Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis Nr. 96

Automatic Splitting in Data-Parallel Complex Event Processing Systems

Tim Sanwald

Course of Study:	Software Engineering
Examiner:	Prof. Dr. rer. nat. Kurt Rothermel
Supervisor:	Dipl.-Inf. Ruben Mayer
Commenced:	March 29, 2016
Completed:	September 28, 2016
CR-Classification:	C.2.4

Abstract

Parallel Complex Event Processing (CEP) systems handle today's heavy loaded event streams from smart homes, network traffic systems or stock trading systems by distributing the incoming event stream to several pattern detection systems. The correct splitting is currently done by CEP experts which ensure the consistent splitting without generating false-positive or false-negative complex events in comparison with centralized CEP systems. In this work an approach is developed which automatically generates a splitting model from the pattern definition which ensures the consistent distribution without generating false positives or false negatives. This approach enables a parallel CEP system to be configured and used the same way as a centralized CEP system. Further, a method which combines window based splitting and key based splitting is presented to reduce the network load and the CPU load on pattern detection operators. The functionality of the automatic splitting and the optimization is validated with common CEP scenarios based on generated and real world data to ensure a wide applicability of the approach.

Contents

List of Figures

List of Tables

Listings

1 Introduction

In today's data centres more and more data needs to be processed. Some needs to be analysed in a very short time like market values in stock data, traffic in network data or data in sensor networks. In stock data exchange systems fractions of seconds can make the difference between millions of dollars, depending on the change of the stock value. But not only in data centres more data is arriving, smart home systems gain more and more attention. Sensors are integrated into houses to observe the living conditions and improving daily routines, e.g. running the washer at night to reduce the cost of electricity. Therefore, a bunch of systems have been developed, e.g. Complex Event Processing (CEP) and Publish Subscribe systems (Pub-Sub). Pub-Sub systems handle messages from so called publishers and transmits them based on flags in a message to subscribers interested in this flag. Pub-Sub systems are fast in processing and delivering but work on a low level or are limited in their functionality. CEP systems handle combinations of several events in the stream, whereas public subscribe systems behave on one message at a time without knowledge of previous messages. CEP systems are configured from the user by specifying a query which defines the pattern detected in a stream of events. Queries are used to derive events with extended complexity from low level events, e.g. recognize a fire from smoke detection sensors and temperature sensors. The process from gathering information into events and deriving complex events up to the informing of the information requester are combined in centralized CEP systems. Due to the increased data rate nowadays, centralized CEP systems are at their limit for most applications.

Centralized CEP systems lack the possibility to distribute the pattern detection. Hinder the processing of heavily frequented event streams. Different solutions emerged, some partition the detection pattern itself others divide the incoming event stream. Each have their advantages and also their limitations. Window based CEP systems distributes batches of the event stream to individual pattern detection operators. Thereby the definition of the splitting rules is a major problem. Currently expert knowledge is required to transform the queries into usable splitting rules, which eliminates the possibility to change or add queries at runtime. Also makes the administration of a decentralized CEP system complex and cost intensive.

In this work an automatic method is proposed to generate the event stream splitting from the complex event query. Common language components are analysed and transformed into a language independent model which decouples the query language and the splitting mechanisms. The model is automatically transformed to support the consistent partitioning of an event stream in a decentralized CEP system. Further, a combination of window-based and key-based splitting is introduced to reduce the amount of events in each window. Restricting the events sent to each operator reduces the network load and also the CPU load on each operator.

In the next chapters a system is proposed to automate the process of generating a splitting algorithm for distributed CEP systems. To explain the basics, first common CEP systems and methods to distribute them are envisaged. Further the languages to configure them are discussed. Then the problems and the proposed approach is described, followed by an evaluation of the approach by the use of practical scenarios.

2 Complex Event Processing

Complex event processing is used to analyse streams of events and extract a conclusion as high level events, known as complex events. Also most CEP system are capable to react on a combination of events and handle independent from user input. The system is initialized and configured by a user given query, which is written in a complex event query language. A more technically definition is given in the Glossary subsection. Common tasks of CEP Systems are the filtering, the aggregation and transformation of events or parts of the event stream further detecting patterns and relationships. The first complex event processing concepts were developed in the 1990s in projects at the Stanford University [Lea09] and California Institute of Technology. The wide applicability of complex event processing area makes it complicated to define a System with its own language to support all of the possible application area. Therefore, a bunch of systems have emerged and specialized on a particular area, e.g. stock exchange [TRP11], sensor networks [BDG+07, WDR06], database integration [WL05, DBB+88, GD94, FVWZ02] or business models [HSD10].

2.1 Glossary

Following some definitions of commonly used words in this work are given to clear up misunderstandings:

Event An event is an occurrence in time. In CEP systems an event contains the time at which it is occurred, a type identifier and some additional data called payload e.g. the current temperature or the density of smoke in the air. The payload can consist of several key value pairs. The key can be used as a name, so if e is an Event and attribute is a key of a payload entry then let e.attribute be the corresponding value to the attribute. In this work Events are represented as Objects and each type of event is represented as a separate class which all inherits from an abstract type of event.

Event stream An event stream is a collection of occurred events sorted by their occurrence in time. It can contain different types of events from several systems which are connected into one event stream.

Complex event A complex event is an event which is generated by a specific combination of events. This generation is specified by an event query. A complex event contains the same information as a common event. The payload of a complex event can contain information of several events, which led to its generation, e.g. Sum up all prices in several orders and use them as new price.

Event query A query in CEP context is a structured text which describes, what the CEP operator should detect in a given event stream. It contains a definition of at least one complex event, the combination of events which lead to the generation of a complex event and how to construct the complex event from the recognized events.

Event query language An event query language or simple query language is a defined language on meta level. An implementation of an event query language is an event query.

2.2 Infrastructure

CEP systems are nowadays used as middleware and communicate between information source and information users, shown in figure 2.1. CEP handles the queries of the users and filter, aggregate and combine the incoming data and informs the users with a complex event when their queries are fulfilled. This is done continuously by the middleware. Internally most CEP systems consists of a language to define the queries and a model to handle the input and check the incoming data on the given queries. Most models build upon a state machine graph as in [CM12].

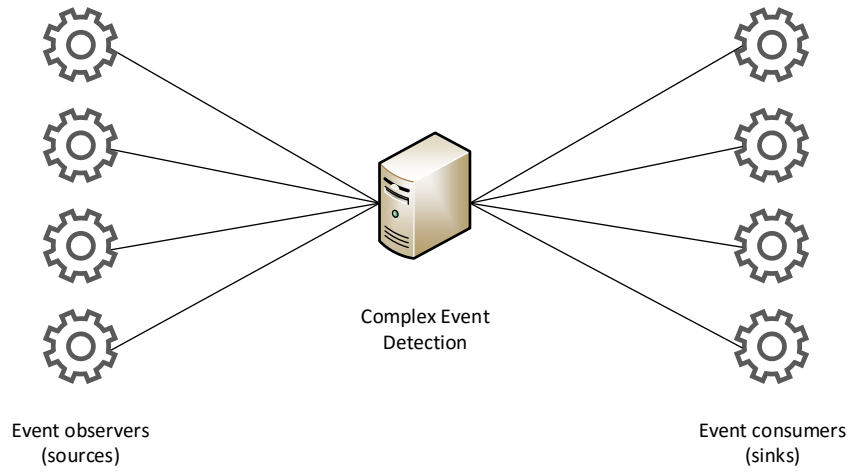


Figure 2.1: Infrastructure of a centralized Complex Event Processing system.

2.3 Example Scenarios

Complex event processing is used in a wide area of application. From applications with a highly flexible load to pattern detection at sensor networks in real time systems. To show the applicability of CEP systems some exemplary scenarios are given:

Sensor Networks The most commonly use case for CEP is in the processing of sensor networks. This can be in the use of smart homes, up to the control of production lines in the industry. Therefore, several sensors are constantly producing events like Temperature, Smoke or Velocity. These events are processed by the CEP system and combined to Complex Events, e.g. to recognize the occurrence of a fire. CEP systems also can make predefined actions on complex event detection, as activate the fire alarm when a fire is detected.

Stock Exchange Business Companies pay a lot of money to gain direct access to stock market data to sell and buy stocks at the best time. They require a powerful and fast processing and direct information passaging, but also must react on sequences of events to detect if the stock is further increasing or is starting to decrease.

Business Case As business Monitoring, techniques are described to keep track of company resources, recognize problematic patterns and solve them. This can be useful for industry companies which keep track of their materials and automatically orders new ones before running low and the production needs to be stopped. But not only industry can use this, it's also possible to keep track of project states and detect problems thereby.

2.4 Pros and Cons

The advantages of CEP systems are their fast response time. The systems administration also is pretty simple, because just a central system has to be linked with all event source and event consumer. This limits the complexity of the whole system. Another point is their wide applicability in several scenarios.

By far CEP systems are no solution for all problems and have their very own problems. The pattern detection at high frequent event streams is limited by the low distribution in many CEP systems. Especially with complex queries common CEP systems are overwhelmed. Also the network load is high, all data from event generators must be transmitted to a central system. A single operator is therefore limited by its communication interface.

2.5 Distributed Complex Event Processing

Distributed complex event processing (DCEP) is the combination of several operators to combine event streams similar to the divide and conquer method. It generates complex events from low level event streams, which then can be used in a second operator together with other streams to detect higher level complex events. Using it with only one query require a Query rewriting to distribute the operations [SMMP09].

In figure 2.2 the architecture of a DCEP is shown, which detects patterns at 3 different operators. First a pattern detection operator uses event streams from source 1 and source 2 and generates complex events from it. Second operator 2 uses those generated complex events and the events from source 3 to generate higher level complex events. Then top-level

complex events are detected by those complex event from operator 2 and events from source 4 in operator 3.

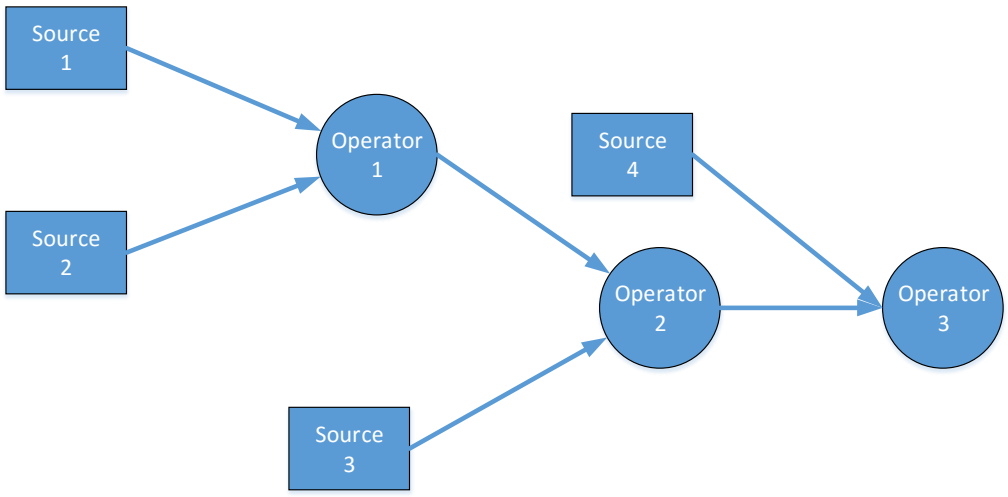


Figure 2.2: Architecture of a distributed complex event processing.

2.6 Distinction

2.6.1 Publish Subscribe

Publish subscribe is an architecture model in software systems which enables subscriber to get notified about information they are interested to from several publishers. The system continually receives messages from the linked publishers and organize them in classes. Subscriber then can sign in for classes and get notified whenever a messages fulfils their requirements. In difference to CEP systems publish subscribe systems only work on the current message and therefore have no information of the previously recognized messages. This makes it impossible for Pub/Sub systems to identify a sequence in a stream or other combinations between messages, e.g. timely dependencies. Also most Pub/Sub systems [EFGK03, TKBR14, BKR08] work on network messages instead of higher level objects as events, which makes them fast in transporting messages but lack in usability by specifying the classes. Against Pub/Sub systems common CEP systems can directly react to incoming events to notify a system, interact with business logic, call other processes and so on.

3 Parallelization of CEP

When centralized CEP systems reach their limit, distributed CEP systems are the solution. In difference parallel CEP systems can distribute all the steps from source concatenation to informing the information requester to different processes. The most critical part of centralized CEP systems is the pattern detection. Thus, several methods of distributing the pattern detection have been developed, those are explained in section 3.1.

3.1 Methods of Distribution

Several methods of distribution in CEP systems have emerged. In this work we will discuss and use the operator splitting methods. The target is to distribute pattern detection to several processes and reduce the amount of events each operator has to analyse. In the last years' two different approaches have been explored, the so called intra operator splitting and the data parallelization method. In the next sections both are described in their details.

3.1.1 Intra Operator Splitting

Intra operator splitting or pipelining analyses the query and split it into several steps, which then are processed in parallel [SGLN+11, BDWT13]. An example can be seen in figure 3.1. It shows the processing of stock events from IBM. First an event must contain the name "IBM", second step is to compare the latest two found events and make sure that the name is equal. At last the price difference is calculated, when the last price is higher than the event before a complex event is found. Intra operator splitting would divide these parts of detection into several steps as seen in figure 3.2. Each step can be processed on several processes but must be called in the given sequence. The reduced complexity results in a higher throughput at each step.

A simple query as in figure 3.1 with low variables cannot be split into a highly distributed process. Thus, this method of distribution is restricted in its distribution degree by the desired query and its variables and therefore another method is needed to yield a highly distributed system.

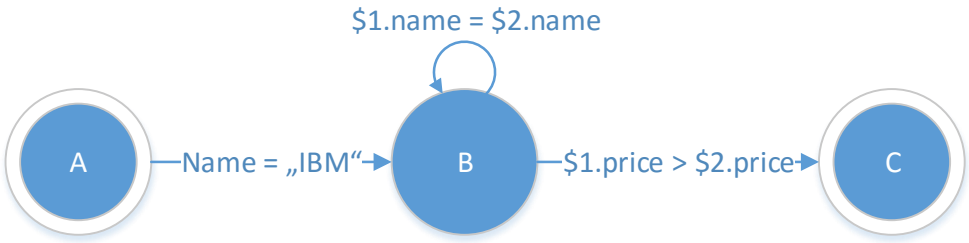


Figure 3.1: Example of an event query [BGHJ].

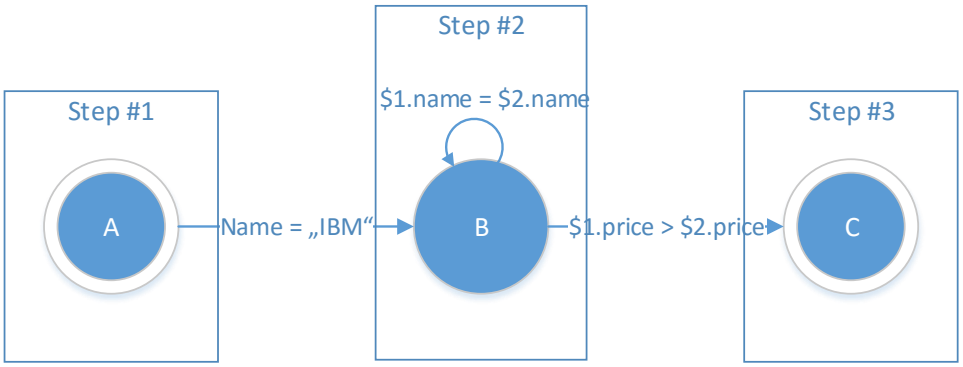


Figure 3.2: Distribution example of intra operator splitting [BGHJ].

3.1.2 Data Parallelization

Data parallelization partition the incoming event stream to serve data for multiple operator instances. A typical architecture can be seen in figure 3.3. The event stream is processed and partitioned at a splitting process. In this point the event stream is separated and several operators can work on their events. Different approaches have emerged to split the event stream the so called key-based splitting and the window-based splitting method, both are described below. In comparison to intra operator splitting, each part of the stream is processed by a full viable operator. Each operator notifies the relevant authorities whenever a complex event is found. This can be achieved directly by the operators or first transferred to a centralized process which then notifies all authorities.

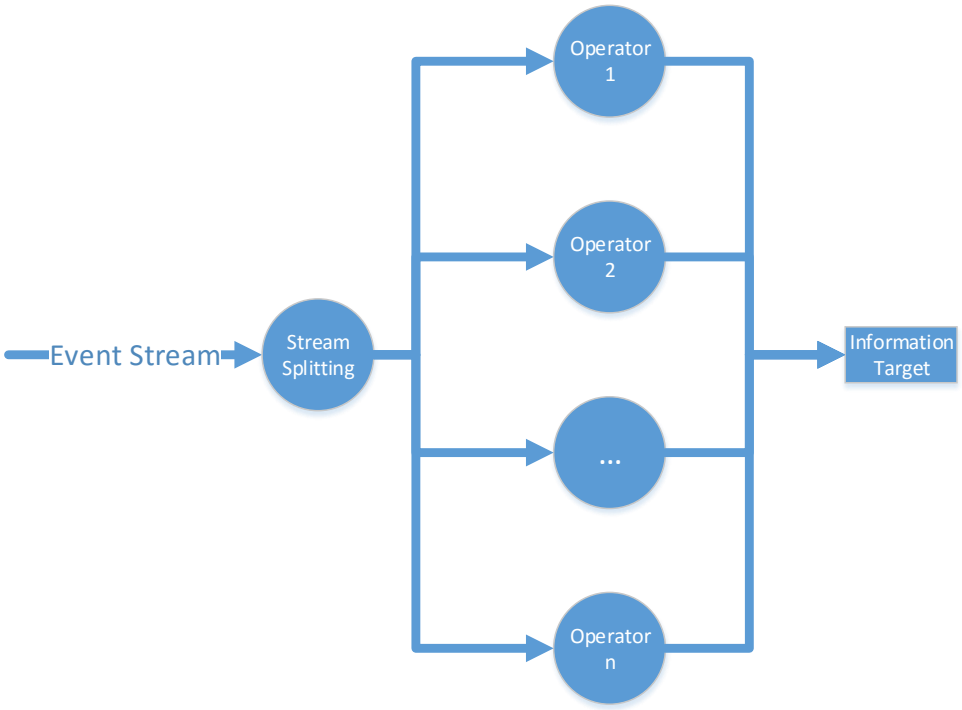


Figure 3.3: Example Architecture of a data parallelization method.

Key-based splitting

Key-based splitting is the base for almost all currently available data parallelization approaches [BEH+10, SHGW12, MBF14]. As key-based splitting, a method is meant, where each event is send to different operators based on a key in its attributes. The approach is insufficient to guarantee a high parallelization degree for all queries. This is a result of a limited number of different key values. Especially when there is no common key in the events or the pattern which is needed to detect is changed based on the appeared events, key-based splitting is limited in its parallelization degree.

Window-based Splitting

Window based splitting partitions the event stream into several windows. A window is defined as a continuous part of the event stream. In some approaches windows are of the same size or must be determined at the beginning. Generally, two predicates can be defined. First called P_{start}, determines whether a new window should be opened at each incoming event. Second called P_{close}, determines for all currently opened windows whether a window should be closed whenever an event has arrived. A possible partition of an event stream can be seen in figure

3.4. Each event of type 1 is used to open a new window and each event of type 3 is used to close a specific window, where the id of type 1 equals id of type 3. A scenario could be the login and logout procedure in a system and all type 2 events are actions made by a specific user while logged in. So each window reflects the login state of a user, when the window is opened a user has logged in and when its closed the user logged out. Note also actions from other users can be inside a window.

The constructed windows can be processed by individual operators. Each operator is fully functional and detects the complete query. An event can be part of several windows when they overlap each other, as seen in figure 3.4.

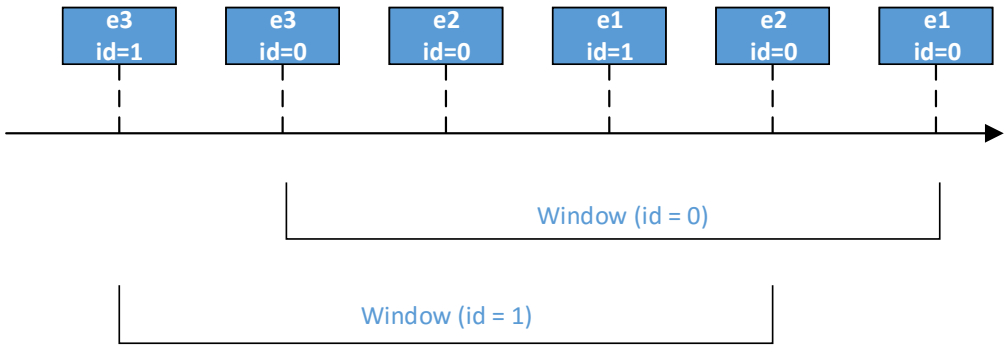


Figure 3.4: Example of a window based split algorithm.

3.2 ParSe

ParSe is a system developed in the CEPiL project, a cooperation between the University of Stuttgart and Georgia Institute of Technology. CEPiL targets at a highly scalable, flexible, robust and secure CEP system build in Java. It is designed to be flexible enough to react on event stream changes as increasing/decreasing event load [MKR14, MKR15], new event sources with variable operator instances by fulfilling today's security features at the same time. Further it provides a dynamic distribution to improve the network load in the system [MKR14].

3.2.1 Architecture

In figure 3.5 the architecture of ParSe is presented. In theory ParSe can be seen as decentralized CEP system with window-based splitting. Several event sources are combined to a single event stream by the so called source. The source collects the data and transforms it into event

objects. The events are sorted by occurrence time and transformed by a network interface to the splitter. The splitter is the most complicated part, responsible to partition the event stream and distribute the selections to operator instances. A selection is a window of the event stream, which contains all events required to decide if a complex event should be detected. The operator instances interpret the data and searches for the pattern specified by the event query. All operator instances can detect the complete query rather than just a part of the pattern as in intra operator splitting. The discovered complex events are sent to a merger, which orders all complex events and filters possible duplicates. The result is sent to the interested processes.

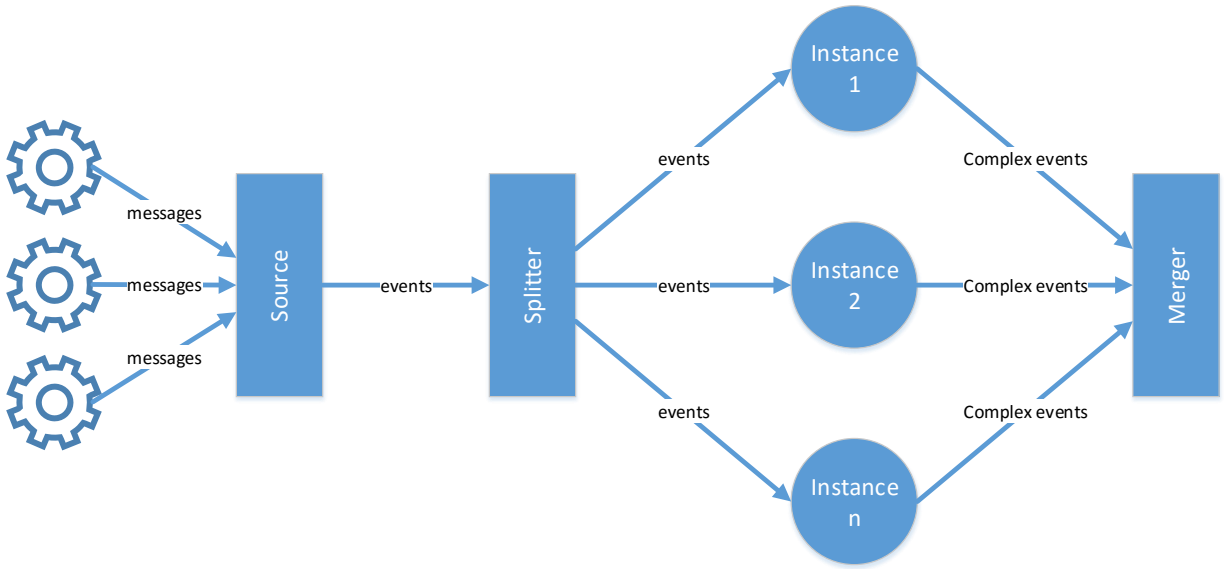


Figure 3.5: Infrastructure of the ParSe Project.

4 Languages

Query languages are a structured way to configure the CEP system. A query defines a pattern the system has to detect in the event stream. Also the actions provided, which should fire in case of a detection.

In the time several query languages have emerged. Generally two sorts of languages exist, transforming languages [GMM90, MMG92, ABW04] and detecting languages [CM94, GD92]. Detecting languages are common in CEP systems and aim to define conditions and actions to be taken when the pattern is detected. Conditions are defined as pattern of logical operators or timing constraints. Actions are specify how detected items have to be combined to produce new events. Transforming languages process input flows by filter, join and aggregate items and produce one or more output flows Most transforming languages are based on relational algebra and SQL [EM99] languages as CQL [ABW04].

4.1 Snoop

Snoop [CM94] is an event specification language developed by Chakravarthy et al. and targets on database systems. The motivation was to combine the knowledge of event detection in complex event systems with an active database. Therefore several methods to interact and react to database events are included and can be combined with primitive events. It can detect if data items are modified, created or deleted and then react automatically based on event-condition-action (ECA) rules. This means whenever an event occurs the condition is evaluated and if it fits an action is executed.

4.1.1 Definition

Snoop distinct between various different event types and composite methods, an event hierarchy can be seen in figure 4.1. Generally, events are classified into two different types first the primitive events, acting as basic structures in a query which detects the occurrence of events. Second the composite events define methods detecting combinations of events, by the use of operators. The grammar of the Snoop event detection system is rather simple and is defined in Backus-Naur-Form (BNF) in listing 4.1. The BNF can be used to generate complex event queries. In difference to other languages Snoop doesn't provide a structured method to define the generation or definition of the resulting complex event. So Snoop only detects a complex event in a structured way and leaves the handling to the user. Following structures forming such an event detection are described in detail.

```
<E1>    ::=    E1 v E2 | E3
<E2>    ::=    E2 ; E3 | E3
<E3>    ::=    All(E4) | E5
<E4>    ::=    E4, E5 | E5
<E5>    ::=    S(E1,E1,E1)
                  | A*(E1,E1,E1)
                  | P(E1,[timeinterval],E1)
                  | <timestring>
                  | (E1) + [timeinterval]
                  | Label:(E1)
                  | (E1)
                  | BOB
                  | EOB
```

Listing 4.1: Grammar of Snoop in BNF.

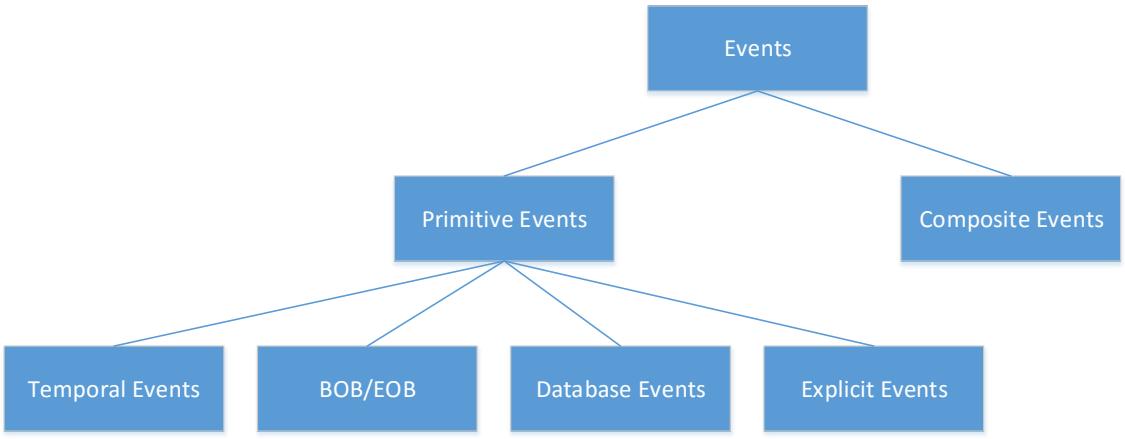


Figure 4.1: Event hierarchy in Snoop.

Primitive Events

Primitive events are events which can be detected by looking at only one event at a time. In Snoop those are database events, temporal events, explicit events, begin of Block (BOB) and end of Block (EOB).

BOB / EOB

BOB and EOB are events specifying the begin and the end of a block statement, similar to methods in conventional programming languages. Parameters are event-type, timestamp, block-id and the user specified parameters. BOB and EOB are used when an event can't be treated as a single event due to an unknown execution time. Those events are defined with the start of the operation and the end of it.

```
Event: [ (h:m:s)mm/dd/yy ]
```

Listing 4.2: Structure of an absolute temporal event.

```
Event: < event + (h:m:s)mm/dd/yy >
```

Listing 4.3: Structure of a relative temporal event.

Database Event

Database events are operations on a database and are treated as normal events. Snoop recognizes four different types of database operations, Access occurs when a data item is accessed, Insert when new data is put into the database, Delete when a data item is removed and Update when a data item is overwritten.

Temporal Event

Temporal events are further divided into absolute events, occurring at a specific point in time, e.g. 5am, and relative events which are specified by another event plus a timespan. An absolute event as seen in listing 4.2 is a defined value in time it can contain a wild card denoted as '-' which represents any value in that field. So it can be specified a time which occurs once a day or similar. Relative events as seen in listing 4.3 contain an event pattern as reference point and a given timespan as offset time. It's alarmed when an event occured, which matches the event pattern and then the offset time passed.

Explicit Event

An explicit event is a simple event with type and time of occurrence, but also user defined parameters can be specified to refine the matches.

Composite Event

Just primitive events aren't sufficient for several scenarios of applications. For example sequences of events cannot be specified by only using primitive events. Therefore the composite events are declared. In difference to primitive events composite events combine several event patterns into another event pattern. So the event detection mechanism must observe at least the current event but mostly also needs to look for matches found in the past. In Snoop five classes of composite events are declared Disjunction, Sequence, Conjunction, Aperiodic and Periodic.

```
Event: A*(S, Ack, S+[(10::)//])
// The event of this rule occurs 10 seconds after the occurence of Event Ack.
```

Listing 4.4: An example of a Snoop query.

```
A*(All(Login, Order) ; <(10:-:-)-/-/->, Order, Purchase)
```

Listing 4.5: Example of a Snoop pattern detection.

Disjunction

In Snoop a disjunction is specified by $E1 \vee E2$, which triggers either when E1 is signalled or E2 is signalled. This operator is useful if one or more events should trigger a rule to fire.

Sequence

With the sequence operator $E1; E2$ a sequence of events can be specified. So it is only triggered when E2 follows E1. More technically the first occurrence of E1 is guaranteed to be less then the time of E2's occurrence. It's not guaranteed that after E1 has triggered any E2 will occur and leave the rule in a waiting stage. This can be eliminated with further combination of composite events.

Conjunction

In difference to disjunction where either E1 or E2 must be triggered, by a conjunction both E1 and E2 must be triggered to get the rule to fire a complex event. The order in which both sides of the operator must be signalled doesn't matter. A separate Conjunction operator isn't required, because it can be generated by a combination of Sequence and Disjunction operators. So $All(E1, E2)$ could be written as $((E1; E2) \vee (E2; E2))$, however this is not suitable for long combination, when more than two event types needs to be verified.

Aperiodic

An aperiodic event is specified by $A(E1, E2, E3)$ or A*$(E1, E2, E3)$. E1 and E3 define an interval which is opened when E1 occurred and closed on an occurrence of E2. The A operator is signalled each time an event E2 occurs inside of the interval, whereas the A* operator combines all occurrences of E2 and signals the combination only once at the end of the interval.

Periodic

A periodic event p is specified as $P(E1, t, E2)$ where t is a constant time period without wild cards and E1 and E2 are events. E1 denotes the beginning of an interval and E2 denotes the appropriate ending. Inside this interval each t time the event p is signalled.

4.2 TESLA

TESLA is a language based on TRIO [GMM90, MMG92] temporal operators, which targets at a high degree of expressiveness to users while using a simple syntax with defined semantics. TESLA concentrate on the definition of sequences. In difference to Data Stream Processing (DSP) languages, where the query is working with streams from database tables as CQL [ABW04], TESLA doesn't directly support the connection to databases and relies on an object oriented event stream. Also TESLA uses a simple syntax while supporting indirect Sequences, Negations and Iterations. Most other languages doesn't support Negations [BDG+07, LJ05] and therefore can't model some scenarios.

Based on TESLA the complex event processing system T-REX [CM12] were developed to demonstrate the expressiveness and efficiency of TESLA. It transforms the TESLA rule into an ordering graph. This ordering graph is hard to process in an operator, so it is transformed into an linear model of automaton. Each path from a leaf in the ordering graph to the root is represented as a single linear sequence. Each of it can be evaluated by its own.

Following first the definition of a TESLA rule is explained and followed by a brief introduction to the ordering graph.

4.2.1 Definition

The definition of a query in TESLA, a so called rule, consists of four parts the 'define', 'from', 'where' and 'consuming' clauses and is presented in listing 4.6. The 'define' clause declares the complex event with a label and all attributes of the complex event type are specified. In differ to other languages as Snoop, events can be clearly defined and therefore used in further processing steps. The 'from' part gives the pattern how the event can be detected. Therefore first a so called 'terminator' is given. This is the last event in the pattern which leads to an occurrence of the complex event. Then several predicates can be defined, each is a pattern to detect one type of event and needs to be placed in a sequence to a previously defined pattern. There are two types of predicates the positive, which declares a pattern which must found and negatives, to declare events which aren't allowed in the sequence. Positive predicates consist of a pattern matching an event type and a maximum delay between this predicate and the terminator or a previously defined predicate. Negative predicates define an event which isn't allowed between two other predicates. Through those predicates a detection graph is constructed with a single target node, the terminator. This procedure is explained in chapter 4.2.2. After the detection is declared, the 'where' clause defines the way how the attributes of the complex event can be set from the detected predicates. For each attribute a function $f_1, ..., f_n$ can be asserted which depends on the detection pattern defined in the 'from' clause. Finally, the optional 'consuming' clause is given, which invalidates events from the event stream and prevents the pattern matcher to generate another complex event based on the invalidated events.

```
define        ComplexEvent(Att_1 : Type_1, ..., Att_n : Type_n)
from          terminator (AND predicate)*
where         Att_1 = f_1, ..., Att_n = f_n
consuming     e_1, ..., e_n
```

Listing 4.6: Structure of a TESLA rule.

```
Assign 2000 => Smoke, 2001 => Temp, 2002 => Rain, 2100 => Fire
Define Fire(area: string, measuredTemp: float)
From Smoke(value=>$a)
       and last Rain() within 300000 from Smoke
       and last Temp([int]value>$a * 2 +17) within 300000 from Smoke
Where area:="foo", measuredTemp:=AVG(Rain.value([float]value > $a /2)) between Smoke and
    Temp;
Consuming Rain, Temp;
```

Listing 4.7: Fire detection query in tesla.

4.2.2 Ordering Graph

T-Rex uses the TESLA rules to generate an ordering graph, which then can be used to detect the pattern. The ordering graph G(V,E) is a acyclic connected graph with a single root node. The ordering graph is designed to represent all sequences allowed to produce the complex event of the query. Following the generation of such an ordering graph is explained by an example.

In listing 4.8 an example of a simple shop system is given, where a users can login and order several items and finish the shopping with a purchase. Further it is restricted to logout before buying the items. On purchase an Delivery event is generated which contains the name and the address of the user and the summed costs of items. The appropriate ordering graph is displayed in figure 4.2

The ordering graph is constructed by generating a node n_t as representation of the terminator, in this case the Purchase event. Then all predicates referring to already processed predicates or the terminator generate another node n_i with an edge e_i which connects n_i with the appropriate nodes. In the example first the node for Order will be created and connected to the terminator node n_t with an edge value of 300 seconds. On the same way a node for Login is created and connected to the Order node with an edge value of 300 seconds. Negative predicates, in this case the Logout predicate, refer to two other predicates Login and Purchase, so a node is generated and is connected with both as seen in the figure. In difference to the positive predicates, edges created by a negative predicate don't have an edge value.

```
Assign 4000 => Login, 4001 => Order, 4002 => Logout, 4003 => Purchase, 4100 => Delivery
Define Delivery(name: string, address: string, ordersvalue: int)
From Purchase(id=>$id)
       and each Order([int]id=$id) within 300000 from Purchase
       and first Login(name=>$name, address=>$address, [int]id=$id) within 500000 from Order
       and not Logout([int]id=$id) between Purchase and Login
Where name:=$name, address:=$address, ordersvalue:=SUM(Order.price()) between Login and
    Purchase;
```

Listing 4.8: Basic online shop query in tesla.

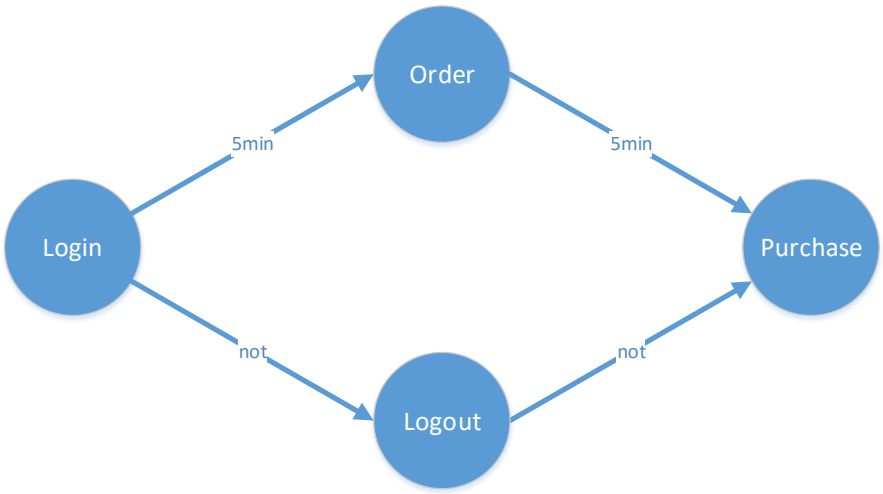


Figure 4.2: Example of an TESLA ordering graph.

5 Problem Description

5.1 Definition

Parallel complex event processing is used and required in large scenarios with thousands of events per second. For the distribution of the events several requirements must be fulfilled. In window-based systems the splitter is the only part of the system which needs to process all events in the event stream. In a system with several operators the splitter must be a lot faster to allow an optimal distribution, therefore its necessary to keep the CPU requirements low. Further the splitter must ensure that the parallel operators detects the same complex events as a single operator would. In this work a method is introduced to automate the splitting of the event stream based on common CEP language queries. The approach has to guarantee a fully functional distributed system. The automation must ensure that common language elements as Negations, Iterations, Sequence and so forth must be detected in the queries and resolved in a window based partitioning scheme. Further it must be flexible enough to easily enlarge the system with new query languages. As a prove of concept an implementation should be implemented, which extends the Parse system and verifies the approach in a common scenario.

5.2 Requirements

Parallel CEP systems have a large set of use cases and are confronted with several requirements. In particular the system has to behave like a common centralized CEP operator, but must be faster in processing. In this work three main requirements have to be fulfilled. First the correctness of the CEP system has to be guaranteed, second the system should allow an easy extensibility with new languages. Further the network traffic should be minimized and the CPU requirement of the splitter should be as low as possible.

5.2.1 Correctness

In this work the correctness requirements can be divided into following parts:

- The parallel operator should behave like a single operator who gets the whole event stream and shouldn't detect less, more or other events than a single operator.
- The splitting should be consistent and deterministic. That means if an event stream is processed twice the results should be the same.

5.2.2 Extensibility

In software development extensibility is a systems ability to allow extensions and changes without major rewriting or changes in code. In CEP systems several languages to describe queries exist as seen in chapter 4. In this work it is sufficient to show the theory with the use of two languages Snoop and TESLA. To support the use of other languages the language itself must be decoupled from the solution, so language experts can build their own automatic model easily.

5.2.3 Network Load

The approach should be aware of limited resources in network traffic. This can be the bottle neck in distributed systems, due to the fact that each event in the incoming event stream can produce several events after the splitting method, this can overwhelm the splitter. Therefore the amount of distributed events should be minimized.

5.2.4 CPU Overhead

In the ParSe system the splitter is the main process, which processes each event in the event stream. Through the processing of each event the splitter can become a bottleneck in the system. For this its important that the splitting at all needs to be fast. This depends on the query used and is rather a soft requirement than a strict requirement, due to no time can be specified for all queries.

6 Proposed Approach

With this approach a solution is presented which automates the transformation from CEP queries to splitting rules for window-based splitting. As denoted in chapter 3 window-based splitting can be done by evaluating the two boolean predicates P_{start} and P_{close}. P_{start} evaluates if a selection is opened and P_{close} marks the end of a selection. Every selection can then be processed by an individual operator instance. As an extension to this model a new predicate P_{in} is introduced, which filters events in the splitter for each selection instead of sending it to the operator and producing unnecessary network load. In (6.1) the equations for the three predicates are shown. P_{close} and P_{in} are processed with the current event e and for all currently opened selections σ_{open} whereas P_{start} is triggered for each incoming event independently from opened selections.

$$
\begin{aligned}
P_{start} &: e && \rightarrow BOOL \\
P_{close} &: (e, \sigma_{open}) && \rightarrow BOOL \\
P_{in} &: (e, \sigma_{open}) && \rightarrow BOOL
\end{aligned}
\tag{6.1}
$$

The approach is divided into three different parts the language parsing, an universal graph and a splitting model and it's architecture is shown in figure 6.1. The language parsing module reads the query in as text and constructs an universal graph, due to the language dependence this part must be implemented for each supported language. The universal graph is a construct based on the TESLA [CM10] ordering graph to provide an abstraction method between the query language and the splitting model. It supports the major language elements and is easy to understand for language designers. From an universal graph, splitting models can be constructed for each predicate P_{start}, P_{close} and P_{in}. A splitting model is a tree based structure where each of the items decide a boolean predicate. Splitting models are constructed with predefined model components supporting most language decisions necessary for the predicates.

In section 6.1 the structure of the universal graph and in section 6.2 structure of the splitting model are explained. Followed by section 6.3 explaining the generation of the splitting models from the universal graph. At last section 6.4 describes an implementation of the approach in the context of the ParSe system.

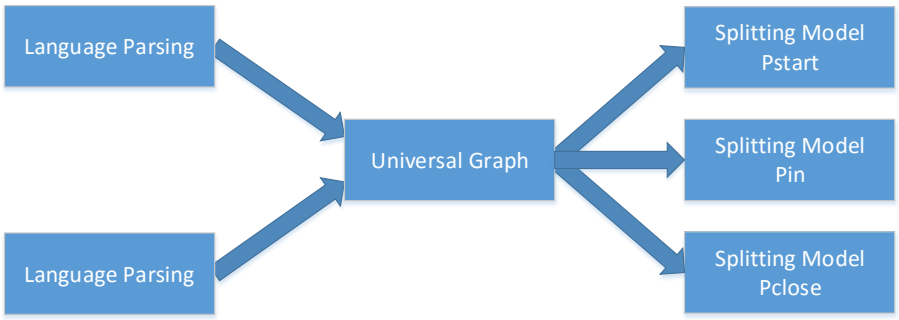


Figure 6.1: Infrastructure of the proposed solution.

6.1 Universal Graph

The universal graph is a structure between the language dependent query and the language independent splitting model and is used to represent CEP queries in a language independent manner. It's a graph structure based on the ordering graph from TESLA as described in chapter 4.2.2. The ordering graph shows all possible combinations of events triggering a complex event generation. Each path from a start node to the terminator (only target node) represents a sequence of events fulfilling the requirements in the query and therefore generate a new complex event. By the use of not edges, also restrictions to those paths can be made. Although the ordering graph is proposed together with TESLA it's language independent, but need adaptations to support all common language elements. The universal graph extends the ordering graph to support the missing language elements by introducing new types of nodes and edges. Nodes of the universal graph are presented in figure 6.2 and the edge types displayed in figure 6.3. In the following all node types and edge types are described.

6.1.1 Nodes

Simple

A simple node is used as a dummy node for various kinds of constructs. Simple nodes don't contain information about the query itself, but are e.g. used by relative temporal events in the Snoop [Mis91] language where an event type is specified plus a timespan, the event type is specified as an Event-node and is connected with a Time-edge to a Simple-node.

Begin of Block & End of Block

Begin of Block (BOB) and End of Block (EOB) are used to mark blocks inside the universal graph, typically used for iterations of sequences. The block declared have defined start nodes and one defined end node. Each connections with Time, Not and Direct edges have to target

(a) Simple node.

(b) Begin of Block (BOB) node.

(c) End of Block (EOB) node.

(d) Logic node of which represents an and operator.

(e) Event node of a Login-event with id 3000.

(f) Time node representing 10 o'clock.

Figure 6.2: Representations for the different types of nodes used for the universal graph.

the BOB or EOB node. It is not allowed that Time, Not and Direct edges target a node inside the block.

Logics

Many query languages allow the use of logical operators in the rules. Therefore a representation of typical operators are supported by the universal graph as node types for And, Or and Xor. Others can easily be extended but currently not part of the approach.

Event

Event types are the most common types of filters in query languages. Event nodes represent those filters with the name of the event type and if given their appropriate id as it can be seen in figure 6.2e. In this example only Login events would be accepted, all other nodes are rejected.

Time

Besides the event filters also points in time are one of the most commonly used features in pattern detections. Time nodes represent the time but can also have wildcards and accept everything above or below the timestamp. The format of the timestamp is shown in the format "hh:mm:ss, dd,mm,yyyy" Wildcards mentioned with '-' can be used to ignore a given value so as in figure 6.2f the time node would accept each events with time $>$ 10 o'clock independently from the date.

6.1.2 Edges

Direct

Direct edges symbolize direct sequences between nodes without further restrictions in time or attributes.

Not

As in the ordering graph from TESLA, not edges denote an event which is not allowed to occur at the appropriate position. A Logout from an account should not be done before the Purchase of an Order, so there would be a Not edge from Logout to purchase.

Time

Time-edges denote time constraints between nodes, the time mentioned is the maximum time difference allowed between recognizing the source and recognizing the target of the edge. When detecting fire from smoke and temperature sensors, a high temperature and a high density of smoke in the air should occur in a time interval of a few seconds, otherwise there is no fire.

Complex

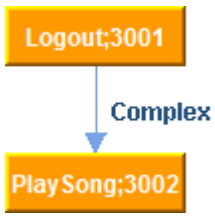
Complex edges represent variable conditions between different types of event filters. This could be that ids of Login-event and Logout-event must be the same, as demonstrated in 6.3a. Targets and sources of this edge must be Event-nodes.

Operator

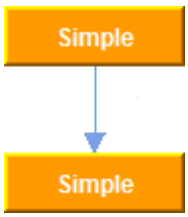
Operator edges are similar to Complex-edges and denote a variable restriction. In difference only restrictions from one event is specified, e.g. specifying all stock events with price > 20.

6.1.3 Example

In figure 6.4 an universal graph is presented with different kinds of nodes and edges. In the example a sequence is searched which begins with A_Event, B_Event or C_Event but each of the events must be contained. Further A_Event must be at least 40 seconds before the next step of the sequence, which is triggered at 10 o'clock each morning. Begin and End nodes denote a block which is a loop to detect an infinite amount of D_Event's. At the end of a sequence an E_Event should be detected, which would then trigger the generation of a complex event.



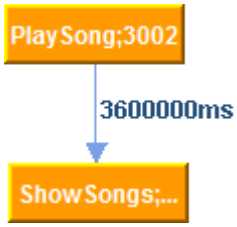
(a) Complex edge between Login and Logout events.



(b) Direct edge between two simple nodes.

(c) Not edge forbidding a Logout-node.

(d) Operator edge.



(e) Time edge which limits the sequence to 1 hour or 3,600,000ms.

Figure 6.3: Representations for the different types of nodes used for the universal graph.

6.2 Splitting Model

Additional to the Universal Graph another model was constructed named Splitting Model. It defines the way how incoming event streams are distributed to several instance operators. Therefore the components of this model answers the boolean predicates P_{start}, P_{in}, P_{close}. Splitting Models are a tree based structure with a defined root element, each item in the tree has a defined behaviour. Leaf items directly answer the predicates and internal nodes can change the behaviour of the descendants or combine their descendants, e.g. an AND operator. All components represent common features used to split the event stream in manual implementations. Following the different model components, which representation is shown in figure 6.5 are described in their detail.

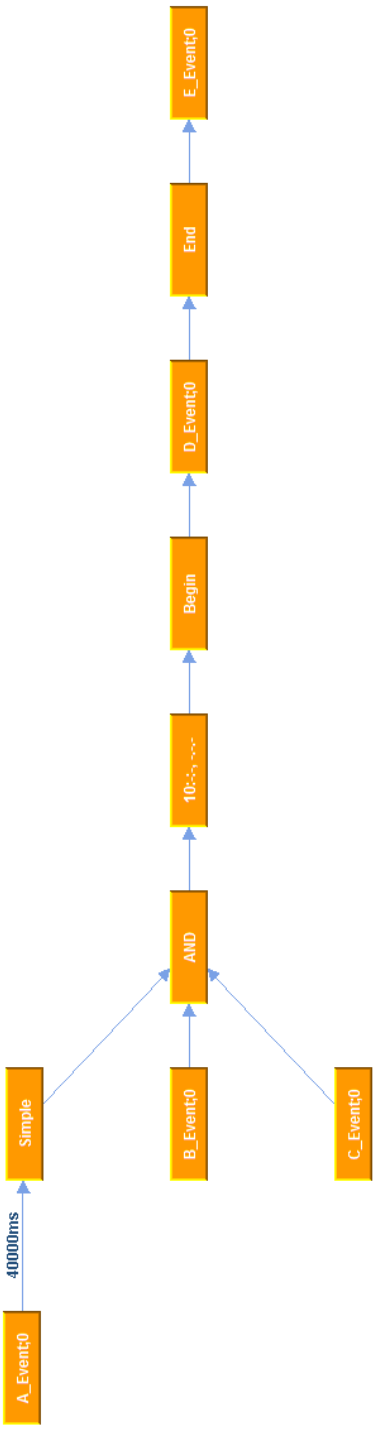


Figure 6.4: Example of an Universal graph with different kinds of nodes and edges.

6.2.1 Model Components

Event Type Filter

Event type filters test an incoming event and decide based on it's type whether to accept the event or not and is used in almost every model.

Time Comparison

This component uses the arrival time of an event and compares it to a timestamp. It is used to specify time constraints which are often used in queries, as temporal events in the Snoop language. If the query needs to detect Login event only after 10 o'clock, then a time comparison could be combined with an event type filter, to filter Login events which occurred before 10 o'clock.

Variable Comparator

Variable comparators check a given variable inside the current event and compares it to a defined value. For example a stock value have to be greater than 45 or the name of the stock must be "IBM".

Time Stop

In difference to time comparison component, time stop component uses a dynamical time for each comparison. The dynamical time constraint is explained in the chapter 6.3. Time stop components are explicit used in the P_{close} splitting model.

Compare

Compare component is used to compare variables between the current event and the start event of each selection. So e.g. the ids of order events can be compared to the id of the Login event, which opened the selection.

Branch

Usually each component in the splitting model is used globally for each selection, branch model changes this aspect by using all children individually for each selection. This is mostly used together with the Unique component to allow a specific combination of events once for each selection.

Logic

To combine several model components also logical operators as AND, OR, XOR and NOT have to be considered and are represented by logical components. Ad an example the And operator can combine an event type filter with a variable comparator to connect the variable directly to a specific type of event.

Switch

Switch component exchanges the currently compared event for all children by the start event of the current selection. It can be used to specify individual behaviour for different starting event types. This allows a more specific behaviour for each start node type, so the paths in the universal graph can be represented with more detail.

Unique

Unique model is used as an internal node and restricts the accepts of the children to one for each selection. So the first accept of the children is accepted, further accepts are declined as long as the selection is opened.

6.3 Generation of the Splitting Models

6.3.1 Generation of P_{start}

The splitting model for the predicate P_{start} is required to detect events which opens a new selection. Starting a selection can only be triggered by an event type, which can be the first in a pattern detection sequence. Those events can be gained from the universal graph, by finding nodes without incoming edges by ignoring complex edges, operator edges, simple nodes and logic nodes. Thereby a set of time nodes and event nodes is gained, the so called start nodes, each can be a starting event of a selection.

In figure 6.6 the generated splitting model P_{start} of the universal graph in figure 6.4 is shown. It can be seen that each of the events (A_Event, B_Event, C_Event) are represented by an event type filter and can open a new selection. Time nodes in the universal graph are represented by a time comparison component. When there are no further restrictions through complex or operator edges in the universal graph, an automatic distribution is not possible then a unique node has to be placed as a root node to the splitting model. This is important, because by distribution a possible false-negative detection can occur and therefore offend the system requirements stated in chapter 5. When using a complex edge or an operator edge with an equation, for each individual value of the variable a new selection can be constructed without producing false-positive events. If several complex and operator edges to a start node with different variables exist, the combination of used variables needs to be unique. This is directly checked inside the event type filter.

(a) Representation of an event type filter.

(b) Representation of a time comparison component.

(c) Representation of a variable comparator.

(d) Representation of a time stop component.

(e) Representation of a compare component.

(f) Representation of a branch component.

(g) Representation of a logic component.

(h) Representation of a switch component.

(i) Representation of an unique component.

Figure 6.5: Representations for the different types of nodes used for the universal graph.

6.3.2 Generation of P_{close}

Intuitive someone would use all nodes, without outgoing edges, in the universal graph and close a selection on occurrence. This would imply that the splitter can identify this event without doubts, for this the splitter have to detect the whole pattern, which would eliminate the use of operators. Without distributed operators the CEP system would by centralized. Therefore, only the time is a valid method to detect the closing of a selection. In this approach the maximum time which can pass between a selection is opened and a selection is closed is calculated with the universal graph. Therefore, the same start nodes as in the splitting model of P_{start} are used. For each of the nodes the maximum path length to the target node, which is unique, is calculated. In this calculation time edges and direct edges are used. Time edges add the appropriate value to the path length. If in the path between two event nodes only direct edges and no time node exist, the time which could pass between the occurrence of those event could be infinite. A selection opened by this start node can never be closed in this case, because there might be the possibility of an upcoming complex event. Paths with time nodes are dynamically and change the allowed time depend on the current timestamp.

As seen in figure 6.7 for each combination of start nodes with the target node, an individual time stop component is used as a child to a branch component. The branch component ensures

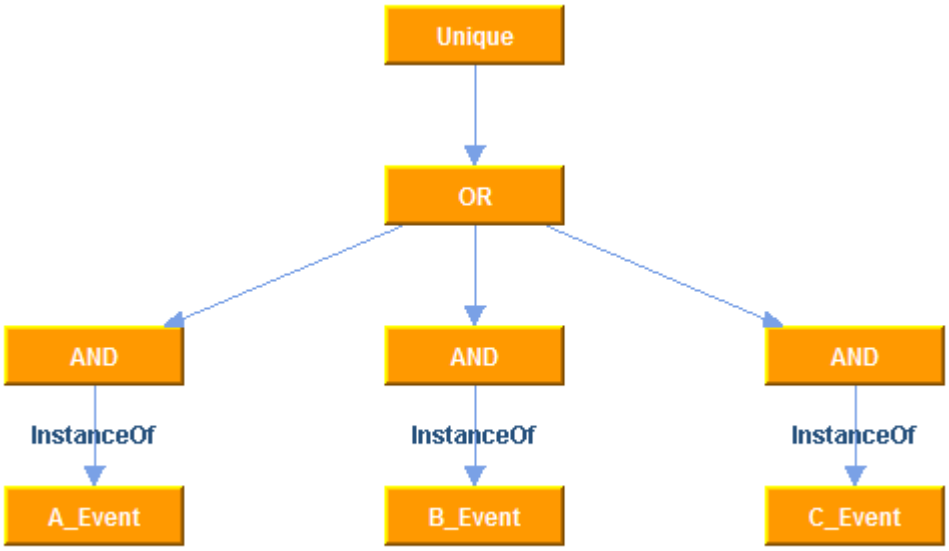


Figure 6.6: Example of a generated splitting model for P_{start}.

the individual detection for each selection, by referring the call of the predicate to a unique copy of time stop components.

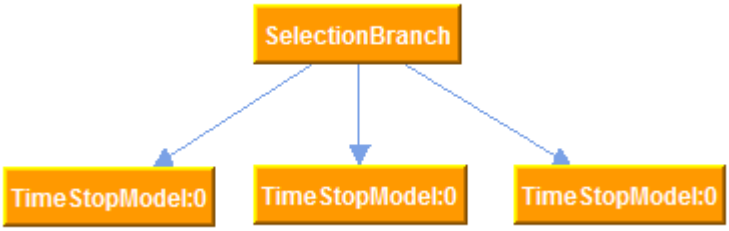


Figure 6.7: Example of a generated splitting model for P_{close}.

6.3.3 Generation of P_{in}

Splitting model generation for P_{in} is done in two steps. First all possible start events need to be accepted. Second for each other event filter in the universal graph an accepting filter must be constructed.

As in the splitting models for P_{start} and P_{close} the same start events are used. Each of them with their appropriate conditions from compare nodes are represented in event type filters with additional variable comparators. Event filters and compare nodes are combined with

the logic node for AND. In the example of figure 6.8 which is the P_{in} splitting model for the universal graph of figure 6.4. Those components are further placed besides a time comparison node which states the global type. In this case the start nodes have to occur before 10 o'clock as mentioned in the universal graph by the use of a time node.

The second part is generated for each event node in the universal graph which is no start node. Therefore, an event filter which matches the type of the event in the universal graph is combined with the global time at which the event can occur by an AND module. In the figure 6.8 this is done for D_Event and E_Event each can only be detected after 10 o'clock, denoted with a time comparison component. Further restrictions are the possible start events for this type of event. To do this a switch component is added with all start event filters having a path to the current node in the universal graph.

6.4 Implementation

For evaluation purposes and as proof of concept for the approach a Java implementation as a ParSe extension has been created. It supports TESLA and Snoop as input languages and automatically creates universal graph and splitting models from queries. Within the ParSe system the extension allows the system to split incoming event streams automatically. However the implementation of the operator instances has to be provided by a developer, which was out of scope in this work.

6.4.1 Architecture

The architecture of the implementation is described in figure 6.9. Same as in the approach the three main parts, language parsing, universal graph and splitting models, are separated to guarantee the best extensibility and follow best practices in software engineering as separation of concerns.

ANTLR (ANother Tool for Language Recognition) [PQ95] is used in this approach to parse the languages and construct an object model of event queries. ANTLR uses a grammar similar to EBNF to parse an input string into objects. The object types are automatically generated based on the grammar. In this implementation a grammar for TESLA (see appendix A) and for Snoop (see appendix B) was created and used to parse the languages. From the objects generated by ANTLR the universal graph has been created. As TESLA and Snoop are in continuous development the usage of ANTLR allows an easy adaptation to improved or changed languages by abstracting the grammar and the language object created. So whenever the language changes in its grammar only the used grammar to parse it has to be adapted.

The universal graph is implemented as a graph in the library JGraphT [jgr]. Each Node type as explained in the section 6.1 is created as a separate class. The graph library allows an easy implementation of the P_{close} method by providing algorithms as Dijkstra to dynamically detect the distance in time between two nodes.

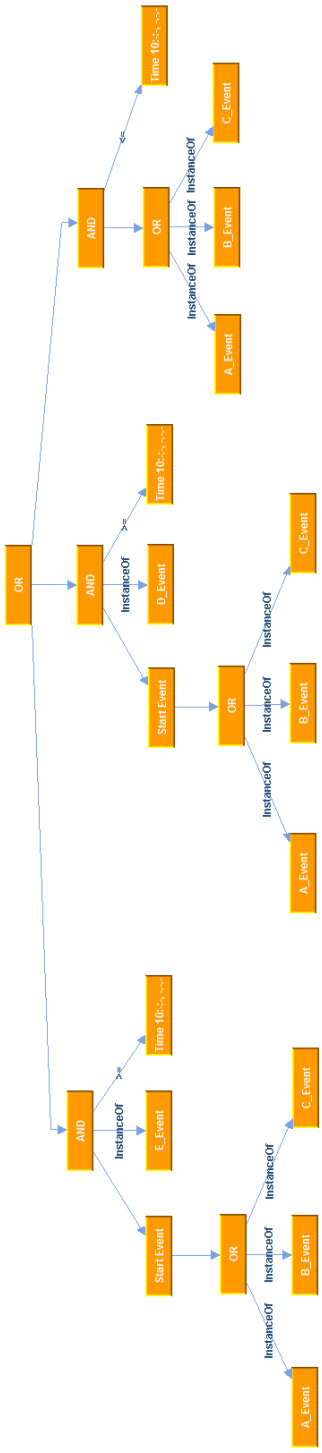


Figure 6.8: Example of a generated splitting model for P_{in}.

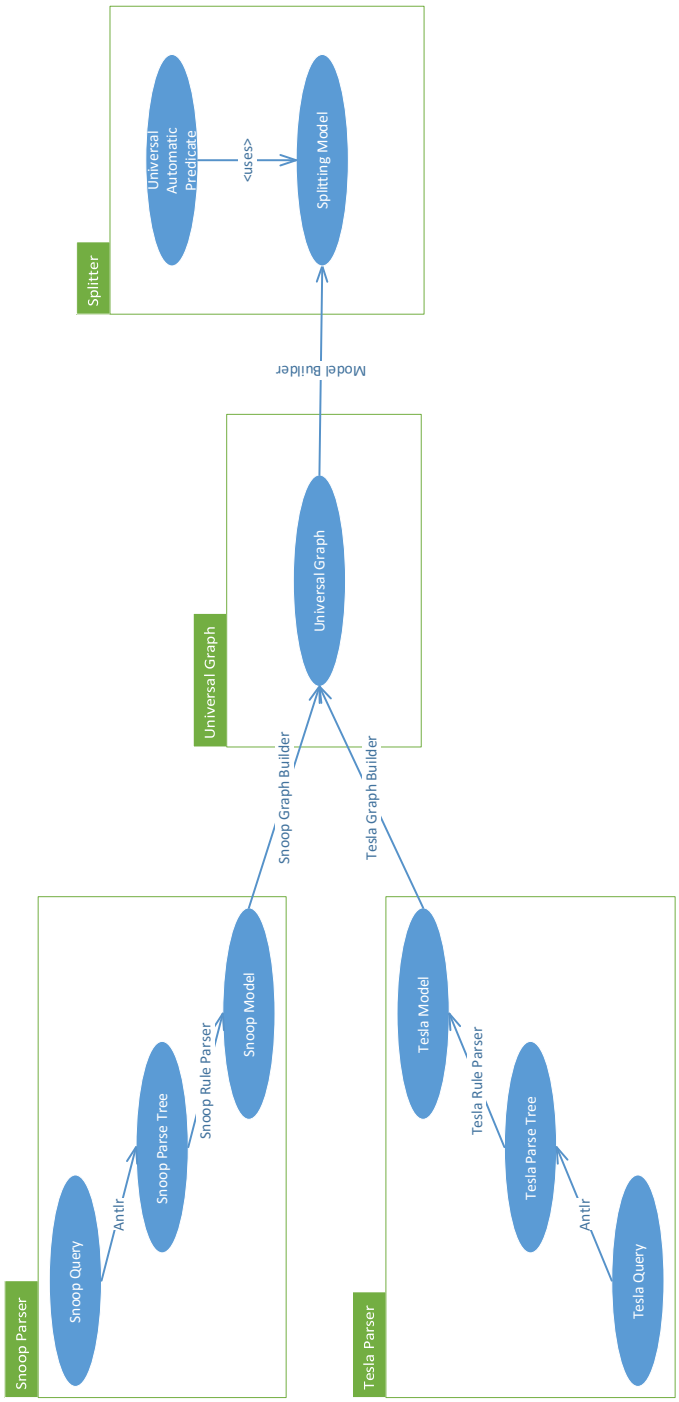


Figure 6.9: Infrastructure of the proposed solution.

6.4.2 Integration into ParSe

ParSe uses predicate objects to handle the incoming events and distribute them to running operator instances. In terms of this implementation the universal automatic predicate has been created, which gets the query as string input and the specific language type as enum and triggers the model generation. From parsing the language into language dependent objects and generation of the universal graph to the construction of the three splitting models is done automatically.

The generated splitting models are then used by this predicate for each incoming event in the following procedure:

1. Parse the events from binaries.
2. Each event is proven by the splitting model of P_{start} on accept the event generates a new selection, otherwise nothing happens.
3. Iterates on all active selections and does following:
   a) Prove the events with the P_{in} splitting model and on accept marks the event to the operator handling this selection.
   b) If accepted by P_in prove if its closing the selection by using the splitting model of P_close. Events not contained in a selection aren't needed for the P_close predicate, they can't close a selection.
4. The event is sent to each marked operator.

7 Evaluation

The Implementation based on the approach presented in chapter 6 is evaluated to show the fulfilment of the requirements presented in chapter 5. The focus of the evaluation is on the correctness of the splitting method in the automatic generated splitting model. The extension with the P_{in} predicate is inspected by the network load and the processed events. As it is impossible to test all configurations of a generic model, two scenarios are provided, which could be used in real applications. The first scenario is a basic model of an online shop which tracks the users' behaviour. The second is located in music stream networks and notifies all your friends on your currently listened song. The first scenario is used to validate the splitting method in terms of correctness. The second scenario uses real data from last.fm and Facebook to show the benefits of the P_{in} predicate and its use in parallel CEP systems.

In section 7.1 the scenarios and their queries, universal graphs and splitting models are presented in detail. Followed by section 7.2 which shows the correctness of the approach by the use of the shop scenario. The extension is evaluated in section 7.3 with the music network scenario. In the end the results are discussed based on the requirements of chapter 5.

7.1 Scenarios

7.1.1 Shop Scenario

In the last years the numbers of web shops have emerged and nearly all producing companies offer their products also directly to their customers by online shops. But rather than simply making money through sales, also using personalized advertised is a common money making method. Therefore, the users' buying behaviour must be tracked and investigated for patterns. Users behaviour is packed into events like login, logout, put items in basket, looks at an item and so on and those events are put into a CEP system to detect the users' behaviour. Shops as Amazon have a large user base and therefore need to track lots of people simultaneously. In this case a parallel CEP system can be the perfect systems by providing high parallelization degree and easy configuration.

A simple query is demonstrated in listing 7.1 which handle the purchase process from the login to several orders till the payment or the logout without buying. The query handles the basic behaviour of an online shop as log into an account and log out, put items in a basket and payment a basket. The query generates a Delivery-event, with the name and address of the person and the price of the items, whenever the pattern is detected. Name and address of the Payment-event is taken from the Login-event and the value of the order should be the

```
Assign 4000 => Login, 4001 => Order, 5000 => Anonymous_Login, 4002 => Logout, 4003 =>
    Payment, 4100 => Delivery
Define Delivery(name: string, address: string, ordersvalue: int)
From Payment()
       and each Order() within 300 from Payment
       and first Login(name=>$name, address=>$address, id=>$id) within 500 from Order
       and not Logout([int]id=$id) between Payment and Login
       and first Anonymous_Login([int]id=$id) within 600 from Order
Where name:=$name, address:=$address, ordersvalue:=SUM(Order.price()) between Login and
    Payment;
```

Listing 7.1: TESLA Query of the shop scenario.

sum of each price in the Order-events. The pattern searches for a sequence of a Login-event, several Order-events and a Payment-event. Further, between the login and the payment no Logout-event should be tracked. Instead of a Login-event also an Anonymous_Login-event can occur, which is created whenever a logged out user of the site puts an item into the basket. Sometimes user's login and don't put something in the basket, for this case a timeout of 300 seconds is given before the user is logged out, also in the case of a timeout after the first order a timeout of 500 seconds is given before the basket is cleared. Those values could be the same in a real system but are different in this test scenario to simplify the understanding. The universal graph, seen in figure 7.2 easily shows the pattern searched for with the two starting events Anonymous_Login and Login and the intermediate event Order. A sequence is finished with a Payment event or previous events are ignored at an occurrence of a Logout event.

The splitting model for P_{start} in figure 7.3 shows the acceptance of LoginEvents and Anonymous_LoginEvents as start of a window. The compare components allow only new id's in the login events to start a new window. The model for P_{close} in figure 7.1 consists of two Time stop components each responsible for one start event. The left one calculates the maximum time a window is opened when an Anonymous_LoginEvent started the window, initially a window can't last longer than 900 seconds but can be enlarged. If another Anonymous_LoginEvent with the same id would open a window, the current window is extended instead. The P_{in} splitting model shown in figure 7.4 filter all event types not used in the scenario. Further all LogoutEvent with a different id than the starting node isn't included in the window.

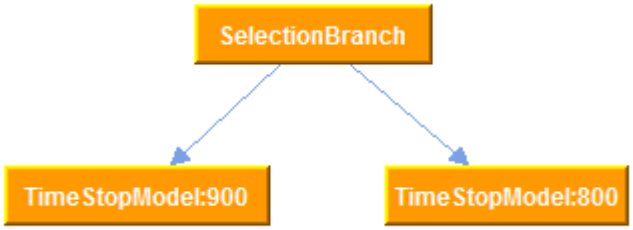


Figure 7.1: Splitting model (P_{close}) of the shop scenario.

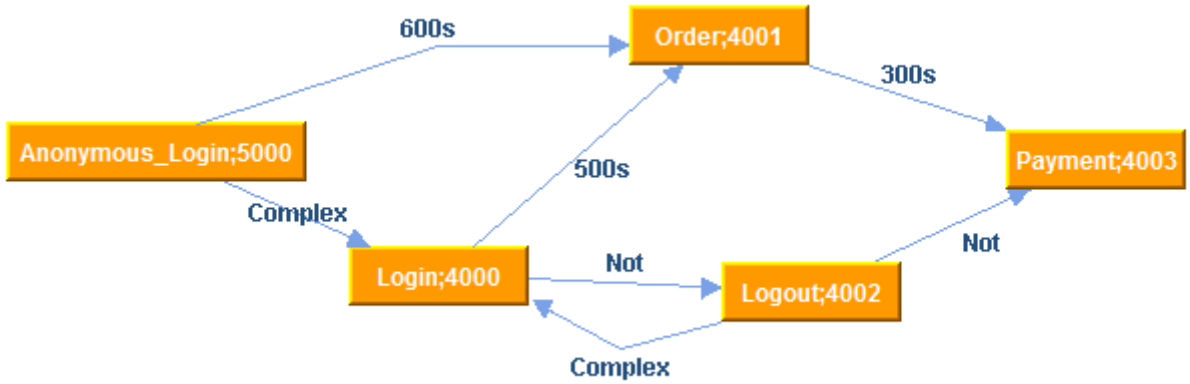


Figure 7.2: Universal Graph of the shop scenario.

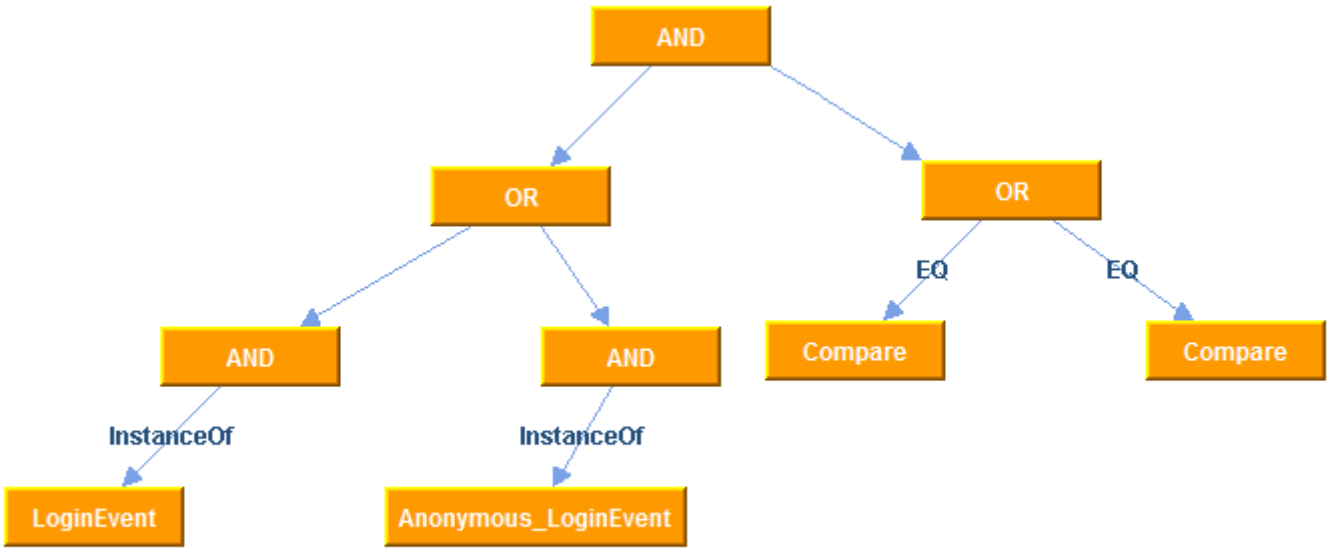


Figure 7.3: Splitting model (P_{start}) of the shop scenario.

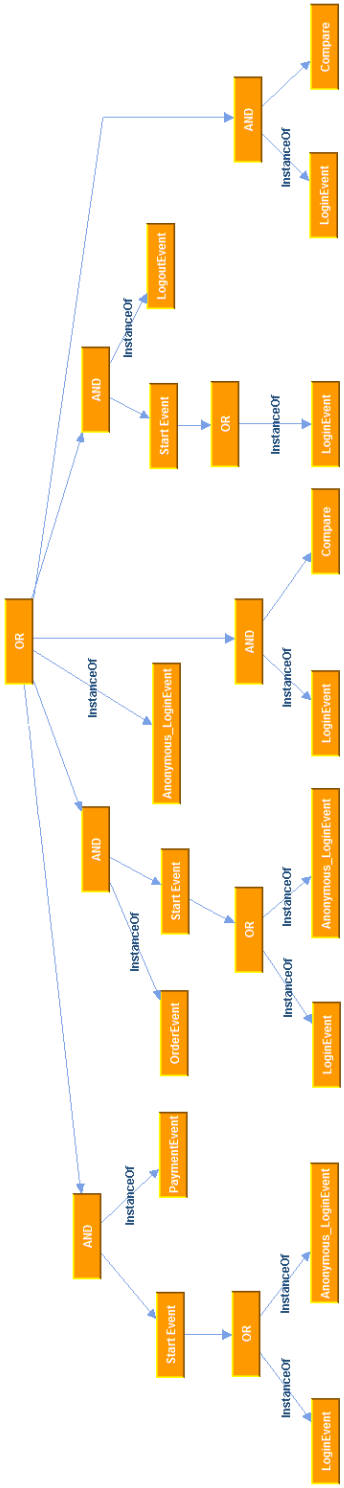


Figure 7.4: Splitting model P_{in} of the shop scenario.

```
Assign 3000 => Login, 3001 => Logout, 3002 => PlaySong, 3003 => ShowSongs, 3100 =>
    FriendPlayedSong
Define FriendPlayedSong(friendid: int, songname: string)
From ShowSongs()
       and last PlaySong(playerID=>$d, [int[]]fiendids=contain($b), songname=>$a, id=>$c)
           within 3600000 from ShowSongs
       and last Login(id=>$b) within 3600000 from PlaySong
       and not Logout([int]id=$b) between PlaySong and Login
Where friendid:=PlaySong.playerID, songname:=PlaySong.songname;
```

Listing 7.2: TESLA Query of the music scenario.

7.1.2 Music Network Scenario

Today's music industry is in a big change, in the past the most value was generated in music stores by selling cd's, but now more and more users use streaming services as Spotify, Apple Music, Amazon Prime or TIDAL. The business is in a high competition and all streaming services try to reduce the fees paid by the users. To compensate the low fees services increase their revenue by exploiting information about the users, e.g. to improve advertisements. To generate more data about their users most of the streaming service combine the music experience with a social part. For example, users get recommendations based on the music friends like and listen to or users share playlists among friend circles. A feature almost each social streaming service provide is the possibility to see what friends have recently listened to. In this scenario the method is implemented in a CEP system.

Listing 7.2 shows a TESLA language which represents the recently listened method. In a first step a user have to log into the service by sending a Login-event, since then the system should recognizes all PlaySong-events send by friends. To do this the friendids in the PlaySong-event must contain the current user id. Also when the user logs out the information is not needed anymore and no new events should be generated. Because of TESLA doesn't allow the parameterization of the terminator, another event, the ShowSongs-event, is added to the scenario to trigger the complex event generation which then contains the last song played by a friend.

In figure 7.5 the universal graph of the rule can be seen. The allowed sequences starting with Login events followed by PlaySong events and ShowSongs events, but Logout isn't allowed between Login and PlaySong events, an occurrence would reset the pattern detection. In figure 7.6 the splitting model for P_{start} is shown, through the complex edge in the universal graph, just for individual ids of a LoginEvent a new selection is generated. The splitting model of P_{in}, shown in figure 7.8, restricts the events in a selection to the event types used in the query. Also, each PlaySong event with a different id than the start event is filtered by the compare component. The closing of the selections with the P_{close} splitting model, shown in figure 7.7, assigns an time stop component to each selection restricting a selection to a maximum of 7,200,000ms or 120 minutes, if no new logins occur.

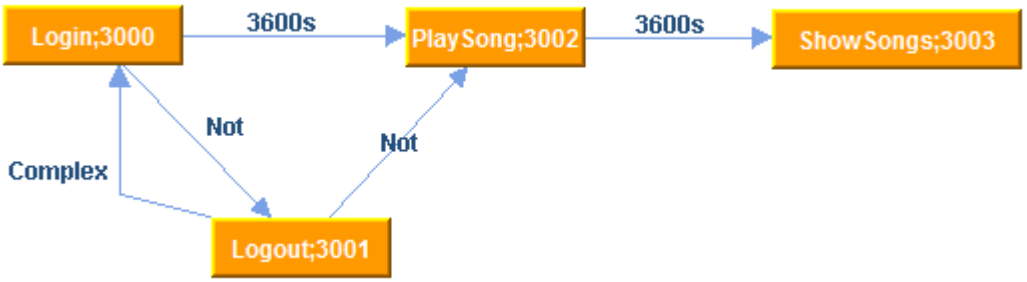


Figure 7.5: Universal Graph of the music scenario.

Figure 7.6: Splitting model (P_{start}) of the music scenario.

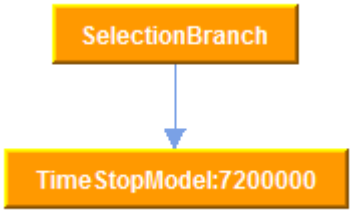


Figure 7.7: Splitting model (P_{close}) of the music scenario.

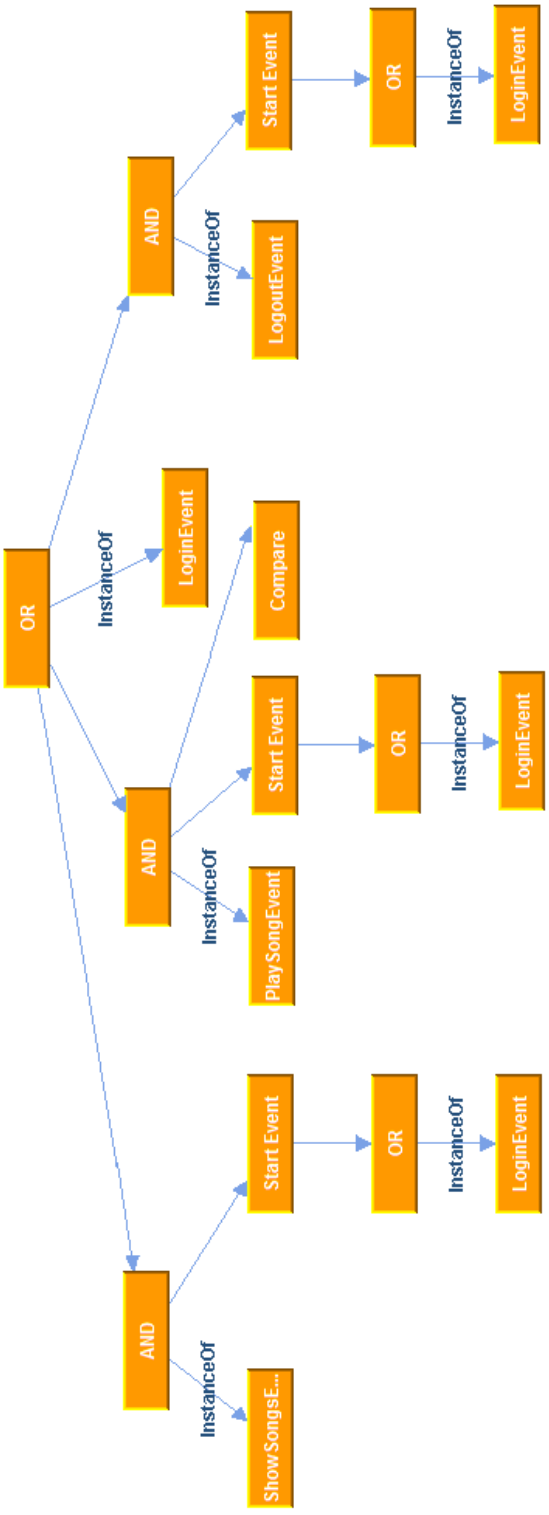


Figure 7.8: Splitting model P_{in} of the music scenario.

7.2 Functional Tests

7.2.1 Automatic Generated

Based on the previously presented Shop scenario events streams have been generated to validate the functionality of the approach. Therefore, the common testing approach of path based testing is adapted to test the universal graph of the query.

Path based testing is a white box method for test cases. Normally it analyses the control flow graph of a program and divide it into separate paths of execution. Each path is then an individual test case. In this case the method is used to verify the universal model of the shop scenario shown in figure 7.2. Instead of dividing the control flow graph we use the universal graph instead, each path leading from a start node to the target node is verified by its own. Event streams are generated to validate each path. Therefore, for each edge e in the path p with e.from as the edges source and e.target as the edges target with a time t an event stream is generated where a e.from event is followed by an e.target event with three different times $t-1$, t and $t+1$. A full example of this procedure can be seen in figure 7.9 where a path of the shopping scenario is reconstructed as several event streams. In this figure only the relation between a Login-event and an Order-event are used in further process this procedure would also generate events streams for 300ms and 301ms between an Order and a Purchase. All generated event streams then are tested with the generated splitting models.

7.2.2 Manual Tests

In addition to the automatic generated tests several manual tests have been developed targeting combinations of language elements or uncommon situations. In each of the tests the amount of opened selections and generated events after each event are confirmed against calculated values. In following some test categories are presented which increased the validation of the approach:

Several Selections In this simple tests several selections are tested concurrently. Therefore, several different ids for Login-events are used.

Selection Enlargement These tests enlarge a selection by sending another Login-event before the last one is timed out.

Reuse of Selections In difference to the Selection Enlargement the selection is getting timed out and then another selection is opened with the same id as the timed out one. In this tests there should be left over items of the last selection in the current one.

Other language elements The scenario presented in this evaluation only targets on a small part of the language elements. Therefore, several other test cases have been developed to test most of the language elements.

Each of the tests have passed without a failure and strengthened the validation of implementation and the approach.

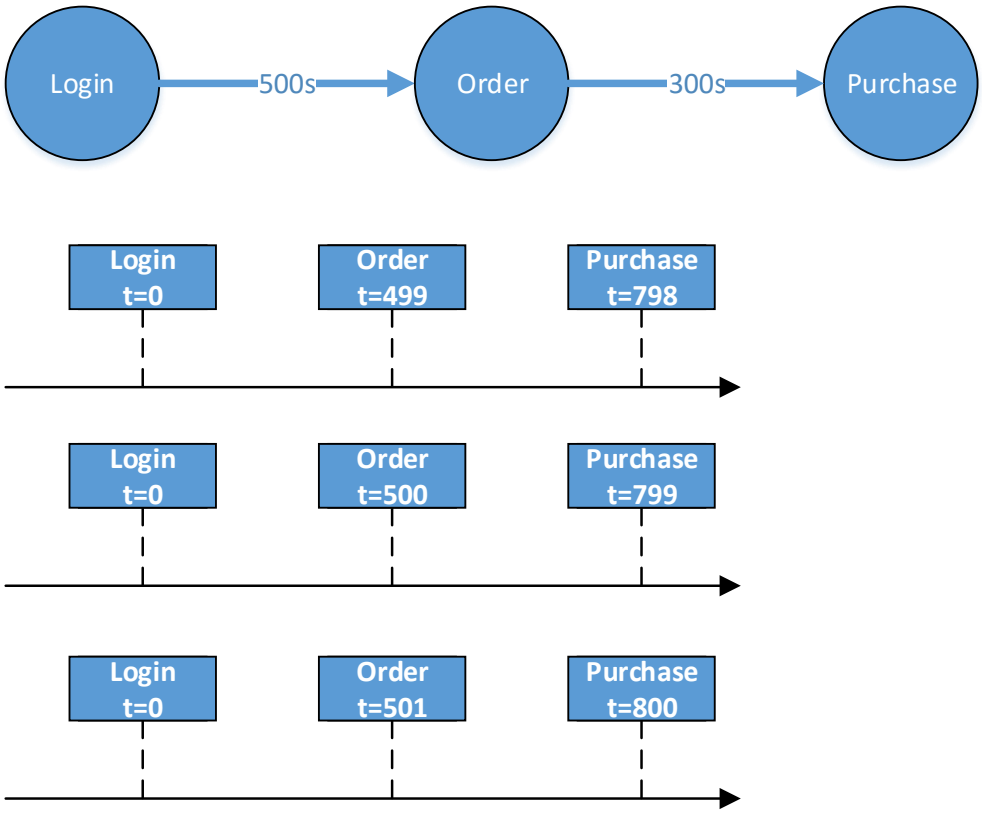


Figure 7.9: Generation of event streams out of a Path in the online shop scenario.

7.3 Performance Tests

The extension which introduced the P_{in} predicate is a performance optimization and therefore have to be validated by its improvements. Based on the previously presented music network scenario event streams have been created. Real data were taken from last.fm by Òscar Celma [Cel10] who recorded the listened music of nearly 1,000 users over several years. The data contains a timestamp, the name of the song, the artist and a userid for each participant. Another dataset from Facebook published by Stanford university [Les, ML12] was taken to simulate the relationship amongst last.fm users. The data contained information of approximately 4,000 users with 88,000 relationships. Each userID from the last.fm dataset was represented by a random user of the Facebook dataset to add the social component to the last.fm data required by the scenario. Unfortunately, the last.fm dataset doesn't contain information about the login status of each user. Events have been generated to simulate the login and logout of users. In particular each user logged in once a day at a random time before they played their first song, and logged out at a random time after their last song played at this day.

Measurements taken are the number of events sent to each operator instance and the amount of bytes sent to each operator instance. Whenever a selection is created or destroyed also the timestamp with the current amount of opened selections was recorded.

7.3.1 Results

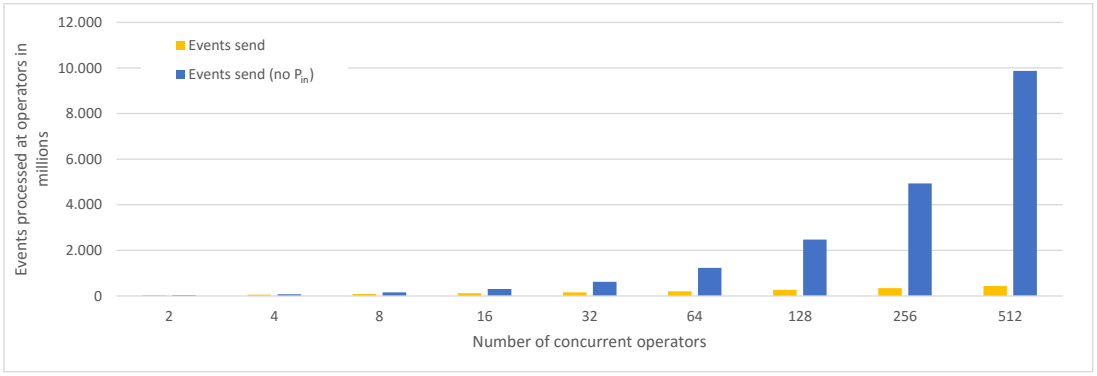


Figure 7.10: Number of processed events with and without P_{in}.

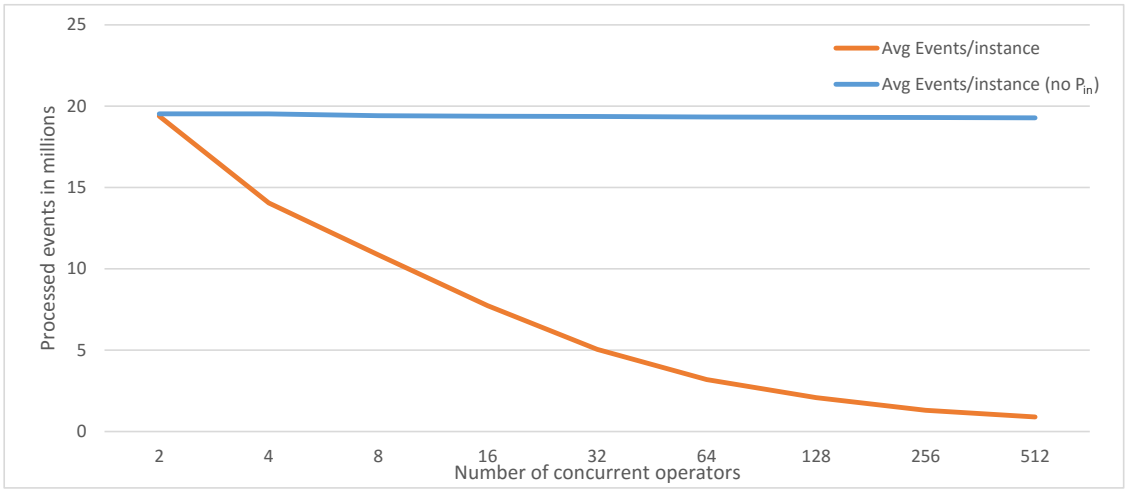


Figure 7.11: Average numbers of events processed by each operator instance with and without P_{in}.

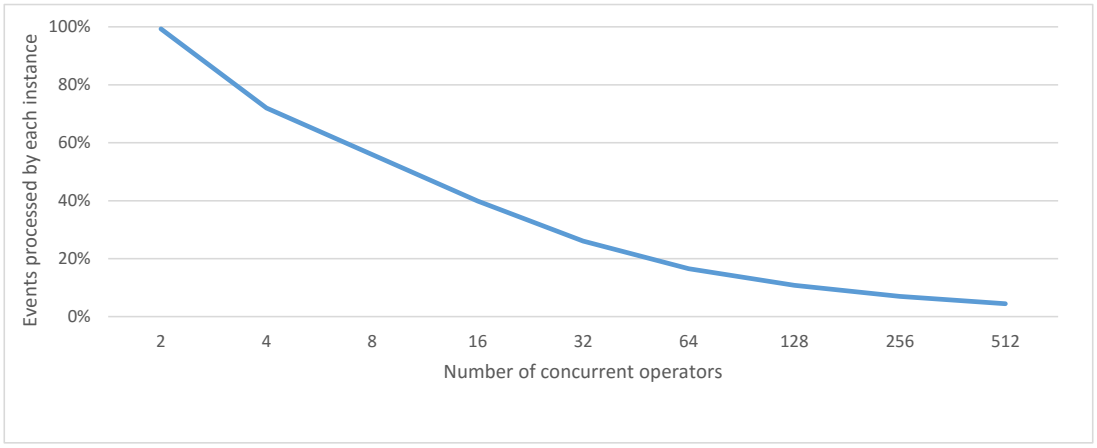


Figure 7.12: Percentage of events processed with P_{in} instead of simple window based splitting.

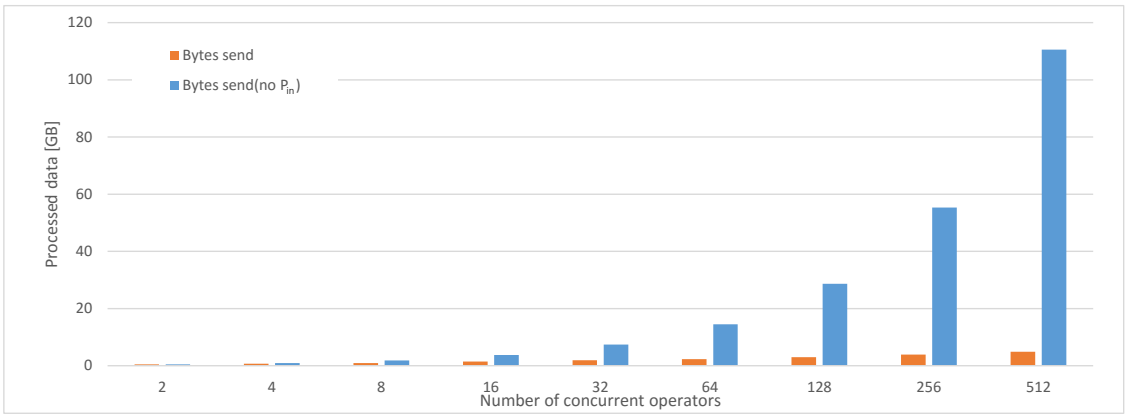


Figure 7.13: Bytes send over the network to the operator instances with and without P_{in}.

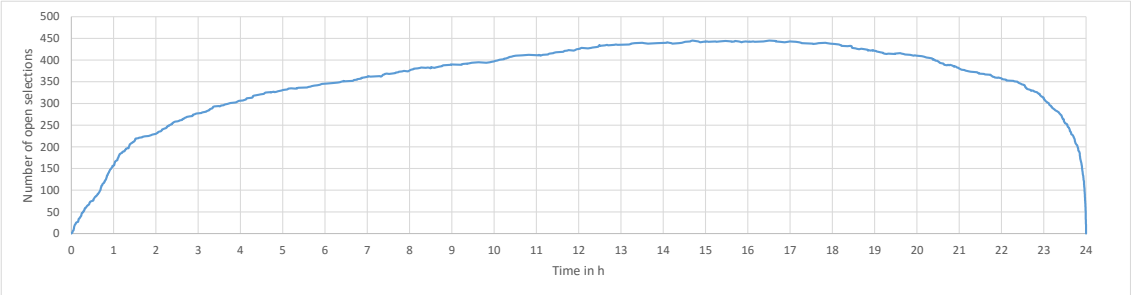


Figure 7.14: Opened selections in a representative day of the sample data.

7.4 Interpretation of Results

The functional correctness of the system has been shown with the online shop scenario. The scenario is a possible candidate for using this approach by analysing several concurrent users. The test method stated in chapter 7.2.1 based on the accredited path-based testing method proven the functionality of the approach, nevertheless it only uses one concurrent user. For several concurrent users the manual tests based on use cases tested the system, but as most generating methods, a guaranteed correctness cannot be given. At least the tests shown that for this particular scenario no false-positive or true-negative complex events have been generated. As most implementations in real systems use a combination of simple queries to construct a more complex query, the online shop scenario shows the correctness of the system when used with short queries.

The combination of window-based and key-based splitting with the predicate P_{in} was tested by the music network scenario. In figure 7.14 a representative day of the scenario is shown. The curve seen can be divided in four parts, the fast increase of users between hour 0 and hour 2, the more slowly increase of users from hour 2 to hour 15, the slow decrease from hour 17 to hour 23 and the high decrease in the afternoon between hour 23 and hour 24. Each of those categories can be seen in real life systems, e.g. use of the radio which is a strong increase at the rush hours and a big decrease afterwards between rush hours small increased and decreases can be seen. This shows the practical relevance of the test scenario with the usage of the real life data from last.fm.

In figure 7.10 and figure 7.11 the huge improvement of sent events can be seen. Each event sent without P_{in} would have been analysed by an instance operator and therefore would cost computation time on the operators. The operators could have filtered the events the same way as with P_{in} at the splitter, but the network traffic still remains as high and also each event would be filtered at several operators. With a low degree of operator parallelization, the benefits of the approach is rather low, but as seen in figure 7.12 even with 4 concurrent operators only 70% of events are sent to operators in difference to conventional window based splitting. With a high degree of parallelization as in this scenario with 512 concurrent operators, which allows each selection to be transferred to a separate operator, the only 4% of the events

Operators	Bytes processed	Bytes processed (no Pin)	Events processed	Events processed (no Pin)	events processed	Avg Events/instance	Avg Events/instance (no P_{in})
2	4.653.630.240	4.570.109.037	38.780.252	39.060.761	99,28%	19.390.126	19.530.381
4	6.725.137.995	9.373.713.120	56.202.360	78.114.276	71,95%	14.050.590	19.528.569
8	9.157.148.241	18.328.362.272	86.902.333	155.325.104	55,95%	10.862.792	19.415.638
16	14.500.538.336	37.223.546.880	123.710.416	310.196.224	39,88%	7.731.901	19.387.264
32	19.392.754.324	73.758.492.416	161.234.324	619.819.264	26,07%	5.050.762	19.369.352
64	22.857.368.576	144.816.549.696	204.083.648	1.237.748.288	16,49%	3.188.807	19.339.817
128	29.882.374.144	286.889.952.768	266.806.912	2.473.189.248	10,79%	2.084.429	19.321.791
256	38.707.979.072	553.449.336.832	345.606.956	4.941.511.936	6,99%	1.306.543	19.302.781
512	49.160.703.424	1.105.811.775.488	438.934.852	9.873.319.424	4,45%	896.234	19.283.827

Table 7.1: Results of the music network scenario

must be processed at the operators. By guaranteeing each selection an individual operator, this reduction value marks the minimum achievable in this scenario.

Also the network traffic can be reduced as seen in figure 7.13 which nearly scales the same way as with sent events. Below 8 concurrent operators the amount of network traffic is nearly the same but with a higher degree the network data reach a tremendous amount of data with over 110GB of data. In comparison to the size of the event stream with approximately 2.5GB the amount of data sent to each operator is enormous. In these cases, the predicate P_{in} is a lot more network efficient.

The requirements mentioned in chapter 5 correctness and the improvement is show by the two test scenarios. The correctness is proven for small queries as the web shop scenario and can be used in such scenarios without trouble. The usage of P_{in} should be activated when more then 4 concurrent operators are used or the expected event stream offers a good simplification. This decision must be taken by a domain expert and can be different from query to query. Also some queries can provide no improvement with P_{in} event at high degree of operator parallelization. An automated process to the decision is currently not provided but could be possible in further development.

8 Conclusion

Using CEP systems in highly frequented event streams require the use of a distributed architecture. The high effort of domain experts needed to allow the parallelism of pattern detection mechanism by guaranteeing correctness is a big problem. The resulting distribution of the event stream and the operators' implementation have to be consistent and aren't allowed to produce false-positive or true-negative complex events. Implementing those methods requires the expert to understand the CEP system and the language it is written in. Based on those requirements a method to split the event stream in data-parallel complex event processing systems have been developed and evaluated.

In this context first the currently used parallelism methods have been discussed, key-based splitting and window-based splitting. Thereby positive and negative aspects of the systems were shown. In this work the window-based approach was combined with the key-based splitting by filtering the events inside a window based on their keys. For the extensibility of the approach common query languages, Snoop and TESLA, were analysed and common language elements have been identified. In order to support the common language elements for several languages a graph based structure, the universal graph, were developed which acts as an interface between the query languages and the splitting model.

The splitting models, language independent structures with several predefined structures, can directly generated from the universal graph and divides the predicates in several smaller predefined decision. The components represent common decisions in programming languages and CEP specific methods of patterns for splitting.

The resulting splitting approach guarantees a consistent and correct splitting of event streams which is flexible enough to handle several query languages with an easy extensibility. With the new predicate p_{in} the approach granting a highly parallel system and reducing the network load in difference to current data-parallel CEP systems.

8.1 Future Work

8.1.1 Automatic Generation of Operator Instances based on Universal Graphs

The universal graph is a powerful tool to express the pattern detection queries instead of using it only for the splitting mechanism would be close-minded. So it could also be used as an universal model for generating the pattern detection itself. As the base of the universal graph is used by T-Rex [CM12] to generate the pattern detection, also the enhanced model should be capable too. Using the same method as T-Rex uses to generate it's pattern detection in

parallel CEP system however would possibly lead to multiple detections of events. Because a selection s_1 send to operator o_1 can be completely contained in a selection s_2 send to operator o_2. So each detection made in s_1 would also be made in s_2 and therefore o_1 and o_2 would generate the same complex events.

8.1.2 Automatic activation of P_{in}

As stated in chapter 7 P_{in} is very useful in some scenarios and can reduce the amount of data send to each operator tremendously. Mostly when a high degree of parallelization is required, but also in some smaller contexts the use of P_{in} can be profitable. Nevertheless in some scenarios the use of P_{in} won't improve the system at all. Currently the decision to activate or deactivate the use of P_{in} has to be done by a domain expert, which hinders the use in a fully automatic system. A method has to be developed which analyses the query if the use of P_{in} can improve the processing. But not only the query can affect the beneficing of P_{in}, also the incoming event stream is an important factor.

A ANTLR grammar of TESLA

```
grammar TESLA;

@members {
StringBuilder buf = new StringBuilder(); // can't make locals in lexer rules
}

ASSIGN    : 'Assign';
DEFINE    : 'Define';
FROM      : 'From';
WHERE     : 'Where';
CONSUMING : 'Consuming';
VALTYPE   : 'string' | 'int' | 'float' | 'bool' ;
SEL_POLICY : 'each' | 'last' | 'first' ;
AGGR_FUN  : 'AVG' | 'SUM' | 'MAX' | 'MIN' | 'COUNT' ;
OPERATOR  : '=' | '>' | '<' | '>=' | '<=' | '!=' | '&' | '|' ;
BINOP_MUL : '*' | '/';
BINOP_ADD : '+' | '-' ;
INT_VAL   : ('0' .. '9')+;
FLOAT_VAL : ('0' .. '9')+ '.' ('0' .. '9')+ ;
BOOL_VAL  : 'false' | 'true' ;
STRING_VAL :  '"'
       (   '\\'
           (   'r'    {buf.append('\r');}
           |   'n'    {buf.append('\n');}
           |   't'    {buf.append('\t');}
           |   '\\'   {buf.append('\\');}
           |   '\"'  {buf.append('"');}
           )
       |   ~('\\'|'"') {buf.append((char)_input.LA(-1));}
       )*
       '"'
       {setText(buf.toString()); buf.setLength(0); System.out.println(getText());}
    ;
EVT_NAME  : ('A' .. 'Z') (('A' .. 'Z') | ('a' .. 'z') | ('0' .. '9') | '_')*;
ATTR_NAME : ('a' .. 'z') (('A' .. 'Z') | ('a' .. 'z') | ('0' .. '9') | '_')*;
PARAM_NAME : '$' ('a' .. 'z') (('A' .. 'Z') | ('a' .. 'z') | ('0' .. '9') | '_')*;
WS        : [ \t\r\n]+ -> skip ;

static_reference : (INT_VAL | FLOAT_VAL | STRING_VAL | BOOL_VAL);
packet_reference : (EVT_NAME '.' ATTR_NAME);
param_mapping: ATTR_NAME '=>' PARAM_NAME;
param_atom : (packet_reference | PARAM_NAME | static_reference);
agg_one_reference : ('within' INT_VAL 'from' EVT_NAME);
agg_between : ('between' EVT_NAME 'and' EVT_NAME);
aggregate_atom : AGGR_FUN '(' packet_reference '(' ((attr_parameter | attr_constraint) (','
    (attr_parameter | attr_constraint))* )? ')' ')' (agg_one_reference | agg_between) ;
```

```
expr: expr BINOP_MUL expr | expr BINOP_ADD expr | '(' expr ')' | (param_atom |
    aggregate_atom);
attr_declaration : ATTR_NAME ':' VALTYPE;
staticAttr_definition: ATTR_NAME ':=' static_reference;
attr_definition: ATTR_NAME ':=' expr;
attr_constraint: ATTR_NAME OPERATOR static_reference;
attr_parameter: '[' VALTYPE ']' ATTR_NAME OPERATOR expr;
predicate : EVT_NAME '(' ((param_mapping | attr_constraint | attr_parameter) (','
    (param_mapping | attr_constraint | attr_parameter))*)? ')' event_alias? ;
event_alias : 'as' EVT_NAME;
terminator : predicate;
positive_predicate : 'and' SEL_POLICY predicate 'within' INT_VAL 'from' EVT_NAME;
neg_one_reference: ('within' INT_VAL 'from' EVT_NAME);
neg_between: ('between' EVT_NAME 'and' EVT_NAME);
negative_predicate : 'and' 'not' predicate (neg_one_reference | neg_between);
pattern_predicate : positive_predicate | negative_predicate;
event_declaration : INT_VAL '=>' EVT_NAME;
event_declarations : event_declaration (',' event_declaration)*;
ce_definition : EVT_NAME '(' (attr_declaration (',' attr_declaration)* )? ')';
pattern : terminator (pattern_predicate)*;
definitions : (staticAttr_definition | attr_definition) (',' (staticAttr_definition |
    attr_definition))*;
consuming : EVT_NAME (',' EVT_NAME)*;
ending_rule : ';';
trex_rule : ASSIGN event_declarations DEFINE ce_definition FROM pattern (WHERE definitions)?
    (CONSUMING consuming)? ending_rule;
```

Listing A.1: TESLA grammar in ANTLR.

B ANTLR grammar of Snoop

```
grammar Snoop;

@members {
StringBuilder buf = new StringBuilder(); // can't make locals in lexer rules
}
snooprule : e1;
e1 : e1 or_event | e2;
e2 : e2 sequence_event | e3;
e3 : all_event | e5; // Composite All event
e5 : aperiodic_event // Aperiodic event
       | periodic_event // Periodic event
       | timestring
       | ex_event
       | relative
       | labeled
       | bracketed;
ex_event : EVT_NAME;
bracketed : '(' e1 ')';

relative             : '(' e1 ')' '+' timestring_intervall;
labeled              : LABEL ':' '(' e1 ')' ;
or_event             : '|' e3;
sequence_event  : ';' e3;
all_event            : 'All(' all_elements ')';
all_elements  : e5 (',' e5)*;
aperiodic_event : 'A' aperiodic_star '('e1 ',' e1 ',' e1')';
aperiodic_star : '*'*;
periodic_event  : 'P(' e1 ',' timestring_intervall ',' e1 ')';

timestring                          : '<' TIME '>';
timestring_intervall  : '[' TIME ']';

// TODO todo-list start
LABEL  : ('A' .. 'Z') (('A' .. 'Z') | ('0' .. '9') | '_')*;
EVT_NAME  : ('A' .. 'Z') (('A' .. 'Z') | ('a' .. 'z') | ('0' .. '9') | '_')*;

TIME   : '(' HOURS ':' MINUTES ':' SECONDS ')' MONTH '/' DAY '/' YEAR;
YEAR   : ('0'..'9')('0'..'9')('0'..'9')('0'..'9') | '-';
MONTH  : ('0'..'1')('0'..'9') | '-';
DAY           : ('0'..'3')('0'..'9') | '-';
HOURS  : ('0'..'2')('0'..'9') | '-';
MINUTES : ('0'..'5')('0'..'9') | '-';
SECONDS : ('0'..'5')('0'..'9') | '-';

WS        : [ \t\r\n]+ -> skip ;
```

Listing B.1: Snoop grammar in ANTLR.

Bibliography

[ABW04] A. Arasu, S. Babu, J. Widom. *CQL: A Language for Continuous Queries over Streams and Relations*, pp. 1–19. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. (Cited on pages 21 and 25)

[BDG+07] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, W. White. Cayuga: A High-performance Event Processing Engine. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pp. 1100–1102. ACM, New York, NY, USA, 2007. (Cited on pages 11 and 25)

[BDWT13] C. Balkesen, N. Dindar, M. Wetter, N. Tatbul. RIP: Run-based Intra-query Parallelism for Scalable Complex Event Processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pp. 3–14. ACM, New York, NY, USA, 2013. (Cited on page 15)

[BEH+10] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, D. Warneke. Nephele/PACTs: A Programming Model and Execution Framework for Web-scale Analytical Processing. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pp. 119–130. ACM, New York, NY, USA, 2010. (Cited on page 17)

[BGHJ] L. Brenna, J. Gehrke, M. Hong, D. Johansen. Distributed event stream processing with non-deterministic finite automata. In A. Gokhale, D. C. Schmidt, editors, *the Third ACM International Conference*, p. 1. (Cited on page 16)

[BKR08] J. A. Briones, B. Koldehofe, K. Rothermel. SPINE: Publish/subscribe for Wireless Mesh Networks through self-managed intersecting paths. In *International Conference on Innovative Internet Community Systems*. 2008. (Cited on page 14)

[Cel10] Òscar Celma. Last.fm Dataset - 1K users. http://www.dtic.upf.edu/~ocelma/MusicRecommendationDataset/lastfm-1K.html, 2010. [Online; accessed 27-September-2016]. (Cited on page 53)

[CM94] S. Chakravarthy, D. Mishra. Snoop: An expressive event specification language for active databases. *Data & Knowledge Engineering*, 14(1):1–26, 1994. (Cited on page 21)

[CM10] G. Cugola, A. Margara. TESLA: A Formally Defined Event Specification Language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, pp. 50–61. ACM, New York, NY, USA, 2010. (Cited on page 31)

[CM12] G. Cugola, A. Margara. Complex event processing with T-REX. *Journal of Systems and Software*, 85(8):1709–1728, 2012. (Cited on pages 12, 25 and 59)

[DBB+88] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny, R. Jauhari. The HiPAC Project: Combining Active Databases and Timing Constraints. *SIGMOD Rec.*, 17(1):51–70, 1988. (Cited on page 11)

[EFGK03] P. T. Eugster, P. A. Felber, R. Guerraoui, A.-M. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003. (Cited on page 14)

[EM99] A. Eisenberg, J. Melton. SQL: 1999, formerly known as SQL3. *ACM Sigmod record*, 28(1):131–138, 1999. (Cited on page 21)

[FVWZ02] I. Foster, J. Vockler, M. Wilde, Y. Zhao. Chimera: a virtual data system for representing, querying, and automating data derivation. In *Scientific and Statistical Database Management, 2002. Proceedings. 14th International Conference on*, pp. 37–46. 2002. (Cited on page 11)

[GD92] S. Gatziu, K. R. Dittrich. SAMOS: An active object-oriented database system. *IEEE Data Eng. Bull.*, 15(1-4):23–26, 1992. (Cited on page 21)

[GD94] S. Gatziu, K. R. Dittrich. *Events in an Active Object-Oriented Database System*, pp. 23–39. Springer London, London, 1994. (Cited on page 11)

[GMM90] C. Ghezzi, D. Mandrioli, A. Morzenti. TRIO: A Logic Language for Executable Specifications of Real-time Systems. *J. Syst. Softw.*, 12(2):107–123, 1990. (Cited on pages 21 and 25)

[HSD10] G. Hermosillo, L. Seinturier, L. Duchien. Using Complex Event Processing for Dynamic Business Process Adaptation. In *Services Computing (SCC), 2010 IEEE International Conference on*, pp. 466–473. 2010. (Cited on page 11)

[jgr] JGraphT - A free Java Graph Library. http://jgrapht.org/. [Online; accessed 27-September-2016]. (Cited on page 41)

[Lea09] N. Leavitt. Complex-Event Processing Poised for Growth. *Computer*, 42(4):17–20, 2009. (Cited on page 11)

[Les] J. Leskovec. Social circles from Facebook. https://snap.stanford.edu/data/. [Online; accessed 27-September-2016]. (Cited on page 53)

[LJ05] G. Li, H.-A. Jacobsen. *Composite Subscriptions in Content-Based Publish/Subscribe Systems*, pp. 249–269. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. (Cited on page 25)

[MBF14] A. Martin, A. Brito, C. Fetzer. Scalable and Elastic Realtime Click Stream Analysis Using StreamMine3G. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pp. 198–205. ACM, New York, NY, USA, 2014. (Cited on page 17)

[Mis91] D. Mishra. *SNOOP: An event specification language for active database systems.* 1991. (Cited on page 32)

[MKR14] R. Mayer, B. Koldehofe, K. Rothermel. Meeting Predictable Buffer Limits in the Parallel Execution of Event Processing Operators. In *Proceedings of the 2014 IEEE International Conference on Big Data, BigData '14*, pp. 402–411. IEEE, 2014. (Cited on page 18)

[MKR15] R. Mayer, B. Koldehofe, K. Rothermel. Predictable Low-Latency Event Detection with Parallel Complex Event Processing. *IEEE Internet of Things Journal*, pp. 1–13, 2015. (Cited on page 18)

[ML12] J. J. McAuley, J. Leskovec. Learning to Discover Social Circles in Ego Networks. In *NIPS*, volume 2012, pp. 548–56. 2012. (Cited on page 53)

[MMG92] A. Morzenti, D. Mandrioli, C. Ghezzi. A Model Parametric Real-time Logic. *ACM Trans. Program. Lang. Syst.*, 14(4):521–573, 1992. (Cited on pages 21 and 25)

[PQ95] T. Parr, R. Quong. ANTLR: A Predicated. *Software—Practice and Experience*, 25(7):789–810, 1995. (Cited on page 41)

[SGLN+11] S. Suhothayan, K. Gajasinghe, I. Loku Narangoda, S. Chaturanga, S. Perera, V. Nanayakkara. Siddhi: A Second Look at Complex Event Processing Architectures. In *Proceedings of the 2011 ACM Workshop on Gateway Computing Environments*, GCE '11, pp. 43–50. ACM, New York, NY, USA, 2011. (Cited on page 15)

[SHGW12] S. Schneider, M. Hirzel, B. Gedik, K.-L. Wu. Auto-parallelizing Stateful Distributed Streaming Applications. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pp. 53–64. ACM, New York, NY, USA, 2012. (Cited on page 17)

[SMMP09] N. P. Schultz-Møller, M. Migliavacca, P. Pietzuch. Distributed Complex Event Processing with Query Rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pp. 4:1–4:12. ACM, New York, NY, USA, 2009. (Cited on page 13)

[TKBR14] M. A. Tariq, B. Koldehofe, S. Bhowmik, K. Rothermel. PLEROMA: A SDN-based High Performance Publish/Subscribe Middleware. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pp. 217–228. ACM, New York, NY, USA, 2014. (Cited on page 14)

[TRP11] K. Teymourian, M. Rohde, A. Paschke. Processing of Complex Stock Market Events Using Background Knowledge. *RuleML2011@ BRF Challenge*, p. 95, 2011. (Cited on page 11)

[WDR06] E. Wu, Y. Diao, S. Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pp. 407–418. ACM, 2006. (Cited on page 11)

[WL05] F. Wang, P. Liu. Temporal management of RFID data. In *Proceedings of the 31st international conference on Very large data bases*, pp. 1128–1139. VLDB Endowment, 2005. (Cited on page 11)

All links were last followed on September 27, 2016.

Decleration

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

---

place, date, signature