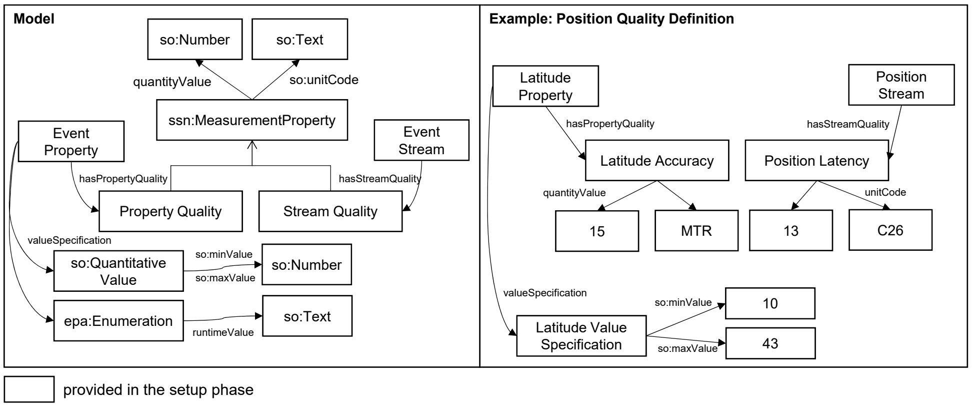


Figure 7.6 Quality Definitions: Model and Example

Vocabulary

Concepts and properties related to the definition of quality attributes are summarized in tables 7.7 and 7.8.

Table 7.7 Concepts: Quality Definitions

Concept	rdfs:subClassOf	Description
epa:StreamQuality	ssn:MeasurementProperty	A quality attribute of an event stream
epa:PropertyQuality	ssn:MeasurementProperty	A quality attribute of an event property
ssn:Frequency	epa:StreamQuality	The smallest possible time between one observation and the next [Compton et al. 2012]
ssn:Latency	epa:StreamQuality	The time between a request for an observation and the sensor providing a result [Compton et al. 2012]
ssn:Accuracy	epa:PropertyQuality	The closeness of agreement between the value of an observation and the true value of the observed quality [Compton et al. 2012]
ssn:Precision	epa:PropertyQuality	The closeness of agreement between replicate observations on an unchanged or similar quality value [Compton et al. 2012]

Continued on next page

Concept	rdfs:subClassOf	Description
		Continued from last page
ssn:Resolution	epa:PropertyQuality	The smallest difference in the value of a quality being observed that would result in perceptably different values of observation results [Compton et al. 2012]
so:QuantitativeValue	rdfs:Class	A point value or interval
epa:Enumeration	so:Enumeration	An entity defining a fixed set of possible values for an event property

Table 7.8 Properties: Quality Definitions

Property	so:domainIncludes / so:rangeIncludes	Description
epa:hasPropertyQuality	epa:EventProperty / epa:PropertyQuality	Assigns a property quality to an event stream
epa:hasStreamQuality	epa:EventStream / epa:StreamQuality	Assigns a stream quality to an event stream
epa:valueSpecification	epa:PrimitiveEventProperty epa:ListEventProperty / epa:QuantitativeValue epa:Enumeration	Assigns a value specification to an event stream
so:minValue	so:QuantitativeValue / so:Number	Indicates the minimum value of a quantitative value interval
so:maxValue	so:QuantitativeValue / so:Number	Indicates the maximum value of a quantitative value interval
epa:runtimeValue	epa:Enumeration / so:Text	Indicates the value of an event property at runtime
epa:quantityValue	ssn:MeasurementProperty / so:Number	Indicates the quantitative value of a quality attribute
so:unitCode	ssn:MeasurementProperty so:QuantitativeValue / so:Text	Assigns a measurement unit to a property

Example

The following listing 7.5 aims to clarify the usage of quality definitions in an event stream definition. First, we define two value specifications to model expected values of event properties. The event property definition representing the vehicle plate number is assigned to a specific enumeration type consisting of two specific vehicles. For

Listing 7.5 Example: Quality Definition

```
:PlateNumberProperty epa:valueSpecification :VehicleEnumeration .
:LatitudeProperty epa:valueSpecification :LatitudeValueSpecification .
:LatitudeProperty epa:hasPropertyQuality :LatitudeAccuracy .
:PositionStream epa:hasPropertyQuality :PositionLatency .

:VehicleEnumeration rdfs:subClassOf epa:Enumeration .
:Vehicle1 rdf:type :VehicleEnumeration .
:Vehicle1 epa:runtimeValue : "AB-C" .
:Vehicle2 rdf:type :VehicleEnumeration .
:Vehicle2 epa:runtimeValue : "AB-D" .

:LatitudeValueSpecification rdf:type so:QuantitativeValue .
:LatitudeValueSpecification so:minValue 10 .
:LatitudeValueSpecification so:maxValue 43 .

:LatitudeAccuracy epa:quantityValue : 15 .
:LatitudeAccuracy so:unitCode : "MTR" .

:PositionLatency epa:quantityValue : 13 .
:PositionLatency so:unitCode : "C26" .
```

the latitude property, we define a value range between 10 and 43 to indicate that latitude coordinates are always expected to be within this range. Finally, we define an accuracy value of 15 meters (identified by the UN/CEFACT Code "MTR") to the latitude property and a latency of 13 milliseconds.

7.3.4 Stream Grounding

The *stream grounding* defines technical aspects of event streams. As we aim to provide an implementation-independent description of event streams, it becomes necessary to define the protocol which is being used at run-time as an event channel as well as the run-time format in which events are represented. Model and example to define a stream grounding are illustrated in figure 7.7.
For instance, a stream could be transmitted via multiple publish/subscribe protocols such as Java Message Service (JMS)[10], MQTT[11] or it might send events to an Apache Kafka[12] broker. Also the message format might differ ranging from a broad spectrum from different text-based message formats (e.g., XML or JSON) to binary formats such as Apache Thrift[13].

[10] http://docs.oracle.com/javaee/6/tutorial/doc/bncdq.html
[11] http://mqtt.org/
[12] http://kafka.apache.org/
[13] http://thrift.apache.org/

In order to assign a protocol to an event stream definition, a subconcept of epa:TransportProtocol can be defined. For message formats, the concept epa:TransportFormat is provided.

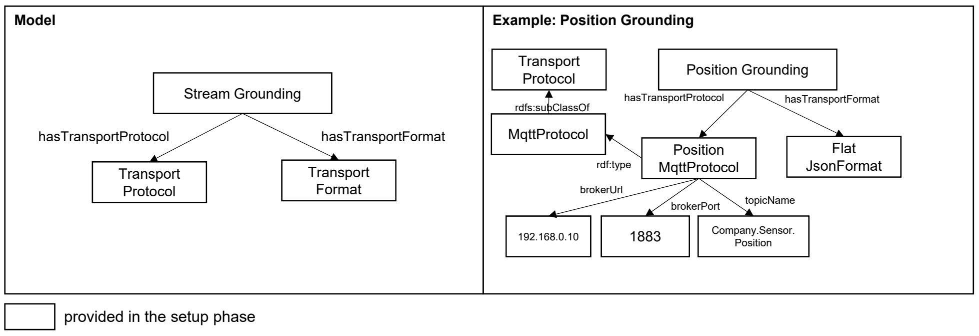


Figure 7.7 Stream Grounding: Model and Example

Vocabulary

Concepts and properties related to definition of stream groundings are summarized in tables 7.9 and 7.10.

Table 7.9 Concepts: Grounding Definitions

Concept	rdfs:subClassOf	Description
epa:TransportProtocol	rdfs:Class	An entity representing the run-time communication protocol
epa:TransportFormat	rdfs:Class	An entity representing the run-time event format

Table 7.10 Properties: Grounding Definitions

Property	so:domainIncludes / so:rangeIncludes	Description
epa:hasTransportProtocol	epa:StreamGrounding / epa:TransportProtocol	Assigns a transport protocol to a stream grounding
epa:hasTransportFormat	epa:StreamGrounding / epa:TransportFormat	Assigns a transport format to a stream grounding

Listing 7.6 Example: Stream Grounding Definition

```
:PositionStreamGrounding epa:hasTransportProtocol :PositionMQTTProtocol .
:PositionStreamGrounding epa:hasTransportFormat :PositionJsonFormat .

:PositionMQTTProtocol rdf:type :MQTTProtocol .
:MQTTProtocol rdfs:subClassOf epa:TransportProtocol .
:PositionMQTTProtocol :topicName "Company.Sensor.Position" .
:PositionMQTTProtocol :brokerUrl "192.168.0.1" .
:PositionMQTTProtocol :brokerPort 1883 .

:PositionJsonFormat rdf:type :FlatJsonFormat .
:FlatJsonFormat rdfs:subClassOf epa:TransportFormat .
```

Example

The following listing 7.6 defines a grounding for the position stream. Events are published to an MQTT server in a flat JSON-based format as illustrated in the beginning of this section.

7.4 Semantic Event Processing Agents

We start with introducing the notion of *Semantic Event Processing Agents (S-EPA)* by explaining their conceptual architecture as illustrated in figure 7.8. Similar to event producers, S-EPAs consist of a description layer and an implementation layer. The description layer defines how an EPA works. It provides information on required event streams, required static data and the supported event transformation leading to an output stream. While this description is exposed via HTTP similar to event producers, S-EPAs provide another endpoint to receive invocation graphs. An invocation graph contains information on the instantiation of an S-EPA, i.e., it defines required configuration parameters for the implementation layer needed to execute the event processing logic and is generated during the execution phase, further explained in chapter 8.
The implementation layer controls the run-time behavior of the EPA. In contrast to event producers, it consists of several runtime-independent components, i.e., components which do not depend on the actual processing logic that can be re-used among different EPAs. This includes a topic which connects to a message broker in order to consume events from a topic specified in the invocation graph. Depending on the message format of the event, an appropriate input adapter transforms this format into a generic, format-independent data structure (in our implementation, we use adapters to transform from arbitrary message formats to a *Map* data type). Output events generated by the runtime-specific event processing systems are transformed

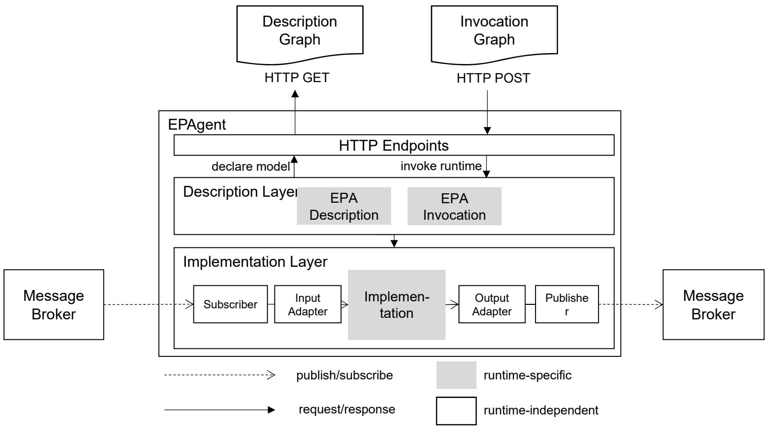


Figure 7.8 Semantic Event Processing Agents: Conceptual View

to the target output format and published to a message broker using the protocol specified in the invocation graph.

The vocabulary to describe semantic event processing agents is summarized in figure 7.9. Similar to event producers, a title and a description can be assigned to an S-EPA. In addition, an S-EPA can define *required streams*, which specify minimum requirements an incoming event stream of an instantiated S-EPA needs to provide. Besides stream requirements, required static data can be defined by relating the S-EPA to a StaticProperty.

As the specific output event stream of an S-EPA is, in most cases, not known at the time the description is generated (as an event processing agent performs a transformation of an input stream to an output stream, and the exact input stream is not yet known in the setup phase as it is selected by pattern engineers in the execution phase), we have introduced the concept of OutputStrategies. An output strategy specifies this transformation process on an abstract level. In conjunction with a specific input event stream, an expected output stream can be computed during the pipeline authoring process.

Finally, a S-EPA might support multiple transportation formats and protocols on both input (subscriber) and output (publisher) sides. To define such properties, a SupportedGrounding can be assigned to an S-EPA.

Vocabulary

Basic concepts and properties to describe Semantic Event Processing Agents are summarized in tables 7.11 and 7.12.

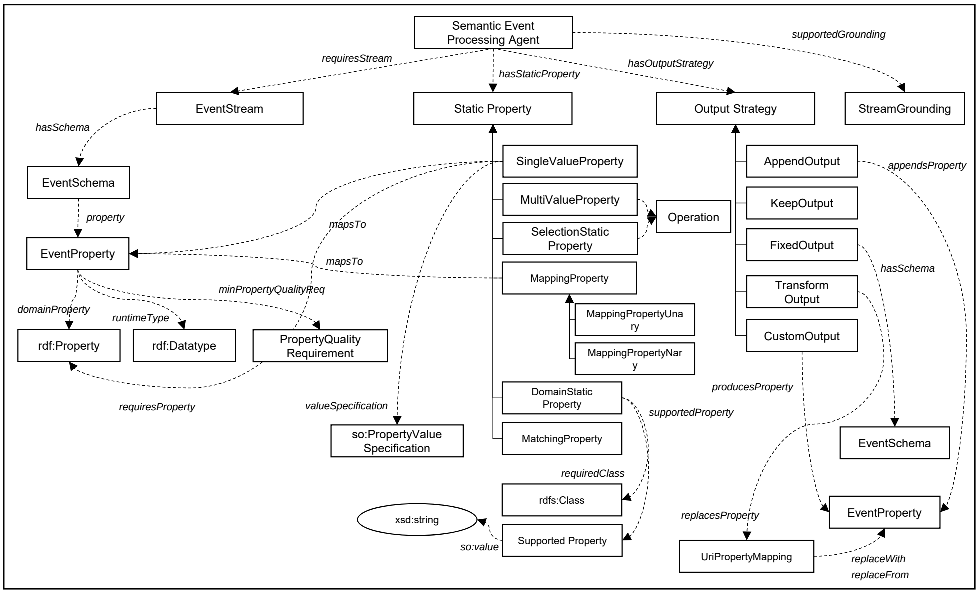


Figure 7.9 Semantic Event Processing Agents: Vocabulary

Table 7.11 Concepts: Semantic Event Processing Agents

Concept	rdfs:subClassOf	Description
epa:SemanticEventProcessingAgent	rdfs:Class	An entity which provides event processing capabilities
epa:StaticProperty	rdfs:Class	Defines user input required to instantiate an event processing agent
epa:OutputStrategy	rdfs:Class	Defines the output transformation strategy of an event processing agent

Table 7.12 Properties: Semantic Event Processing Agents

Property	so:domainIncludes / so:rangeIncludes	Description
dc:title / dc:description	epa:SemanticEventProcessingAgent epa:EventStream / so:Text	Assigns a human-readable title/description to a SEMANTIC EVENT PROCESSING AGENT or an EVENTSTREAM
epa:requiresStream	epa:SemanticEventProcessingAgent / epa:EventStream	An event stream definition required by a SEMANTIC EVENT PROCESSING AGENT
epa:hasStaticProperty	epa:SemanticEventProcessingAgent / epa:StaticProperty	Assigns a static property to a SEMANTIC EVENT PROCESSING AGENT
epa:hasOutputStrategy	epa:SemanticEventProcessingAgent / epa:OutputStrategy	Assigns an output strategy to a SEMANTICEVENT-PROCESSINGAGENT
epa:supportedGrounding	epa:SemanticEventProcessingAgent / epa:StreamGrounding	Defines a grounding a SEMANTICEVENTPROCESSIN-GAGENT supports

Example

In this section, we use the following example to illustrate the definition of an S-EPA: Based on the event producer presented in the previous section, a geofencing EPA should be developed. The goal of this EPA is to detect whether a vehicle arrives within a certain radius around a specific location. In addition, the EPA should support a second operation and indicate if a vehicles leaves a geofenced area. In order to provide this functionality, three different forms of static data input are required: 1) the location of the center of the geofence needs to be defined, 2) the radius of the geofence from the center and 3) the operation type, i.e. whether the arrival within or departure from a geofence should be detected.
As we aim to create re-usable event processing agents, the EPA should not be bound to specific vehicle position events as shown in Listing 7.1. Thus, we need to create an S-EPA which operates on any event type which provides geospatial coordinates. The following listing first defines a *Geofencing S-EPA* along with one required event stream, three static property definitions and an output strategy. The supported grounding is restricted to a MQTT protocol.

Listing 7.7 Example: S-EPA Definition

```
:GeofencingEpa rdf:type epa:SemanticEventProcessingAgent .

:GeofencingEpa dc:title "Geofencing" .
:GeofencingEpa dc:description "Performs geofencing operations on geospatial
    event types" .

```

```
:GeofencingEpa epa:requiresStream :GeospatialStream .

:GeofencingEpa epa:hasStaticProperty :GeofencingOperation .
:GeofencingEpa epa:hasStaticProperty :GeofenceCenter .
:GeofencingEpa epa:hasStaticProperty :GeofenceRadius .

:GeofencingEpa epa:hasOutputStrategy :GeofencingOutput .

:GeofencingEpa epa:supportedGrounding :MqttGrounding .
```

7.4.1 Stream Requirements

In order to describe the expected input of an S-EPA, we can specify requirements on event streams. A stream requirements can be modeled in form of an event stream template, i.e., an event stream which provides the minimum capabilities the S-EPA expects. Therefore, most concepts and properties from event streams can be re-used to define stream requirements. In addition, the vocabulary to define quality aspects is extended with concepts and properties to define minimum and maximum quality requirements. For instance, a maximum requirement assigned to a frequency-related stream quality allows to define a maximum frequency an S-EPA is able to process.

Vocabulary

Tables 7.13 and 7.14 explain additional concepts for event streams to define quality requirements.

Table 7.13 Concepts: Stream Requirements

Concept	rdfs:subClassOf	Description
epa:StreamQualityRequirement	rdfs:Class	Defines a stream quality requirement
epa:PropertyQualityRequirement	rdfs:Class	Defines a property quality requirement

Table 7.14 Properties: Stream Requirements

Property	so:domainIncludes / so:rangeIncludes	Description
epa:hasStreamQualityReq	epa:EventStream / epa:StreamQualityRequirement	Assigns a quality requirement to an event stream
epa:hasPropertyQualityReq	epa:PrimitiveEventProperty epa:ListEventProperty / epa:PropertyQualityRequirement	Assigns a property quality requirement to an event stream

Continued on next page

Property	so:domainIncludes / so:rangeIncludes	Description
		Continued from last page
epa:minStreamQuality	epa:StreamQualityRequirement / epa:StreamQuality	Defines a minimum stream quality requirement
epa:maxStreamQuality	epa:StreamQualityRequirement / epa:StreamQuality	Defines a maximum stream quality requirement
epa:minPropertyQuality	epa:PropertyQualityRequirement / epa:PropertyQuality	Defines a minimum property quality requirement
epa:maxPropertyQuality	epa:PropertyQualityRequirement / epa:PropertyQuality	Defines a maximum property quality requirement

Example

Listing 7.8 creates a stream requirement for the Geofencing S-EPA. In order to be able to provide viable results, any event stream which can be bound to the S-EPA must provide an event schema that contains a geospatial location. In the same way, quality restrictions could be assigned to an event stream.

Listing 7.8 Example: Stream Requirement Definition

```
:GeospatialStream rdf:type epa:EventStream .

:GeospatialStream epa:hasSchema :GeospatialSchema .
:GeospatialSchema epa:hasEventProperty :SomeLatitudeProperty .
:GeospatialSchema epa:hasEventProperty :SomeLongitudeProperty .

:SomeLatitudeProperty epa:domainProperty geo:lat .
:SomeLongitudeProperty epa:domainProperty geo:long .
```

7.4.2 Static Properties

Static properties define, besides stream restrictions, further data an S-EPA requires for its instantiation. Static properties are therefore not part of the event payload, but are typically user-defined during pipeline authoring, or are related to concepts defined in the knowledge modeling task.

Single-value Properties The most basic static property type are single-value properties. These properties can be used to define a required user input in form of a single data value. As illustrated in figure 7.10, the concept SingleValueProperty allows to define such data requirements along with a requiredDatatype specifying the expected data type. In addition, a so:PropertyValueSpecification, which allows to define

value ranges in form of minimum, maximum and step values, can be assigned to a single-value property. Furthermore, an event property can be assigned to a SingleValueProperty by using the mapsTo relation. This allows to restrict the value range of the required value to the value range of the assigned event property of the incoming event stream.
Example. *An S-EPA provides filter capabilities on text-based event properties. A filter keyword is required, this can be modeled as a single-value property.*

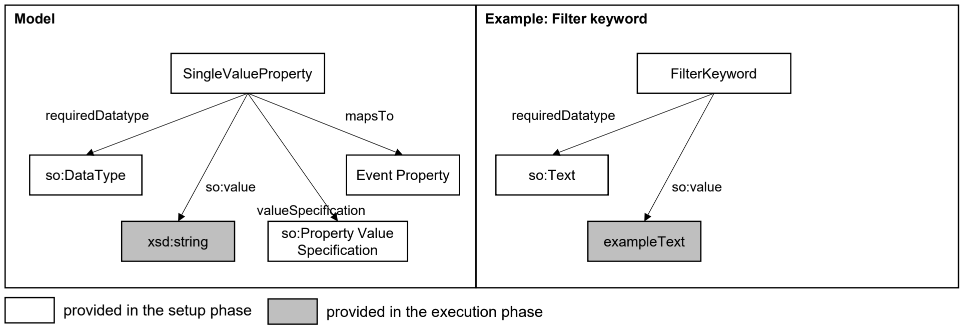


Figure 7.10 Single-value Properties: Model and Example

Multiple values and selections In many cases, an S-EPA might provide multiple operations or alternative configurations users can choose from during pipeline definition. To support these use cases, we have created the concepts MultiValueProperty and SelectionStaticProperty (illustrated in figure 7.11). Both concepts can be related to a number of *options* specifying an operation alternative or configuration type users can select from. The difference of both elements are the underlying semantics: While MultiValueProperties allow to select multiple values from the list of available options, SelectionStaticProperties require users to select exactly one available options.
Example. *The same filter S-EPA as described above is able to filter events based on an exact match, i.e., the property equals the provided filter keyword, or an inexact match, i.e., the property value contains the provided keyword. This choice can be modeled as a SelectionStaticProperty.*

Domain properties and background knowledge By now, we have focused on static data requirements which are only required for a specific user input. In contrast, such data frequently comes from knowledge bases. In order to provide a mechanism to align requirements of event processing agents with concepts and instances defined in a knowledge base, we created the concept DomainStaticProperty. A DomainStaticProperty defines a required resource of an external vocabulary by providing a

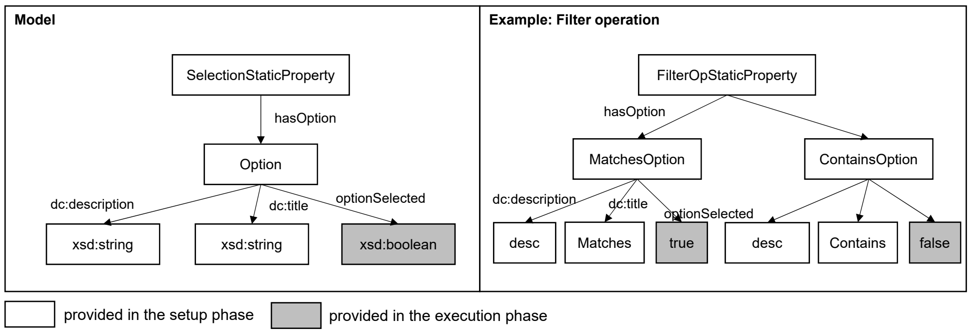


Figure 7.11 Selection Static Properties: Model and Example

requiredClass and one or more supportedProperties. During the pipeline authoring process, an RDF-based knowledge base is first queried for instances belonging to the concept defined in requiredClass. Afterwards, for each instance it is verified whether they provide all rdf:Property assignments that are defined as supportedProperties. The usage of domain static properties illustrated in figure 7.12.

***Example**. We might be interested in filtering the VehiclePositionStream for specific plate numbers. This could be easily done by specifying a single-value property which allows users to enter a vehicle plate number as a filter condition. However, this approach is error-prone as pattern engineers need to look up the list of available vehicles in order to manually enter the plate number. By using a domain static property, we can link the required filter condition to a Vehicle concept in the knowledge base and additionally define a hasPlateNumber property the Vehicle concept need to provide. As a result, during the pipeline authoring process the pattern engineer is provided with a search interface to look up suitable instances in the knowledge base.*

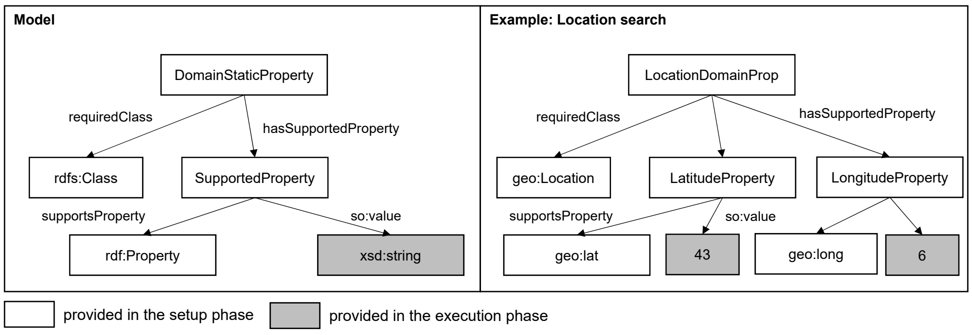


Figure 7.12 Domain Static Properties: Model and Example

MappingProperty Stream requirements define properties a specific event stream needs to provide. In many cases, event properties defined as stream requirements need to be used at run-time to parameterize the event processing logic. In order to map an event property defined as a stream requirement to a specific event property of an event stream during the invocation of an element, we provide the concept MappingProperty. A mapsFrom relation assigns a mapping property to an event property of a stream requirement (see figure 7.13).
We provide two specializations of mapping properties: MappingPropertyUnary defines a mapping from an event property in a stream requirement to a single event property of a specific event stream, while a MappingPropertyNary defines a mapping to multiple event properties of a specific event stream.
***Example**. We illustrate the usage of MappingProperties based on a basic text filter S-EPA presented in the beginning of this section. The only stream requirement of such a component is typically the occurrence of an event property in a bound event stream which provides a text-based data type (so:Text). If such a stream contains more than one event property matching this criterion, users at pipeline authoring time need to select a specific event property the filter operation should be performed upon. The mapping between this specific property and the requirement is modeled as a unary mapping property.*

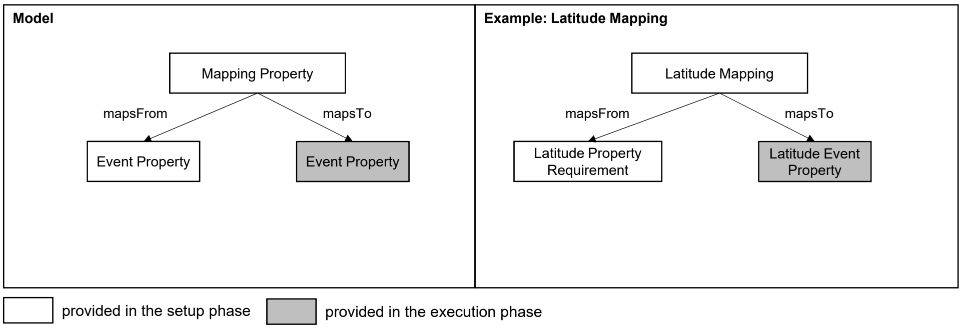


Figure 7.13 Mapping Properties: Model and Example

MatchingProperty MatchingStaticFeatures (figure 7.14) are a static property used for S-EPAs with more than one input channel, i.e., S-EPAs which require for at least two input streams. Typical use cases for such S-EPAs are event-at-a-time operators such as detection of co-occurrences or sequences of events. In such use cases, events are frequently correlated based on a common event property value. Incoming streams are then partitioned, i.e., results are computed separately per property value. In order to define a matching between property values of two streams that should provide the same value, MatchingProperties can be used. MatchingProperties can define two relations indicating one event property from each incoming event stream which should provide the same values.

Example*. We consider a Sequence S-EPA which detects the sequential occurrence of two events within a specified time windows. This component can be used to detect the arrival of a vehicle at a warehouse followed by the departure (in the form Arrival followed by Departure within 5 minutes). In order to indicate that the correlation between arrival and departure should be matched based on the same vehicleId, a matching property can be used.*

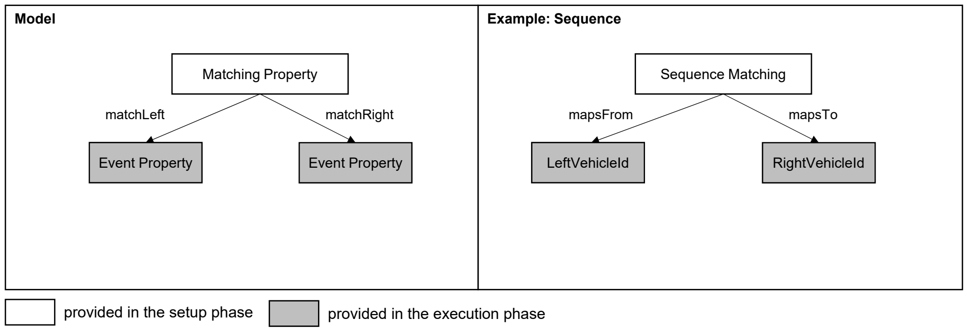


Figure 7.14 Matching Properties: Model and Example

Vocabulary

The complete vocabulary provided to model static properties is further detailed in tables 7.15 and 7.16.

Table 7.15 Concepts: Static Properties

Concept	rdfs:subClassOf	Description
epa:SingleValueProperty	epa:StaticProperty	An entity representing required user input in form of a single data value
epa:MultiValueProperty	epa:StaticProperty	An entity representing required user input in form of multiple data values
epa:SelectionStaticProperty	epa:StaticProperty	An entity representing required user input in form of a single data value based on a fixed set of selection options
epa:DomainStaticProperty	epa:StaticProperty	An entity representing required user input from instances of a knowledge base

Continued on next page

Concept	rdfs:subClassOf	Description
		Continued from last page
epa:MappingProperty	epa:StaticProperty	Defines a mapping between properties from stream requirements and properties from event streams
epa:MappingPropertyUnary	epa:MappingProperty	Defines a mapping from one event property from a stream requirement to one event property from an event stream
epa:MappingPropertyNary	epa:MappingProperty	Defines a mapping from one event property from a stream requirement to multiple event properties from an event stream
epa:MatchingProperty	epa:StaticProperty	Defines a relationship between two event properties from different event streams
epa:Option	rdfs:Class	An entity representing a selection option
epa:SupportedProperty	rdfs:Class	Defines an RDF:PROPERTY of a concept in the knowledge base that is supported by a domain static property.

Table 7.16 Properties: Static Properties

Property	so:domainIncludes / so:rangeIncludes	Description
epa:requiredDatatype	epa:SingleValueProperty / so:DataType	Defines the required data type of a SINGLEVALUEPROPERTY
epa:hasOption	epa:MultiValueProperty epa:SelectionStaticProperty / epa:Option	Assigns a selection option to a MULTIVALUEPROPERTY or an SELECTIONSTATICPROPERTY
epa:requiredClass	epa:DomainStaticProperty / rdfs:Class	Indicates the required RDFS:CLASS of a required instance from the knowledge base
epa:hasSupportedProperty	epa:DomainStaticProperty / epa:SupportedProperty	Assigns a SUPPORTEDPROPERTY to a DOMAINSTATICPROPERTY
epa:requiredProperty	epa:SupportedProperty / rdf:Property	Indicates a required RDF:PROPERTY of a required instance from the knowledge base
epa:mapsFrom	epa:MappingProperty / epa:EventProperty	Assigns a property requirement to a mapping property
epa:mapsTo	epa:MappingProperty epa:SingleValueProperty / epa:EventProperty	Assigns an event property to a mapping property or a single value property
epa:valueSpecification	epa:SingleValueProperty / so:PropertyValueSpecification	Assigns a value specification to a single value property
		Continued on next page

Property	so:domainIncludes / so:rangeIncludes	Description
		Continued from last page
epa:matchLeft	epa:MatchingProperty / epa:EventProperty	Assigns an event property from the first event stream to a matching property
epa:matchRight	epa:MatchingProperty / epa:EventProperty	Assigns an event property from the second event stream to a matching property

Example

As already mentioned, the *Geofencing S-EPA* requires the selection of a geofencing operation, a location which represents the center of the geofence and a radius indicating the size of the geofence. In addition, we define two mapping properties linking to the required event properties defined as stream restrictions. Human-readable labels which identify the purpose of static properties to pipeline authors are omitted. Listing 7.9 illustrates the usage of these static properties:

Listing 7.9 Example: Static Property Definition

```

:GeofencingEpa epa:hasStaticProperty :GeofencingOperation .
:GeofencingEpa epa:hasStaticProperty :GeofenceCenter .
:GeofencingEpa epa:hasStaticProperty :GeofenceRadius .
:GeofencingEpa epa:hasStaticProperty :LatitudeMapping .
:GeofencingEpa epa:hasStaticProperty :LongitudeMapping .

:GeofencingOperation rdf:type epa:SelectionStaticProperty .
:GeofencingOperation epa:hasOption :ArrivesOption .
:ArrivesOption dc:title "Arrive" .
:GeofencingOperation epa:hasOption :DepartsOption .
:DepartsOption dc:title "Depart" .

:GeofenceCenter rdf:type epa:DomainStaticProperty .
:GeofenceCenter epa:requiredClass geo:Location .
:GeofenceCenter epa:hasSupportedProperty :LatitudeSupported .
:GeofenceCenter epa:hasSupportedProperty :LongitudeSupported .

:LatitudeSupported rdf:type epa:SupportedProperty .
:LatitudeSupported epa:supportedProperty geo:lat .
:LatitudeSupported so:valueRequired true .

:LongitudeSupported rdf:type epa:SupportedProperty .
:LongitudeSupported epa:supportedProperty geo:lat .
:LongitudeSupported so:valueRequired true .

:GeofencingRadius rdf:type epa:SingleValueProperty .
:GeofencingRadius epa:requiredDatatype so:Integer .
:GeofencingRadius epa:valueSpecification :RadiusValueSpecification .

:RadiusValueSpecification rdf:type so:PropertyValueSpecification .
:RadiusValueSpecification so:minValue 0 .
```

```
:RadiusValueSpecification so:maxValue 200 .
:RadiusValueSpecification so:step 5 .

:LatitudeMapping rdf:type epa:MappingPropertyUnary .
:LatitudeMapping epa:mapsFrom :LatitudeProperty .

:LongitudeMapping rdf:type epa:MappingPropertyUnary .
:LongitudeMapping epa:mapsFrom :LongitudeProperty .
```

7.4.3 Output Strategy

By providing an output strategy, developers are able to describe the intended output of an S-EPA. As already explained, in most cases it is not possible to define the specific output event stream in the setup phase, as the exact output stream depends on the input stream selected in the execution phase. Therefore, the output depends on both the schema of an input event stream and the transformation performed by the S-EPA itself. In order to be able to describe this transformation, we have developed the concept of *output strategy*. Output strategies allow developers to define the transformation decoupled from specific business logic. In addition, at pipeline execution time the output stream can be easily computed during pipeline definition and therefore allows to form pipelines consisting of heterogeneous processing elements.
In general, five different output strategies are part of our vocabulary to describe S-EPAs:

KeepOutput (Figure 7.15) The most basic output strategy is a KeepOutput strategy. This strategy allows to define filter operations, where the output stream corresponds to the specific input stream of a processing element, i.e., it provides the same event schema. This output strategy does not require for any further configuration. If an S-EPA has more than one input event stream, a KeepOutput produces an output event which contains the union of all event properties from all input streams.

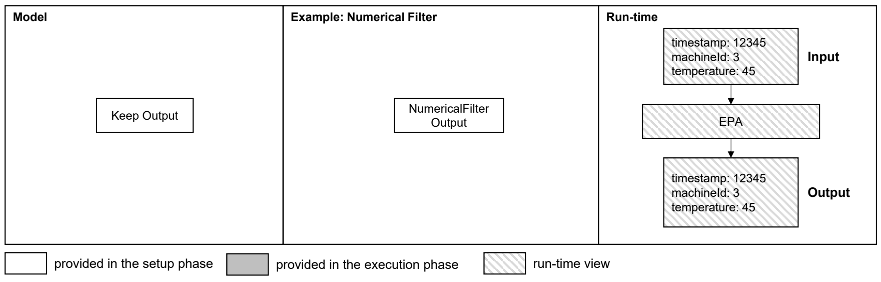


Figure 7.15 Keep Output: Model and Example

AppendOutput (Figure 7.16) S-EPAs using an AppendOutput strategy are typically processing elements which correspond to the *Enrich EPA*, i.e., one or more event properties are added to an existing input event schema. An AppendOutput can be defined by providing one or more additional event properties by using the relation appendsProperty.

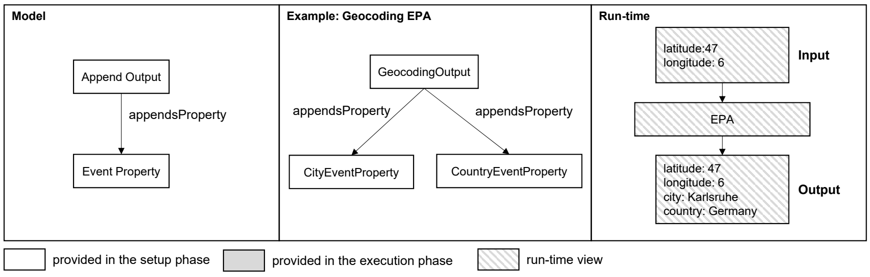


Figure 7.16 Append Output: Model and Example

FixedOutput (Figure 7.17) FixedOutput strategies do not depend on the schema of an input event stream. The output event schema is solely defined by developers during the setup phase. For instance, an activity recognition algorithm which identifies physical activities based on an acceleration sensor typically outputs an event schema which is known in advance and can therefore provide a FixedOutput strategy.

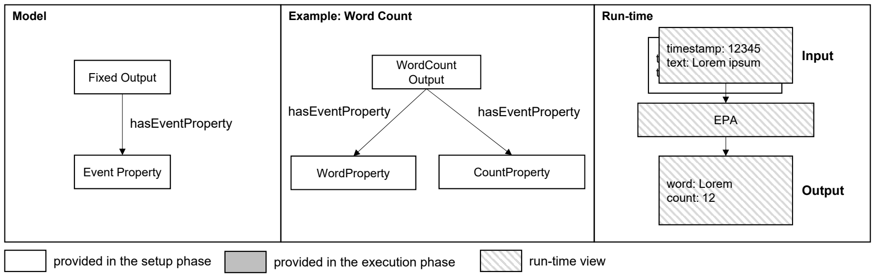


Figure 7.17 Fixed Output: Model and Example

TransformOutput (Figure 7.18) EPAs belonging to the *Transformation* category transform one or more event properties from an input stream to one or more properties in the output stream. In general, transformations potentially affect multiple aspects of an event schema, e.g., the run-time name in order to rename properties for data harmonization purposes, the run-time type or the domain property. An S-EPA which defines a transformation-based output strategy must therefore be able to express *supported* transformations. Based on these definitions and an inbound event stream, specific transformation options can then be computed and presented to users. To support transformations in our vocabulary, a TransformOutput can be created. For each property restriction the S-EPA has defined, an UriPropertyMapping can be assigned. This mapping requires a replaceFrom property which represents a property restriction, and additional properties to define supported transformations (renamingAllowed, typeCastAllowed and domainPropertyCastAllowed).

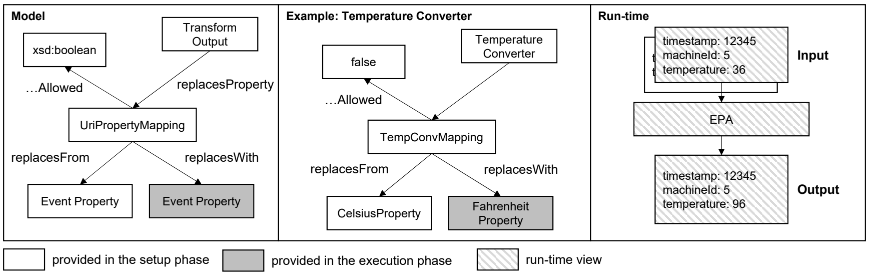


Figure 7.18 Transform Output: Model and Example

CustomOutput (Figure 7.19) Finally, an S-EPA might leave the choice which input requirements from the input streams should be available in the output stream up to users. In this case, an output schema can be manually defined during the pipeline authoring process.

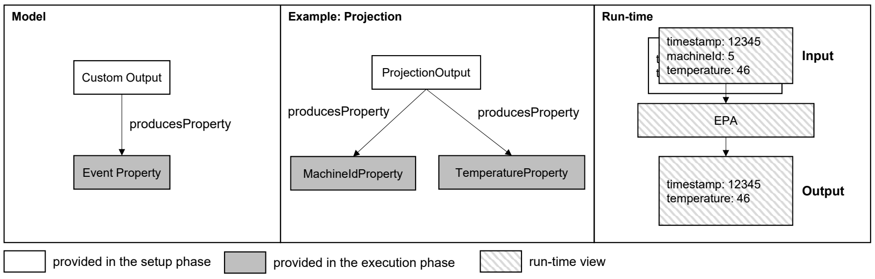


Figure 7.19 Custom Output: Model and Example

Table 7.17 Concepts: Output Strategies

Concept	rdfs:subClassOf	Description
epa:AppendOutput	epa:OutputStrategy	An entity representing an append output strategy
epa:CustomOutput	epa:OutputStrategy	An entity representing a custom output strategy
epa:FixedOutput	epa:OutputStrategy	An entity representing a fixed output strategy
epa:KeepOutput	epa:OutputStrategy	An entity representing a keep output strategy
epa:TransformOutput	epa:OutputStrategy	An entity representing a transform output strategy
epa:UriPropertyMapping	rdfs:Class	An entity representing a mapping between two event properties which should be replaced with each other

Table 7.18 Properties: Output Strategies

Property	so:domainIncludes so:rangeIncludes	Description
epa:appendsProperty	epa:AppendOutput epa:EventProperty	Assigns an event property which is appended to an input stream
epa:hasEventProperty	epa:FixedOutput epa:EventProperty	Defines an event property that is part of an output stream
epa:producesProperty	epa:CustomOutput epa:EventProperty	Defines an event property from the input stream that is kept in the output stream
epa:replacesFrom	epa:UriPropertyMapping epa:EventProperty	Assigns an event property which should be replaced by a transform output strategy
epa:replacesWith	epa:UriPropertyMapping epa:EventProperty	Defines the event property that should replace an event property from an input stream
epa:replacesProperty	epa:TransformOutput epa:UriPropertyMapping	Assigns an UriPropertyMapping to a transform output strategy
epa:typeCastAllowed	epa:UriPropertyMapping xsd:boolean	Determines whether the output strategy allows a type cast
epa:domainPropertyCastAllowed	epa:UriPropertyMapping xsd:boolean	Determines whether the output strategy allows a domain property cast
epa:renamingAllowed	epa:UriPropertyMapping xsd:Boolean	Determines whether the output strategy allows to rename an event property

Listing 7.10 Example: Output Strategy Definition

```

:GeofencingOutput rdf:type epa:AppendOutput .

:GeofencingOutput epa:appendsProperty :EnterTimeProperty .
:EnterTimeProperty rdf:type epa:PrimitiveEventProperty .
:EnterTimeProperty epa:runtimeName "timeOfArrival" .
:EnterTimeProperty epa:runtimeType so:Text .
:EnterTimeProperty epa:domainProperty so:DateTime .
```

Example

We show the usage of output strategies in Listing 7.10 based on an enriched output strategy the *Geofencing S-EPA* supports: For each vehicle which enters or leaves the defined geofence, an output event is generated which contains the original event payload of the input stream and, in addition, the time the vehicle has crossed the geofence.

7.5 Semantic Event Consumers

Event consumers represent sinks in an event processing network. Sinks often provide connections to other third party systems, such as databases, external message brokers or visualization components. Figure 7.20 illustrates the conceptual architecture of *Semantic Event Consumers (S-EC)*. In general, an S-EC provides similar characteristics as S-EPAs. Stream restrictions define requirements an input event stream needs to provide, and static properties needed to configure the consumer implementation can be defined. However, in contrast to S-EPAs, events are not forwarded to other processing elements within an event processing network, i.e., no output event stream is produced.
Therefore, the vocabulary to define S-ECs can mostly be re-used from already introduced concepts and relations to define S-EPAs, but do not offer a possibility to define an output strategy.

7.6 Tools

In order to support the development of S-EPs, S-EPAs and S-ECs, we have developed extensive tool support which aims at reducing the programming effort to define both the description layer and implementation layer of processing elements. In this section, we introduce these tools and briefly explain their usage.

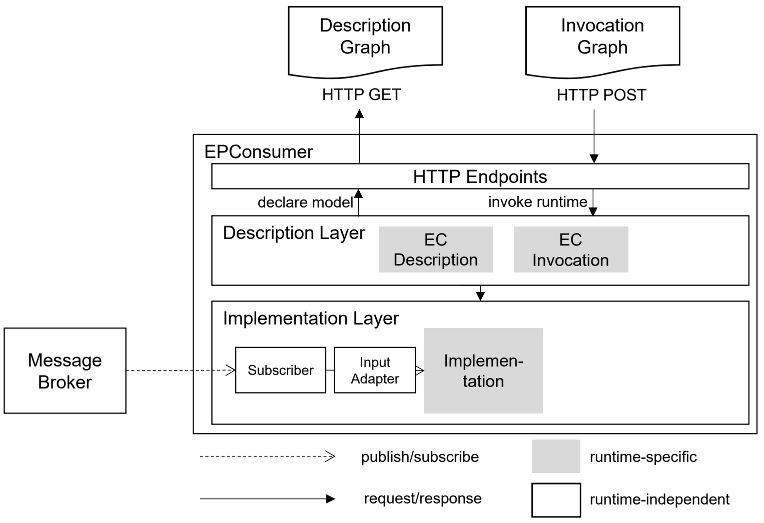


Figure 7.20 Semantic Event Consumers: Conceptual View

7.6.1 Model Generation

Model generation refers to the first task of the setup phase, namely pipeline element modeling. From a conceptual point of view, pipeline element modeling requires developers to build the description layer for processing elements by providing an appropriate instantiation of the concepts introduced in the previous section.
In general, the description layer can also be built using standard tooling available for ontology modeling. However, one of our design goals was to enable developers to build models without the need for any knowledge related to semantic web technologies. We believe this is an important cornerstone as many developers concerned with the development of event-based applications are not trained for semantic web programming. This approach aims to reduce the implementation effort needed to make existing event processing logic available using our model.
Developer support to create event processing logic is available in form of a Software Development Toolkit (SDK) and a graphical model editor.

SDK

The SDK is available as a Apache Maven[14] artifact and provides high-level programming access for creating and deploying semantic descriptions as well as methods

[14] http://maven.apache.org

to retrieve configuration parameters from invocation graphs. In general, the SDK supports especially three different use cases:

- **Model Definition.** Models can be created without the need for RDF and semantic web technologies in general. A pure Java interface can be used to create descriptions by using a Java-to-RDF mapper. This part of the SDK supports the full vocabulary presented in the previous sections and includes consistency checking of models in order to assist developers in creating the description layer.
- **Deployment.** Based on these Java-based models, description graphs are automatically generated. In addition, the SDK provides methods to support the deployment process, e.g., the generation of RESTful endpoints which provide the description of a model and retrieve invocation graphs. Several deployment options are supported, so that implementations can be hosted standalone using an embedded web server or in an already existing servlet container.
- **Configuration**. For S-EPAs and S-ECs, the SDK provides endpoints which are prepared to receive and transform invocation graphs. These graphs contain parameters needed to configure the underlying implementation layer. By using the SDK, high-level methods for retrieving these parameters can be used.

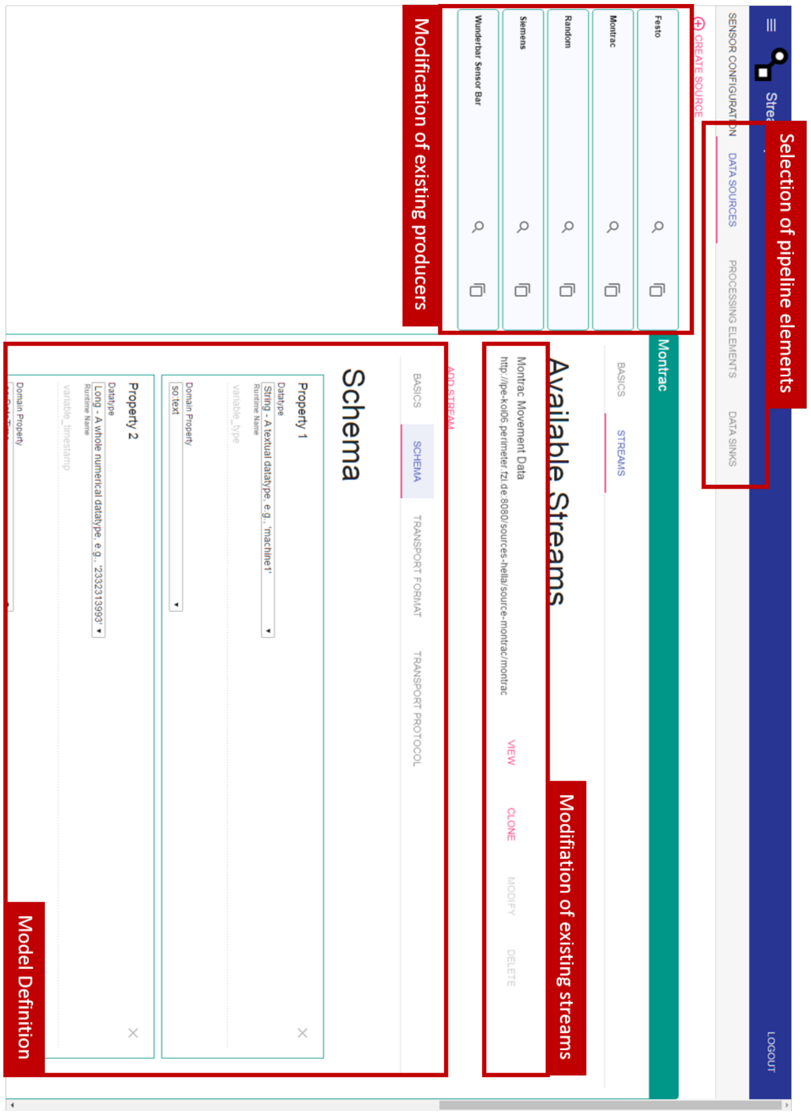
Appendix A illustrates the usage of our SDK to create a description for the Geofencing S-EPA introduced in section 7.4.

Model Editor

Instead of the SDK, we also provide graphical tool support to create pipeline elements. The model editor is a web-based interface which allows to create descriptions for event producers, event processing agents and event consumers. The model editor and its features are illustrated in Figure 7.21. In general, the editor leverages users to inspect existing pipeline elements and to create new elements. New elements can either be created from scratch, or existing elements can be cloned, i.e., the description of another element can be extended and/or modified.

Definition of new elements is implemented as a guided step-by-step process, which is described below:

1. **Pipeline Element Selection.** As illustrated in figure 7.21, the first task is to select a pipeline element which should be created. By selecting "Data Sources" from the top-level menu, configuration options for event producers and streams become visible.
2. **Producer Definition.** Two ways exist for creating new event producers: An element can be selected from the list of existing producers. The “clone” feature creates a new element in the background and copies the description of the selected element, which is useful for producers which have a description similar to an already existing element. In other cases, a new producer must be defined. The editor first requires users to provide basic settings (such as title and



Montrac Movement Data
http://ipe-koi06.perimeter.fzi.de:8080/sources-hella/source-montrac/montrac

String - A textual datatype, e.g., 'machine1'
variable_type
so:text

Long - A whole numerical datatype, e.g., '2332313993'
variable_timestamp

Figure 7.21 Model Editor: Features

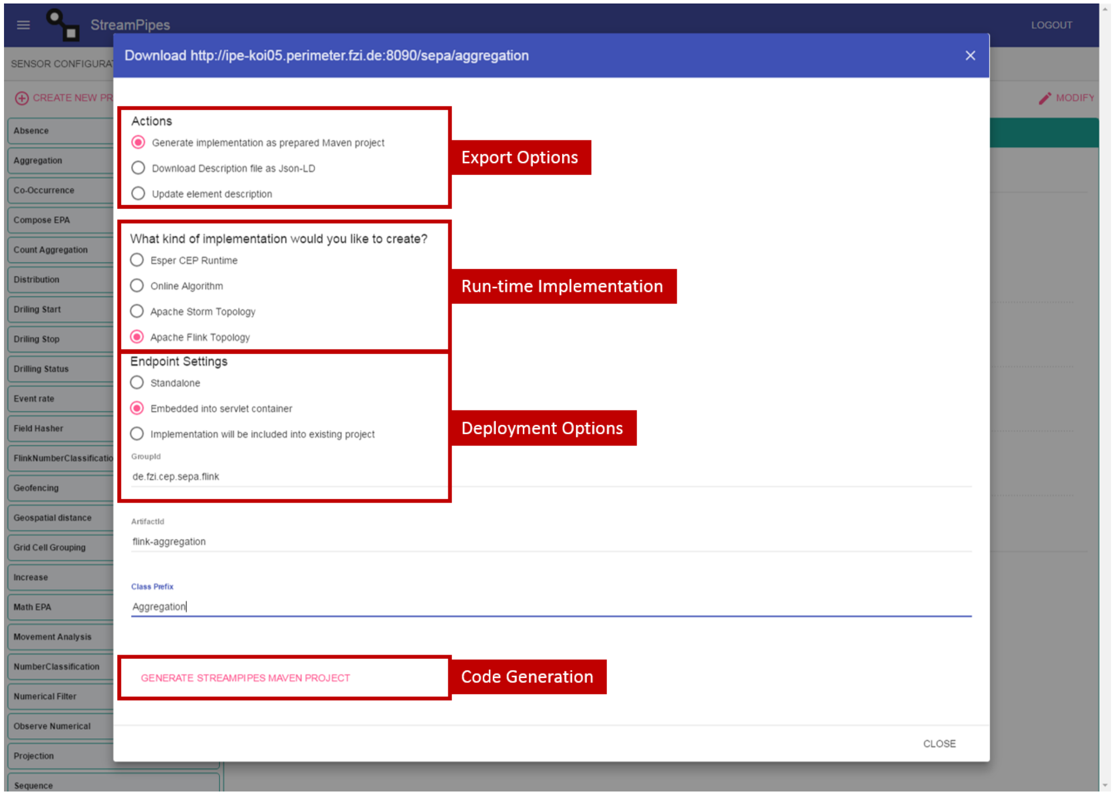
description) for the producer, which corresponds to the vocabulary presented in section 7.3.

3. **Stream Definition.** Afterwards, streams can be assigned to the producer. The definition of event streams is a guided process starting with basic descriptions. Afterwards, event schema, transport protocol and transport format need to be provided by users. The event schema can be built by adding an arbitrary number of event properties including their runtime name, type and assigned domain property, while the system allows to re-use properties which have been previously defined in the knowledge base.
4. **Export.** After the model has been created, it can be exported in several ways. The dialog which allows users to choose from multiple export options is shown in figure 7.22. The first task is to select an export type. Three different export types are offered: First, an implementation can be generated, which creates programming code packaged as a Maven project which contains all required instances to further refine the model or to implement the producer-specific implementation layer. Second, the description graph can be exported as a JSON-LD document, which can subsequently be embedded into existing software modules. Third, if an implementation is not needed (e.g., as streams are already published to a message broker), the description can be directly imported, so that the pipeline element is immediately available for pattern engineers in the execution phase. After selection of an export type, a run-time implementation can be chosen. For event processing agents, implementations for multiple event processing systems are available. For event producers, the resulting implementation provides interfaces in order to implement connections to external systems acting as event sources. Afterwards, a deployment option can be selected. *Standalone* deployment generates a software project with an embedded web server capable to host the element description and to receive invocation graphs. An *embedded* deployment creates an implementation which can be deployed in an existing servlet container such as Apache Tomcat[15]. Finally, the export can be started, which triggers a code generation module.

Runtime-Wrapper

While the SDK primarily focuses on the description layer providing high-level access to description graphs, run-time wrappers focus on the implementation layer and encompass its implementation-specific part. We provide tool support for four different run-time implementations. We've selected these implementations described in more detail below in order to show the ability of our approach to support different programming models: Declarative languages, which are often part of event processing systems focusing on pattern detection, distributed systems, which focus on processing of event

[15] http://tomcat.apache.org



Download http://ipe-koi05.perimeter.fzi.de:8090/sepa/aggregation

Actions
- [✓ Generate implementation as prepared Maven project]
- Download Description file as Json-LD
- Update element description

What kind of implementation would you like to create?
- Esper CEP Runtime
- Online Algorithm
- Apache Storm Topology
- [✓ Apache Flink Topology]

Endpoint Settings
- Standalone
- [✓ Embedded into servlet container]
- Implementation will be included into existing project

GroupId: de.fzi.cep.sepa.flink

ArtifactId: flink-aggregation

Class Prefix: Aggregation

Figure 7.22 Model Editor: Export

streams with high throughput in a cluster and standalone algorithms which do not rely on a specific event processing system, but implement their own programming model. Although the wrappers we have developed are implemented in Java, our approach is not bound to a specific programming language as our model generally relies on the exchange of RDF-based graphs.
The wrappers described below use a common run-time system which encapsulates implementation-independent logic. In order to show the independence of our approach in terms of specific run-time communication protocols and formats, support for multiple message brokers and converters for several event formats is provided. On the protocol side, we have developed implementations for MQTT, JMS, Kafka and Websocket. Concerning event formats, implementations for JSON, XML and several Apache Thrift-based formats are available.

Esper Wrapper Esper[16] is a complex event processing engine with a strong focus on pattern detection. Event processing logic in Esper is defined using a declarative domain-specific language called *EPL*. Our Esper wrapper provides several helper functions to create EPL descriptions from invocation graphs in a re-usable way by abstracting from specific event streams bound to an EPL declaration.

Flink Wrapper The Flink[17] wrapper encapsulates a run-time implementation for Apache Flink programs. In contrast to Esper, Flink focuses on distributed stream processing and therefore follows a different programming model. A Flink program must be provided as a packaged jar file which contains all dependencies required by the program. This file is submitted to a *JobManager*, where it is distributed to *TaskManagers* which execute the application logic. A single Flink program can be executed on multiple TaskManagers at the same time, enabling parallel execution of the event processing application.
Our Flink wrapper provides support to write Flink programs by taking care of event subscription, publishing, message transformation and deployment, allowing developers to implement specific implementation logic from a single interface. At invocation time, the program is parameterized based on the configuration retrieved from the invocation graph and submitted to the JobManager.

Storm Wrapper Apache Storm[18] is another system supporting distributed processing of event streams. A Storm topology logically consists of a set of *spouts* acting as event producers and a set of *bolts* acting as event processing agents. A storm topology needs to be submitted to a master node called *Nimbus*. This nodes distributes the topology to multiple *worker nodes*, where the it can be executed.

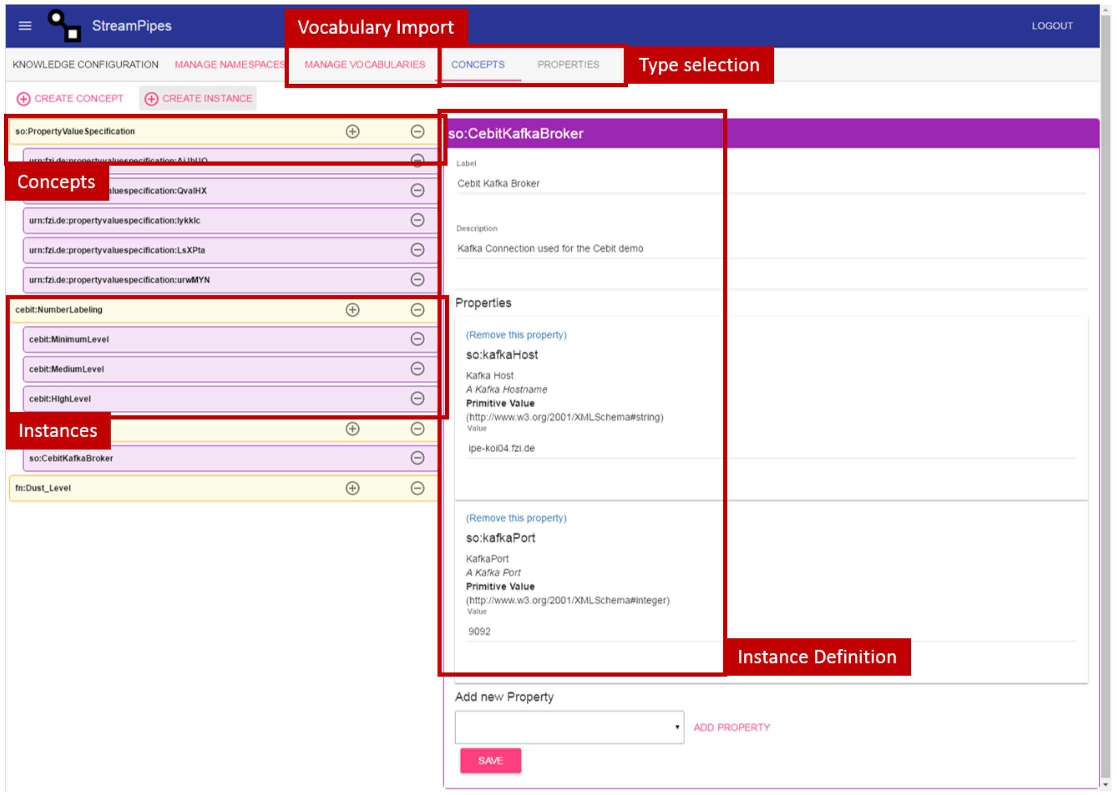
[16] http://esper.codehaus.org
[17] http://flink.apache.org
[18] http://storm.apache.org

In contrast to Flink, a Storm topology needs to be packaged before it is submitted to the Nimbus node. In order to allow the development of re-usable elements for Storm, an invocation graph is stored in a database. At deployment time, a graph id is passed to the topology which allows to retrieve the invocation graph from the database. This enables us to configure the topology at run-time.

7.6.2 Knowledge Modeling

Finally, tool support for knowledge modeling is provided. The knowledge editor is a web-based component embedded into the StreamPipes web application. Its main purpose is to capture knowledge which is re-used across multiple processing pipelines, not to provide a complete knowledge base for the domain of interest.
The knowledge editor is illustrated in figure 7.23. In general, the editor follows an approach known from existing ontology editors. It provides ways to create concepts, relations between concepts and instances. However, higher-level support for users that are not experts in the usage of semantic web tools is provided. For instance, users are able to define in which concepts a property can be *used*. This corresponds to the creation of a data model which describes the intended use of a property. Although this approach differs from most knowledge modeling approaches, which assume an open world assumption and therefore rather define to which concepts a property *belongs*, it is more intuitive for the targeted user roles to define a model based on assigned properties.
Modeling of domain knowledge using the editor can be performed as follows:

1. **Vocabulary Import.** If needed, existing RDF vocabularies can be imported into the knowledge base.
2. **Property definition.** The knowledge editor provides an interface to create properties and to define their range. Multiple options are available for range definition, for instance, a range can be a primitive type (similar to datatype properties in OWL), a fixed set of values (by applying schema.org's Enumeration concept) or a quantitative value representing numerical properties with an additional range of values such as minimum, maximum and step. Although such functionality can also be provided by existing ontology modeling tools, our editor provides high-level concepts targeted at modeling knowledge that can be re-used in pipeline definitions.
3. **Concept definition.** Concepts can be created by providing a label, a description and a set of properties which should be assigned to the concept. Internally, we represent property assignments by adding a so:domainIncludes statement to a property.
4. **Instance definition.** For each concept, instances can be created. The editor receives all properties which have a so:domainIncludes property assigned to the concept and renders forms which allow users to provide the required values.



so:PropertyValueSpecification
urn:fzi.de:propertyvaluespecification:AUhUO
urn:fzi.de:propertyvaluespecification:QvalHX
urn:fzi.de:propertyvaluespecification:lykklc
urn:fzi.de:propertyvaluespecification:LsXPta
urn:fzi.de:propertyvaluespecification:urwMYN
cebit:NumberLabeling
cebit:MinimumLevel
cebit:MediumLevel
cebit:HighLevel
so:CebitKafkaBroker
fn:Dust_Level

so:CebitKafkaBroker

Label
Cebit Kafka Broker

Description
Kafka Connection used for the Cebit demo

Properties

(Remove this property)
so:kafkaHost
Kafka Host
A Kafka Hostname
Primitive Value
(http://www.w3.org/2001/XMLSchema#string)
Value
ipe-koi04.fzi.de

(Remove this property)
so:kafkaPort
KafkaPort
A Kafka Port
Primitive Value
(http://www.w3.org/2001/XMLSchema#integer)
Value
9092

Figure 7.23 Knowledge Editor: Features

7.7 Summary

This chapter introduced concepts supporting the setup phase. The first contribution presented in this chapter are conceptual models that define Semantic Event Producers (S-EP), Semantic Event Processing Agents (S-EPA) and Semantic Event Consumers (S-EC). Each of these elements consists of a description layer, specifying properties of event producers, processing agents and consumers and an implementation layer, specifying the actual implementation logic. The description layer exposes these properties using standard RESTful interfaces and therefore allows an element description to be made available in the web.

The second contribution is associated with the description itself. We provided an RDF-based vocabulary that allows to define event processing building blocks. Related to event producers, the vocabulary comprises the definition of event streams by providing concepts and properties to assign event schemas, quality-oriented properties and grounding information. Furthermore, we proposed a vocabulary to model event processing agents. This model takes into account stream requirements, as well as static properties expressing input required from users in the execution phase. In addition, the concept of output strategies allows developers to express the transformation an S-EPA performs on an input event stream.

Finally, tools supporting both main processes in the setup phase were introduced.

8 Execution Phase

The last chapter has already introduced a vocabulary used to define capabilities of pipeline elements and an architecture to publish these as a description graph. In this chapter, we move on to the question how processing pipelines can be *authored* and *executed*. After a walkthrough in section 8.1, we define the notion of processing pipelines and explain the usage of invocation graphs in section 8.2. Section 8.3 focuses on the second task in the execution phase and describes the pipeline authoring process. Finally, the pipeline deployment task is described.

8.1 Walkthrough

After pipeline identification, the execution phase primarily deals with the definition and execution of processing pipelines. Processing pipelines define a data flow between the pipeline elements defined in the previous chapter. Building upon the conceptual architecture of pipeline elements, figure 8.1 illustrates our approach. The *application layer* shows the conceptual architecture for individual pipeline elements. The RDF description each element offers after it has been deployed within the setup phase can be imported into a repository as part of the *management layer*, which makes the element available in the *modeling layer*. The modeling layer itself consists of a graphical tool to define processing pipelines. The graphical editor solely operates based on description graphs provided by individual pipeline elements in the application layer. Pipeline elements available in the repository can therefore be connected to form a pipeline consisting of event sources publishing a specific event stream provided by a corresponding producer, processing agents and consumers. For each connection which is made during the pipeline authoring process, we verify the ability of two elements to be connected with each other based on provided and required schema-, quality- and stream grounding-related aspects; if the verification is unsuccessful, the connection is omitted. Once a pipeline has been created, it can be started. The management layer creates one RDF-based invocation graph for each processing element of a pipeline. This graph contains configuration options for the individual processing elements, their exact stream binding and information on the output schema. The graph is subsequently sent to the HTTP endpoints provided by each processing element. At run-time, events are exchanged by connecting to message brokers and topics as provided in the invocation graph. The management layer itself assigns run-time communication protocols and topics to each processing element based on provided capabilities.

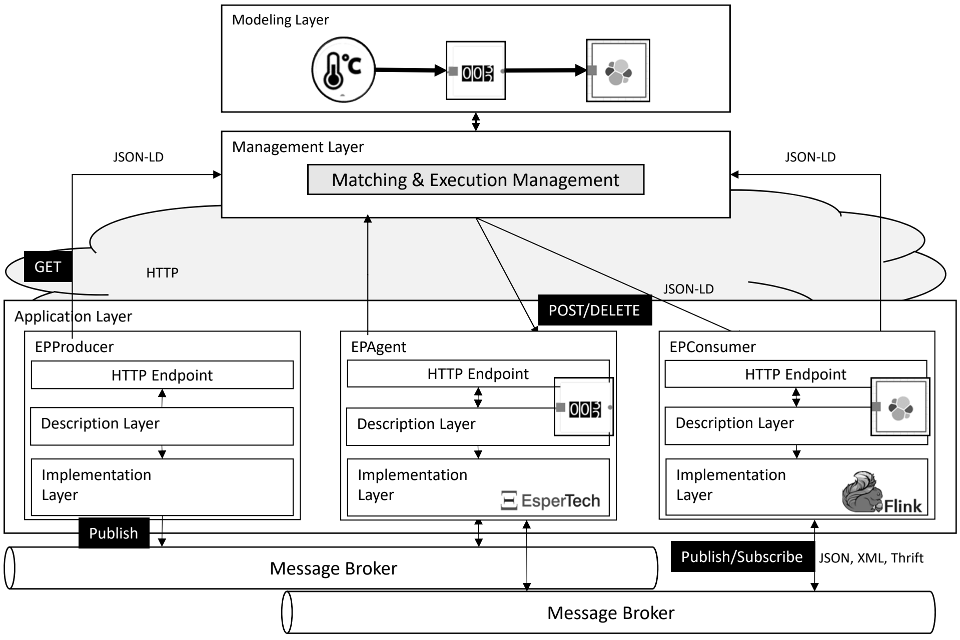


Figure 8.1 Processing Pipelines: Conceptual Architecture

8.2 Processing Pipelines

As a basis for an in-depth view on the management of processing pipelines, within this section processing pipelines are formally defined. In addition, we introduce structure and vocabulary of invocation graphs.

8.2.1 Definition

In this section, we give a formal description of a processing pipeline, which is composed from the three building blocks presented in the previous section. We consider a model base M which contains a set of event sources (S), event processing agents (A),

event consumers (C) and a set of processing pipelines P such that $M = (S,A,C,P)$. A Processing Pipeline $p \in P$ is a quadruple $p = (S_p,A_p,C_p,F_p)$, such that S_p denotes a set of event sources with $S_p \subseteq S$, A_p denotes a set of semantic event processing agents with $A_p \subseteq A$, C_p denotes a set of semantic event consumers with $C_p \subseteq C$. F_p denotes a binary relation defining the connections between these sets in a directed acyclic graph, such that $F_p \subseteq (S_p \times A_p) \cup (A_p \times A_p) \cup (A_p \times C_p)$ and $\forall a \in A_p : \{a,a\} \notin F_p$.
Furthermore, $N = S \cup A \cup C$ is defined as the set of nodes and each $f \in F$ is called an arc. $d^+(n)$ denotes the number of output arcs and $d^-(n)$ denotes the number of input arcs of $n \in N$. p is syntactically valid if the following applies:

1. $|S_p| \geq 1$,$|A_p| \geq 1$,$|C_p| \geq 1$, p is built by at least one source, one S-EPA and one S-EC.
2. $|C_p| = 1$, each P consists of exactly one event consumer.
3. $\forall a \in A_p : d^+(a) = 1$, each agent has exactly one output arc.
4. $\forall a \in A_p : d^-(a) \geq 1 \wedge d^-(a) \leq 2$, each agent has exactly one or two input arcs.
5. $\forall s \in S_p : d^+(s) \geq 1$, a source has at least one output arc.
6. $\forall s \in S_p : d^-(s) = 0$, a source has zero input arcs.
7. $\forall c \in C_p : d^-(c) = 1$, a consumer has exactly one input arc.
8. $\forall c \in C_p : d^+(c) = 0$, a consumer never has an output arc.

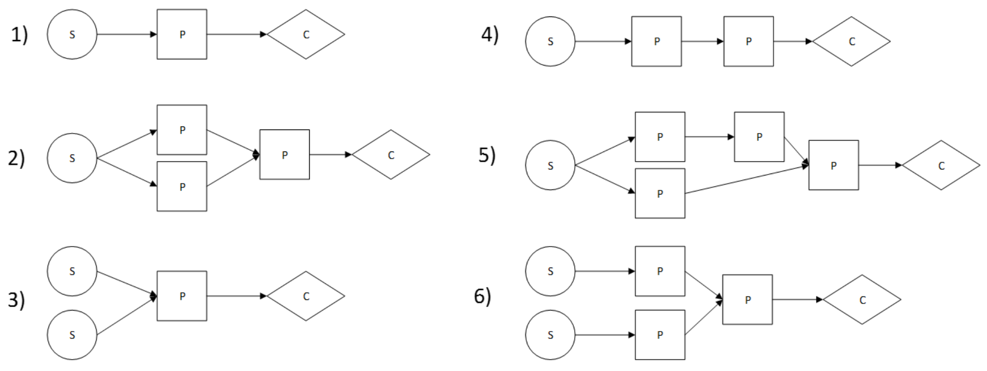


Figure 8.2 Syntactically Valid Processing Pipelines: Examples

Examples for syntactically valid processing pipelines are illustrated in figure 8.2. The attentive reader will have noticed that we do not allow splits in pipelines, i.e., an S-EPA is not allowed to have more than one output arc. We made this design decision due to the assumption that a pipeline should generally pursue a single scope, e.g., transforming one or more input streams to exactly one output stream. However, as splits might be useful to route events based on filter expressions to different consumers, we support the concept of *partial pipelines*. A partial pipeline $p' \in P'$ is formally defined as a triple $p' = (S_{p'},A_{p'},F_{p'})$ where $F_{p'} \subseteq (S_{p'} \times A_{p'}) \cup (A_{p'} \times A_{p'})$

and $\forall a \in A_{p'} : \{a,a\} \notin F_{p'}$., i.e., a partial pipeline is a graph similar to a processing pipeline, but does not provide a consumer element. Instead, a partial pipeline can be used as a *block* within a processing pipeline, which basically means that every part of a processing pipeline can be re-used in another pipeline which practically leads to the same functionality as a *Split EPA*, but ensures more intuitive modeling of pipelines. Therefore, we extend the definition of F_p to include partial pipelines such that $F_p \subseteq (S_p \times A_p) \cup (A_p \times A_p) \cup (A_p \times C_p) \cup (P' \times A_p) \cup (P' \times C_p)$.

8.2.2 Invocation Graphs

Invocation graphs encapsulate configuration information for processing elements. In contrast to description graphs for Semantic Event Processing Agents (S-EPAs) and Semantic Event Consumers (S-ECs), an invocation graph explicitly specifies how the implementation logic of a processing element can be instantiated.
In figure 8.3, we illustrate how invocation graphs are generated. A software module, the integration and execution engine, receives a description graph as an input. Based on the specification contained in the description graph, required user input is calculated. Required user input can be presented to pattern engineers involved with pipeline authoring. Tools such as graphical editors are then able to generate views which request users to provide the required input. Once user input has been provided, the invocation graph can be calculated. This graph contains information about the input event stream the S-EPA should subscribe, the output event stream, configuration values depending on the user input and grounding information on input and output streams.
In section 8.3.5, the generation of invocation graphs is presented in detail.

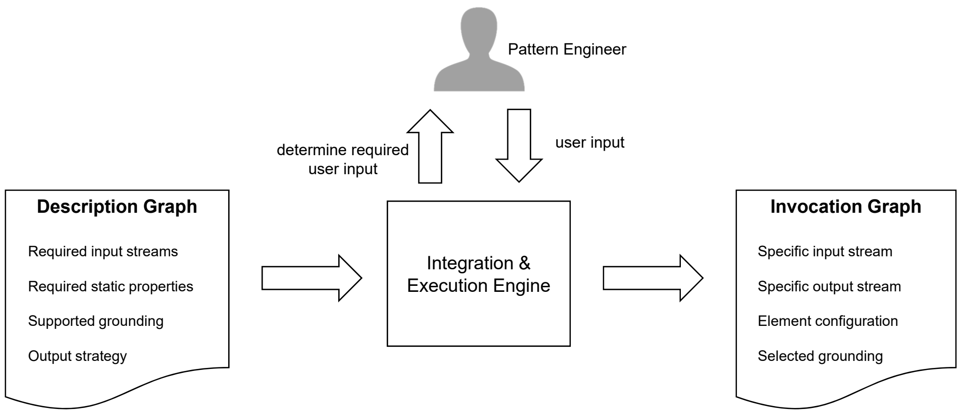


Figure 8.3 Invocation Graphs: Generation

Although many elements of the vocabulary presented in chapter 7 can be re-used for invocation graphs, some extended concepts and properties are necessary. An invocation is represented either by an AgentInvocation to invoke S-EPAs or a ConsumerInvocation to invoke S-ECs. While both concepts are related to one or more input streams, only an AgentInvocation provides information on an output stream.

Vocabulary

Concepts and properties that are intended to be used with invocation graphs are listed in tables 8.1 and 8.2.

Table 8.1 Concepts: Processing Element Invocation

Concept	rdfs:subClassOf	Description
epa:Invocation	rdfs:Class	An entity representing information on the invocation of a pipeline element
epa:AgentInvocation	epa:Invocation	An entity representing information on the invocation of an S-EPA
epa:ConsumerInvocation	epa:Invocation	An entity representing information on the invocation of an S-EC

Table 8.2 Properties: Processing Element Invocation

Property	so:domainIncludes / so:rangeIncludes	Description
epa:hasInputStream	epa:Invocation / epa:EventStream	Assigns an input event stream to an EPA:INVOCATION
epa:hasOutputStream	epa:AgentInvocation / epa:EventStream	Assigns an output event stream to an EPA:AGENTINVOCATION
epa:hasStaticProperty	epa:Invocation / epa:StaticProperty	Assigns a static property to an EPA:INVOCATION

Example

Our example is based on the examples given in chapter 7. Based on the *PositionStream* and the *Geofencing S-EPA*, the following listing shows how an invocation graph is being defined in order to instantiate the Geofencing S-EPA.
In general, the description graph of the Geofencing element has defined multiple static properties containing configuration parameters needed for the instantiation of the element's implementation logic.

- **Geofencing Operation**, which expects users to select the operation that should be performed. The description offers two operations, the arrival within a geofence and the departure from a geofence.
- **Geofence Center**, which expects users to provide a center coordinate of the geofence. This has been modeled as a *DomainStaticProperty*, i.e., it links to a required concept from the knowledge base.
- **Geofence Radius**, which expects users to provide the size of the geofence as a single value.
- **Latitude and Longitude Mapping**, which expect users to select a specific property from a list of possible properties from the input stream (in this example the *PositionStream*) which match the property requirement defined in the S-EPA description. In our case, as only a single property in the input stream matches each of the requirements, no further user input is required.

According to Listing 8.1, we first specify that the Geofencing S-EPA should detect an arrival of a vehicle within the geofence by assigning a *true* value to the optionSelected property.

Next, both supported properties are assigned numbers to indicate the geofencing center. The specific numbers are directly derived from the knowledge base, section 8.3.5 illustrates the implementation of a lookup in the knowledge base. In addition, the center of the geofence is specified by assigning a value to the static property. Finally, the invocation includes information about the specific latitude and longitude properties from the *PositionStream* that should be used to calculate the geofence detection.

Listing 8.1 Example: Invocation Graph

```
:GeofencingInvocation rdf:type epa:AgentInvocation .

:GeofencingInvocation epa:hasInputStream :PositionStream .

:GeofencingInvocation epa:hasStaticProperty :GeofencingOperation .
:GeofencingEpa epa:hasStaticProperty :GeofenceCenter .
:GeofencingEpa epa:hasStaticProperty :GeofenceRadius .
:GeofencingEpa epa:hasStaticProperty :LatitudeMapping .
:GeofencingEpa epa:hasStaticProperty :LongitudeMapping .

:GeofencingOperation epa:hasOption :ArrivesOption .
:ArrivesOption epa:optionSelected true .

:GeofenceCenter rdf:type epa:DomainStaticProperty .
:GeofenceCenter epa:requiredClass geo:Location .
:GeofenceCenter epa:hasSupportedProperty :LatitudeSupported .
:GeofenceCenter epa:hasSupportedProperty :LongitudeSupported .

:LatitudeSupported so:value 49 .
:LongitudeSupported so:value 7 .

:GeofencingRadius rdf:type epa:SingleValueProperty .
:GeofencingRadius epa:requiredDatatype so:Integer .
:GeofencingRadius epa:valueSpecification :RadiusValueSpecification .
:GeofencingRadius so:value 500 .
```

```

:LatitudeMapping rdf:type epa:MappingPropertyUnary .
:LatitudeMapping epa:mapsFrom :LatitudePropertyRequirement .
:LatitudeMapping epa:mapsTo :LatitudeProperty .

:LongitudeMapping rdf:type epa:MappingPropertyUnary .
:LongitudeMapping epa:mapsFrom :LongitudePropertyRequirement .
:LongitudeMapping epa:mapsTo :LongitudeProperty .
```

8.3 Pipeline Authoring

In this section, we describe the pipeline authoring process in more detail. Although we suggest that pipeline authoring targeted at non-programmers can be best supported by using graphical editors to define processing pipelines, our vocabulary in general is not restricted to be used with a single tool for pipeline authoring. However, in order to better illustrate the pipeline authoring process within this section, our examples are given by showing examples from the pipeline editor we developed as part of the StreamPipes framework.

8.3.1 StreamPipes Pipeline Editor

The StreamPipes pipeline editor is a web-based application to create processing pipelines using a blocks-and-arrows-based notation. Figure 8.4 illustrates the main interface of the pipeline editor. The top navigation bar allows users to select an element type. The *Data Streams* section displays event streams extracted from all available S-EP descriptions, *Processing Elements* lists available S-EPAs and *Data Sinks* shows available S-ECs. Elements users are able to select are shown in the pipeline element section.
The lower area of the pipeline editor, subsequently called the *assembly area*, is the main canvas to define processing pipelines. In order to create a pipeline, individual pipeline elements can be dragged from the pipeline element selection into the assembly area. Starting with the selection of streams, pipelines can be formed by connecting individual pipeline elements with other elements according to the pipeline authoring process described in the next section.

8.3.2 Authoring Process

Pipeline authoring follows a defined process which is illustrated in figure 8.5. This process includes both user-related tasks and system-related tasks. In this section, we briefly summarize the authoring process, while the following sections present selected tasks in more detail. In general, the pipeline authoring process starts with the selection of a *data sources* of the pipeline. The next user-related task is to select an

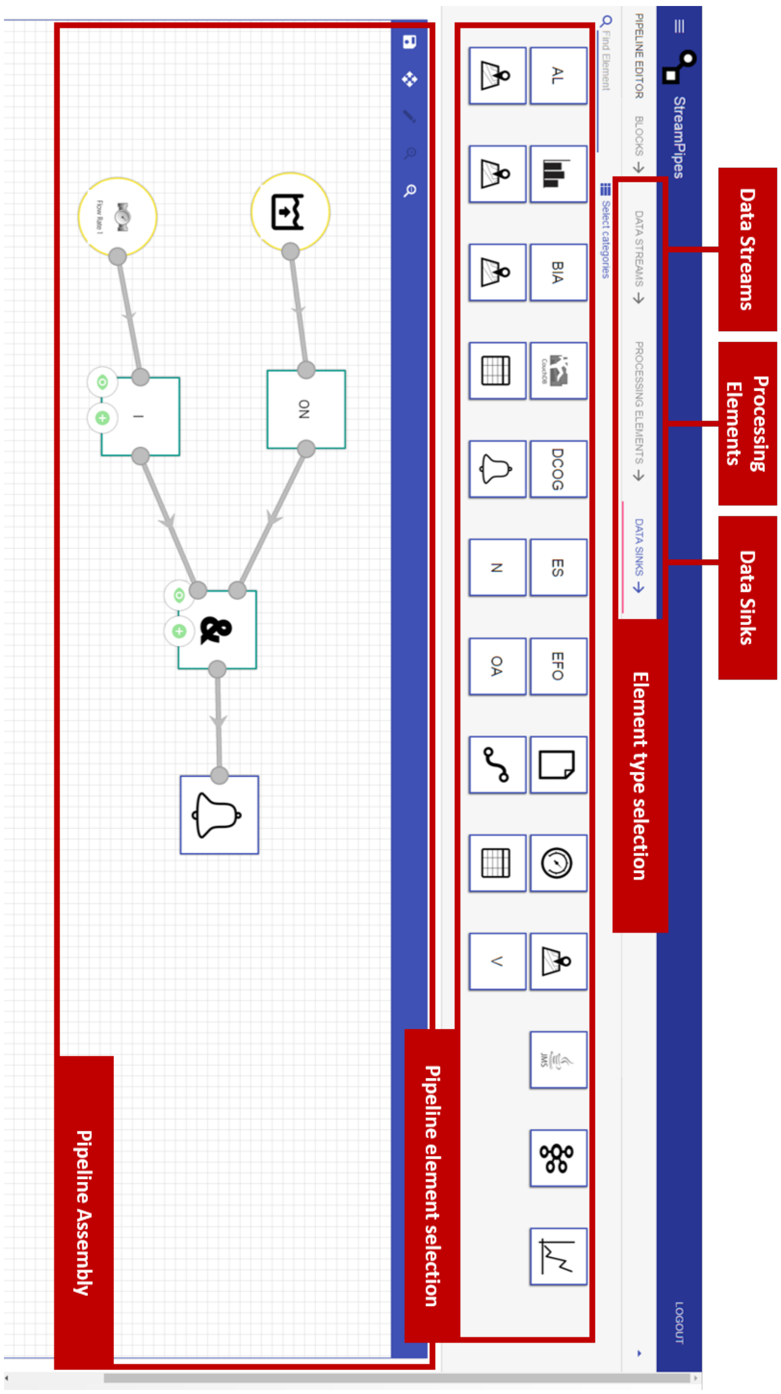


Figure 8.4 Processing Pipelines: Authoring Tool

S-EPA. Depending on the number of input nodes, both elements are connected with each other.
Once a connection has been made, this connection is verified based on the description of both elements. The verification process considers schema-related, grounding-related and quality-related requirements as introduced in chapter 7. If a connection is considered valid, the *element customization* process is entered. In this process, stream-specific mapping properties are computed for the element. Based on the possible mapping between event property requirements and event properties provided from the incoming streams, in addition to defined static properties of the S-EPA, required user input is computed. After users have configured the element with the required parameters, the output stream can be computed. Once all required S-EPAs have been added to the pipeline, an S-EC needs to be selected, which acts as the sink of the pipeline. Connections between S-EPAs and S-ECs are also verified and customized before the pipeline can be started. At pipeline startup time, invocation graphs are computed for all processing elements of the pipeline. Afterwards, these graphs are sent to the HTTP endpoints of each processing element, where the individual runtimes implementations are parameterized based on the configuration parameters included in the invocation graph.

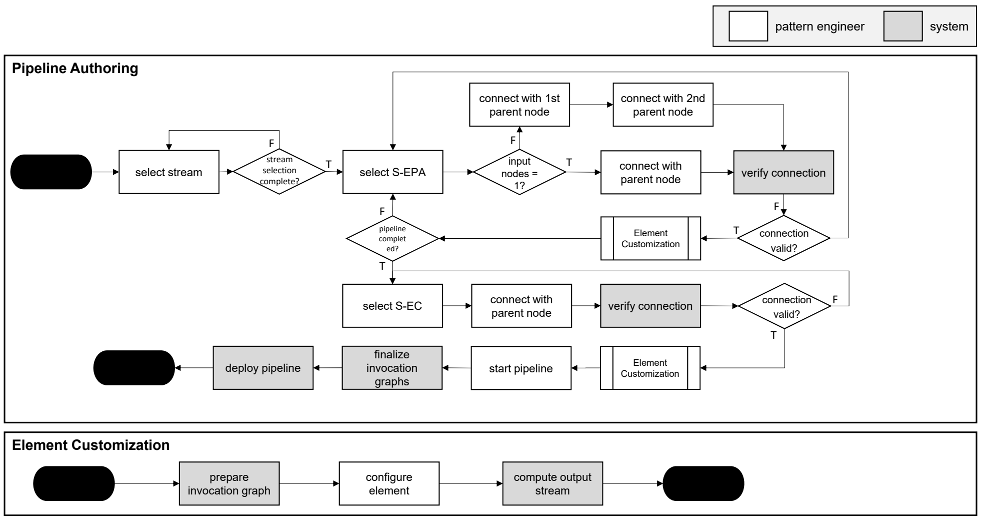


Figure 8.5 Pipeline Authoring: Process

8.3.3 Stream/S-EPA Selection

The first two tasks within the pipeline authoring process are user-related. By using the graphical editor, an event source can be dragged into the assembly area. Afterwards, either additional sources or an S-EPA can be selected. This process is illustrated in figure 8.6. By using the StreamPipes editor, users are further provided with two features assisting to select appropriate elements. First, for a given source, elements are recommended based on an analysis of the pipeline repository, which simply counts the number of successful connections of previously created pipelines between two elements and generates a recommendation consisting of the top-k recommended elements. Second, each element provides an option to filter for pipeline elements which are compatible to each other based on the verification. By choosing this option, the *pipeline element selection* area only displays elements that can be connected with the active element.
Source elements and S-EPAs provide an output port. This port can be used to create a connection to another element, which triggers the verification process as described below.

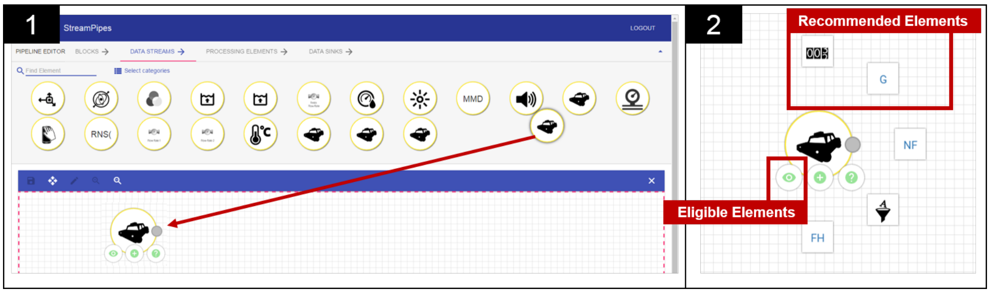


Figure 8.6 Pipeline Authoring: Stream Selection

8.3.4 Verification

Once a connection between two pipeline elements has been made, we are interested in figuring out whether these elements can be connected. Instead of a pure syntactical-oriented approach based on data types, we aim to verify a connection based on the vocabulary defined in chapter 7. Therefore, a connection is considered valid if an input stream satisfies all stream requirements of a processing element. In addition, an input stream must at least support one grounding which is required by a processing element.

In order to formally describe the verification of two pipeline elements, which is also being used in our provided tools to determine two compatible elements, we extend the model base defined in section 8.2 as follows:

$$M = (S, A, C, P, E, \Gamma, \Pi, K, Q, T, D, \Psi, \Phi)$$

with

- E as the set of all event schemas,
- Γ as the set of all stream groundings,
- Π as the set of all event properties,
- K as the set of all stream qualities,
- Q as the set of all property qualities,
- T as the set of all data types,
- D as the set of all domain properties,
- Ψ as the set of all transport protocols and
- Φ as the set of all transport formats.

Furthermore, O represents a *StreamOffer* and R represents a *StreamRequirement*. We further define two functions $offers$ and *requires*, whereas a *StreamOffer* can be made by sources and event processing agents and a *StreamRequirement* can be made by event processing agents and event consumers. Note that, as defined above, event processing agents and consumers might have either one or two input ports and therefore can define one or two *StreamRequirements*.

$$\begin{aligned} offers &: S \cup A \rightarrow O \\ requires &: A \cup C \rightarrow R \times R \end{aligned}$$

In order to decide whether an offer fulfills a requirement, the event schema, stream qualities and the stream groundings are taken into account.

$$O, R \subseteq E \times \Gamma \times 2^K$$

In general, a *StreamOffer* fulfills a *StreamRequirement* if its event schema *matches* the schema requirement ($ematch$), its stream qualities match the stream quality requirements ($kmatch$) and if a stream grounding ($gmatch$) which both sides support can be found.

$$\begin{aligned} \forall r \in R, o \in O : (matchAll(r = (\varepsilon_R, \gamma_R, \kappa_R), o = (\varepsilon_O, \gamma_O, \kappa_O)) \\ \Leftrightarrow ematch(\varepsilon_R, \varepsilon_O) \wedge gmatch(\gamma_R, \gamma_O) \\ \wedge (\forall \kappa_R \in K_R \exists \kappa_O \in K_O : kmatch(\kappa_R, \kappa_O))) \end{aligned}$$

From a grounding point of view, a *StreamOffer* fulfills a *StreamRequirement* if at least one common transport protocol (ψ) and one common transport format (ϕ) can be found. Therefore, the function $sup(a,b)$ defines whether a grounding supports a specific format or protocol.

$$\begin{aligned}\forall\gamma_R,\gamma_O\in\Gamma :&(gmatch(\gamma_R,\gamma_O)\Leftrightarrow\\ &(\exists\psi\in\Psi : sup(\gamma_R,\psi)\wedge sup(\gamma_O,\psi))\\ &\wedge(\exists\phi\in\Phi : sup(\gamma_R,\phi)\wedge sup(\gamma_O,\phi)))\end{aligned}$$

Quality attributes defined as requirements on the stream level need to be fulfilled by an input event stream. As defined in chapter 7, a stream quality requirement might provide a minimum value, indicating that an input stream must at least support the provided value, or a maximum value, indicating that an input stream must not exceed the provided value. Therefore, we define a function *ktype*, which maps a stream quality κ to a min/max type and a function *kvalue*, which maps a stream quality κ to a numerical value.

$$\begin{aligned}ktype :&K\to\{\{min\},\{max\}\}\\ kvalue :&K\to\mathbb{R}\end{aligned}$$

A stream quality match is defined as follows: Whenever an offered stream quality κ_O exceeds a required minimum stream quality κ_R or an offered stream quality κ_O does not exceed a required maximum stream quality, then κ_R, κ_R and κ_O match.

$$\begin{aligned}\forall\kappa_R,\kappa_O\in K :&(kmatch(\kappa_R,\kappa_O)\Leftrightarrow\\ &(ktype(\kappa_R)=\{min\}\wedge(kvalue(\kappa_R)\le kvalue(\kappa_O)))\\ &\wedge(ktype(\kappa_R)=\{max\}\wedge(kvalue(\kappa_R)\ge kvalue(\kappa_O))))\end{aligned}$$

Besides grounding and stream quality, the schema of an event stream is relevant to determine a matching between a *StreamOffering* and a *StreamRequirement*. An event schema consists of one or more event properties.

$$E\subseteq 2^{\Pi}$$

In order to determine a schema match between a stream offering and a stream requirement, for each event property defined in the requirement specification an event property needs to be present in the offering which matches the property requirement. A *pmatch* defines such a property match.

$$\begin{aligned}\forall \varepsilon_R, \varepsilon_O \in E :&(ematch(\varepsilon_R, \varepsilon_O) \Leftrightarrow \\ &\quad (\forall \pi_R \in E_R \exists \pi_O \in E_O : pmatch(\pi_R, \pi_O)))\end{aligned}$$

An offered event property matches a required event property, if three conditions are fulfilled. First, if the property requirement specifies a data type, the schema of a stream offering needs to provide an event property with the same data type (*tmatch*). Second, if the property requirement specifies a domain property, the schema of a stream offering needs to provide an event property with the same domain property (*dmatch*). Third, if a property requirement specifies one or more property qualities, these quality requirements must also be fulfilled by an event property that belongs to a stream offering.

$$\Pi \subseteq 2^Q \times D \times T$$

$$\begin{aligned}\forall \pi_R, \pi_O \in \Pi :&(pmatch(\pi_R = (Q_R, d_R, t_R), \pi_O = (Q_O, d_O, t_O)) \Leftrightarrow \\ &(\forall q_R \in Q_R \exists q_O \in Q_O : qmatch(q_R, q_O)) \wedge \\ &dmatch(d_R, d_O) \wedge tmatch(t_R, t_O))\end{aligned}$$

A *tmatch* indicates whether a required data type is matched by an offered data type.

$$\forall t_R, t_O \in T : (tmatch(t_R, t_O) \Leftrightarrow (t_R = t_O)$$

A *dmatch* indicates whether a required domain property is matched by an offered domain property. In contrast to data types, a domain property match is not only satisfied if a domain property specified in the requirement is provided in the offering, but also matches all domain properties in the offering that are sub properties (based on an RDFS:SUBPROPERTYOF relation). The function $spo(a, b)$ determines whether a domain property b is a sub property of a.

$$\forall d_R, d_O \in D : (dmatch(d_R, d_O) \Leftrightarrow (spo(d_R, d_O))$$

Finally, property quality is matched similar to stream qualities. For each property quality assigned in a property requirement, an event property in the offering must fulfill the quality specification.

$$\begin{aligned}
qtype :& Q \rightarrow \{\{min\},\{max\}\} \\
qvalue :& Q \rightarrow \mathbb{R} \\
\forall q_R, q_O \in Q :& (qmatch(q_R, q_O) \Leftrightarrow \\
& (qtype(q_R) = \{min\} \wedge (qvalue(q_R) \leq qvalue(q_O))) \\
& \wedge (qtype(q_R) = \{max\} \wedge (qvalue(q_R) \geq qvalue(q_O))))
\end{aligned}$$

Based on this approach, we are able to determine whether two pipeline elements can be connected. If this is the case, the next step is to compute the input required from users. Technically, this task implies the preparation of an invocation graph. Although the exact output stream is not yet known before the user-related customization of an element has been finished, some static properties need to be re-configured based on the exact input stream. For instance, for each *mapping property* defined in the description of a processing element, specific event properties from the input event stream need to be found.
Algorithm 1 defines the algorithm used to create an invocation graph. The input of this algorithm is a temporary pipeline containing all pipeline elements that have been added so far. Therefore, the minimum size of a temporary pipeline is 2, an input stream and the first element that has just been connected. The output of the algorithm is a *prepared invocation graph*, i.e., an invocation graph that still does not contain information on output event streams, but already includes both input event stream definitions and user-related static property requirements.
As follows, we define the *root element* of a pipeline as the first element of a pipeline which is not yet been fully customized by the user, i.e., the last pipeline element that has been connected. The initial step of the algorithm is to extract partial pipelines. Starting from the root element, for each input channel that already has an arc to a predecessor, a partial pipeline is built. The output stream of each partial pipeline is then added as an input stream of the invocation graph. Afterwards, static properties are attached to the invocation graph. For each static property that is part of the root element's description graph, we determine whether the static property is either of type *MappingProperty* or *MatchingProperties*. As introduced in chapter 7, mapping properties are used in order to map an operation of an S-EPA or S-EC to a specific event property from an input stream. For instance, considering an S-EPA which filters numerical event properties in order to detect whether a value exceeds a custom-defined threshold value, the mapping properties' *mapsFrom* relation links to an event property defined in the stream requirement in the S-EPA's description graph. For the invocation graph, the *mapsFrom* relation needs to be rewritten in order to link to "real" event properties from the input event stream, i.e., all event properties which match the property defined in the *mapsFrom* relation. This is performed in lines 11-16 of the algorithm. In addition, matched properties from the input event stream need to be

Algorithm 1 Compute Invocation Graph

```
Input:
    incomplete temporary pipeline $p_t$.
Output:
    prepared invocation graph $i$.

 1: $partialPipelines[] \leftarrow$ EXTRACTPARTIALPIPELINES($p_t$)
 2: $r \leftarrow$ GETROOT($p_t$)
 3: $d \leftarrow$ GETDESCRIPTIONGRAPH($r$)
 4: $i \leftarrow$ empty invocation graph

 5: for all $p'$ in $partialPipelines[]$ do                ▷ incoming streams of $r$
 6:     $s \leftarrow$ GETOUTPUTSTREAM($p'$)
 7:     ADDINPUTSTREAM($i$,$s$)
 8: ADDOUTPUTSTRATEGIES($i$, $d$)
 9: for all $sp$ in GETSTATICPROPERTIES($d$) do
10:     if $sp$ typeof $MappingProperty$ then          ▷ modify Mapping Properties
11:         $ep_m \leftarrow$ GETMAPSFROM($sp$)
12:         $sp_m \leftarrow$ new mapping property
13:         for all $s_i$ in GETINPUTSTREAMS($i$) do
14:             $ep[]_s \leftarrow$ COLLECTMATCHEDPROPERTIES($s_i$, $ep_m$)
15:             ADDMAPSFROM($sp_m$, $ep[]_s$)
16:         ADDSTATICPROPERTY($i$, $sp_m$)
17:     else if $sp$ typeof $MatchingProperty$ then     ▷ modify Matching Properties
18:         $ep_l \leftarrow$ GETMATCHLEFT($sp$)
19:         $ep_r \leftarrow$ GETMATCHRIGHT($sp$)
20:         $sp_c \leftarrow$ new matching property
21:         for all $s_i$ in GETINPUTSTREAMS($i$) do
22:             $ep[]_l \leftarrow$ COLLECTMATCHEDPROPERTIES($s_i$, $ep_l$)
23:             $ep[]_r \leftarrow$ COLLECTMATCHEDPROPERTIES($s_i$, $ep_r$)
24:             ADDMATCHLEFT($sp_c$, $ep[]_l$)
25:             ADDMATCHRIGHT($sp_c$, $ep[]_r$)
26:         ADDSTATICPROPERTY($i$, $sp_c$)
27:     else
28:         ADDSTATICPROPERTY($i$, $sp$)
          return $i$
```

Algorithm 2 Collect matched properties

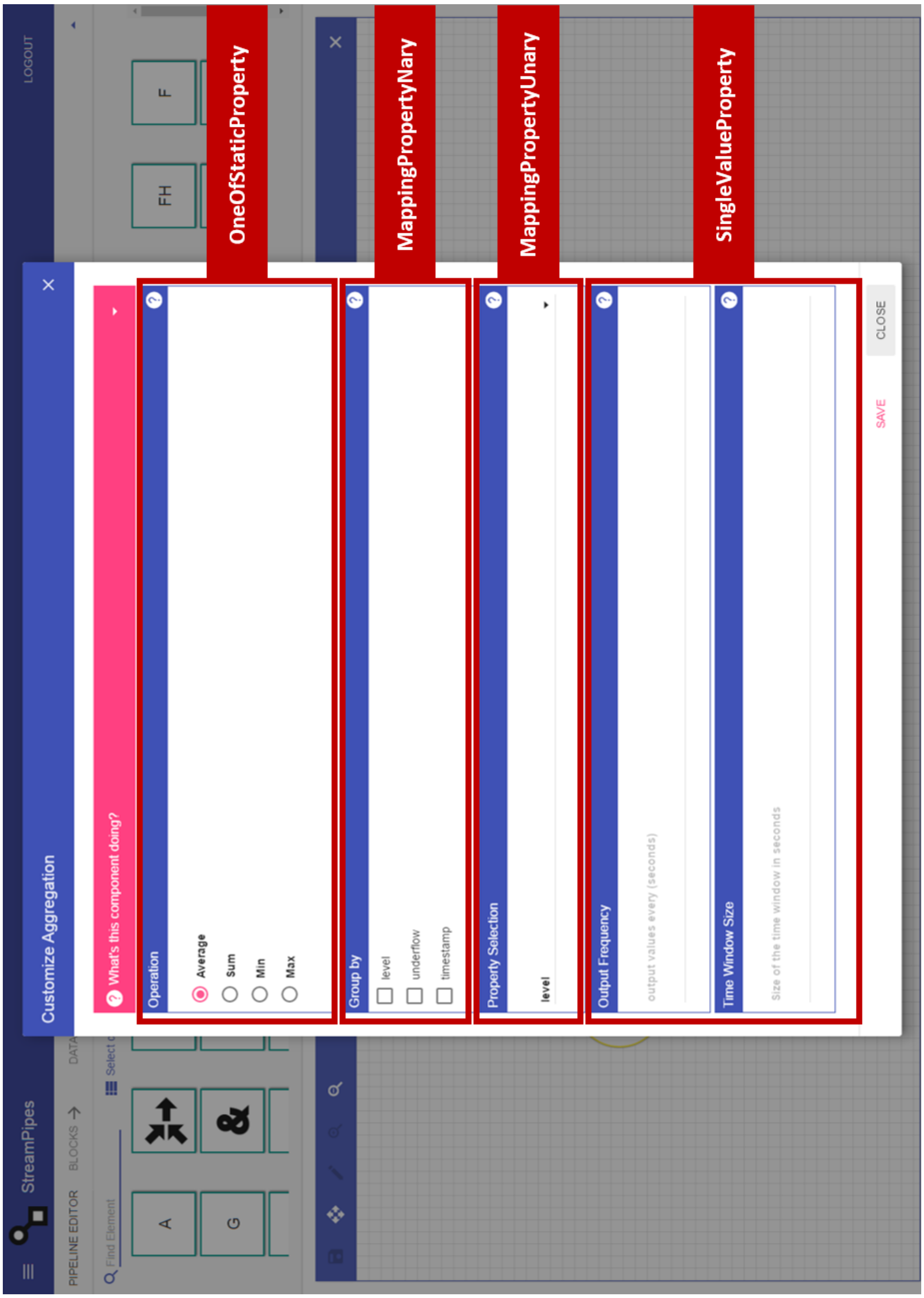
```
Input:
    event stream $s$.
    event property requirement $ep_r$
Output:
    set of matched event properties $ep[]_m$

1: $ep[]_m \leftarrow \emptyset$
2: $es_s \leftarrow \textsc{getEventSchema}(s)$
3: $ep[]_s \leftarrow \textsc{getEventProperties}(es_s)$
4: for all $ep_s$ in $ep[]_s$ do
5:     if $\textsc{pmatch}(ep_r, ep_s)$ then                ▷ property match
6:         $ep[]_m \leftarrow ep[]_m \cup ep_s$
   return $ep[]_m$
```

found for both properties defined in the *matchLeft* and *matchRight* relations assigned to *MatchingProperties*, which is done in lines 18-26 of the algorithm.
Algorithm 2, being used both for rewriting *MappingProperties* and *MatchingProperties* within algorithm 1 provides an algorithm to find matching properties from an event stream based on a given event property.
The algorithm returns a prepared invocation graph which can be used by the graphical editor in order to guide the user to correctly instantiate the run-time implementation of the processing element.

8.3.5 Element Customization

After a connection between two elements has been verified, and the *prepared invocation graph* has been computed, a customization dialog is presented to users as illustrated in figure 8.7. Available customization options depend on static properties and the provided output strategy. For each static property and each output strategy requiring for further user customization, a web-based form input element is generated.
Table 8.3 shows how static properties and output strategies are transformed to form elements. We briefly summarize the elements supported in the StreamPipes framework and describe their usage. It is noteworthy that the assignment from a static property to a generated form element is not necessarily fixed, but depends on the individual semantics of the underlying property. For instance, if a domain static property requires properties from an instance of a GEO:LOCATION class, a map can be displayed showing available locations gathered from instances found in the knowledge base.



Customize Aggregation

What's this component doing?

Operation
- [✓ Average]
- Sum
- Min
- Max

Group by
- level
- underflow
- timestamp

Property Selection: level

Output Frequency: output values every (seconds)

Time Window Size: Size of the time window in seconds

Figure 8.7 Pipeline Authoring: Element Customization

Table 8.3 Generation of Form Elements

Required User Input	Additional Constraints	Generated Form Element
SingleValueProperty	-	Text Input
	PropertyValueSpecification	Slider Input
	mapsTo (Quantitative Value)	Slider Input
	mapsTo (Enumeration)	Select Input
MultiValueProperty	-	Checkbox Input
SelectionStaticProperty	-	Radio Input
DomainStaticProperty	-	Instance Search
MappingPropertyUnary	-	Select Input
MappingPropertyNary	-	Checkbox Input
MatchingStaticProperty	-	Multiple Select Input
CustomOutputStrategy	-	Checkbox Input
TransformOutputStrategy	-	Transform Input

- **Text Input** requires a free text input and can additionally be restricted to a required data type as provided in the associated SingleValueProperty.
- **Slider Input** requires a numerical value with specified minimum and maximum values. A slider input is displayed if the associated SingleValueProperty either contains a mapsTo relation which links to an event property having a quantitative value definition. In addition, a slider input is shown if the SingleValueProperty itself provides a PropertyValueSpecification.
- **Select Input** requires the selection of one or more options. SingleValueProperties which have an assigned mapsTo relation linking to an event property with an enumerated value specification are displayed as select inputs. In addition, possible property mappings defined in a mapping property are presented as select inputs.
- **Checkbox Input** lets users select multiple options from a list of available options. This corresponds to the specification of MultiValueProperties, where users can select one or more supported operations the run-time implementation supports and instances of MappingPropertyNary allowing users to select multiple mappings.
- **Radio Input** corresponds to the functionality of a Select Input and is used in order to present multiple configuration options a SelectionStaticProperty offers, whereas users can select from one of these options.
- **Instance Search**. User input required for domain static properties linking to concepts and supported properties from a knowledge base are rendered as an instance search component. This component allows users to search for instances in the knowledge base based on keywords. The search component auto-suggests

matching instances found in the knowledge base. Once an instance has been selected by the user, supported properties are automatically extracted from the instance and attached to the domain static property definition.

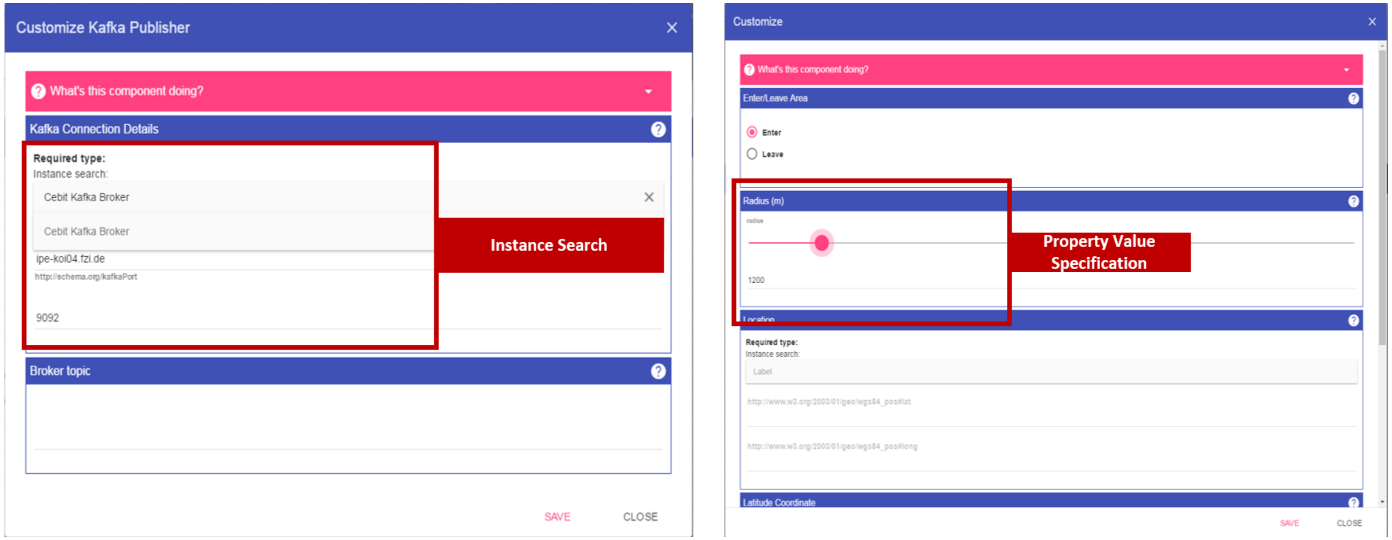
- **Multiple Select Input** is used to support MatchingStaticProperties. A multiple select input basically consists of two select inputs which require users to select one event property from each stream which should provide the same value.
- The **Transform Input** allows users to specify the transformation. Depending on the specification of the corresponding TransformOutputStrategy, input forms are generated to provide a new runtimeName, runtimeType or domainProperty.

Figure 8.8 illustrates examples for generated forms based on DomainStaticProperties as well as SingleValueProperties providing a PropertyValueSpecification. The left example shows the configuration of a DomainStaticProperty. The customization dialog is shown for the case of a *Apache Kafka Sink* which forwards an incoming event to a topic of an Apache Kafka message broker. In this example, specific broker settings, such as the URL of the broker and the port that should be used to initiate the broker connection can be selected from a list of available brokers previously configured in the knowledge base. The instance search dialog presents a search interface to look up such instances in the knowledge base. The right example shows the appearance of a SingleValueProperty which has an assigned PropertyValueSpecification. In this case, instead of a plain text input field, a slider input is generated enabling users to directly select a numerical value according to the allowed range of the value specification. In this example, the *radius* property of the already mentioned *Geofencing S-EPA* can be instantiated using a slider input.

8.3.6 Computation of Output Streams

Once users have provided inputs for all required values, the output stream can be computed. Algorithm 3 summarizes the steps required to compute an output event stream based on input event streams and provided user input. The input of this algorithm are all input event stream definitions of a processing element and the output strategy the element provides.

First, in lines 4-5, the event properties of both input streams are collected. Depending on the output strategy, a set of output event properties is defined. In case of a KeepOutput, the set of input event properties also represents the set of output event properties. An AppendOutput defines the output stream as the union set of all input event properties and additional event properties defined in the output strategy. In case of a FixedOutput, output event properties correspond to the set of event properties defined by the output strategy. CustomOutput uses the set of user-selected event properties, which are a subset of all event properties from the input event streams. Finally, if the output strategy is of type TransformOutput, event properties from the input event stream are replaced with new event properties according to the transfor-


Customize Kafka Publisher
What's this component doing?
Instance Search
SAVE
CLOSE
Customize
What's this component doing?
Property Value Specification
SAVE
CLOSE

Kafka Connection Details

Required type:
Instance search:
Cebit Kafka Broker
Cebit Kafka Broker
ipe-koi04.fzi.de
http://schema.org/kafkaPort
9092

Broker topic

Enter/Leave Area

[✓ Enter] Leave

Radius (m)
radius
1200

Location

Required type:
Instance search:
Label
http://www.w3.org/2003/01/geo/wgs84_pos#lat
http://www.w3.org/2003/01/geo/wgs84_pos#long

Latitude Coordinate

Figure 8.8 Processing Pipelines: Semantics-based Form Generation

mation type (partly) indicated for by users. Therefore, for each UriPropertyMapping a set of replaceFromProperties and a set of replaceWith properties is collected. The set of output event properties contains all input event properties that are not affected by the transformation in addition to transformed event properties.

Algorithm 3 Compute Output Stream

```
Input:
    input stream definitions $S_i$.
    output strategy $o$
Output:
    output stream definition $s_o$

 1: $ep[]_i \leftarrow \emptyset$                                          ▷ input event properties
 2: $ep[]_o \leftarrow \emptyset$                                          ▷ output event properties

 3: for all $s_i$ in $S_i$ do
 4:     $es_i \leftarrow$ getEventSchema($s_i$)
 5:     $ep[]_i \leftarrow ep[]_i \cup$ getEventProperties($es_i$)
 6: if $o$ typeof $KeepOutput$ then
 7:     $ep[]_o \leftarrow ep[]_i$
 8: else if $o$ typeof $AppendOutput$ then
 9:     $ep[]_{o'} \leftarrow$ getAppendOutputProperties($o$)
10:     $ep[]_o \leftarrow ep[]_i \cup ep[]_{o'}$
11: else if $o$ typeof $FixedOutput$ then
12:     $ep[]_{o'} \leftarrow$ getFixedOutputProperties($o$)
13:     $ep[]_o \leftarrow ep[]_{o'}$
14: else if $o$ typeof $CustomOutput$ then
15:     $ep[]_{o'} \leftarrow$ getUserDefinedOutputProperties($ep[]_i$)
16:     $ep[]_o \leftarrow ep[]_i \cap ep[]_{o'}$
17: else if $o$ typeof $TransformOutput$ then
18:     $ep[]_{r'} \leftarrow \emptyset$                                   ▷ properties to remove
19:     $ep[]_{i'} \leftarrow \emptyset$                                   ▷ properties to include
20:     for all $u$ typeof getUriPropertyMappings($o$) do
21:         $ep[]_{r'} \leftarrow ep[]_{r'} \cup$ getReplaceFromProperty($u$)
22:         $ep[]_{i'} \leftarrow ep[]_{r'} \cup$ getReplaceWithProperty($u$)
23:     $ep[]_o \leftarrow (ep[]_i \setminus ep[]_{r'}) \cup ep[]_{i'}$
24: $es_o \leftarrow$ makeSchema($ep[]_o$)
25: $s_o \leftarrow$ makeStream($es_o$)
26: return $s_o$
```

8.4 Pipeline Deployment

Finally, once a pipeline has been completed, its deployment process can be started. Pipeline deployment is manually triggered by users by requesting a pipeline to be started. At this time, a pipeline is represented by one or more source definitions indicating the input event streams and a set of *prepared invocation graphs* which already include input stream definitions, an output stream definition and configuration parameters. Therefore, the final missing building block is the *communication* between pipeline elements, i.e., the transport protocol and format event streams are exchanged in form of their individual stream groundings. Although we have already ensured within the verification task that connected pipeline elements provide at least one common grounding, a specific *protocol*, *transport format* and a *topic* used for publishing and subscription needs to be chosen.

Algorithm 4 Complete Invocation Graphs

```
Input:
    temporary processing pipeline $p_t$.
Output:
    processing pipeline $p_t$

 1: $tree \leftarrow$ buildTree($p_t$)
 2: $root \leftarrow$ getRoot($tree$)
 3: $pipelineElements \leftarrow$ traversePostOrder($tree$)
 4: for $pipelineElement$ in $pipelineElements$ do
 5:     if $pipelineElement = root$ then
 6:         $break$;
 7:     else
 8:         $parentElement \leftarrow$ getParent($pipelineElement$)
 9:         $s_o \leftarrow$ getOutputStream($pipelineElement$)
10:         $s_i \leftarrow$ getInputStream($pipelineElement, parentElement$)
11:         $sgr[]_l \leftarrow$ getSupportedGrounding($pipelineElement$)
12:         $sgr[]_r \leftarrow$ getSupportedGrounding($parentElement$)
13:         $sgr_c \leftarrow$ selectStreamGrounding($sgr[]_l, sgr[]_r$)
14:         assignSelectedGrounding($s_i$, $sgr_c$)
15:         assignSelectedGrounding($s_o$, $sgr_c$)
    return $p$
```

This is done as summarized in algorithm 4. The input of this algorithm is a temporary processing pipeline. First, a tree is built with the S-EC as the root node. Afterwards, we traverse this tree in post-order. For each output-input connection between two pipeline elements, a specific stream grounding needs to be selected. The algorithm finishes once the root element has been reached. A stream grounding is selected by comparing the supported grounding of a pipeline element and the supported grounding of its direct ancestor element in the pipeline. Afterwards, the function SELECTSTREAMGROUNDING, described in algorithm 5, is called which finds a common grounding supported by both elements and assigns a topic to the transport protocol of the grounding. If a source (which was already assigned a topic in the setup phase) is the current pipeline element, the topic from the source is being re-used. Otherwise, a random topic is generated. The grounding is then assigned to both the output event stream definition from the current pipeline element and the input stream definition from the parent element.

Algorithm 5 Select stream grounding

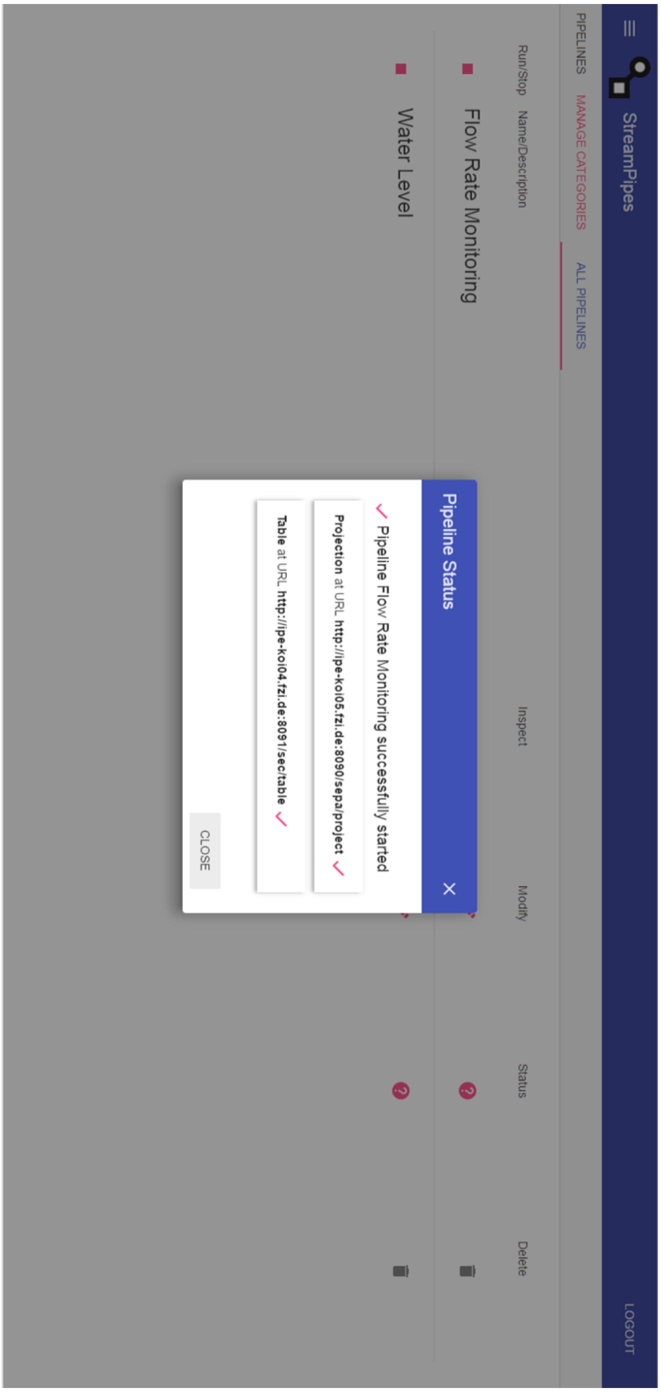
```
Input:
    supported stream groundings left $sgr[]_l$
    supported stream groundings right $sgr[]_r$
Output:
    selected stream grounding $sgr_c$

1: $sgr_c \leftarrow \emptyset$
2: for all $sgr_r$ in $sgr[]_r$ do
3:     for all $sgr_l$ in $sgr[]_l$ do
4:         if GMATCH($sgr_r, sgr_l$) then
5:             $sgr_c \leftarrow sgr_r$
6:             $topic \leftarrow$ CHOOSETOPIC( )
7:             ASSIGNTOPIC($sgr_c, topic$)
8:             break
   return $sgr_c$
```

Finally, for each processing element of an invocation graph exists which contains all necessary configuration required for the instantiation of the element. Graphs are subsequently sent to the HTTP endpoints of each processing element where the implementation logic is invoked. A processing pipeline is successfully started if all processing elements of the pipeline are successfully invoked. Our provided tool support includes an interface to trigger the start of a pipeline as illustrated in figure 8.9. This interface also indicates whether all pipeline elements could be started successfully.

8.5 Summary

In this chapter, the execution phase was discussed in detail. We first gave a formal definition of processing pipelines which consist of a number of potentially distributed Semantic Event Producers (S-EP), Semantic Event Processing Agents (S-EPA) and



Pipeline Flow Rate Monitoring successfully started

Projection at URL **http://ipe-koi05.fzi.de:8090/sepa/project**

Table at URL **http://ipe-koi04.fzi.de:8091/sec/table**

Figure 8.9 Processing Pipelines: Pipeline Start Dialog

Semantic Event Consumers (S-EC). Afterwards, the pipeline authoring process was presented in details. Along a process model describing the individual steps required to create processing pipelines, we formally presented details on the matching process between stream offerings (provided by an S-EP or an S-EPA) and stream requirements (provided by an S-EPA or an S-EC). Finally, algorithms were introduced that facilitate the generation of invocation graphs, which contain the necessary information required to invoke the run-time implementation of geographically distributed pipeline elements in order to execute a processing pipeline.

Part IV

Finale

9 Evaluation

In this chapter, we evaluate our approach. Evaluations are performed based on the artifacts we developed in order to implement the methodology that has been introduced in chapter 6 and further detailed in chapter 7 and 8. Section 9.1, the first part of this chapter introduces the evaluation methodology and describes our implementation which is used as the artifact to evaluate the proposed methodology. Section 9.2 presents three case studies where we present experiences made by applying our methodology in various application domains in terms of the *development process*. In section9.3, we perform a conceptual investigation in order to analyse the *expressivity* of our approach by discussing the fulfillment of requirements collected in chapter 5. We evaluate the *usability* of our approach in section 9.4. Finally, section 9.5 deals with *performance* measurements to compare the performance of our approach to existing event processing systems.

9.1 Evaluation Methodology

In this section, we discuss the design of evaluations performed in order to show to what extend our contributions are able to solve the identified problems. Therefore, an *evaluation strategy* is required. As we followed the design science research (DSR) paradigm as a research methodology during the course of this thesis, a strategy supporting the evaluation of design science artifacts is required [Pries-Heje et al. 2008]. An overview of variables and values for the evaluation of DSR artifacts is given in [Cleven et al. 2009], which include, among others, the selection of an *artifact type* (e.g., a *model* or an *instantiation*, an evaluation *method* (e.g., a *Case study*), and a *function* the evaluation serves.
According to [Venable et al. 2012], an evaluation design process starts with a requirements elicitation. Our evaluation design is defined along the research questions identified in chapter 1 and briefly recapped below. For each research question, we define evaluation metrics which are able to give evidence of the fulfillment degree of our approach in relation to the research question. Based on each metric, we select an evaluation method which is able to acquire knowledge on the fulfillment of an evalua-

tion metric. Finally, a software artifact which implements the metric under evaluation is chosen. Details on the evaluation methods itself are given in the respective sections. Our evaluation methodology is summarized in figure 9.1.

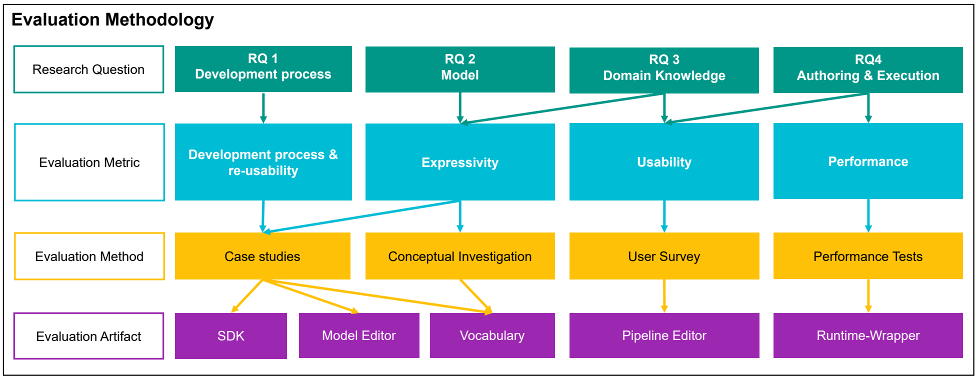


Figure 9.1 Evaluation: Methodology

We will now identify metrics, methods and artifacts for the research questions identified in chapter 1.

Research Question 1 (Development Process). *How can we improve the development process of event processing applications?*

This research question targets the efficiency of event processing development processes. As a main artifact, in chapter 6 we introduced a novel methodology supporting the development of event processing applications. The main advantage of this methodology is to leverage software engineers to provide event processing building blocks (i.e., producers, event processing agents and consumers) which can be re-used by non-programmers in various settings without further software development effort. Therefore, we can answer this question by showing that a) the effort to develop re-usable building blocks is not significantly higher than the development of hard-coded building blocks and b) building blocks developed with our methodology are in fact re-usable. An evaluation method which is able to answer this research question could therefore compare the development effort typically needed for existing development processes with the effort needed when applying our methodology in order to make results measurable. However, in order to get comparable results, such an approach would require developers to implement event processing logic twice. Thus, we were not able to perform such an evaluation. Instead, we decided to conduct case studies. In these studies, we developed multiple building blocks by applying the developed methodology in various use cases and discuss experiences made during the development. The main software artifacts under evaluation in the case study are tools we provide to support the *Setup Phase*, namely the *StreamPipes Software Development Kit (SDK)* and the *Model Editor*. Results from this study are presented in section 9.2.

Research Question 2 (Model)**.** *How can we model event processing blocks independent from their specific run-time implementation?*

Research Question 3 (Domain Knowledge)**.** *How can we separate domain knowledge from the technical specification of event processing languages?*

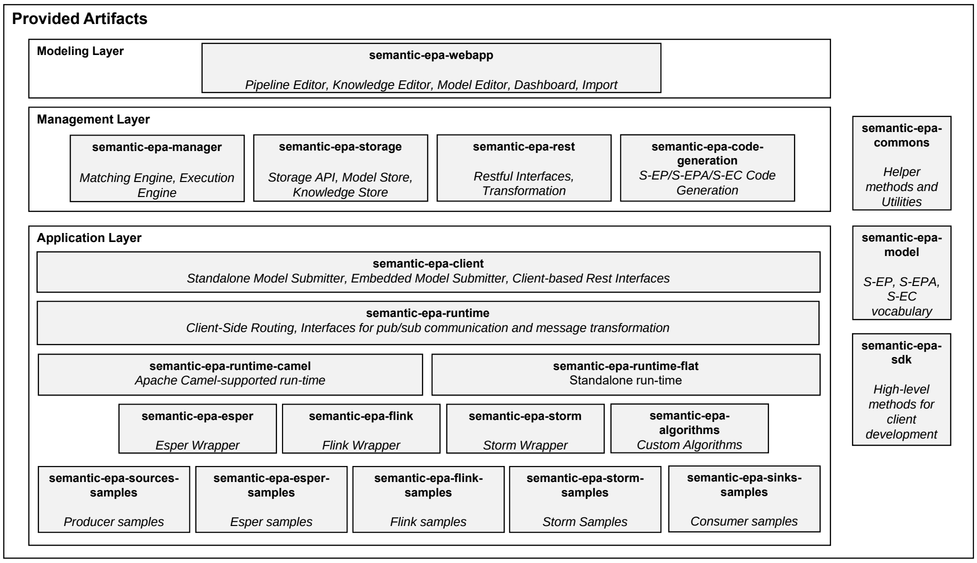
Research questions 2 and 3 deal with the development of a model supporting implementation-independent description of event producers, processing agents and consumers and the separation of background knowledge from an event pattern specification. In order to indicate the fulfillment degree of our approach related to these research questions, the expressivity of our vocabulary needs to be investigated. Expressivity deals with the ability of our model to support the targeted use cases and indicates whether or vocabulary is missing any required concepts. Therefore, the conduction of case studies can also give information on the expressiveness of our vocabulary. In addition, we perform a conceptual investigation by analyzing the fulfillment degree of our approach compared to the requirements elicited in chapter 5 as well as a comparison to the theoretical hierarchy of event processing agents from the literature as introduced in chapter 2. The conceptual investigation is conducted in section 9.3.

Research Question 4 (Execution)**.** *How can we author and execute event processing pipelines consisting of heterogeneous processing blocks?*

The last research question deals primarily focuses on the *Execution Phase*, especially the authoring and execution of processing pipelines. Pipeline authoring is a task performed by pattern engineers. The primary evaluation metric of interest related to pipeline authoring is the usability of the pipeline editor itself. We investigate usability by presenting a user study we conducted together with partners from the industry in one of our research projects. This study also covers the usability of the knowledge editor used by business analysts during the setup phase. Finally, we conduct performance tests. Performance tests are used in order to gain insights on the run-time performance. Compared to the deployment of pipelines in a single-host system, our approach adds overhead in terms of message broker communication due to its distributed execution model. Therefore, we present performance tests which use several run-time wrappers we developed as part of our framework, which we compare against a single-host solution.

9.1.1 Software Artifact

Although our methodology is not bound to a specific tool, we have developed StreamPipes as a reference implementation. Most evaluation methods therefore make use of StreamPipes as the evaluated software artifact. The framework provides complete tool coverage for our methodology and consists of about 140,000 lines of code packaged into 30 Maven[1] modules containing the implementation itself in addition to samples we developed applying the methodology in various projects. The component architecture is illustrated in figure 9.2. Thus, we briefly describe the individual components.



Provided Artifacts
Modeling Layer
semantic-epa-webapp
Pipeline Editor, Knowledge Editor, Model Editor, Dashboard, Import
Management Layer
semantic-epa-manager
Matching Engine, Execution Engine
semantic-epa-storage
Storage API, Model Store, Knowledge Store
semantic-epa-rest
Restful Interfaces, Transformation
semantic-epa-code-generation
S-EP/S-EPA/S-EC Code Generation
Application Layer
semantic-epa-client
Standalone Model Submitter, Embedded Model Submitter, Client-based Rest Interfaces
semantic-epa-runtime
Client-Side Routing, Interfaces for pub/sub communication and message transformation
semantic-epa-runtime-camel
Apache Camel-supported run-time
semantic-epa-runtime-flat
Standalone run-time
semantic-epa-esper
Esper Wrapper
semantic-epa-flink
Flink Wrapper
semantic-epa-storm
Storm Wrapper
semantic-epa-algorithms
Custom Algorithms
semantic-epa-sources-samples
Producer samples
semantic-epa-esper-samples
Esper samples
semantic-epa-flink-samples
Flink samples
semantic-epa-storm-samples
Storm Samples
semantic-epa-sinks-samples
Consumer samples
semantic-epa-commons
Helper methods and Utilities
semantic-epa-model
S-EP, S-EPA, S-EC vocabulary
semantic-epa-sdk
High-level methods for client development

Figure 9.2 Implementation: Provided Software Artifacts

The modeling layer is implemented by the module SEMANTIC-EPA-WEBAPP, which contains all user-related components that include graphical editors, i.e., the pipeline editor, the model editor and the knowledge editor. In addition, this module contains an implementation of a dashboard showing real-time visualizations for pipelines which use a visualization as event consumer. The webapp also allows to import new pipeline elements at run-time by requiring users to specify the link to a URL which returns a description graph.

The management layer serves as a backend to the modeling layer and consists of several modules leveraging the definition and execution of pipelines. SEMANTIC-EPA-MANAGER realizes the matching engine triggered during pipeline definition and the execution engine responsible for the generation and distribution of invocation graphs. The component SEMANTIC-EPA-STORAGE serves as an API for storage-related services and

[1] http://maven.apache.org

implements the database connection to OpenRdf Sesame[2] which we use as a triple store. SEMANTIC-EPA-REST provides RESTful interfaces for user interface interaction, while SEMANTIC-EPA-CODE-GENERATION realizes the generation of Java code in order to prepare implementations for new elements that have been created by using the model editor. On the application layer, the first three layers from the top implement implementation-independent run-time logic, e.g., HTTP endpoints used to serve and receive description and invocation graphs (SEMANTIC-EPA-CLIENT) and to realize the connection to message brokers and message transformation (SEMANTIC-EPA-RUNTIME). The layer below consists of several modules implementing the run-time wrappers we already introduced in chapter 6. Finally, sample projects are available which demonstrate the definition and usage of S-EPs, S-EPAs and S-ECs. These samples also implement most of the functionality used within the case studies.
Finally, three modules contain layer-independent implementations, most importantly, SEMANTIC-EPA-SDK which contains the software development kit that assists developers to define new pipeline elements and SEMANTIC-EPA-MODEL containing the model used to create description and invocation graphs.

9.2 Case Studies

In order to derive information whether our methodology is able to reduce the development effort of event processing applications, we have applied our methodology during the development of event processing applications in several use cases. In total, we have created more than 100 pipeline elements including producers, S-EPAs for all supported run-time wrappers and S-ECs in form of data storage components, visualizations and actuators. Some examples are presented in this section.
This section is structured as follows: First, we present generic pipeline elements we initially developed to support a wide variety of use cases. Afterwards, we present three case studies and explain pipeline elements we have developed specifically for these use cases. The first case study, presented in section 9.2.2 is taken from the application of our methodology for an industry partner from the automotive industry in the manufacturing domain. The second case study elaborates on the application of our methodology for a demonstration production facility (section 9.2.3). Finally, in section 9.2.4 we present a case study conducted for the Debs Grand Challenge, a competition aimed at evaluating event-based systems using complex real-world scenarios. At the end of this section, we discuss experiences we gathered from these case studies in relation to the development effort.

[2] http://openrdf.org

9.2.1 Generic elements

Before diving deeper into the specific problems within each use case, we introduce some pipeline elements we have developed as an initial set of generic elements. These elements implement basic operators and features closely related to standard event processing functionality such as pattern detection and filtering. Table 9.1 summarizes these elements.

Generic elements in this sense are S-EPAs and S-ECs with few stream requirements, e.g., they work in general for a large variety of input event streams. For instance, pattern detection is supported by the S-EPAs Absence to detect missing events, And to detect co-occurrences within a specified amount of time and Sequence to detect two events occurring one after the other within a specified time window. In addition, basic EPA types are provided, e.g., to join events based on a Compose EPA and to remove event properties from a stream using a Project EPA. The goal of developing generic elements was not to provide a complete set, but rather to start with an initial set of components which we believed might be frequently re-used across different application scenarios.

Table 9.1 Case studies: Initial set of generic elements

Name	Type	Description
Absence	S-EPA	Detects the absence of an event in the form a AND NOT b WITHIN time_window
Aggregation	S-EPA	Aggregates (min, max, sum, avg) an event property over a sliding window
Count	S-EPA	Counts a numerical event property value over a sliding window
Rate	S-EPA	Calculates the rate in which events are arriving per second
Compose	S-EPA	Implementation of the Compose EPA (see section 2.3.1)
Math	S-EPA	Performs basic math operations on numerical event properties
Numerical Filter	S-EPA	Filters numerical event properties based on a filter condition
Projection	S-EPA	Implementation of the Project EPA (see section 2.3.1)
Text Filter	S-EPA	Filters text-based event properties based on a filter condition
And	S-EPA	Detects the co-occurrence of two events in the form a AND b within time_window
Sequence	S-EPA	Detects a sequence of events in the form a FOLLOWED_BY b within time_window
Table	S-EC	Visualizes an event schema in a real-time table
Line Chart	S-EC	Visualizes numerical event properties in a real-time line chart
Jms Publisher	S-EC	Publishes an event stream to a Java Message Service (JMS)-enabled message bro-ker
Gauge	S-EC	Visualizes numerical event properties in a gauge chart

9.2.2 Case study 1: Manufacturing

The first case study was conducted in collaboration with an automotive supplier in the manufacturing domain. Our goal was to provide an integrated view on the current status of production processes. This implied the need for a) monitoring machine-related key performance indicators, b) to detect early warnings on upcoming situations, which might cause production failures and c) it was required to continuously store gathered data in a database for off-line analytics purposes.

Setup Phase

In the setup phase, we first had to create models for various sensors. This included sensors for several production machines and sensors equipped along a transportation system used to move products between machines and environmental sensors. In addition, a human sensor type had to be integrated in order to process data gathered from employees during the production process. While specific adapters for the consumption of data from these sensors have already been put in place, we only needed to develop our model. Modeling of sensors was mainly done by using the model editor, which allowed to directly import sensors into the system as no further implementation was required.
In addition, several additional S-EPAs and S-ECs were developed in order to support the targeted use cases. First, we created generic elements to detect relative increases of numerical data values. The S-EPA Increase implements this functionality. It allows users to define a percentage value which represents a relative decrease or increase. In addition, a data window type (time and count) can be defined along with a window size. For instance, this S-EPA allows to recognize steadily rising temperature values measured in a machine. In addition, a NumberClassification S-EPA was developed which allows to assign a label to an event property based on the value of a numerical event property (e.g., by assigning the label *low* to an event that contains a sensor measurement between 0 and 20). This label can be used in order to perform off-line analyses based on previously classified data. In this use case, we also had to integrate some external algorithms which apply a previously learned model on data streams in order to derive predictions. While we did not develop the algorithm itself, we have provided the model of the S-EPA in order to use the algorithm as part of a pipeline. Besides S-EPAs, we developed a new consumer element that allows to store data in Elasticsearch[3], which was intended to be used for analysis of historical data and another element that is capable to forward data to a message broker. Finally, a notification component was developed which enables business analysts to receive immediate notifications on desired situations.

[3] A distributed schemaless search engine, see http://elasticsearch.co

The full set of custom implementations provided for this use case is summarized in table 9.2.

Table 9.2 Manufacturing: Elements developed in the Setup Phase

Name	Type	Description
Machine Sensor 1	S-EP	Machine parameters
Transportation Shuttle	S-EP	Movement of transportation shuttles
Environmental Sensor	S-EP	Environmental sensor parameters
Human Sensor	S-EP	Data gathered from human input
Number Classification	S-EPA	Labels numerical data based on a customizable value range
Increase	S-EPA	Detects an increase of a numerical value over a customizable time window
Algorithm #1	S-EPA	Confidential algorithm #1
Algorithm #2	S-EPA	Confidential algorithm #2
Field Renamer	S-EPA	Renames event property keys
Elasticsearch Sink	S-EC	Stores events in an Elasticsearch cluster
Kafka Publisher	S-EC	Forwards events to an Apache Kafka broker
Notification	S-EC	Triggers a notification in the StreamPipes Dashboard

Execution Phase

The execution phase was planned to be performed by the partner itself. In order to enable business analysts of the company to create processing pipelines, we conducted a training session of about one hour and explained the purpose of individual pipeline elements. In general, many generic elements could already be re-used within this use case. For instance, aggregation and pattern detect operators could be directly re-used without customization. An example pipeline we used within this use case is illustrated in figure 9.3. This pipeline is used for continuous ingestion of sensor data and consists of two input sources, machine data (1, the exact purpose is omitted in this case due to confidentiality) and environmental sensor data (2). Both streams are first joined (3), before a parameter is aggregated (4). Afterwards, event property names are renamed (5), a subset of properties is selected (6) and events are continuously stored in Elasticsearch (7).

9.2.3 Case study 2: Smart Automation

The second case study also deals with manufacturing systems. The *SmartAutomation* system (illustrated in figure 9.4) is a demonstration production facility which serves to provide researchers with a non-productive, but real-world system that allows to deploy and test new technologies. The demonstrator consists of a water pump station and a set of water tanks allowing to pump water from one tank to another. Multiple

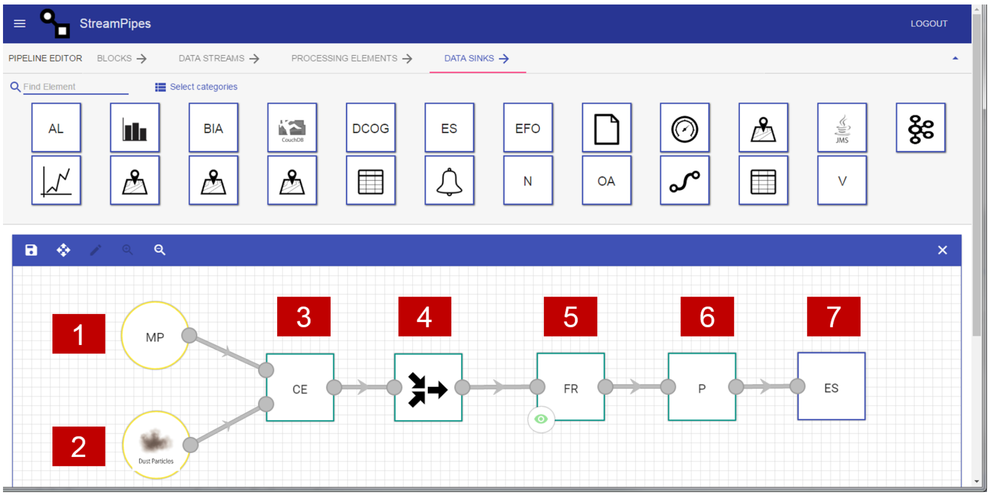


Figure 9.3 Manufacturing Use Case: Example Pipeline

sensors are attached to the demonstrator that can be used to monitor the current state of the system. In addition, various valves can be used to modify the water flow through the system and therefore leverages to simulate potential failures or undesired situations. For instance, water level sensors continuously monitor the water level per tank and flow rate sensors are able to measure the flow rate within water pipes.
We used this system in order to implement a live demonstrator which explains how to define processing pipelines using StreamPipes. For the implementation of the demonstrator itself, we applied our own methodology to define data sources and specific S-EPA and S-EC implementations required for the demonstration.

Setup Phase

In the setup phase, we started with the development of event sources. In contrast to the manufacturing use case, adapters to gather data from hardware sensors did not yet exist. We first developed a model for each producer in the system as depicted in table 9.3. Afterwards, for each producer an adapter was implemented. The next step included the definition of S-EPAs and S-ECs. In terms of S-EPAs we only added one new pipeline element as most required functionality was already covered by existing elements. The ObserveNumerical S-EPA operates on events containing numerical values and detects whether a sensor value is out of a defined range for a customizable time window. Additionally, three different S-ECs have been created. The first consumer element serves for logging purposes and continuously writes events to a file. Second, we created a controller component for an actuator in form of an alarm light.

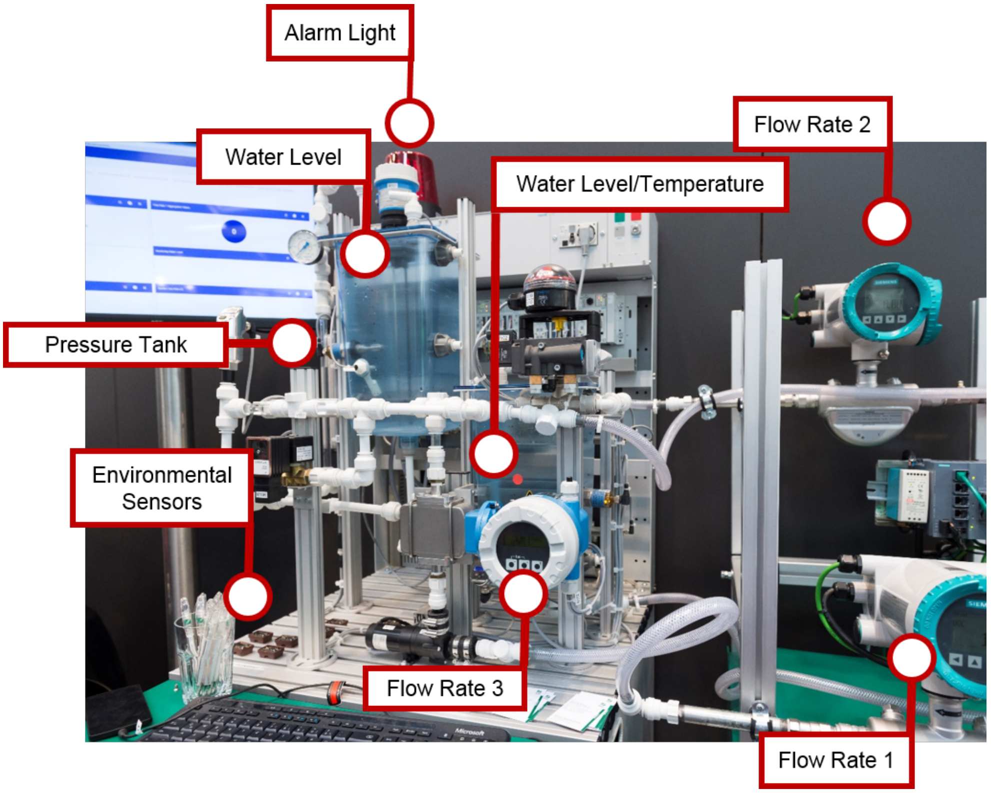


Figure 9.4 Smart Automation Use Case: Demonstrator

The implementation logic of this S-EC comprises the generation and submission of messages to trigger the alarm light, while the model defines a SingleValueSelection static property to let users choose whether to turn the alarm on or off. Finally, another visualization was created in form of a VerticalBarChart, which provides customizable minimum and maximum values and is used in this use case in order to visualize the current water level.

Table 9.3 Smart Automation: Elements developed in the Setup Phase

Name	Type	Description
Water Level Sensor	S-EP	Measures the water level in a water tank
Tank Pressure Sensor	S-EP	Measures the pressure in a pressurized water tank

Continued on next page

Name	Type	Description
		Continued from last page
Flow Sensor (Siemens)	S-EP	Measures the flow rate in a water pipe
Flow Sensor (Festo)	S-EP	Measures the flow rate in a water pipe
Temperature	S-EP	Measures the water temperature
Observe Numerical	S-EPA	Detects whether a numerical value is out of range
File Writer	S-EC	Writes events to a file
Alarm Light	S-EC	Controls an alarm light
Vertical Bar Chart	S-EC	Visualizes a vertical bar chart

Execution Phase

In the execution phase, due to the intended scope of the use case to demonstrate the capabilities of the pipeline editor, we mainly focused on developing pipelines with visualizable elements as data sinks. An example pipeline is illustrated in figure 9.5. This pipeline uses two data sources as an input, water temperature (1) and water level (2). Temperature values are processed by the INCREASE S-EPA (3) mentioned above, while the water level is processed by a OBSERVENUMERICAL S-EPA in order to detect a low water level (4). Once both S-EPA trigger an event within a time window of 30 seconds (5), the alarm light is triggered (6).

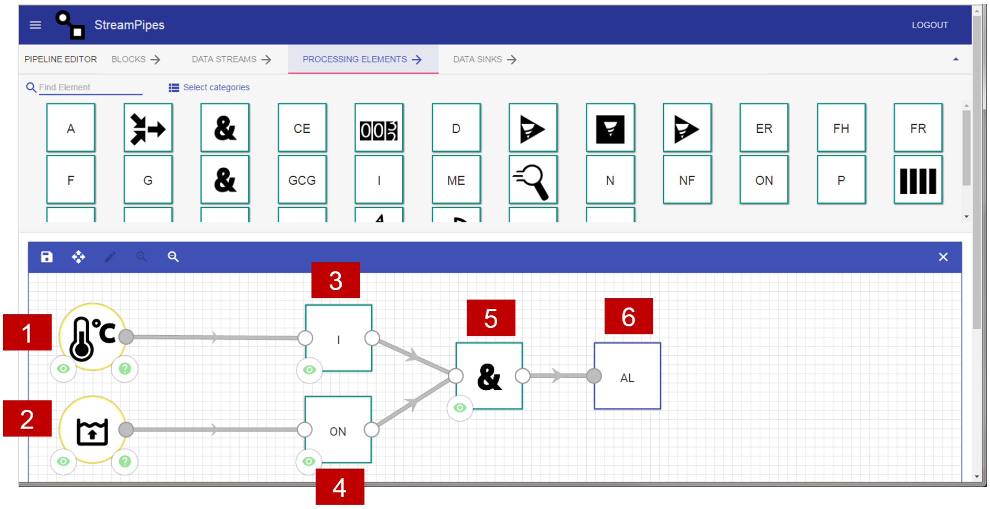


Figure 9.5 Smart Automation Use Case: Example Pipeline

9.2.4 Case study 3: Debs Grand Challenge

In order to show the ability of our approach to also support more complex scenarios, we participated in the Debs Grand Challenge 2015. The ACM Conference on Distributed Event-Based Systems[4] is a major conference in the area of event processing and provides a forum for researchers to exchange recent developments around event-based systems. The overall goal of the DEBS Grand Challenge is to provide a common ground for researchers to evaluate and compare event-driven systems. The 2015 Grand Challenge provided a data set of publicly available taxi usage data from New York City. The whole dataset includes around 173 million events (where an event includes 17 parameters such as pickup time, location, drop-off location, fare amount, tip amount and trip duration) collected for one year in 2013. The main tasks of the Debs Grand Challenge 2015 were defined as follows:

- **Frequent Routes (Query 1).** The goal of the first query was to output the top 10 most frequent routes during the last 30 minutes. A route was defined as a trip between two cells, whereas a cell was defined as a rectangle of 500 square meters starting from a fixed location outside of New York City. A trip could be derived directly from the data set, where a single entry contained the location of the pickup and the drop-off.
- **Profitable Areas (Query 2).** The goal of the second query was to find the most profitable areas within the last 15 minutes based on a profitability value that had to be computed by dividing the median profit (defined as fare amount plus tip) with the number of empty taxis.

Setup Phase

In the setup phase, we had to create a set of new pipeline elements as depicted in table 9.4. For each query, we developed components in two ways. First, for each query we designed an S-EPA which includes the whole implementation logic required to produce the intended result of the query in a single component. Therefore, the Debs Grand Challenge 1 S-EPA was designed to expect an event containing four event properties representing the coordinate values for the pickup and drop-off.
As for our second solution, we aimed to create re-usable pipeline elements, where each pipeline element should perform a single processing step of the whole query. We therefore had to develop the following additional S-EPAs:

- **Grid Enrichment.** The Grid Enrichment S-EPA requires a location-based event stream and assigns a cell identifier to the location. The cell size is modeled as a SingleValueProperty, whereas the starting location used to compute the cell size is modeled as a DomainStaticProperty linking to the knowledge base.

[4] http://www.debs2015.org/

- **Top-k.** The Top-k S-EPA stores incoming events in a sliding time window and sorts events based on an event property. The output stream contains a list of the events containing the top-k values, while the output frequency can be configured during the pipeline authoring process.
- **Univariate Statistics.** This S-EPA offers multiple options to calculate univariate statistics. For instance, the median of a property value can be continuously calculated over a sliding time window. We created this S-EPA in order to support the median fare calculation required for query 2.

These S-EPAs (among additional elements as summarized in table 9.4) were mostly implemented using the wrapper for the Esper engine. The solution for query 2 was also implemented using the Apache Storm run-time wrapper. In order to define the description graphs, we used the StreamPipes SDK, as the model editor was not yet fully functional at this time.

Table 9.4 Debs Grand Challenge: Elements developed in the Setup Phase

Name	Type	Description
Taxi Data Producer	S-EP	Simulator to read taxi data from a CSV file and publish to a message broker
Grid Enrichment	S-EPA	Assignment of location-based events to a grid of a customizable size
Geofencing	S-EPA	Detect whether a location-based event arrives within a customizable radius around a customizable center coordinate
Top-k	S-EPA	Outputs a list sorted by a customizable event property value of a customizable size within a customizable time window
Univariate Statistics	S-EPA	Calculate univariate statistics for an event property
Debs Challenge 1	S-EPA	Calculates the result of Task 1 of the Grand Challenge
Debs Challenge 2	S-EPA	Calculates the result of Task 2 of the Grand Challenge
Map	S-EC	Displays location-based event properties in a real-time map
Route	S-EC	Displays routes in a real-time map
Heatmap	S-EC	Generates a heatmap of aggregated location-based events over a sliding time window
Bar Chart	S-EC	Real-time bar chart

Execution Phase

In the execution phase, we defined the pipelines required to solve the Grand Challenge. Figure 9.6 shows the query 1 modeled as a processing pipeline in the Pipeline Editor. As a data source, the taxi event simulator is being used (1). In order to assign the pickup and drop-off coordinates to a grid cell, the GridEnrichment S-EPA is invoked two times. First, the cell identifier of the pickup coordinate is calculated and added to the event payload (2), afterwards the drop-off coordinate is calculated and added to the event payload (3). Afterwards, the Count S-EPA (4) counts the number of trips between the pickup cell and the drop-off cell by partitioning the input stream based on the cell identifiers. Events are then forwarded to the Top-k S-EPA (5) which ranks routes depending on the count value and outputs a new result once the ranking changes. Frequent routes are displayed in the dashboard using a table element (6).

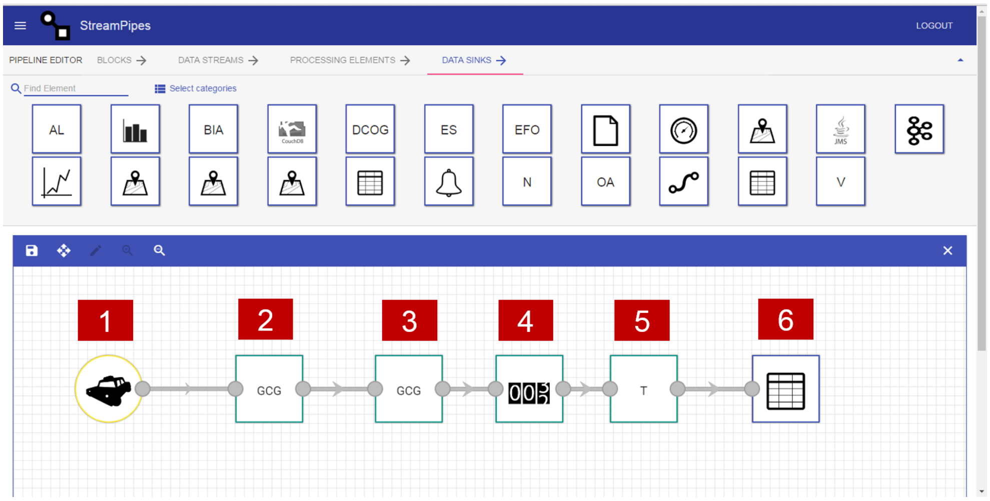


Figure 9.6 Debs Grand Challenge: Query 1

By providing re-usable elements, we were able to show additional scenarios on top of the required queries. For instance, we created various map-based visualization components which were able to show current frequent routes directly in a map. A Heatmap consumer allowed to visualize frequently serves areas. Most importantly, the Debs Grand Challenge provided a good use case to demonstrate the value of re-usable elements. For instance, the GridEnrichment S-EPA was easy to instantiate with other parameters such as other cell sizes and starting locations which allowed us to analyse taxi data based on multiple settings, e.g., finer-grained cells. Compared to the complexity of the scenarios, it is also worth mentioning that only three additional elements had to be created, which demonstrates the usefulness of providing re-usable elements.

9.2.5 Discussion

The goal we pursued when performing the case studies was to get insights on the development effort and to get feedback on possible shortcomings of our model in terms of its expressivity.

Concerning the development effort, our goal was to show that, by using our methodology, the development effort can be reduced by providing re-usable elements that can be used in various settings without developer consultancy. If this is the case, the second assumption that needs to apply is that the development effort to create re-usable elements itself does not increase compared to other implementations. The latter clearly depends heavily on the provided tool support. Therefore, we put effort in providing assistance to developers lowering the barrier to define event processing logic according to our vocabulary to reduce the overhead as much as possible.
Although this is hard to quantify, based on our experience with developing more than 100 pipeline elements, we believe that this is the case. For instance, in our case studies we observed that the development effort for providing event producers mostly depends on the implementation logic, i.e., the development of adapters which connect to the actual systems. This effort is not increased by applying our methodology and is the same as required for standard development of event processing applications. Concerning the development of S-EPAs and S-ECs, the definition of models requires additional effort. However, by providing a model editor to allow graphical modeling of the semantic description, by completely abstracting from specific characteristics and pitfalls of RDF modeling, this effort is significantly reduced.
In addition, the run-time wrappers we developed for various popular event processing systems (as introduced in section 7.6.1) are able to take over most of the additional effort required for the parameterization of pipeline elements and the connection to publish/subscribe systems. By using the provided code generation module, a major part required to lift existing event processing logic to be supported by our methodology is being taken away from developers. In contrast, in order to develop event processing logic using existing systems, effort needs to be spent into the development of input and output adapters, which is not needed when using the tool support we provide.
In terms of re-usability, the case studies have shown that in most cases, it is possible to define event processing logic in a more abstract way by omitting implementation details. Even besides the rather generic components we have developed initially, many use-case specific building blocks could be re-used in other case studies without additional customization effort. We therefore believe that the need to develop new processing elements will be decreased over the time. Also our participation in the Debs Challenge has shown that even more complex tasks can be separated into fine-grained building blocks that serve multiple purposes. Although not further described in this section, we have conducted experiments by employing our methodology in another project in the logistics domain using similar examples as presented throughout the course of this thesis. During these experiments, we were also able to re-use geographical processing elements we initially developed for the Debs Grand Challenge.
However, although our methodology and specifically the tasks related to the setup phase have been successfully performed by multiple software developers including

both people concerned with the implementation of our approach itself and developers who were unfamiliar with the complete framework, we were not able to perform a complete study that quantifies the development effort needed to build event processing using our model to standard development processes. Even though this requires for deeper investigation using longer-term studies, we are confident that our methodology is able to significantly improve the way event processing applications are developed.

9.3 Conceptual Investigation

In this section, we evaluate the expressivity of our approach. Although the case studies have already provided clues in terms of the completeness of our vocabulary, this section elaborates on the expressivity using a conceptual investigation method. In section 9.3.1, we therefore discuss the fulfillment of requirements along the requirements identified in chapter 5. Section 9.3.2 investigates the expressivity of our approach in relation to the hierarchy of event processing agents.

9.3.1 Fulfillment of Requirements

First, we discuss to what extent our approach fulfills the requirements collected in chapter 5.2 by focusing on requirements related to the development process. Table 9.5 summarizes these requirements and their fulfillment degree. The first two requirements are directly implemented in our methodology. Generic event processing logic is supported by the pipeline element modeling tasks which produce event processing building blocks independent from specific event streams. The second requirement, domain-specific event processing is provided by the methodology itself, i.e., the setup phase which is designed to be a developer-oriented preparation phase that has the goal to develop domain-specific event processing logic once requirements arise from the execution phase. Requirement R3, definition of event processing applications using graphical modeling languages, is supported by our methodology within the pipeline authoring task of the execution phase as well. In addition, tool support for graphical modeling is provided as part of the StreamPipes framework, our reference implementation to the methodology. Requirement R4 is fulfilled by the provision of a vocabulary supporting the setup phase and also in form of tool support with the provision of a model editor to instantiate models in order to generate S-EP, S-EPA and S-EC descriptions and the Software Development Kit.
Requirement R5 deals with the separation of domain knowledge from the event processing logic. Our methodology implements this requirement by introducing a separate task targeted at business users within the setup phase which supports the definition of knowledge apart from the development of event processing logic. In addition, our vocabulary includes concepts to define static data requirements that link to concepts available in the knowledge base, allowing pattern engineers

to use domain knowledge during the pipeline authoring task. Finally, adaptivity of EP applications (R6) is supported by the pipeline identification and authoring tasks within the execution phase, which exactly targets this requirement by providing strategies and tools to create processing pipelines based on re-usable elements.
Requirements R7-R13 deal with the model used to describe event producers, processing agents and consumers. While requirement R7 is further investigated in the next section, requirement R8 demands for expressive representation of event schemes We support this requirement by providing a vocabulary which includes concepts and properties to define a schema based on its event properties, whereas event properties can be described using the provided RDF properties propertyType, runtimeName) and domainProperty. In addition, we added support for representing value ranges in form of quantitative values and enumerated property types.
Requirement R9, dealing with the support for quality-oriented aspects related to events and processing agents, is fulfilled by the concept StreamQuality and PropertyQuality in our vocabulary. These concepts can be used to express offered quality attributes on both event property level (e.g., accuracy of measurement values) and stream level (e.g., frequency of streams) by event streams and in form of *quality requirements* to express minimum quality requirements processing elements expected to be provided.
In order to integrate static data, as demanded by requirement R10, our vocabulary provides various concepts to define static data requirements which express additional configuration parameters required to be provided by pattern engineers in order to instantiate the implementation logic of a processing element. We also provide a concept DomainStaticProperty which allows to express static data that can be directly connected to instances from a knowledge base. Further concepts to represent static data have been included in form of SingleValueProperties, MultiValueProperties and MappingProperties, besides others.
Finally, Requirements R11-R12 express the need for a model independent from run-time formats and run-time protocols. We ensured this in our vocabulary by the provision of the concept StreamGrounding, which allows an S-EPA to express multiple supported groundings based on the TransportFormat and TransportProtocol.
R13, which required the usage of RDF as a data model throughout the system is ensured by using an RDF-based knowledge store in addition to RDF-based descriptions for event producers, event processing agents and event consumers. R14 is fulfilled by the presented vocabulary, which provides abstractions from event processing operators in form of multiple static properties.
Requirements R15-R17 deal with system-related requirements. Extensibility (R15) which demands for the system's capability to be extended with new pipeline elements at run-time is ensured by our architecture (see chapter 8.1), which allows to make new elements available in the execution phase by providing the link to an URL which returns a description graph. Requirement R16 is realized by the matching engine, which is intensively discussed in chapter 8.3.

Requirements related to failure recognition (R17) and fault tolerance (R18) are partly fulfilled by our methodology. In terms of detection of failures, our system is able to detect failures during pipeline deployment and to propagate errors occurring at run-time from the individual run-time wrappers back to the pipeline authoring environment. Therefore, failures in processing nodes can be detected. In contrast, fault tolerance is not necessarily supported. A completely fault tolerant pipeline does not only depend on fault-tolerant pipeline elements (which can be ensured by selecting a run-time event processing system which supports fault tolerance, e.g., Apache Flink), but also in-between, which requires for handling of fault tolerance on the message broker level. Although our system supports the development of processing pipelines on architectures that support fault tolerance (e.g., Flink wrappers in combination with Apache Kafka as a message broker), it cannot necessarily be ensured by our system. Requirement R19 requires for support to integrate geographically distributed execution, which is directly implemented in our conceptual architecture. As pipeline elements communicate by the means of description graphs and invocation graphs that are made available over standard web protocols, processing pipelines are able to integrate geographically distributed systems. In terms of heterogeneous run-time technologies (R20), we could show the support for multiple existing event processing systems in form of provided run-time wrappers.

Table 9.5 Requirements Fulfillment

#	Description	Fulfilled by
R1	Generic event processing logic	Methodology, Setup Phase (Chapter 6.5)
R2	Domain-specific event processing logic	Methodology, Setup Phase (Chapter 6.5)
R3	Graphical Modeling Language	Execution Phase, Pipeline Authoring, StreamPipes Pipeline Editor (Chapter 8.3)
R4	Developer Support	Setup Phase, Pipeline Element Modeling, StreamPipes SDK (Chapter 7.6)
R5	Knowledge Modeling	Setup Phase, Domain Knowledge Modeling, StreamPipes Knowledge Editor (Chapter 7.6)
R6	Adaptivity of EP applications	Execution Phase, Pipeline Authoring (Chapter 8.3)
R7	Event processing agents	Vocabulary, S-EPAs (Chapter 7.4)
R8	Event schema	Vocabulary, Event schema (Chapter 7.3)
R9	Quality of events	Vocabulary, Event Schema (Chapter 7.3)
R10	Static data	Vocabulary, S-EPAs, Static Properties (Chapter 7.4)
R11	Run-time representation: Event format	Vocabulary, Stream Grounding, Run-time wrapper (Chapters 7.3 and 7.6)
R12	Communication protocol	Vocabulary, Stream Grounding, Run-time wrapper (Chapters 7.3 and 7.6)
R13	Design-time representation: Model	Vocabulary (Chapter 7)

Continued on next page

#	Description	Fulfilled by
		Continued from last page
R14	Abstraction	Vocabulary (Chapter 7)
R15	Extensibility	Setup Phase, Deployment (Chapter 7.6)
R16	Matching	Execution phase, pipeline authoring task (Chapter 8.3)
R17	Failures	Execution phase, pipeline deployment (Chapter 8.4)
R18	Fault tolerance	depends on run-time implementation
R19	Distributed execution	Architecture (Chapter 8.1)
R20	Heterogeneous run-times	Architecture, run-time wrappers (Chapter 7.1)

9.3.2 Supported Event Processing Agents

This section investigates the expressivity of our vocabulary in terms of its support for the hierarchy of event processing agents as defined by [Etzion and Niblett 2010]. By assuming that any event processing logic can be assigned to one of the agents introduced in section 2.3.1 and summarized in table 9.6, we are able to show that our vocabulary is expressive enough by showing that each of the basic EPAs can be properly described.
In table 9.6, for each EPA a mapping to specific properties of our model is illustrated with *IS* indicating the minimum number of input streams, *OS* indicating the number of output streams and *Static Properties* indicating the minimum set of static properties needed to define such an EPA. *Output Strategy* assigns possible output strategies to each EPA.
In general, all abstract EPA types are supported with the exception of the *Split EPA*. In chapter 8.2, we have discussed why we decided to not support split pipeline elements and have shown that a split element can be replaced by the usage of *partial pipelines* in our model. For the other elements, we regularly assume the most basic use case without any further provided configuration. For instance, a Filter EPA can be modeled implemented by assigning a KeepOutput strategy along with a single MappingPropertyUnary which links to an event property where the filter condition should be applied upon. In this case, no further configuration is possible and we assume that filter conditions are hard-coded in the S-EPA implementation. In order to provide more configuration options, additional static properties (such as SelectionStaticProperty to allow for the selection of multiple supported filter operations) can be assigned. Translate EPAs can be defined by using a TransformOutput strategy along with a mapping property specifying the event property that will be modified by the S-EPA. Aggregations as an example for stateful event processing agents, can be implemented with a single mapping property and an output strategy of type Append or Fixed. Compose EPAs require for two input event streams and can be used with multiple output strategies. An Enrich EPA can be described without the need for providing a mapping

property (in this case, the same value would be added to each incoming event) and needs to provide an Append output strategy. Project EPAs, which output a subset of the event properties of an input stream, can be directly represented in our model by using a Custom output strategy. Finally, pattern detect EPA minimally require for two input streams and may be implemented using multiple output strategies. No further static properties are required assuming that the configuration is completely fixed and hidden from the user.

Table 9.6 Mapping of EPA types to our model

EPA	IS	OS	Static Properties	Output Strategy
Filter	1	1	MPU	Keep
Translate	1	1	MPU	Transform
Aggregate	1	1	MPU	Append, Fixed
Split	-	-	-	-
Compose	2	1	-	Keep, Custom, Fixed
Enrich	1	1	-	Append
Project	1	1	-	Custom
Pattern Detect	2	1	-	Append, Custom, Fixed

In summary, we can show that basic EPA types are supported either by our model directly or are indirectly supported by features of processing pipelines. The result therefore indicates that the set of output strategies we have introduced is complete enough to support all abstract event processing agents from the literature.

9.4 User Study

In this section, we present results gathered in form of a user study in the manufacturing domain. The user study was performed in order to get insights on the acceptance of the pipeline editor. While we were able to study our methodology in the setup phase during the case studies, the goal of this study was to gain insights on the execution phase, i.e., the ability of non-programmers to create processing pipelines in a self-service manner.

9.4.1 Setup

The target user group for our study were business analysts and operators of production companies. Our goal was not to perform a study with a broad spectrum of participants, but to find business experts which were able to identify situations within their domain of interest in order to find out if they were able to model such situations by using

the pipeline editor. The individual participants of our study were selected by the companies themselves. In the end, six experts participated in the survey. In the first step, we prepared an instance of the StreamPipes framework for each company and created pipeline elements (event producers, processing agents and consumers) specific to the targeted use cases. After the installation, we provided the participants with a short documentation explaining the functionality of the pipeline editor and, in addition, provided a short video that demonstrated the usage of the editor.
After a test phase lasting two weeks, participants were given a survey consisting of 20 questions, 14 of them related to their user experience with the pipeline editor. All answers had to be provided using a 5 point Likert scale ranging from 1 (strongly disagree) to 5 (totally agree).

9.4.2 Results

In this section, we present results from the user study. The complete survey and individual answers are shown in appendix B. The first question asked about previous experience with stream processing applications. 4 out of 6 participants had low experience with this kind of applications.
Individual answers to the questions related to the usage of the pipeline editor are illustrated in figures 9.7, 9.8, 9.9 and 9.10. Questions 1-5 are related to general issues concerned with usability of the system. We asked users on their opinion concerning the overall look and feel of the StreamPipes GUI, the overall usage, the ability to understand error messages and the navigation. In general, results are above average in all categories, while the look and feel as well as the intuitiveness is rated rank 4 or higher. An exception in question 4 concerning the ability to understand system messages is remarkable.
Questions 6 and 7 were asked in order to gather insights whether a tool to create event processing pipelines is useful to solve a specific business problem. Question 6 is concerned with the general capability of the pipeline editor to improve the work experience, while question 7 is concerned with the functionality of StreamPipes in order to fulfill the needs of the participant's jobs. Question 6 is rated well above average, while question 7 only receives average grades. This might be due to the fact that we only presented a limited set of pipeline elements to the participants. Therefore, it is likely that this selection is not able to fully support the requirements in terms of functionality. We plan to further evaluate the pipeline editor after the system has been fully deployed in the individual companies.
Questions 8-12 are concerned with the usability of the pipeline authoring tool itself. We first asked participants to determine whether the general approach to provide a graphical notation to develop event processing applications is useful. This question was positively answered by all participants, which contributes to the assumption that our design decision to provide a high-level graphical modeling language is useful

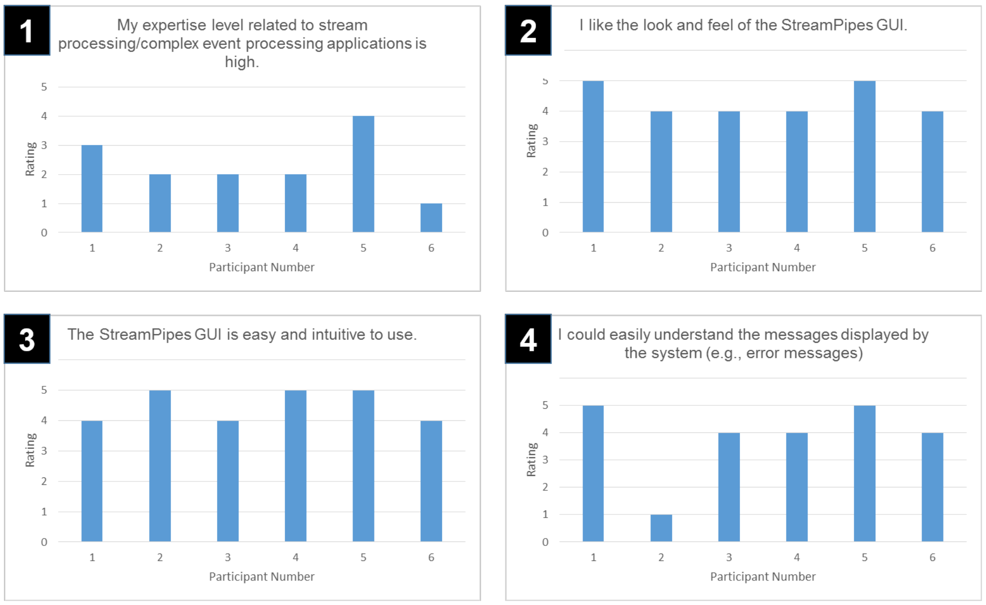


Figure 9.7 User Study: Answers (1)

for the intended target group. Afterwards, we asked users to assess whether they understood the meaning of pipelines. Although most answers were rated above average, one participant did not fully understand the meaning of processing pipelines. This shows that, even though the system was assessed to be simple to use, training is required in order to make users familiar with event processing applications in general and specifically the purpose of pipelines. Question 10, where we asked if users were able to understand how to start and stop pipelines, was completely positively answered by 5 out of 6 participants. Even more important is the result of our user survey in question 11: 4 out of 6 participants strongly agree that the average time to create a pipeline is satisfying. This is a rather interesting result and shows that our approach enables application specialists to create processing pipeline within a satisfying time frame. Question 12 finally tries to figure out how much guidance is needed in order to enable business users to create processing pipelines. Although some users state that they do not need any guidance at all, the answers suggest that training prior to pipeline development might be useful.
Finally, questions 13 and 14 asked for the participant's opinion on specific features of the pipeline editor. In general, users mostly liked the recommendation feature (described in section 8.3), and most users are satisfied with the consistency checking of pipelines, which targets the usefulness of our matching engine. Although we

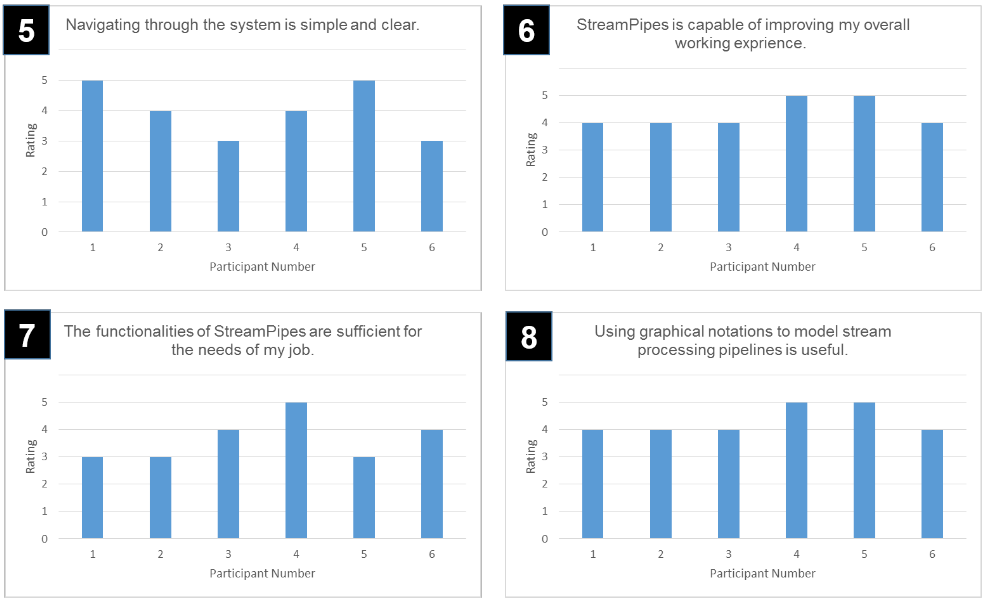


Figure 9.8 User Study: Answers (2)

delivered a fully functional version of the matching engine at the time the user study was conducted, some features related to client-side type checking during pipeline definition were not fully functional and might therefore explain two average and below-average answers.

9.4.3 Discussion

We briefly discuss the results gathered from the user survey. Although a user survey consisting of six participants might provide limited evidence in terms of usability, the results show that our general methodology, to provide re-usable elements to business users in order to enable them to create processing pipelines in a self-service manner, is considered useful.
Figure 9.11 shows the average rating per user. Based on the first question, which asked for the expertise level related to stream processing applications, we additionally categorized the user into two groups. The *Higher Experience* group consists of participants who rated their stream processing experience rank 3 or higher (resulting in a total number of two participants). The *Lower Experience* group consists of participants with an experience level of two or lower (resulting in a total number of four participants). The average rating per user over all questions spans from 3,69 (user 3) to 4,84 (user

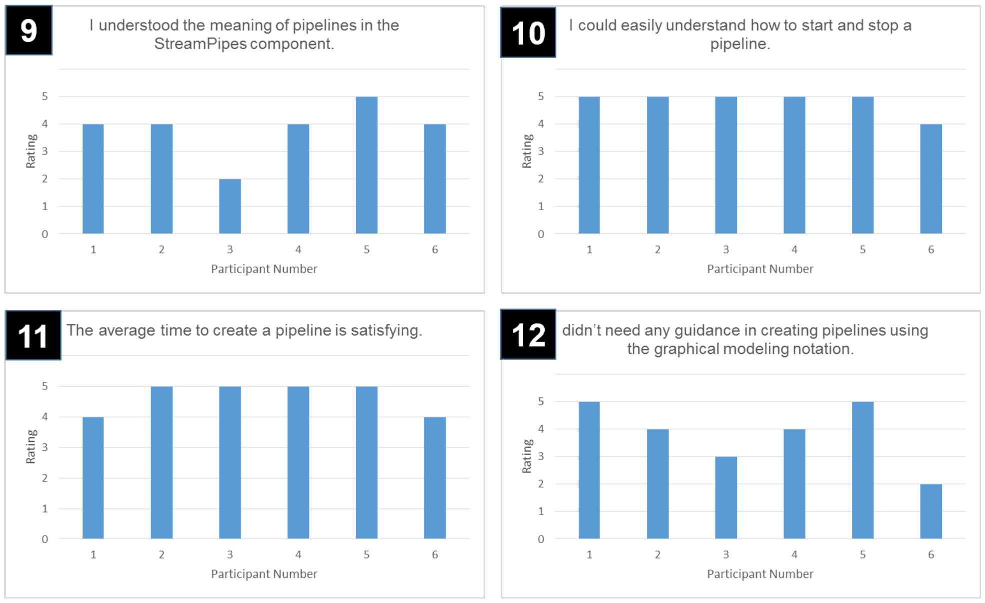


Figure 9.9 User Study: Answers (3)

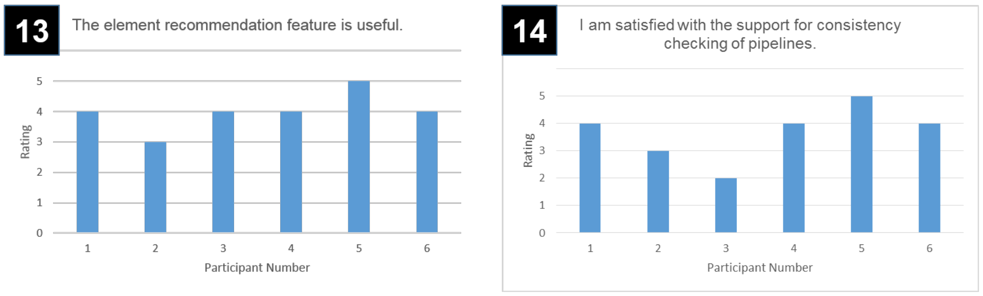


Figure 9.10 User Study: Answers (4)

5). The *Higher Experience* group has an average rating of 4,58 compared to an average rating of 3,92 of the *Lower Experience* group. These results suggest that the usage of our system is considered more useful if there is a higher expertise level related to stream processing applications.
In order to compare the results between two user groups, we further categorized the question type. In general, questions 2-11 mainly asked for the overall impression of our approach to model processing pipelines with graphical tool support and the

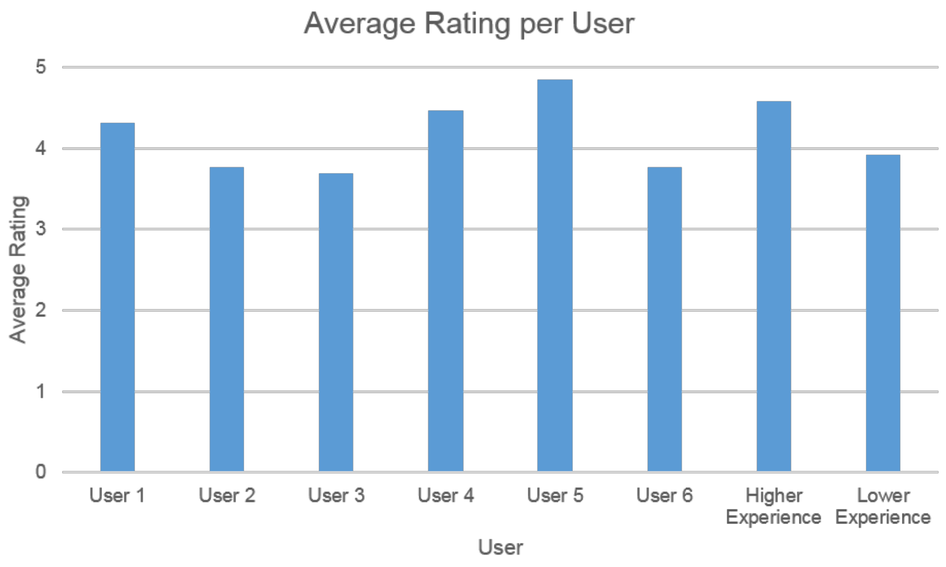


Figure 9.11 User Study: Average Rating per User

system itself, while questions 12-14 are targeted at specific (potentially more advanced) features and the guidance users needed in order to start creating pipelines. Results grouped by these question types are shown in figure 9.12. The group that rated their own experience level higher provides an average rating of 4,55 for questions Q2-Q11 and an average rating of 4,67 for questions Q12-Q14. It becomes obvious that this group rated advanced features such as pipeline recommendation and consistency checking higher than the average ratings of the more basic questions Q2-Q11. By looking at the user group with lower experience, we can see contrary results: Questions Q2-Q11 are rated 4,08 on average, while questions Q12-Q14 only receive average values of 3,42. This result suggests that more training is needed for more inexperienced users and that more advanced features are better understood by more experienced users.
In general, the user survey shows an agreement among all participants that the pipeline editor enables users to create pipelines within a satisfying time period. This strengthens our attempt to divide the development process into two phases and shows that business users generally acknowledge the usefulness of using graphical modeling of processing pipelines in a self service manner (question 8).
The survey also suggests that, even though the navigation and general usage of the pipeline editor is positively assessed by all participants, training is needed in order to enable business users to create pipelines. This is reflected in our methodology, which

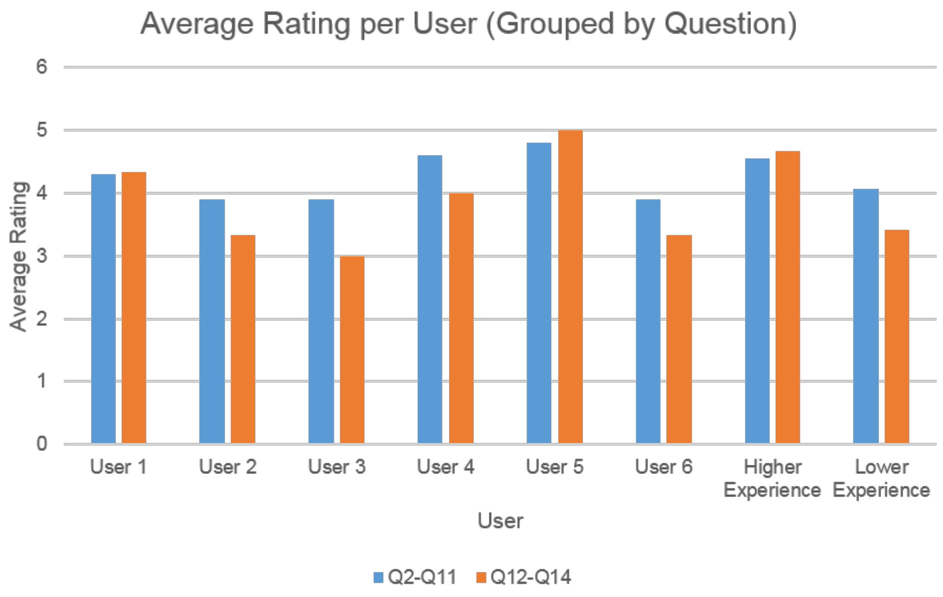


Figure 9.12 User Study: Average Rating per User (Grouped by Question Type)

foresees a new role in form of pattern engineers (as a more specialized role compared to business analysts), who are able to build stream processing pipelines.

9.5 Performance

In this section, we present tests that evaluate how our system performs at run-time. In general, our system relies on the exchange of events between distributed pipeline elements by using publish/subscribe middleware in form of message brokers. Our approach therefore adds overhead to the broker. A comparison to other approaches is illustrated in figure 9.13. In general, single-host systems usually consume events from a message broker and perform the actual event processing logic in a single centralized system, before events are published back to the message broker, where they are available for consumption by external systems. Although this approach has limited load on the message broker, the engine itself, running on a single host, is limited in terms of throughput. Distributed systems, which are able to distribute workload across multiple computing nodes, are able to process hight throughputs of events due to the parallelization of tasks, especially in conjunction with distributed

message brokers. However, distributed programs must be written in a language the system supports and they are depend on a specific run-time implementation.
In contrast, our approach builds on top of these technologies and allows to integrate heterogeneous run-time technologies in a single pipeline. This comes at the cost of broker overhead. Although a single pipeline may facilitate multiple distributed message brokers to exchange data, each pipeline element needs to consume events from the middleware and to publish results back to the broker. In comparison to single-host systems and distributed systems, increased network communication likely has impact on latency.

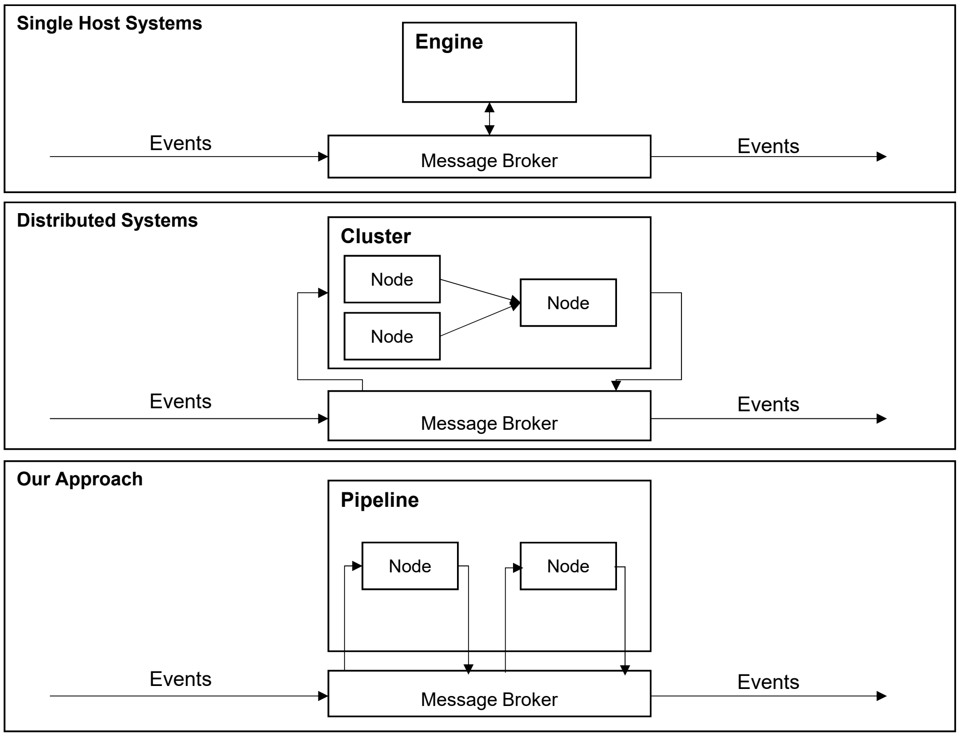


Figure 9.13 Data Flow: Comparison

By conducting performance tests, we aimed to figure out to what extent performance decreases compared to single-host solutions. These results are useful to give decision support concerning the development of event processing systems based on two levels: Depending on the desired performance metrics and the requirement for fast adaptation of processing pipelines in a specific use case, it might be more feasible to implement the application logic as a single-host solution or a single processing element (see our discussion on our solution to the Debs Grand Challenge in section 9.2.4). In addition, performance results also can give advice about the granularity in which re-usable processing elements should be developed. Finer-grained elements tend to produce more broker overhead as they require for larger pipeline sizes, but might be better re-usable, while elements subsuming more logic in a single pipeline elements might be harder to be re-used, but produce less broker overhead.

9.5.1 Settings

Our performance tests were performed on a virtual geographically distributed system consisting of three servers as depicted in table 9.7. These systems were identical but were equipped with different memory sizes. Server 1 was assigned 24 Gigabytes (GB) of memory, server 2 had 12 GB memory and server 3 was equipped with 8 GB.

Table 9.7 Hardware Setup for Performance Measurements

Feature	Server 1	Server 2	Server 3
Memory	24 GB	12 GB	8 GB
CPU	4x 2.3 Ghz	4x 2.3 Ghz	4x 2.3 Ghz
Network	1 Gbit	1 Gbit	1 Gbit
Operating System	Ubuntu 14.04	Ubuntu 14.04	Ubuntu 14.04

Table 9.8 shows the assignment of components used for the performance tests to servers. We briefly summarize the components used:

- **Apache Kafka** was used as a message broker. Although Kafka is designed as a distributed messaging system allowing to spread topics over multiple computing nodes, we conducted our tests using a single instance of Kafka.
- **Apache Zookeeper** is a distributed configuration service required by Kafka.
- **Apache ActiveMQ** was used as a second message broker to simulate the usage of multiple transportation protocols.
- **semantic-epa-actions-samples** contained an S-EC implementation of a file writer component which was designed to write events to a log file.
- **Event Simulator** produced events containing random numbers and was used as an event source for the performance tests.
- **Apache Flink**, an Apache Flink cluster consisting of a single task manager .
- **semantic-epa-flink**, the run-time wrapper for Apache Flink.
- **semantic-epa-esper**, the run-time wrapper for Esper.
- **semantic-epa-backend**, consisting of the pipeline editor and the backend including the matching and execution engine.

- **OpenRdf Sesame**, used as a triple store which contained the description graphs of pipeline elements.

Table 9.8 Assignment of Components to Servers

Server	Components
Server 1	Apache Kafka
	Apache Zookeeper
	Apache ActiveMQ
	semantic-epa-actions-samples
	Event Simulator
Server 2	Apache Flink
	semantic-epa-flink
	semantic-epa-esper
Server 3	semantic-epa-backend
	OpenRdf Sesame

The performance tests were designed as followed: We created pipelines of different sizes, each of them consisting of the same data producer (random number stream, RNS) and a set of S-EPAs of the type EnrichTimestamp. These components correspond to an enrich EPA, where each incoming event is enriched with an additional event property which contains the current time. As a consumer node, we created a file writer component (evaluation file output, EFO), which measures the time difference between the first pipeline element and the last element and outputs the whole event payload to a file. An example pipeline used for the performance tests is illustrated in figure 9.14.

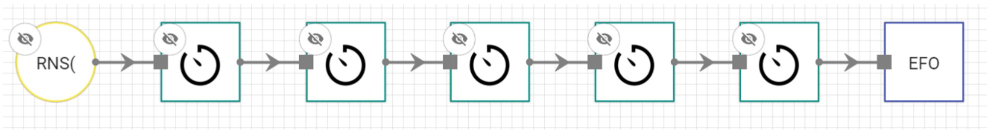


Figure 9.14 Performance Evaluation: Pipeline

We ran several tests by adjusting the pipeline size and the throughput of the random number generator. The throughput was chosen based on performance tests of the Kafka broker itself prior to the evaluation in order to avoid the selection of evaluation settings which exceed the broker's performance capabilities. Finally, we conducted tests for 2 simulation settings (1000 events/sec and 5000 events/sec) using pipeline sizes of 1, 2 and 5. For each test, we produced 100,000 events in order to see whether the processing time remains constant for higher loads. We evaluated three different configurations: The first configuration was a plain Esper (*esper*) instance, which reflects a single-host system. In this system, we manually created event patterns. The second configuration, *semantic-epa-esper*, uses our system and uses Esper nodes as run-time implementation. The third configuration, *semantic-epa-flink*, uses our run-time wrapper for Apache Flink.

9.5.2 Results

First, we present evaluation results from a low-throughput scenario, which was run at a throughput rate of 1000 events per second. Figure 9.15 shows a histogram of each configuration related to the pipeline size. In general, in all configurations almost all events were processed with latency less than 6 milliseconds. Although the *esper* configuration is significantly faster with latency less than a millisecond, this can be explained by the simple use case. The patterns developed for the esper configuration only require to add a timestamp to an existing event, which is a very fast operation on a single-host system. Both configurations relying on our system are comparable in terms of performance with a median value of 2 milliseconds for the pipeline consisting of one node, and about 3 milliseconds for the 2-nodes configuration. For the pipeline of size 5, the latency increases to around 4 milliseconds for the *semantic-epa-esper* and the Flink-based configuration.
The second experiment was performed with higher throughput. In total we simulated 100,000 events with a total throughput of 5000 events per second. The results are illustrated in figure 9.16. While the latency of the Esper configuration almost remains constant at latency measurements below 1 millisecond, we can observe a performance decrease in the distributed configurations. For small pipelines consisting of a single processing element, the median latency remains almost constant, but a higher variance becomes visible. At pipeline size 2, both configuration have slightly increased latency measurements compared to the low-throughput configuration. At higher throughput and a pipeline size of 5, the latency for the semantic-epa-esper configuration increases to around 10 milliseconds, while the semantic-epa-flink configuration has a median latency of 16 milliseconds. We were not able to verify whether this difference originates from the system itself; another explanation might be found due to the circumstance that different broker implementations were used for both configurations.
In order to assess the performance of our solution, it is also necessary to show that the latency remains constant over time. Therefore, in figure 9.17, we show the latency on a per-event basis for all 6 configurations related to the experiment with 5000 events per second. It can be easily seen that the latency remains constant in all six configurations. Compared to the semantic-epa-esper configuration, the higher variance for the semantic-epa-flink configuration also becomes visible. The fluctuation

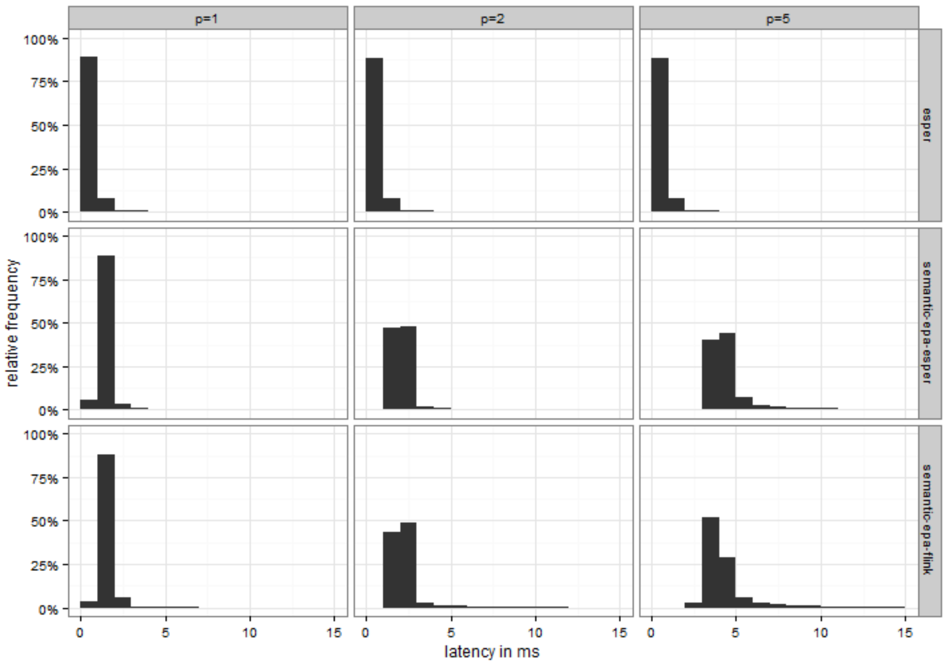


Figure 9.15 Performance Evaluation: 1000 events/second

at certain time intervals can be explained by the garbage collection performed by the Java Virtual Machine (JVM).
In general, the performance tests give good advice about application domains and limits of our system. Although it can be seen that our approach produces increased load on the message broker, which especially decreases latency in high-throughput scenarios in conjunction with larger pipeline sizes, the absolute latency achieved by our system is still competitive. Most scenarios do not require latencies below one second, and we argue that use cases which have stricter requirements in terms of latency usually do not require for quick adaptation of processing pipelines by business analysts. In such cases, it is therefore more feasible to hard-code event processing logic in single-host systems.
The performance tests also justify our argumentation to move semantic descriptions of events and their properties to design-time instead of using native RDF processing at run-time. This allows us to achieve much higher performance compared to existing RDF stream processing systems, which currently are able to process events with latency measurements in the area of hundreds of milliseconds [Stühmer 2015]. It

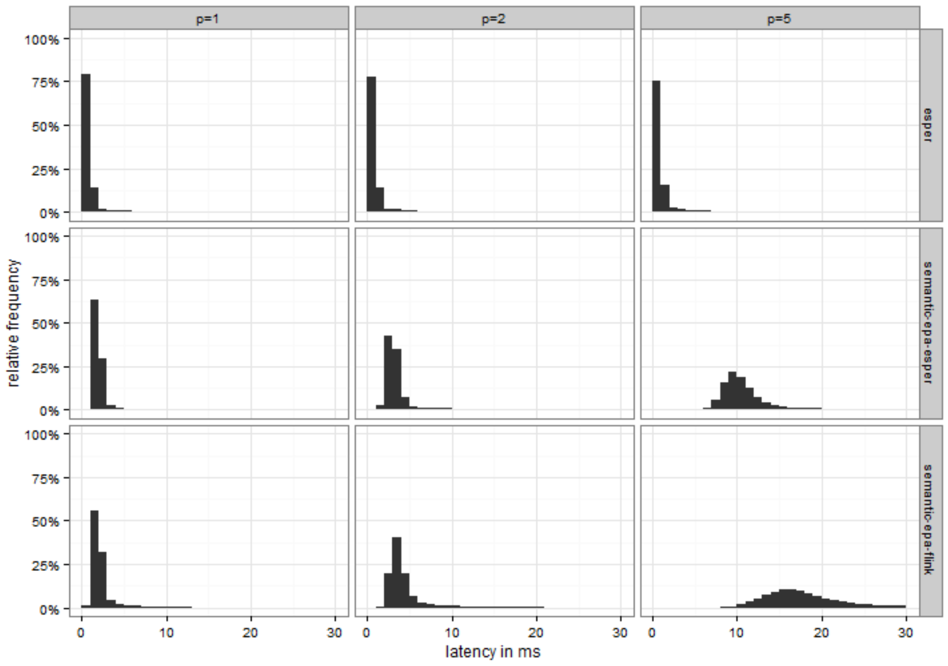


Figure 9.16 Performance Evaluation: 5000 events/second

is also worth to note that our tests were performed using a single broker, causing a potential bottleneck. Therefore, splitting the payload among multiple brokers, and also scaling the brokers itself, which is supported by systems such as Apache Kafka further reduces the overhead leading to potentially lower latency.

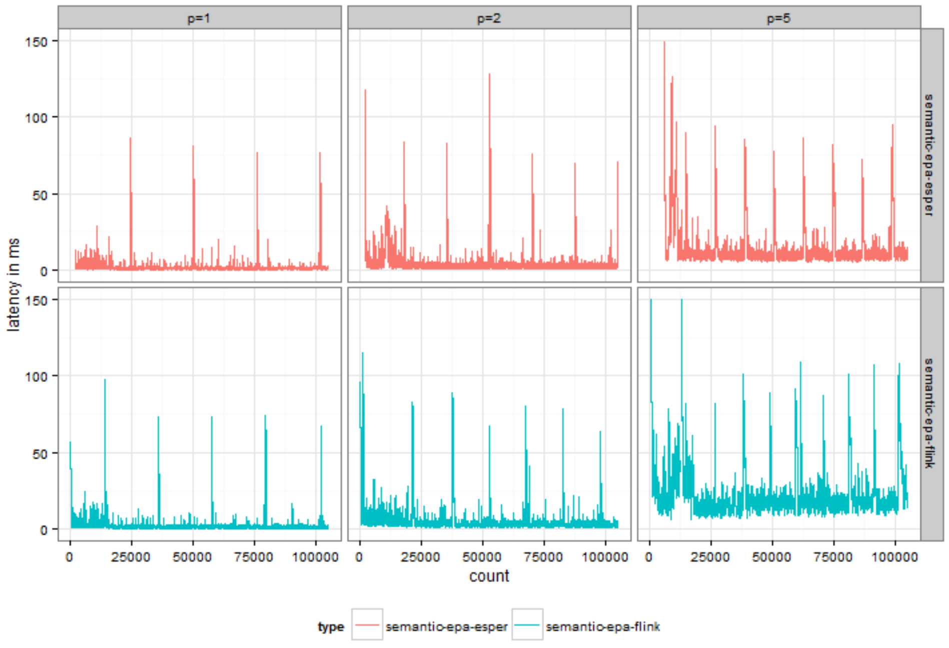


Figure 9.17 Performance Evaluation: Comparison

10 Conclusion

In this thesis, we introduced a methodology and a corresponding instantiation targeting the development process of event processing applications. The presented methodology aims at reducing the development effort in distributed settings by providing methods that allow non-programmers to create processing pipelines consisting of heterogeneous and geographically distributed pipeline elements. In addition, we have developed end-to-end tool support in form of StreamPipes, a reference implementation demonstrating the application of our methodology.
This thesis is concluded with summarizing the main contributions along the research questions identified in section 10.1. In section 10.2, the significance of our results is discussed by elaborating on the applicability of our approach in various emerging application areas. Finally, section 10.3 discusses our point of view on future work in this area to further improve the development process of event processing applications.

10.1 Summary

In summary, we recap the initially identified research questions and discuss our approach and main results that target these questions.

Research Question 1 (Development Process)**.** *How can we improve the development process of event processing applications?*

The first research question, concerned with the development process of event processing applications, was answered in chapter 6. Related to our main goal, enabling application specialists to define event processing applications in a self-service manner, we introduced a novel methodology which basically splits the development process into two parts, the *Setup Phase* and the *Execution Phase*. In general, the setup phase is designed as a preparation phase which includes two main processes. On the one hand, software developers are leveraged to define event producers and re-usable pipeline elements in form of event processing agents and event consumers. Re-usable pipeline elements are not bound to specific input event streams, but express stream requirements needed to parameterize the underlying event processing logic. On the

other hand, domain knowledge required as static data within an event processing pipeline is defined by business analysts. Re-usable pipeline elements and domain knowledge are made accessible in a central repository to the execution phase. Within the execution phase, processing pipelines can be defined without further development effort. Pattern engineers are responsible to author processing pipelines using graphical tool support based on previously registered pipeline elements in conjunction with knowledge gathered in the setup phase, and pipeline are automatically deployed by invoking the underlying run-time implementations.
The main advantage of this approach is, while application specialists are provided with the possibility to define event processing applications abstracted from technical details without developer consultancy, at the same time the event processing system does not rely on a predefined, fixed set of pipeline elements. The setup phase ensures extensibility of the system at any time new requirements arise. Thus, our methodology combines a highly flexible system with an intuitive way to create event processing applications.
Our reference implementation provides tool support covering all tasks of the proposed methodology. The setup phase is supported by providing a model editor and a software development toolkit which support the development of re-usable pipeline elements. In addition, a knowledge editor is proposed assisting business analysts to gather knowledge needed as static data within processing pipelines. The execution phase is supported by a graphical editor that allows to create processing pipelines, and a backend component ensuring proper execution of pipelines.
Our methodology was evaluated by presenting three case studies from various domains, resulting in the development of more than 100 pipeline elements. This allowed us to inspect the applicability of our approach in multiple application domains.

Research Question 2 (Model)**.** *How can we model event processing blocks independent from their specific run-time implementation?*

The second research question dealt with the modeling of event processing blocks independent from their run-time implementation. The main resulting artifact we developed in chapter 7 of this thesis is an ontology-based vocabulary that allows to define a semantics-based description layer which expresses requirements and capabilities of pipeline elements. We introduced a conceptual model of *Semantic Event Producers*, *Semantic Event Processing Agents* and *Semantic Event Consumers*, each of them consisting of a description layer and an implementation layer. In the description layer, concepts and properties have been proposed to define characteristics of event streams independent from a specific run-time event format or communication protocol. In contrast to other semantics-based approaches to event processing, where the semantics are directly included as part of the event payload, the main advantage of our approach is that lightweight and thus potentially faster run-time event formats can be used,

while at the same time a rich semantic model of an event exists that can be used at design-time for matching and integration purposes.
In chapter 9, we evaluated our model based on three case studies and a conceptual investigation of the expressivity of the developed model along the requirements identified in chapter 5.

Research Question 3 (Domain Knowledge)**.** *How can we separate domain knowledge from the technical specification of event processing languages?*

In section 3.3.2, the need was stressed to separate background knowledge from the technical specification of event patterns. Our approach ensures this feature in the proposed methodology by introducing a process that allows business analysts to define knowledge required as part of processing pipelines. This knowledge can be reused in the execution phase during the pipeline authoring process. More specifically, we have developed concepts that allow to define requirements for the integration of background knowledge as part of the description of pipeline elements.

Research Question 4 (Execution)**.** *How can we author and execute event processing pipelines consisting of heterogeneous processing blocks?*

Finally, research question 4, covering the authoring and execution of processing pipelines based on heterogeneous run-time implementations as discussed in chapter 8, was answered based on two defined tasks being part of the execution phase. First, pipeline authoring allows to define processing pipelines based on pipeline elements previously developed in the setup phase. The pipeline authoring task solely relies on the modeling of pipelines based on a pipeline element's description hiding implementation details from the user. In section 8.3, we introduced the process concerned with pipeline authoring and showed how heterogeneous pipeline elements can be integrated in a single pipeline. Finally, we tackled geographically distributed execution of pipelines, which is based on the exchange of invocation graphs between the pipeline authoring environment and individual pipeline elements deployed at geographically distributed locations.
Provided tool support demonstrates the pipeline authoring and execution process. We developed a graphical editor that allows to import pipeline elements at run-time and provides capabilities to author pipelines based on these elements. In addition, we created run-time wrappers for multiple event processing systems showing the ability of our system to integrate heterogeneous event processing logic.
We conducted a user study involving a number of business experts from the manufacturing domain in order to evaluate the ability of the pipeline editor to allow non-programmers to build processing pipelines. In addition, we conducted performance tests in order to show that our approach is able to process events with high throughput and low latency, although it generates workload due to increased message broker communication due to its distributed compared to single-host system.

10.2 Significance of the results

Our approach improves and simplifies the development of event processing applications. The proposed methodology ensures flexibility in the sense that new requirements for integrating event producers in form of new physical devices or software, but also requirements demanding for new event processing logic and consumers can be directly implemented. At the same time, our methodology pushes the development of event processing applications to a higher level making the previously technical-oriented development of processing pipelines better accessible to application specialists.
Many emerging application domains potentially benefit from event processing application development in a self-service manner. For instance, from a technical perspective, the initially discussed concept of marketplaces in the Internet of Things (IoT) domain is fully supported. On the one hand, our approach allows to define processing pipelines spanning technology-independent and geographically distributed event processing logic. In addition, as pipeline elements make their requirements and capabilities available using RDF-based description graphs, interoperability is facilitated.
From a use case perspective, our approach contributes to needs where fast definition and adaptation of real-time processing logic is required. It was already shown in section 3.2 that mainly three different purposes, including integrated monitoring of a wide variety of data sources, but also situation detection (e.g., anomaly detection) and continuous data ingestion (e.g., data harmonization) require for fast development of event processing applications.

10.3 Outlook

This thesis provides methods to improve the development of event processing applications for application specialists. Our methodology improves the development process while it still maintains flexibility in form of an extensible approach. In addition, the pipeline editor we provide as tool support ensures fast definition of processing pipelines by abstracting from technical details specific to event processing. Future work can further simplify the development of event processing applications:

Pipeline Expansion Our matching process, described in section 8.3, considers multiple criteria in order to detect whether two pipeline elements are compatible to each other. Although tool support is provided to assist users during the pipeline authoring process (e.g., in form of recommendations), more sophisticated methods might be able to go beyond direct matching by finding indirect matches, i.e., by automatically including pipeline elements that are able to mediate between two potentially non-compatible elements. Furthermore, it might be worth to be investigated whether a kind of reverse pipeline authoring process might be feasible, for instance, by letting users express their intended analysis purpose first, while the system assists in finding sensors and event processing agents supporting this purpose. Such pipeline expansion methods might require further extensions to the model that need to be researched.

Context-aware Pipeline Management Currently, pipelines defined by users are static, e.g., they do not change over time and are executed until they are manually stopped. In addition, users select pipelines that are started at some point in time. However, it might be useful to investigate (semi-) automatic adaptations of pipelines in the pipeline evolution task of the execution phase by also taking into account contextual information. For instance, a system that observes the current scrap rate in a production facility could automatically adapt thresholds based on the currently produced products. One approach in this area would be to support context-defining pipelines, which detect a specific context in terms of time, location or other parameters. Depending on the currently active context, pipelines could be automatically deployed, removed and modified using different parameters specific to the context.

Appendix

A Runtime-Wrapper: Description

Listing A.1 Description: Geofencing S-EPA

```java
package de.fzi.cep.sepa.esper.geo.geofencing;

import java.net.URI;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import de.fzi.cep.sepa.commons.Utils;
import de.fzi.cep.sepa.esper.config.EsperConfig;
import de.fzi.cep.sepa.model.builder.EpProperties;
import de.fzi.cep.sepa.model.builder.EpRequirements;
import de.fzi.cep.sepa.model.builder.StaticProperties;
import de.fzi.cep.sepa.model.impl.EpaType;
import de.fzi.cep.sepa.model.impl.EventSchema;
import de.fzi.cep.sepa.model.impl.EventStream;
import de.fzi.cep.sepa.model.impl.Response;
import de.fzi.cep.sepa.model.impl.eventproperty.EventProperty;
import de.fzi.cep.sepa.model.impl.graph.SepaDescription;
import de.fzi.cep.sepa.model.impl.graph.SepaInvocation;
import de.fzi.cep.sepa.model.impl.output.AppendOutputStrategy;
import de.fzi.cep.sepa.model.impl.output.OutputStrategy;
import de.fzi.cep.sepa.model.impl.staticproperty.DomainStaticProperty;
import de.fzi.cep.sepa.model.impl.staticproperty.MappingPropertyUnary;
import de.fzi.cep.sepa.model.impl.staticproperty.OneOfStaticProperty;
import de.fzi.cep.sepa.model.impl.staticproperty.Option;
import de.fzi.cep.sepa.model.impl.staticproperty.PropertyValueSpecification;
import de.fzi.cep.sepa.model.impl.staticproperty.StaticProperty;
import de.fzi.cep.sepa.model.impl.staticproperty.SupportedProperty;
import de.fzi.cep.sepa.model.util.SepaUtils;
import de.fzi.cep.sepa.model.vocabulary.Geo;
import de.fzi.cep.sepa.runtime.flat.declarer.FlatEpDeclarer;
import de.fzi.cep.sepa.client.util.StandardTransportFormat;

public class GeofencingController extends FlatEpDeclarer<GeofencingParameters>
    {

  @Override
  public SepaDescription declareModel() {
    EventStream stream1 = new EventStream();
    EventSchema schema = new EventSchema();
    EventProperty e1 = EpRequirements.domainPropertyReq(Geo.lat);
    EventProperty e2 = EpRequirements.domainPropertyReq(Geo.lng);
    EventProperty e3 = EpRequirements.stringReq();
    schema.setEventProperties(Arrays.asList(e1, e2, e3));

    SepaDescription desc = new SepaDescription("geofencing", "Geofencing", "
        Detects whether a location-based stream moves inside a (circular) area
         around a given point described as latitude-longitude pair.");

    stream1.setEventSchema(schema);
    desc.addEventStream(stream1);

    List<OutputStrategy> strategies = new ArrayList<OutputStrategy>();
    List<EventProperty> additionalProperties = new ArrayList<>();
    additionalProperties.add(EpProperties.longEp("geofencingTime", "http://
        schema.org/DateTime"));
    additionalProperties.add(EpProperties.booleanEp("insideGeofence", "http://
        schema.org/Text"));
    AppendOutputStrategy appendOutput = new AppendOutputStrategy();
    appendOutput.setEventProperties(additionalProperties);
    strategies.add(appendOutput);
    desc.setOutputStrategies(strategies);

    List<StaticProperty> staticProperties = new ArrayList<StaticProperty>();

    OneOfStaticProperty operation = new OneOfStaticProperty("operation", "Enter
        /Leave Area", "Specifies the operation that should be detected: A
        location-based stream can enter or leave the selected area.");
    operation.addOption(new Option("Enter"));
    operation.addOption(new Option("Leave"));

    staticProperties.add(operation);
    staticProperties.add(StaticProperties.integerFreeTextProperty("radius", "
        Radius (m)", "Specifies the geofence size (the radius around the
        provided location) in meters.", new PropertyValueSpecification(0,
        10000, 10)));

    SupportedProperty latSp = new SupportedProperty(Geo.lat, true);
    SupportedProperty lngSp = new SupportedProperty(Geo.lng, true);

    List<SupportedProperty> supportedProperties = Arrays.asList(latSp, lngSp);
    DomainStaticProperty dsp = new DomainStaticProperty("location", "Location",
         "Specifies the center of the geofence", supportedProperties);

    staticProperties.add(dsp);

    MappingPropertyUnary latMapping = new MappingPropertyUnary(URI.create(e1.
        getElementId()), "mapping-latitude", "Latitude Coordinate", "Specifies
         the latitude field of the stream.");
    staticProperties.add(latMapping);

    MappingPropertyUnary lngMapping = new MappingPropertyUnary(URI.create(e1.
        getElementId()), "mapping-longitude", "Longitude Coordinate", "
        Specifies the longitude field of the stream.");
    staticProperties.add(lngMapping);

    MappingPropertyUnary partitionMapping = new MappingPropertyUnary(URI.create
        (e3.getElementId()), "mapping-partition", "Partition Property", "
        Specifies a field that should be used to partition the stream (e.g., a
         vehicle plate number)");
    partitionMapping.setValueRequired(false);
    staticProperties.add(partitionMapping);

    desc.setStaticProperties(staticProperties);
    desc.setSupportedGrounding(StandardTransportFormat.getSupportedGrounding())
        ;
    return desc;
  }

  @Override
  public Response invokeRuntime(SepaInvocation invocationGraph) {
    String operation = SepaUtils.getOneOfProperty(invocationGraph, "operation")
        ;

    int radius = (int) Double.parseDouble(SepaUtils.
        getFreeTextStaticPropertyValue(invocationGraph, "radius"));

    DomainStaticProperty dsp = SepaUtils.getDomainStaticPropertyBy(
        invocationGraph, "location");
    double latitude = Double.parseDouble(SepaUtils.getSupportedPropertyValue(
        dsp, Geo.lat));
    double longitude = Double.parseDouble(SepaUtils.getSupportedPropertyValue(
        dsp, Geo.lng));

    GeofencingData geofencingData = new GeofencingData(latitude, longitude,
        radius);

    String latitudeMapping = SepaUtils.getMappingPropertyName(invocationGraph,
        "mapping-latitude");
    String longitudeMapping = SepaUtils.getMappingPropertyName(invocationGraph,
         "mapping-longitude");
    String partitionMapping = SepaUtils.getMappingPropertyName(invocationGraph,
         "mapping-partition");
    GeofencingParameters params = new GeofencingParameters(invocationGraph,
        getOperation(operation), geofencingData, latitudeMapping,
        longitudeMapping, partitionMapping);

    try {
    invokeEPRuntime(params, Geofencing::new, invocationGraph);
    return new Response(invocationGraph.getElementId(), true);
    } catch (Exception e) {
    e.printStackTrace();
    return new Response(invocationGraph.getElementId(), false, e.getMessage());
    }
  }

  private GeofencingOperation getOperation(String operation) {
    if (operation.equals("Enter")) return GeofencingOperation.ENTER;
    else return GeofencingOperation.LEAVE;
  }

}
```

B User Survey

Table B.1 User Survey: Answers

Id	Question	#1	#2	#3	#4	#5	#6
1	My expertise level related to stream processing/complex event processing applications is high.	3	2	2	2	4	1
2	I like the look and feel of the StreamPipes GUI.	5	4	4	4	5	4
3	The StreamPipes GUI is easy and intuitive to use.	4	5	4	5	5	4
4	I could easily understand the messages displayed by the system	5	1	4	4	5	4
5	Navigating through the system is simple and clear.	5	4	3	4	5	3
6	StreamPipes is capable of improving my overall working exprience.	4	4	4	5	5	4
7	The functionalities of StreamPipes are sufficient for the needs of my job.	3	3	4	5	3	4
8	Using graphical notations to model stream processing pipelines is useful.	4	4	4	5	5	4
9	I understood the meaning of pipelines in the StreamPipes component.	4	4	2	4	5	4
10	I could easily understand how to start and stop a pipeline.	5	5	5	5	5	4
11	The average time to create a pipeline is satisfying.	4	5	5	5	5	4
12	I didn't need any guidance in creating pipelines using the graphical modeling notation.	5	4	3	4	5	2
13	The element recommendation feature is useful.	4	3	4	4	5	4
14	I am satisfied with the support for consistency checking of pipelines.	4	3	2	4	5	4

Bibliography

Alonso, Gustavo; Casati, Fabio; Kuno, Harumi; Machiraju, Vijay (2004). *Web services*. Springer.

Andrade, Henrique CM; Gedik, Buğra; Turaga, Deepak S (2014). *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press.

Anicic, Darko; Fodor, Paul; Rudolph, Sebastian; Stojanovic, Nenad (2011). 'EP-SPARQL: a unified language for event processing and stream reasoning'. In: *Proceedings of the 20th international conference on World wide web*. ACM, pp. 635–644.

Anicic, Darko; Rudolph, Sebastian; Fodor, Paul; Stojanovic, Nenad (2012). 'Stream reasoning and complex event processing in ETALIS'. In: *Semantic Web* 3.4, pp. 397–407.

Appel, Stefan; Frischbier, Sebastian; Freudenreich, Tobias; Buchmann, Alejandro (2013). 'Event stream processing units in business processes'. In: *Business Process Management*. Springer, pp. 187–202.

Atzori, Luigi; Iera, Antonio; Morabito, Giacomo (2010). 'The internet of things: A survey'. In: *Computer networks* 54.15, pp. 2787–2805.

Babcock, Brian; Babu, Shivnath; Datar, Mayur; Motwani, Rajeev; Widom, Jennifer (2002). 'Models and issues in data stream systems'. In: *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, pp. 1–16.

Barbieri, Davide Francesco; Braga, Daniele; Ceri, Stefano; Della Valle, Emanuele; Grossniklaus, Michael (2009). 'C-SPARQL: SPARQL for continuous querying'. In: *Proceedings of the 18th international conference on World wide web*. ACM, pp. 1061–1062.

Barone, Daniele; Mylopoulos, John; Jiang, Lei; Amyot, Daniel (2010). 'The business intelligence model: Strategic modelling'. In: *University of Toronto, Canada*.

Binnewies, Sebastian; Stantic, Bela (2011). 'Introducing knowledge-enrichment techniques for complex event processing'. In: *Informatics Engineering and Information Science*. Springer, pp. 228–242.

Binnewies, Sebastian; Stantic, Bela (2012). 'OECEP: enriching complex event processing with domain knowledge from ontologies'. In: *Proceedings of the Fifth Balkan Conference in Informatics*. ACM, pp. 20–25.

Bonino, Dario; Corno, Fulvio (2012). 'SpChains: A declarative framework for data stream processing in pervasive applications'. In: *Procedia Computer Science* 10, pp. 316–323.

Bonino, Dario; Corno, Fulvio; De Russis, Luigi (2013). 'Real-time big data processing for domain experts, an application to smart buildings'. In: *Big Data Computing/Rajendra Akerkar; Taylor & Francis Press: London, UK* 1, pp. 415–447.

Bonino, Dario; De Russis, Luigi (2012). ‘Mastering real-time big data with stream processing chains’. In: *XRDS: Crossroads, The ACM Magazine for Students* 19.1, pp. 83–86.

Boubeta-Puig, Juan; Ortiz, Guadalupe; Medina-Bulo, Inmaculada (2014). ‘A model-driven approach for facilitating user-friendly design of complex event patterns’. In: *Expert Systems with Applications* 41.2, pp. 445–456.

Boubeta-Puig, Juan; Ortiz, Guadalupe; Medina-Bulo, Inmaculada (2015). ‘ModeL4CEP: Graphical domain-specific modeling languages for CEP domains and event patterns’. In: *Expert Systems with Applications* 42.21, pp. 8095–8110.

Bousdekis, Alexandros; Papageorgiou, Nikos; Magoutas, Babis; Apostolou, Dimitris; Mentzas, Gregoris (2015). ‘A Real-Time Architecture for Proactive Decision Making in Manufacturing Enterprises’. In: *On the Move to Meaningful Internet Systems: OTM 2015 Workshops*. Springer, pp. 137–146.

Brennan, K. (2009). *A Guide to the Business Analysis Body of Knowledge (BABOK Guide)*. IT Pro. International Institute of Business Analysis. ISBN: 9780981129228.

Brickley, Dan (2011). ‘Using RDFa 1.1 Lite with Schema. org’. In: *schema. org* 11.

Bruns, Ralf; Dunkel, Jürgen (2010). *Event-driven architecture: Softwarearchitektur für ereignisgesteuerte Geschäftsprozesse*. Springer-Verlag.

Business Motivation Model - Version 1.1 (2010).

Calbimonte, Jean-Paul; Corcho, Oscar; Gray, Alasdair JG (2010). ‘Enabling ontology-based access to streaming data sources’. In: *The Semantic Web–ISWC 2010*. Springer, pp. 96–111.

Chandy, K; Schulte, W (2009). *Event processing: designing IT systems for agile companies*. McGraw-Hill, Inc.

Cleven, Anne; Gubler, Philipp; Hüner, Kai M (2009). ‘Design alternatives for the evaluation of design science research artifacts’. In: *Proceedings of the 4th International Conference on Design Science Research in Information Systems and Technology*. ACM, p. 19.

Compton, Michael; Barnaghi, Payam; Bermudez, Luis; GarcıA-Castro, RaúL; Corcho, Oscar; Cox, Simon; Graybeal, John; Hauswirth, Manfred; Henson, Cory; Herzog, Arthur, et al. (2012). ‘The SSN ontology of the W3C semantic sensor network incubator group’. In: *Web Semantics: Science, Services and Agents on the World Wide Web* 17, pp. 25–32.

Cugola, Gianpaolo; Margara, Alessandro (2012). ‘Processing flows of information: From data stream to complex event processing’. In: *ACM Computing Surveys (CSUR)* 44.3, p. 15.

Erb, Benjamin; Kargl, Frank (2015). ‘A conceptual model for event-sourced graph computing’. In: *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. ACM, pp. 352–355.

Etzion, Opher; Halle, Barbara von (2013). *Modeling the Event-Driven World*. http://de.slideshare.net/opher.etzion/er-2013-tutorial-modeling-the-event-driven-world. Accessed: 2016-03-30.

Etzion, Opher; Niblett, Peter (2010). *Event processing in action*. Manning Publications Co.

Eugster, Patrick Th; Felber, Pascal A; Guerraoui, Rachid; Kermarrec, Anne-Marie (2003). 'The many faces of publish/subscribe'. In: *ACM Computing Surveys (CSUR)* 35.2, pp. 114–131.

Feier, Christina; Polleres, Axel; Dumitru, Roman; Domingue, John; Stollberg, Michael; Fensel, Dieter (2005). 'Towards intelligent web services: The web service modeling ontology (WSMO)'. In:

Fernández-López, Mariano; Gómez-Pérez, Asunción; Juristo, Natalia (1997). 'Methontology: from ontological art towards ontological engineering'. In:

Gessler, Damian DG; Schiltz, Gary S; May, Greg D; Avraham, Shulamit; Town, Christopher D; Grant, David; Nelson, Rex T (2009). 'SSWAP: A Simple Semantic Web Architecture and Protocol for semantic web services'. In: *BMC bioinformatics* 10.1, p. 309.

Guha, Ramanathan (2011). 'Introducing schema. org: Search engines come together for a richer web'. In: *Google Official Blog*.

Hepp, Martin (2008). 'Goodrelations: An ontology for describing products and services offers on the web'. In: *Knowledge Engineering: Practice and Patterns*. Springer, pp. 329–346.

Hevner, Alan R; March, Salvatore T; Park, Jinsoo; Ram, Sudha (2004). 'Design science in information systems research'. In: *MIS quarterly* 28.1, pp. 75–105.

Hoßbach, Bastian (2015). 'Design and implementation of a middleware for uniform, federated and dynamic event processing'. In:

Hoßbach, Bastian; Glombiewski, Nikolaus; Morgen, Andreas; Ritter, Franz; Seeger, Bernhard (2013). 'JEPC: The Java Event Processing Connectivity'. In: *Datenbank-Spektrum* 13.3, pp. 167–178. ISSN: 1610-1995. DOI: 10.1007/s13222-013-0133-y.

Infso (2008). *Internet of Things in 2020*. http://www.smart-systems-integration.org/public/documents/publications/Internet-of-Things_in_2020_EC-EPoSS_Workshop_Report_2008_v3.pdf. Accessed: 2016-03-30.

Jardine, Andrew KS; Lin, Daming; Banjevic, Dragan (2006). 'A review on machinery diagnostics and prognostics implementing condition-based maintenance'. In: *Mechanical systems and signal processing* 20.7, pp. 1483–1510.

Jiang, Lei; Barone, Daniele; Amyot, Daniel; Mylopoulos, John (2011). 'Strategic models for business intelligence'. In: *Conceptual Modeling–ER 2011*. Springer, pp. 429–439.

Kagermann, Henning; Wahlster, Wolfgang; Helbig, Johannes (2008). *Recommendations for implementing the strategic initiative INDUSTRIE 4.0*. http://www.acatech.de/fileadmin/user_upload/Baumstruktur_nach_Website/Acatech/root/de/

Material_fuer_Sonderseiten/Industrie_4.0/Final_report__Industrie_4.0_accessible.pdf. Accessed: 2016-03-30.

Karampiperis, Pythagoras; Mouchakis, Giannis; Paliouras, Georgios; Karkaletsis, Vangelis (2014). 'ER designer toolkit: a graphical event definition authoring tool'. In: *Universal Access in the Information Society* 13.1, pp. 115–123.

Kavakli, Evangelia (2004). 'Modeling organizational goals: Analysis of current methods'. In: *Proceedings of the 2004 ACM symposium on Applied computing*. ACM, pp. 1339–1343.

Kona, Srividya; Bansal, Ajay; Simon, Luke; Mallya, Ajay; Gupta, Gopal; Hite, Thomas D (2009). 'USDL: A Service-Semantics Description Language for Automatic Service Discovery and Composition1'. In: *International Journal of Web Services Research* 6.1, p. 20.

Kreps, Jay; Narkhede, Neha; Rao, Jun, et al. (2011). 'Kafka: A distributed messaging system for log processing'. In: NetDB.

Lanthaler, Markus; Gütl, Christian (2013). 'Hydra: A Vocabulary for Hypermedia-Driven Web APIs.' In: *LDOW* 996.

Lee, O-Joun; You, Eunsoon; Hong, Min-Sung; Jung, Jason J (2015). 'Adaptive Complex Event Processing Based on Collaborative Rule Mining Engine'. In: *Intelligent Information and Database Systems*. Springer, pp. 430–439.

Loucopoulos, Pericles; Karakostas, Vassilios (1995). *System requirements engineering*. McGraw-Hill, Inc.

Luckham, David (2002). *The power of events*. Vol. 204. Addison-Wesley Reading.

Luckham, David C (2011). *Event processing for business: organizing the real-time enterprise*. John Wiley & Sons.

Luckham, David C.; Schulte, Roy (2011). *Event Processing Glossary - Version 2.0*. Online Resource. http://www.complexevents.com/2011/08/23/event-processing-glossary-version-2-0/ Last visited 2014-05-02.

Magoutas, Babis; Riemer, Dominik; Apostolou, Dimitris; Ma, Jun; Mentzas, Gregoris; Stojanovic, Nenad (2013). 'An event-driven system for business awareness management in the logistics domain'. In: *Business Process Management Workshops*. Springer Berlin Heidelberg, pp. 402–413.

Margara, Alessandro; Cugola, Gianpaolo; Tamburrelli, Giordano (2014). 'Learning from the past: automated rule generation for complex event processing'. In: *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. ACM, pp. 47–58.

Martin, David; Burstein, Mark; Hobbs, Jerry; Lassila, Ora; McDermott, Drew; McIlraith, Sheila; Narayanan, Srini; Paolucci, Massimo; Parsia, Bijan; Payne, Terry, et al. (2004). 'OWL-S: Semantic markup for web services'. In: *W3C member submission* 22, pp. 2007–04.

McCarthy, Dennis; Dayal, Umeshwar (1989). 'The architecture of an active database management system'. In: *ACM Sigmod Record*. Vol. 18. 2. ACM, pp. 215–224.

Merriam-Webster (2009). *Merriam-Webster Online Dictionary*.

Metzke, Tobias; Rogge-Solti, Andreas; Baumgrass, Anne; Mendling, Jan; Weske, Mathias (2013). 'Enabling Semantic Complex Event Processing in the Domain of Logistics'. In: *Service-Oriented Computing–ICSOC 2013 Workshops*. Springer, pp. 419–431.

Neely, Andy (2002). *Business performance measurement: theory and practice*. Cambridge University Press.

Obweger, Hannes; Schiefer, Josef; Suntinger, Martin; Breier, Florian; Thullner, Robert (2011a). 'Complex event processing off the shelf-rapid development of event-driven applications with solution templates'. In: *Control & Automation (MED), 2011 19th Mediterranean Conference on*. IEEE, pp. 631–638.

Obweger, Hannes; Schiefer, Josef; Suntinger, Martin; Kepplinger, Peter (2011b). 'Model-driven rule composition for event-based systems'. In: *International Journal of Business Process Integration and Management* 5.4, pp. 344–357.

Olson, Michael; Liu, Annie; Faulkner, Matthew; Chandy, K Mani (2011). 'Rapid detection of rare geospatial events: earthquake warning applications'. In: *Proceedings of the 5th ACM international conference on Distributed event-based system*. ACM, pp. 89–100.

Osborne, Miles; Moran, Sean; McCreadie, Richard; Von Lunen, Alexander; Sykora, Martin D; Cano, Elizabeth; Ireson, Neil; Macdonald, Craig; Ounis, Iadh; He, Yulan, et al. (2014). 'Real-time detection, tracking, and monitoring of automatically discovered events in social media'. In: *52nd Annual Meeting of the Association for Computational Linguistics*.

Paolucci, Massimo; Kawamura, Takahiro; Payne, Terry R; Sycara, Katia (2002). 'Semantic matching of web services capabilities'. In: *International Semantic Web Conference*. Springer, pp. 333–347.

Paton, Norman W; Diaz, Oscar (1999). 'Active database systems'. In: *ACM Computing Surveys (CSUR)* 31.1, pp. 63–103.

Pedrinaci, Carlos; Cardoso, Jorge; Leidig, Torsten (2014). 'Linked USDL: a vocabulary for web-scale service trading'. In: *The Semantic Web: Trends and Challenges*. Springer, pp. 68–82.

Pedrinaci, Carlos; Domingue, John (2010). 'Toward the Next Wave of Services: Linked Services for the Web of Data.' In: *J. ucs* 16.13, pp. 1694–1719.

Pinto, H Sofia; Martins, JP (2000). 'Reusing ontologies'. In: *AAAI 2000 Spring Symposium on Bringing Knowledge to Business Processes*. Vol. 2. 000. Karlsruhe, Germany: AAAI, p. 7.

Potocnik, Martin; Juric, Matjaz B (2014). 'Towards complex event aware services as part of soa'. In: *Services Computing, IEEE Transactions on* 7.3, pp. 486–500.

Pries-Heje, Jan; Baskerville, Richard; Venable, J (2008). 'Strategies for design science research evaluation'. In: *ECIS 2008 proceedings*, pp. 1–12.

Riemer, Dominik; Kaulfersch, Florian; Hutmacher, Robin; Stojanovic, Ljiljana (2015). 'StreamPipes: Solving the DEBS Challenge with Semantic Stream Processing Pipelines'.

In: *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. ACM, pp. 330–331.

Riemer, Dominik; Stojanovic, Ljiljana; Stojanovic, Nenad (2012). 'Using complex event processing for modeling semantic requests in real-time social media monitoring'. In: *Sixth International AAAI Conference on Weblogs and Social Media*.

Riemer, Dominik; Stojanovic, Ljiljana; Stojanovic, Nenad (2013a). 'Demo: ALERT-real-time coordination in open source software development'. In: *Proceedings of the 7th ACM international conference on Distributed event-based systems*. ACM, pp. 339–340.

Riemer, Dominik; Stojanovic, Ljiljana; Stojanovic, Nenad (2014). 'SEPP: Semantics-Based Management of Fast Data Streams'. In: *Service-Oriented Computing and Applications (SOCA), 2014 IEEE 7th International Conference on*. IEEE, pp. 113–118.

Riemer, Dominik; Stojanovic, Nenad; Stojanovic, Ljiljana (2013b). 'A methodology for designing events and patterns in fast data processing'. In: *Advanced Information Systems Engineering*. Springer Berlin Heidelberg, pp. 133–148.

Roman, Dumitru; Keller, Uwe; Lausen, Holger; Bruijn, Jos de; Lara, Rubén; Stollberg, Michael; Polleres, Axel; Feier, Cristina; Bussler, Cristoph; Fensel, Dieter (2005). 'Web service modeling ontology'. In: *Applied ontology* 1.1, pp. 77–106.

Sen, Sinan (2013). 'Efficient and Effective Event Pattern Management'. PhD thesis. TU Karlsruhe.

Sen, Sinan; Stojanovic, Nenad (2010). 'GRUVe: a methodology for complex event pattern life cycle management'. In: *Advanced information systems engineering*. Springer, pp. 209–223.

Sen, Sinan; Stojanovic, Nenad; Shemrani, Bijan Fahimi (2010a). 'EchoPAT: a system for real-time complex event pattern monitoring'. In: *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*. ACM, pp. 107–108.

Sen, Sinan; Stojanovic, Nenad; Stojanovic, Ljiljana (2010b). 'An approach for iterative event pattern recommendation'. In: *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*. ACM, pp. 196–205.

Sharon, Guy; Etzion, Opher (2007). *Event processing network-a conceptual model*. Technion-Israel Institute of Technology, Faculty of Industrial and Management Engineering.

Stojanovic, Ljiljana; Sen, Sinan; Ma, Jun; Riemer, Dominik (2012). 'ALERT: semantic event-driven collaborative platform for software development'. In: *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*. ACM, pp. 385–386.

StreamBase Studio (2016). http://streambase.typepad.com/.a/0d-pi. Accessed: 2016-07-30.

StreamBase Visual Editor (2016). http://docs.streambase.com/sb76/index.jsp?topic=/com.streambase.sb.ide.help/data/html/gettingstarted/termscomponents.html. Accessed: 2016-07-30.

Studer, Rudi; Benjamins, V Richard; Fensel, Dieter (1998). 'Knowledge engineering: principles and methods'. In: *Data & knowledge engineering* 25.1, pp. 161–197.

Stühmer, Roland (2015). *Web-oriented Event Processing*. KIT Scientific Publishing.

Stühmer, Roland; Stojanovic, Nenad; Obermeier, Stefan; Gibert, Philippe (2012a). 'Demo: Where Events Meet Events: PLAY Event Marketplace'. In: *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*.

Stühmer, Roland; Stojanovic, Nenad; Obermeier, Stefan; Gibert, Philippe (2012b). 'Demo: Where Events Meet Events: PLAY Event Marketplace'. In: *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*.

Sure, York; Angele, Juergen; Staab, Steffen (2003). 'OntoEdit: Multifaceted inferencing for ontology engineering'. In: *Journal on Data Semantics I*. Springer, pp. 128–152.

Sure, York; Staab, Steffen; Studer, Rudi (2004). 'On-to-knowledge methodology (OTKM)'. In: *Handbook on ontologies*. Springer, pp. 117–132.

Teymourian, Kia; Paschke, Adrian (2010). 'Enabling knowledge-based complex event processing'. In: *Proceedings of the 2010 EDBT/ICDT Workshops*. ACM, p. 37.

Teymourian, Kia; Rohde, Malte; Paschke, Adrian (2012). 'Fusion of background knowledge and streams of events'. In: *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*. ACM, pp. 302–313.

Vassiliadis, Panos (2009). 'A survey of Extract–transform–Load technology'. In: *International Journal of Data Warehousing and Mining (IJDWM)* 5.3, pp. 1–27.

Venable, John; Pries-Heje, Jan; Baskerville, Richard (2012). 'A comprehensive framework for evaluation in design science research'. In: *Design Science Research in Information Systems. Advances in Theory and Practice*. Springer, pp. 423–438.

Verborgh, Ruben; Harth, Andreas; Maleshkova, Maria; Stadtmüller, Steffen; Steiner, Thomas; Taheriyan, Mohsen; Van de Walle, Rik (2014). 'Survey of semantic description of rest apis'. In: *rest: Advanced Research Topics and Practical Applications*. Springer, pp. 69–89.

Vermesan, Ovidiu; Friess, Peter (2014). *Internet of Things-From research and innovation to Market Deployment*. River Publishers.

Vidačković, Krešimir; Kellner, Ingmar; Donald, John (2010). 'Business-oriented development methodology for complex event processing: demonstration of an integrated approach for process monitoring'. In: *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*. ACM, pp. 111–112.

Vidačkovič, Krešimir; Weisbecker, Anette (2011). 'A methodology for dynamic service compositions based on an event-driven approach'. In: *SRII Global Conference (SRII), 2011 Annual*. IEEE, pp. 484–494.

Walford, W. (2014). *What is Information System Analysis and Design*. Information Technology.

Walsh, Aaron E., ed. (2002). *Uddi, Soap, and Wsdl: The Web Services Specification Reference Book*. Prentice Hall Professional Technical Reference. ISBN: 0130857262.

Weibel, Stuart; Kunze, John; Lagoze, Carl; Wolf, Misha (1998). *Dublin core metadata for resource discovery*. Tech. rep.

Witt, Graham (2012). *Writing Effective Business Rules*. Amsterdam: Elsevier (Morgan Kaufmann). ISBN: 978-0-12-385051-5.

Zowghi, Didar; Coulin, Chad (2005). 'Requirements elicitation: A survey of techniques, approaches, and tools'. In: *Engineering and managing software requirements*. Springer, pp. 19–46.