Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit Nr. 0202-0004

Consistent Splitting of Event Streams in Parallel Complex Event Processing

Nagarjuna Siddam

Course of Study:	INFOTECH
Examiner:	Prof. Dr. Kurt Rothermel
Supervisor:	Ruben Mayer, Dipl.-Inf.
Commenced:	2015-04-20
Completed:	2015-10-20
CR-Classification:	C.2.4,C.1.4,F.1.2

Abstract

Complex Event Processing (CEP) combines streams of simple events from multiple sources and infer high-level knowledge by correlating the simple events. The correlation of events is done according to the requirement of the application. This requirement is specified to the CEP system in the form of *rules*. State of the art implementations of CEP system deploy a distributed network of operators to execute a *rule*. For low latency information retrieval under high event rate of input streams, scalability of operators is required. This is done by the parallelization of operator functionality. The existing PACE system provides a framework for operator parallelization. Parameter context in a *rule* specify which events in the input stream can be used for correlation and are essential for application of CEP systems to many real world situations. In this thesis operator parallelization techniques, for adopting the existing PACE system to execute rules with various parameter contexts, are proposed. In cases where the existing PACE system cannot be adopted, other approaches for operator parallelization are proposed. Finally, the proposed techniques are analysed by implementing the techniques and evaluating their performance.

Contents

List of Figures

List of Algorithms

1 Introduction

In recent years there has been a huge increase in the amount of data produced. This data is produced continuously by geographically distributed software tools. Most of this raw data contains information that is of low-level. Many modern information systems require real-time processing of this continuously flowing data to identify situations of interest occurring in a system or in the real world:

- In a Sensor network deployed to monitor in an industrial environment [5], an individual sensor generates data containing information restricted by the physical location of the sensor and the type of the sensor. This low-level data from the individual sensors has to be processed to derive high-level knowledge of the normal or abnormal situation in the environment that the monitoring application is interested in.
- In Smart grid management [16], measurements from various power-generating, power-distributing, and power-consuming nodes have to be processed to detect abnormal situations that may lead to the failure of the gird system.
- In algorithmic trading [9], continuous analysis of stock related data is required to detect trends in the stock market based on which the required actions are determined.
- In business activity monitoring [10], high volumes of technical data related to activities of various business processes has to be processed to provide business data to higher management.
- In Radio-Frequency Identification (RFID) systems [12], RFID readers collect the data from RFID tags and transmit it to a back-end server. The back-end server requires processing of received data from various locations for inventory management and tracking.

In all the above systems there exists an information gap between the sources that generate the low-level data and the applications that are interested in high-level information. Complex Event Processing (CEP) systems bridge this information gap by correlating the seemingly unintelligible low-level input data to derive the knowledge of high-level situations that are of interest.

The usage of the CEP systems is growing rapidly. A number of open source and proprietary implementations of CEP are available in the market [15]. Every company uses a software tool that implements CEP exclusively or under the covers.

The information retrieval by a CEP system must be in real-time or near real-time, so as to take advantage of the identified opportunities or to avoid the occurrence of forecasted threats. Overloading of the CEP system results in buffering of input data. Data waiting in the input buffers increases the latency of output generation. To maintain the

real-time nature of information retrieval by the CEP system, parallel processing of input data is employed. This requires the splitting of input data stream into multiple streams which can be processed independently in parallel. A framework for such a parallel processing, called PACE system, already exists [14].

Parameter contexts in a CEP system can be seen, informally, as restrictions on the usage of the received input data in generating output. Parameter contexts are used to tailor the output data of a CEP system in accordance with the requirements of an application. Therefore parameter contexts are important in realizing the usage of CEP systems in many application scenarios. Though the existing framework for parallelization, the PACE system, is very expressive it does not have any accommodation for the processing of input data with parameter contexts.

In this thesis I have examined the parallel processing in a CEP system with parameter context. I have studied different implementations of CEP systems and identified a set of commonly used parameter contexts. I have considered each parameter context individually and examined if the current framework can be extended for parallel processing with that parameter context. In the case where such an extension cannot be done, I have looked into the reasons for that and proposed a new technique for parallel processing of incoming event streams.

Thesis Organization

The remaining part of the thesis is organised as follows:

Chapter 2 provides the background for understanding the Complex event processing. It also describes different approaches for implementation of a CEP system and how the events are processed in a CEP system. Also presented in this chapter are the concepts related to parameter contexts and their usage in rule specification in different event specification languages.

In chapter 3 the necessity for operator parallelizatin and conditions to be met for consistent parallelization are described. The working of the PACE system is discussed in detail in this chapter. The system model considered for the CEP system is also presented in this chapter.

Chapter 4 provides the description of the problem statement that is dealt in this thesis.

In chapter 5, the methods for adopting the PACE system for operator parallelization with parameter context are presented. The feasibility for such adoption and the consistency of the resulting parallelized operator are discussed.

In chapter 6, a new method, shared memory approach, for operator parallelization is presented. The shared memory method is presented with respect to the parameter

context for which adoption of the PACE system was not feasible.

Chapter 7 provides the details of the implementation and experimental evaluation of the proposed techniques. The results from the experimental evaluation are presented and discussed.

The thesis is concluded with chapter 8, providing a brief summary of the work done and outline for possible future work.

2 Background

2.1 Introduction to CEP

2.1.1 Basic Definitions

Before looking into details of what a Complex Event Processing does, relevant basic definitions that are required to understand the CEP model are given in this section.

Event and Event Stream. The dictionary definition of an event is given as "a happening in the physical world". In a CEP system an event is an abstraction that contains the data related to a happening in the physical world. Some events for example are:

- The output of a temperature sensor that contains the temperature of a location
- A financial ticker that gives change in stock value of a particular stock.

An event stream is an unbounded collection of events. Events in an event stream are ordered based on a specific criteria. Usually the ordering criterion is time of occurrence of the event. Multiple event streams can be merged to form a single event stream.

An event is represented by a tuple of attribute-value pairs. These attribute-value pairs are called parameters of the event and contain data that gives information about event occurrence. The most common event attributes are event type, time stamp, and sequence number.

An event type represents a class of events. Every event is an instance of an event type. An event type describes the essential features that identify an event of that type and specifies the parameters that sufficiently describe the specific features of the event.

Time stamp gives the time of event occurrence or detection. The value of time stamp attribute can be a point of time or a time interval depending on the semantics of the CEP system.

Sequence numbers are assigned to an event based on their time stamp. The sequence number of an event defines the position of an event in an event stream.

An event may have many other attributes that are specific to an event type, encapsulating all the necessary information required for processing in the CEP system.

Complex Event. In his book [13], The Power of Events, David Luckham defines a complex event as an event whose occurrence depends on the occurrence of other events. A complex event represents a set of other events. These events on which the occurrence of a complex event depends are called component events. For example if the temperature sensor of a location reports big raise in temperature and the smoke sensor of the same

location reports presence of smoke then a CEP system detects the fire event at that location. Here fire is the complex event detected and the component events are raise in temperature and smoke events.

Operator. An operator describes the relation between the component event types of a complex event. Some of the most common operators are disjunction, conjunction, sequence, window, not, etc. These operators can be combined to form complex operators.

Event Pattern. The event pattern describes the procedure in which the events of incoming event types should occur for a complex event to occur. For example, consider an event type E_{12} that occurs whenever an event of type E_1 occurs followed by an event of type E_2. E_{12} can be described by a event pattern with a sequence operator and the event types E_1 and E_2 as component event types: $E_{12} = (E_1; E_2)$.

2.1.2 Complex Event Processing

Complex event processing(CEP) is defined as a technology for analysing event streams from multiple event sources to extract high-level information in the form of detected complex events that the applications are interested in.

The complex events in today's information systems are made up of many simple events, which can be observed via sensors or messages within the system. These observed events are called primitive events. The nodes in the network that produce these events are called event sources. Gathering high-level insights about the status of the system or situations occurring in the system requires analysis of high volumes of primitive events. In the analysis, the primitive events are correlated to detect event patterns that correspond to the situations the applications are interested in. These detections are forwarded to the applications as complex events. The applications that receive the detected complex events are called sinks. The rules for processing of primitive events are submitted to the CEP system by the sinks. A rule defines an event pattern , the output complex event type, and how the data of the output event has to be calculated.

Therefore a CEP system receives event streams from event sources, analyses the events for patterns by correlating the received events, and forwards the detected complex events to the sinks. A CEP system acts as a middleware providing services to the application layer hiding from them the event production layer where distributed event sources generate a vast amount low-level events. The position of CEP with respect to related entities is depicted in figure 2.1.

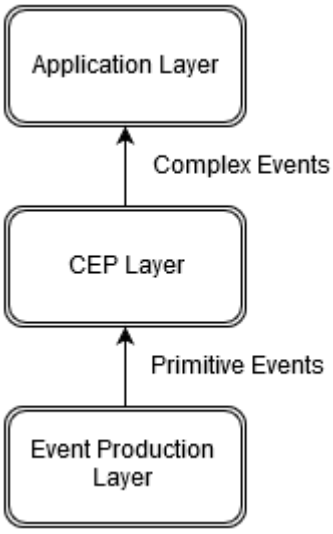

Figure 2.1: Complex Event Processing System

2.2 CEP System

2.2.1 Centralized CEP system

This is the simplest form of a CEP system. In this model the CEP system runs as a centralized process on a single node in the network. The event streams from all the event sources are directed towards this centralized process. All the rules for the correlation of primitive events are programmed in this process. The processing of all the produced events is done in this process and the sinks connect to this process to receive the detected complex events. A centralized CEP model with connections to sources and sinks is depicted in figure 2.2.

With a centralized CEP system, the network architecture is simple and therefore is easy to implement and maintain. But the processing of all the rules has to be done on a single node; such a system lacks scalability and quickly becomes a throughput bottleneck in the network.

2.2.2 Distributed CEP system

In distributed model of the CEP system, the event processing is done by a set of processes, that are running in a distributed manner on cluster of connected hosts. A rule submitted to the CEP system is divided into different operators and the functionality of an operator is executed by a process. A distributed CEP model with connections to sources and sinks is depicted in figure 2.3.

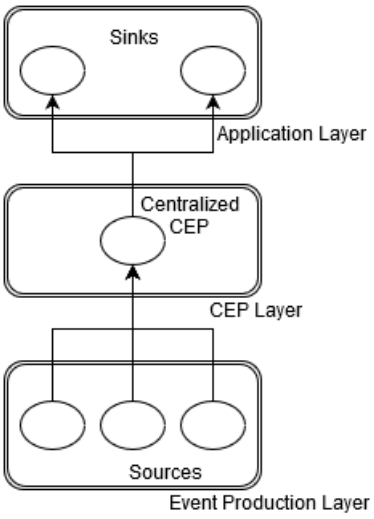

Figure 2.2: Centralized CEP System

Instead of correlating the primitive events to obtain desired complex events, in a distributed system the processing of events is done step by step. An operator receiving primitive events correlates them to intermediate complex events and forwards them to other operators which in turn correlates the incoming intermediate complex events, and primitive events in some cases, to desired complex events.

The disadvantage of the distributed CEP system is its high complexity which increases the cost of implementation and maintenance. An advantage of distributed CEP system is by splitting the event processing load among different operators, the distributed model allows for the increase in scalability of the overall CEP system. In addition, with this model of CEP system the network usage can be minimised by placing an operator as close as possible to the sources of its incoming streams.

Therefore in a large system with many high-frequency sources distributed across large spatial distances a distributed CEP system is more suitable than a centralized CEP system. For this reason hereafter this thesis exclusively refers to distributed CEP system. However it is to be noted that the methods proposed in this thesis are equally applicable to a centralized CEP system.

2.3 Event Processing

Operator Graph. A distributed CEP system can be modelled by a directed graph $G(\Omega \cup S \cup C, L)$, called an operator graph [11]. In the operator graph the set of sources S, the set of operators Ω , and the set of sinks C are interconnected by the set of event streams, $L \subset (S \cup \Omega) \times (\Omega \cup C)$, as edges.

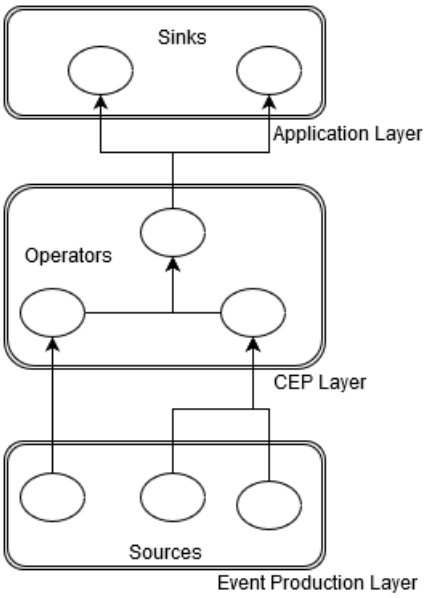

Figure 2.3: Distributed CEP System

Merging of input streams. An event stream $(p, d) \in L$ of the operator graph is directed from a producer to a destination and ensures that events are delivered to the destination in the order they are produced. (p, d) is called an output stream of p and an input stream of d. The events in an input event stream are delivered at the destination in the order of their sequence number. Events from different input event streams have a well-defined global order at the destination, that is independent of their physical time of delivery at the destination. This ordering depends on the time stamp of the event, source from which event is received, and the sequence number of the event in the event stream it is received.

An operator $\omega \in \Omega$ performs processing with respect to the set of all incoming streams $(in, \omega) \in L$. The events from all the incoming streams of the operator ω are combined into a single stream of events denoted by I_ω. The order of the events in I_ω is determined by the global ordering of the events.

Event Correlation. An operator in a distributed CEP system performs correlation on the incoming event streams to produce outgoing events. During its execution, the operator ω performs a sequence of correlation steps on I_ω. In each correlation step, the operator determines a selection σ which is a finite subset of events in I_ω. A correlation function $f_\omega : \sigma \rightarrow (e_1, ..., e_m)$ specifies a mapping from a selection to a finite set of events produced by the operator. The operator writes the produced events to its output stream O_ω in the order of their detection.

Production Set. Each output event has a production set associated with it. The production set of an output event is a subset of the selection σ in which the output event is detected. The production set of an output event contains events of component event types of that output event. For any two output events of an operator ω their production sets are not equal and differ by atleast one event.

Parameter Computation. The parameters of an output event are calculated from the parameters of the events in its production set. Apart from specifying the type of the output event,the rule also specifies the procedure for calculating the parameters of the output event. In a CEP system where the time stamp is given by a point of time, the time stamp of the output event is the time stamp of the last event in its production set. In a CEP system where the time stamp is given by time interval, the time stamp of the output event is from the start time of the first event production set to end time of the last event in production set.

2.4 Parameter Context

Together with the event pattern to be detected and the procedure for calculating the parameters of the output events, the event rule also specifies a selection policy and a consumption policy for event detection.

Selection policy. In the presence of situation in which a selection σ of an operator ω can produce more than one output event, the selection policy specifies if a single or multiple output events have to be produced and which events from the selection are included in the production set of the output events. There are three types of selection policies that a rule can specify [8] : *each* selection policy, *latest* selection policy, and *earliest* selection policy.

A rule with *each* selection policy allows the production of multiple output events from one selection. With this selections policy, all the possible complex events are detected.

Conversely, a rule with *latest* selection policy and *earliest* selection policy allows the production of at most one event from one selection. The *latest* and *earliest* selection policies differ in the way the production set of the output event is determined. With *latest* selection policy the events included in the production set of the output event are the latest instances of the component event types. Conversely, with *earliest* selection policy the events included in the production set of the output event are the earliest instances of the component event types.

For example, consider the complex event $E_{12} = (E_1; E_2)$. Let e_i^j be the j^{th} event of type E_i. Let the order of the incoming events at the input of the operator be e_1^1, e_1^2, e_2^1. The selection σ will be determined as (e_1^1, e_1^2, e_2^1). The number of output events and their production sets for the above selection policies is as follows:

- In the case of *each* selection policy, there will be two output events with production sets (e_1^1, e_2^1) and (e_1^2, e_2^1).
- In the case of *latest* selection policy, there will be one output event with production set (e_1^2, e_2^1).
- In the case of *earliest* selection policy, there will be one output event with production set (e_1^1, e_2^1).

Consumption policy. The consumption policy of a rule specifies whether or not an event of a selection σ included in the production set of the output event of σ can be part of future selections. There are two types of consumption policies that a rule can specify [8]: *selected* consumption policy and *zero* consumption policy.

In *selected* consumption policy, the events that are included in the production set of an output event of a selection are said to be consumed and all the consumed events will not be part of future selections. This means that an event can be consumed in at most one selection.

In *zero* consumption policy the events included in the production set of an output event are not invalidated and can be part of future selections.

For example, consider the previously discussed complex event $E_{12} = (E_1; E_2)$. Let the order of the incoming events at the input of the operator be $e_1^1, e_2^1, e_1^2, e_2^2$. The first selection σ_1 will be determined as (e_1^1, e_2^1). The selection σ_1 will be mapped to single output event with production set (e_1^1, e_2^1). The events in the second selection σ_2 depends on the consumption policy of the rule and is given as follows:

- In the case of *zero* consumption policy, there will be no consumption of events and σ_2 will be determined as $(e_1^1, e_2^1, e_1^2, e_2^2)$.
- In the case of *selected* consumption policy, the events e_1^1, e_2^1 will be consumed and σ_2 will be determined as (e_1^2, e_2^2).

As the events are never invalidated in the *zero* consumption policy, the selection's size will continuously grow as the incoming events arrive. Therefore, with *zero* consumption policy there is a necessity to limit the portion of the incoming event streams that can be considered for inclusion in a selection. Such a limitation can be achieved by a time-based or a tuple-based window operation.

Parameter Context. The parameter context of a rule is the combination of its selection policy and consumption policy. As there are three selection policies and two consumption policies, these can be combined to form six parameter contexts.

For example, consider the previously discussed complex event $E_{12} = (E_1; E_2)$. Let the order of the incoming events at the input of the operator be $e_1^1, e_1^2, e_2^1, e_2^2$. The number

of output events and their parameter contexts for the six parameter contexts is given in the table 1

		Consumption Policy	
		Zero	*Selected*
Selection Policy	*Each*	(e_1^1, e_2^1), (e_1^2, e_2^1), (e_1^1, e_2^2), (e_1^2, e_2^2)	(e_1^1, e_2^1), (e_1^2, e_2^1)
	Latest	(e_1^2, e_2^1), (e_1^2, e_2^2)	(e_1^2, e_2^1)
	Earliest	(e_1^1, e_2^1), (e_1^1, e_2^2)	(e_1^1, e_2^1), (e_1^2, e_2^2)

Table 1: Parameter Contexts

A number of other parameter contexts can be defined other than the six mentioned above. For example a consumption policy can be defined where a non-empty proper subset of the events in production set are consumed. For the purpose of limiting the discussion, the six parameter context from table 1 are considered in this thesis.

2.5 Event Specification Languages

As discussed previously, an event rule describes how a CEP system should process the incoming events. An Event specification language (ESL) describes how to specify an event rule to a CEP system. There exists a number of CEP systems, each implemented for a specific application. Many CEP systems have their own event specification languages. Every event specification language describes a set of basic operators and how these operators can be combined to form complex operators. An event specification language also describes a set of parameter contexts that are supported.

In this section we shall discuss about a few event specification languages and their features in specifying parameter contexts in a rule.

Snoop. Snoop [6] is a well known event specification language. Snoop is originally designed for active databases, well before the advent of CEP systems. As Snoop is independent of semantics of the database systems, it can be used for defining rules in a CEP system.

In Snoop a parameter context can be specified for each rule. The specified parameter context determines the selection policy and consumption policy of the rule. The selection policy and consumption policy can not be specified explicitly and have to be specified through parameter context. Snoop supports four parameter contexts: recent, chronicle, continuous, and cumulative. The selection policy and consumption policy these parameter contexts specify is given in the table 2.

The selection policy of cumulative parameter context allows of at most one event from a selection and multiple occurrences of a component event type are accumulated as a single event. This parameter context specifies the *selected* consumption policy.

		Consumption Policy	
		Zero	*Selected*
Selection Policy	*Each*	Not supported	Continuous
	Latest	Recent	Not supported
	Earliest	Not supported	Chronicle

Table 2: Parameter Contexts

In snoop a rule is specified in the form of Event-Condition-Action. The Event part of the rule specifies the event pattern that needs to be detected. The Condition part specifies a boolean condition that needs to be satisfied for executing the Action part. An example rule defined in Snoop is given in figure 2.5. The scenario that the example rule defines is: if an event of type E_1 is followed by E_2, produce an event of type E_{12}

On	$(E_1; E_2)$
Condition	$E_1.x == E_2.x$
Action	*Output* E_{12}
Parameter context	*Chronicle*

Figure 2.4: An example rule in Snoop

Snoop considers the events as occurring at a specific point of time. Later the semantics of Snoop are extended to capture events that occur over an interval of time. The result is a new language called SnoopIB [1] with events that have interval-based time stamps.

Tesla. Tesla [7] is another event specification language with the language semantics defined using first-order temporal logic. T-Rex is a CEP system implemented for processing rules expressed in Tesla.

In Tesla the selection policy and consumption policy can be explicitly specified. Tesla supports the three selection policies described in 2.4. The *each*, *latest*, and *earliest* policies are named as *each*, last, and first respectively. It also supports the two consumption policies described in 2.4. In Tesla the selection and consumption policies can be specified for each component event of a rule. A simplified example rule defined in Tesla is given in figure 2.5. The scenario that the example rule defines is: if an event of type E_1 is followed by E_2, produce an event of type E_{12}. The quantifiers of each component event in the from clause of the rule gives the selection policy for that rule. The events given in the consumption clause of the rule are consumed after detection and other events are not consumed.

$define$	E_{12}
$from$	E_2 *and* $first$ E_1 *within* $5min$ $from$ E_2
consuming	$E_1,\ E_2$

Figure 2.5: An example rule defined in Tesla

Amit. Amit [2] defines an event specification language and also a run-time environment for processing the rules of the event specification language. In Amit a lifespan is defined for each rule. A lifespan is a time window and the events occurring in this lifespan are correlated to detect complex events.

In Amit the selection policy and consumption policy can be explicitly specified. Amit supports the three selection policies described in 2.4. The *each*, *latest*, and *earliest* policies are named as *each*, last, and first respectively. It also supports the two consumption policies described in 2.4. In a rule specified in Amit, the selection and consumption policies are specified for each component event. A simplified example rule defined in Amit is given in figure 2.5. The component event in the rule is called operand. For each operand in a rule, the quantifier gives the selection policy and the consumption condition specifies gives the consumption policy.

operator	*sequence*
$first$ *operand*	E_1
	$quantifier$: *each*
	consumption condition : *true*
second operand	E_2
	$quantifier$: *last*
	consumption condition : *true*

Figure 2.6: An example rule defined in Amit

Other ESLs. There are numerous other languages that are developed in academia and industry. For example, Stream mill [3] is a CEP system that executes the rules defined in Expressive stream language. This language allows for the specification of selection and consumption policies rule by rule.

All these languages show that parameter contexts are an important aspect of rule specification and are required to define a wide variety of situations from the real-world.

3 PACE System

In this section the parallelization of operator functionality is discussed and the PACE system [14], the state-of-the-art parallelistation technique, is presented.

3.1 Parallel CEP

3.1.1 Need for Parallel Processing

Applications using the CEP systems require low latency event detection as they need to react to the situations in real-time. If the production rate of events at sources is high, an operator that is expensive in terms of time may become a bottleneck in the CEP system. The overloading of operator requires either discarding of events or buffering of events at the input of the operator. Discarding of events will result false negative or/and false positive detection of complex events. The buffering of input events results in increased latency in event detection. Also, if the operator is overloaded continuously, the buffer length grows without a bound and no bound on the latency of the event detection can be established.

Therefore, in the presence of the overloading, parallelization of CEP functionality is required to detect events with low latency.

3.1.2 Consistent Parallelization

A parallization technique is consistent if the output stream of an operator with parallel processing is equivalent to the output stream of the operator without parallel processing. The equivalence can be established if the output stream of the parallel operator satisfies the following conditions:

- There must be no false positives i.e., the events that are not in the output stream of the operator with out parallelization must not be there in the output stream of the operator with parallization
- There must be no false negatives i.e., all events that are in the output stream of the operator with out parallelization must be there in the output stream of the operator with parallization
- There must be no duplicates i.e., an event must not be present more than once in the output stream of operator with parallelization.
- The order of the events must not change i.e., for any two events e_1 and e_2 if e_1 happens before e_2 in output stream of operator with out parallelization then e_1 must happen before e_2 in the output stream of the operator with parallization.

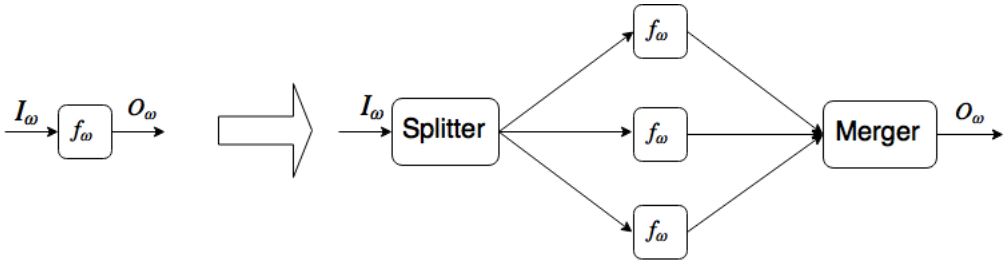

Figure 3.1: Data parallelization framework

3.2 Parallelization Approaches

3.2.1 Intra-operator Parallelization

In this parallelization approach, the event pattern to be detected is modelled as a finite state machine. The steps that can be run in parallel, in processing the incoming event stream, are identified based on the states in the finite state machine [4], [17]. The identification is done based on the fact that the evaluation of each state predicate can be parallelized. The operator logic is split into different identified stages and executed in parallel. In this approach the degree of parallelization depends on the finite state machine of the event pattern and therefore offers only a limited achievable parallelization degree depending on the number of states in the finite state machine.

3.2.2 Data Parallelization

In this parallelization approach, the operator functionality is replicated with a number of instances of the same operator running in parallel. The incoming event stream I_ω of an operator ω is split into a number of partitions and each partition is processed by an operator instance. The operator instance performs the operator correlation function f_ω on the assigned partitions and produces the output events. The operator instances running in parallel are supported by splitter and merger. Splitter divides the incoming stream I_ω into partitions based on a partitioning model. The partitioning model describes how splitter divides the incoming stream into partitions. The splitter assigns each partition to one of the operator instances. The merger receives the output streams from all the operator instances and combines them into a single operator output stream O_ω. The architecture of such an operator is depicted in figure 3.1.

3.3 PACE System

PACE system is the state of the art parallelization framework for CEP systems [14]. In this system the operator parallelization is achieved by data parallelization method. This system performs a consistent parallelization of operator functionality and provides high degree of parallelism for a wide class of CEP operators. The stream partitioning technique used in this system is named as Pattern sensitive stream partitioning.

3.3.1 Pattern sensitive stream partitioning

As discussed in section 2.3, an operator is executed according to its correlation function: mapping selections of events from the incoming streams to output events that are emitted on the outgoing streams. As the correlation function is a mapping, between two correlation steps no computational state is maintained. That means that any two selections can be processed by different operator instances independent of each other. Pattern sensitive stream partitioning uses this idea to split the event streams into partitions based on selections [14]. Therefore the criterion to be fulfilled during partition is: each selection must be contained completely in atleast one partition. That is to say all events that are part of a selection must be present in atleast one partition.

To ensure that a selection is completely comprised in a partition, all events between the first event and the last event of the selection are included in the partition. Thus, to partition the input stream I_ω, the points where selections start and end must be determined. For each event in I_ω, one or more out of three possible conditions are true:

1. The event triggers that a new selection is opened.
2. The event triggers that one or more open selections are closed.
3. The event triggers neither opening of a new selection nor closing of an open selection.

To evaluate which condition is true for an incoming event e, the splitter offers an interface that can be programmed according to the operator functionality. The interface comprises of two predicates:

- $P_s : e \rightarrow BOOL$
  For each incoming event e, P_s is evaluated to determine whether e starts a selection. If the predicate returns true, e starts a selection. An event type, whose event instance e starts a selection, is called an *initiator* event type. Accordingly, the event e is called an *initiator* event.
- $P_c : (\sigma_{open}, e) \rightarrow BOOL$
  For each incoming event e, P_c is evaluated with each open selection σ_{open} to determine whether e closes σ_{open}. If the predicate returns true, e closes the selection σ_{open}. An event type, whose event instance e closes a selection, is called a *terminator* event type. Accordingly, the event e is called the *terminator* event.

The splitter also provides the ability to store variables that capture internal state of a selection. These variables depend on the operator and the predicates, P_s and P_c, modify and use these predicates.

The flowchart for the processing of an incoming event is depicted in figure 3.2. Since a selection must be completely contained in one partition, all the events from start event of a selection to the end event of the selection are placed in one partition. The state of a partition is the state of the selection assigned to that partition. A partition is closed when the selection assigned to the partition is closed. Therefore, P_s and P_c are the predicates for opening and closing of partitions.

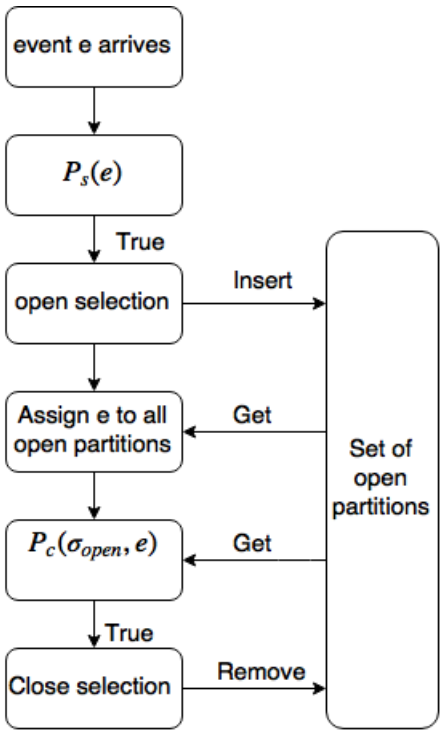

Figure 3.2: Flow chart of stream partitioning

3.3.2 Runtime Environment

The operator instance receives the assigned partitions and the corresponding events. The operator determines the selections in the assigned partitions and produces the output events based on the correlation function f_ω as discussed in section 2.3. The runtime environment of an operator instance manages the execution of an operator instance. It manages the partitions assigned to an operator instance by communicating with the splitter. In particular, when a complete selection is present in more than one partition, the runtime environment prevents the processing of a selection by multiple operator instances. This is done in the following way: When the operator instance processes an event that potentially starts a new selection, a function in the runtime environment isAssigned(startevent) is called, that signals whether it is a start event of an assigned selection or not.

3.3.3 Merging of output events.

The splitter assigns each partition an identifier. The identifier is an increasing sequence number. Also the splitter assigns each event in the incoming stream I_ω an increasing sequence number. The operator instances forwards the produced events to the merger. The output event contains, as an attribute, the identifier of the partition in which it is detected. The sequence number of the output event is the sequence number of the last event in the selection in which it is detected. The merger establishes the ordering of the output events based on the partition identifier and the sequence number of the output event.

3.3.4 Example Predicates

With the ability to program the predicates with respect to the operator, the pattern sensitive stream partitioning can be used for parallelizing a wide class of operators. Below the predicates for parallelizing the two most common operators in CEP, sequence and conjunction operators, are given as examples.

Algorithm 1 gives the predicates for a sequence operator $Sequence(E_1; E_2)$ with two event types E_1 and E_2 [14]. A partition is opened when an event of type E_1 is received and closed when an event of type E_2 is received following the event of type E_1.

Algorithm 1 Predicates for *sequence* operator of PACE system

```
1: procedure P_s(Event e)
2:     if e.type == E_1 then
3:         return TRUE
4:     else
5:         return FALSE
6:     end if
7: end procedure
8:
9: procedure P_c(Event e Selection σ)
10:    if e.type == E_2 AND e.timestamp ≥ s.startTime then
11:        return TRUE
12:    else
13:        return FALSE
14:    end if
15: end procedure
```

Algorithm 2 gives the predicates for a conjunction operator $Conjunction(E_1 \& E_2)$ [14]. A partition is opened when an event of type E_1 or E_2 is received and closed when an event of type E_1 is followed by an event of type E_2 or when an event of type E_2 is

followed by an event of type E_1.

Algorithm 2 Predicates for *conjunction* operator of PACE system

```
1: procedure P_s(Event e)
2:     if e.type == E_1 OR e.type == E_2 then
3:         return TRUE
4:     else
5:         return FALSE
6:     end if
7: end procedure
8:
9: procedure P_c(Event e Selection σ)
10:    if e.type == E_2 AND e.timestamp ≥ s.startTime then
11:        if s.startType == E_1 then
12:            return TRUE
13:        end if
14:    else if e.type == E_1 AND e.timestamp ≥ s.startTime then
15:        if s.startType == E_2 then
16:            return TRUE
17:        end if
18:    end if
19:    return FALSE
20: end procedure
```

3.4 System Model

In this section, the assumptions made regarding the system model of a parallel CEP are specified in detail.

Nodes. As discussed in section 2.2.2, a distributed CEP system consists of a network of operators. In a parallelized CEP system an operator consists of the following processes:

- Splitter
- Instances
- Merger

These processes are deployed on a network of nodes. These nodes are considered to be failure-free and provide a homogeneous computing capability, i.e., the same CPU and memory capabilities. The sources and sinks are not considered a part of the CEP system. Therefore the nodes on which these processes are deployed are assumed to be only failure-free.

Communication Channel. The sources, operators, and sinks communicate among themselves by event streams. In a parallelized CEP the communication between the processes of an operator is also done by event streams. The event streams are transmitted through the communication channels interconnecting the nodes on which these processes are deployed. It is assumed that the communication channels are failure-free and provides reliable communication with the following properties the following properties:

- *No loss of events.* An event e sent from a node N_1 to a node N_2 will be eventually received at the receiver node N_2.
- *No duplicate events.* An event e sent from a node N_1 to a node N_2 will not be received by N_2 more than once.
- *No out-of-sequence events.* If a node N_1 sends event e_1 before sending event e_2 to node N_2, then N_2 receives e_1 before receiving e_2.
- *No modification of events.* An event e transmitted over a communication channel is not modified in the channel during transmission.

4 Problem Description

From the discussion of parameter contexts and event specification languages in section 2, it is clear that every CEP system should have the ability to execute rules with parameter context. The PACE system discussed in section 3 does not provide the features for partitioning of event streams for parallel processing operators with parameter context. So, the main objective of this thesis is to design a partitioning procedure for parallel processing operators with parameter context by extending the PACE system. The parameter contexts considered in this thesis are the parameter contexts presented in table 1 of section 2.4. In designing such a partitioning procedure the consistent parallelization property of the PACE system is maintained.

5 Stream Partitioning with Parameter Context

As mentioned in section 4, the PACE system does not allow for specifying the rules with parameter context. The PACE system executes all the rules with its default parameter context. Let us first analyse the default parameter context of the PACE system.

The processing of events from incoming stream of an operator in the PACE system is summarised as follows:

Split

1. Open a partition for every *initiator* event occurrence.
2. Assign an event to all open partitions.
3. Close an open partition on the occurrence of first *terminator* event after the opening of partition.

Process

An instance processes each partition independently; detecting the output events from the selection in the assigned partition, which has the first event of the partition as *initiator* event.

Merge

Combine all the output events from the instances into a single output stream.

With the above procedure for event processing, the existing PACE system executes *rules* with *Each Selection - Zero Consumption* parameter context. As discussed in section 2.4, when a rule with zero consumption policy is executed, it is required to restrict the portion of incoming stream to consider in determining the next selection. As an open partition is closed on the occurrence of first *terminator* event and the first event in the next partition is next event of *initiator* event type, an event of *initiator* event type will act as an initiator event for only one selection. Therefore the *initiator* event in every selection is invalidated and not considered in determining the next selections. This keeps the size of the selection under check from growing arbitrarily large by invalidating the events before the latest *initiator* event.

As the event in an incoming stream is assigned to all open partitions and the partitions are processed by instances independently, an event from the incoming stream is used in the production set of output events of multiple selections. Therefore, the PACE system does not support executing a rule with selected consumption policy.

In the rest of this section, parallelization techniques, based on the existing PACE system, for the following parameter contexts is presented:

1. Latest Selection - Zero Consumption
2. Each Selection - Selected Consumption

3. Latest Selection - Selected Consumption

4. Earliest Selection - Selected Consumption

The *Earliest Selection - Zero Consumption* parameter context is not discussed in this section, as this parameter context does not find any applications for the situations in the real world. This because because, the selected events are never consumed and all the output events are derived from the same set of earliest events.

5.1 Latest Selection - Zero Consumption

In this parameter context, because of *zero consumption* policy, the events are not consumed after being used in a production set. However, because of *latest selection* policy, an event is invalidated when a new event of same type occurs. This is because, the latest event of an event type is used in production set of output events.Therefore, when a selection is determined, all the events that occurred before the *initiator* event of the selection are invalidated. For this reason, this parameter context does not require the restriction of selection's size by using a window operation over the incoming stream.

5.1.1 Partitioning Model

In this parameter context, an event of *initiator* event type acts as *initiator* event for zero, one, or multiple partitions. This is because, an *initiator* event, will continue to be the *initiator* event of new selections until the next event of *initiator* event type becomes the *initiator* event of a selection.

For partitioning the incoming stream, of an operator with this parameter context, with one selection per one partition, multiple partitions must be opened when an event of *initiator* event type occurs. The number of selections for which an event of *initiator* event type will act as *initiator* event can only be determined when the next event of the *initiator* event type occurs. Therefore, it is not possible to partition the incoming stream with one selection per one partition. As a result, all the selections with same *initiator* event have to be placed in same partition. For this reason, the predicates P_s and P_c are used to determine the opening and closing of partitions rather than selections.

Consider an *initiator* event e_i^j, which is the j_{th} instance of the *initiator* event type. If a partition has all the events occurring between the *initiator* event e_i^j and *terminator* event of the last selection initiated by e_i^j, then such a partition will have all the events of all the selections for which e_i^j is the *initiator* event. Therefore, a partition is opened when an event of *initiator* event type occurs. An open partition has to be closed when the *terminator* event of last selection occurs. But whether a selection with the *initiator* event e_i^j is the last of the selections with e_i^j as *initiator* is known only when the next selection is determined with e_i^{j+1} as *initiator* event. To determine whether the *initiator* event of next selection is different from the first event of a partition, for each open partition a finite state machine is maintained based on the events included in the partition.

If the state of a partition is in the final state of the state machine, the next event of the *terminator* event type is the *terminator* event of a selection. When the finite state machines of two consecutive partitions are in same state, then the first partition is closed. This is because, the *initiator* event of the next selection will be the first event of second partition and therefore, the next selection will be present in the second partition.

Therefore, the partitioning model can be given as:

- A new partition is opened when an *initiator* event occurs. That is to say, $P_s(e)$ returns true, if e is an event of initiator event type.
- A partition P_j, where j is the partition identifier, is closed when the state of the partition P_j is same as the state of partition P_{j+i}, where $i > 0$.

5.1.2 Proof of Consistent Parallelization

In this section, the proof for consistent paralleization, when incoming stream is partitioned according to the above discussed partitioning model, is given.

False Positives. Assume that a false positive event e_{out} is detected in selection σ of partition p. Assume that the production set D of the detected false positive event e_{out} is given as $(e_1^{i1}, e_2^{i2}, \ldots, e_N^{iN})$. Since e_{out} is a false positive event, there must be an event $e_j^{ij} \in D$ where $1 \leq j < N$ such that there exists an event e_j^{ij+1} in incoming stream I_ω with the time stamp $e_j^{ij+1}.timeStamp < e_N^{iN}.timeStamp$. Since every event between the start event and event of σ is in p, e_j^{ij+1} is in p. If so, then e_{out} is never detected since, in an operator instance with *latest selection*, the latest event e_j^{ij+1} will be in production set instead of e_j^{ij}. This contradicts our initial assumption that e_{out} is an output event. This implies that our initial assumption that e_{out} is a false positive event is also wrong. Therefore, no false positive events are detected.

False Negatives. Assume that a positive event e_{out}, detected without parallelization, is not detected with parallelization. Assume that the production set D of this false negative event e_{out} is given as $(e_1^{i1}, e_2^{i2}, \ldots, e_N^{iN})$. Since e_{out} is a postive event in the operator without parallelization,

$$\forall 1 \leq j < N \quad e_j^{ij+1}.timeStamp > e_{j+1}^{i(j+1)}.timeStamp$$

This implies that all the events from e_1^{i1} to e_N^{iN} must be in one of the partitions p. If so, an operator instance with *latest selection* processing partition p will detect the event e_{out}. This contradicts our initial assumption that e_{out} is a false negative event. Therefore all the positive events detected without parallelization are detected and there are no false negatives.

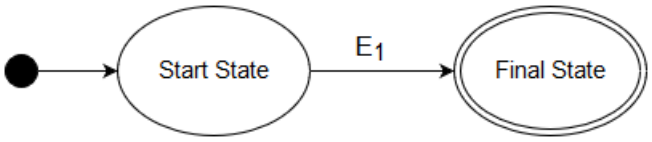

Figure 5.1: The finite state machine of a partition of $Sequence(E_1, E_2)$ operator

Duplicates. An output event e_{out} is detected more than once, if the selection σ, in which e_{out} is detected, is completely present in more than one partition. Since the state of the finite state machines of two open partitions can never be the same, a selection cannot be present completely in two partitions. Therefore, duplicates of an output event are not detected.

Ordering of Events. Consider two selections σ_i and σ_j. Assume that the terminating events e^{ti} and e^{tj} respectively of σ_i and σ_j are such that $e^{ti}.timeStatmp < e^{tj}.timeStamp$. Let e^i_{out} and e^j_{out} be the output events of σ_i and σ_j, respectively. The ordering of the output stream is maintained, if e^i_{out} occurs before e^j_{out} in the output stream.

If the two selections σ_i and σ_j are in the same partition, then the ordering of the two output events is maintained in the event stream from instance that processed the partition to the merger. As the merger does not change the ordering of the events from an instance, the ordering is maintained in the output stream as well.

If the two selections σ_i and σ_j are in two different partitions p_i and p_j, then from the partitioning model as described above, the identifier of p_i is smaller than the identifier of p_j as p_i is opened before p_j. As discussed in section 3.3.3, merger orders the events based on the partition identifier of the output events. Therefore, $\forall\ i, j\ e^i_{out}$ will always occur before e^j_{out} in the output stream, maintaining the ordering of the events.

5.1.3 Example Predicates

Sequence. Algorithm 3 gives the predicates for a sequence operator $Sequence(E_1; E_2)$ with two event types E_1 and E_2. A partition is opened when an event of type E_1 is received. This is because E_1 is an *initiator* event type and therefore, an event of type E_1 starts a selection. All the events following the *initiator* event are included in the partition until the closing predicate P_c returns true for this partition.

For this operator the finite state machine will have only one state other than the start state and it is final state. The finite state machine of a partition of this operator is shown in figure 5.1. The final state is reached when a partition is opened on the occurrence of an event of type E_1. This is because any following event of type E_2 will act as *terminator* event of a selection in this partition. The next partition is opened when the next event of type E_1 occurs and the next partition's state of the finite state machine will be the

Algorithm 3 Predicates for *sequence* operator with *Latest selection and Zero consumption*

```
1: procedure P_s(Event e)
2:     if e.type == E_1 then
3:         return TRUE
4:     else
5:         return FALSE
6:     end if
7: end procedure
8:
9: procedure P_c(Event e, Partition p)
10:    if e.type == E_1 AND e.timestamp > p.startTime then
11:        return TRUE
12:    else
13:        return FALSE
14:    end if
15: end procedure
```

final state after the opening of partition. As a result, the present partition has to be closed when the next partition is opened. Therefore, an open partition is closed when an event of type E_1 is received after opening the partition.

Conjunction. Algorithm 4 gives the predicates for a conjunction operator, with two component event types, $Conjunction(E_1 \& E_2)$. A partition is opened when an event of type E_1 or E_2 is received. This is because, the events of both event types E_1 and E_2 act as *initiator* events for this $Conjunction$ operator. All the events following the *initiator* event are included in the partition until the closing predicate P_c returns true for this partition.

A partition of this operator has two final states and the start state. The finite state machine of a partition of this operator is shown in figure 5.2. From the start state based on the type of *initiator* event of this partition one of the two final states is reached. When the partition is opened with *initiator* event of type E_1, the *terminator* event of the selections in the partition is an event of type E_2. Conversely, when the partition is opened with *initiator* event of type E_2, the *terminator* event of the selections in the partition is an event of type E_1. The two final states of the finite state machine represent the two states: the next event of type E_1 is the *terminator* event and the next event of type E_2 is the *terminator* event. when a future partition is opened with an event of same *initiator* event type as this partition, then such a partition's finite state machine will be in the same final state as this partition. As a result, this partition has to be closed. Therefore, an open partition is closed when an event of type E_1 is received in a partition where the first event is of type E_1 or when an event of type E_2 is received in

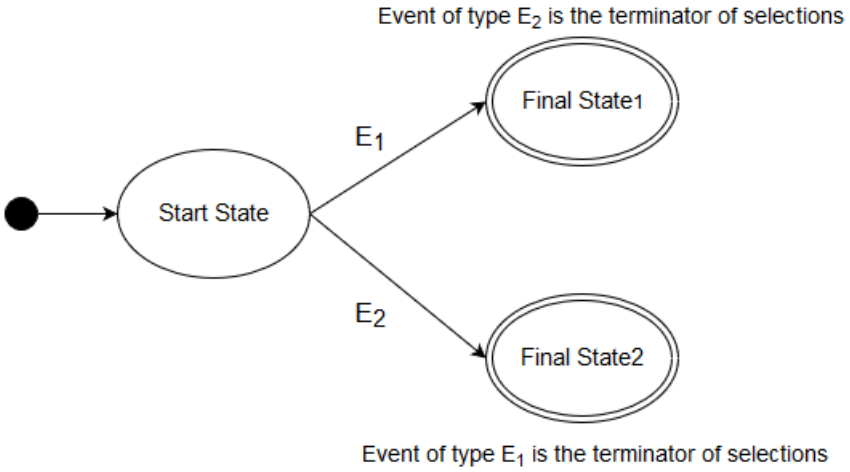

Figure 5.2: The finite state machine of a partition of $Conjunction(E_1, E_2)$ operator

Algorithm 4 Predicates for *conjunction* operator for *Latest selection and Zero consumption*

```
1:  procedure P_s(Event e)
2:      if e.type == E_1 OR e.type == E_2 then
3:          return TRUE
4:      else
5:          return FALSE
6:      end if
7:  end procedure
8:
9:  procedure P_c(Event e Selection σ)
10:     if e.type == E_1 AND e.timestamp > s.startTime then
11:         if s.startType == E_1 then
12:             return TRUE
13:         end if
14:     else if e.type == E_2 AND e.timestamp > s.startTime then
15:         if s.startType == E_2 then
16:             return TRUE
17:         end if
18:     end if
19:     return FALSE
20: end procedure
```

a partition where the first event is of type E_2.

5.2 Each Selection - Selected Consumption

In this parameter context, a selection is mapped to multiple output events. Because of *each selection*, each event in a selection is used atleast once in the production set of an output event. Because of *selected consumption*, all the events in a selection are consumed and no event of this selection is used in a future selection.

5.2.1 Partitioning Model

By changing the partitioning procedure, in the splitter, of the PACE system, the existing framework can be extended to parallelize an operator with this parameter context. As all the events of a selection are consumed and no two selections have an event common between them, the selections in this parameter context do not overlap. Therefore, at any point of time in splitter, there is at most one open selection. As a result, there is at most one open partition. The flowchart in figure 5.3 shows the partitioning procedure.

The predicates for partitioning work in the following way:

- A new partition is opened when an *initiator* event occurs. That is to say, $P_s(e)$ returns true, if e is an *initiator* event.
- An open partition is closed when a *terminator* event occurs. That is to say, $P_c(\sigma_{open}, e)$ return true, if e is the *terminator* event of σ_{open}

5.2.2 Proof of consistent partitioning

False positives. Assume that a false positive event e_{out} is detected in selection σ of partition p. Assume that the production set D of the detected false positive event e_{out} is given as $(e_1^{i1}, e_2^{i2}, \ldots, e_N^{iN})$. Since e_{out} is a false positive event, there must be an event $e_j^{ij} \in D$ where $1 \leq j \leq N$ such that e_j^{ij} is consumed in a previous selection. This implies that, e_j^{ij} is included in two partitions. Since the partitions are disjoint an event cannot be part of two partitions. This contradicts our initial assumption that, e_{out} is a false positive event. Therefore, no false positive events are detected.

False negatives. Let e_{out} be a positive event with production set D containing events $(e_1^{i1}, e_2^{i2}, \ldots, e_N^{iN})$ from the incoming queue.
Since e_{out} is a positive event, for e_N^{iN-1}, the latest *terminator* event before e_N^{iN}, the condition $e_N^{iN-1}.timeStamp < e_1^{i1}.timeStamp$ must hold. This is because, if $e_N^{iN-1}.timeStamp$ is greater than $e_1^{i1}.timeStamp$,then all the events e_j^{ij} in D, for which $e_j^{ij}.timeStamp < e_N^{iN-1}.timeStamp$, would have been part of the selection in which e_N^{iN-1} is the *terminator* event. This means that all these events would be consumed in

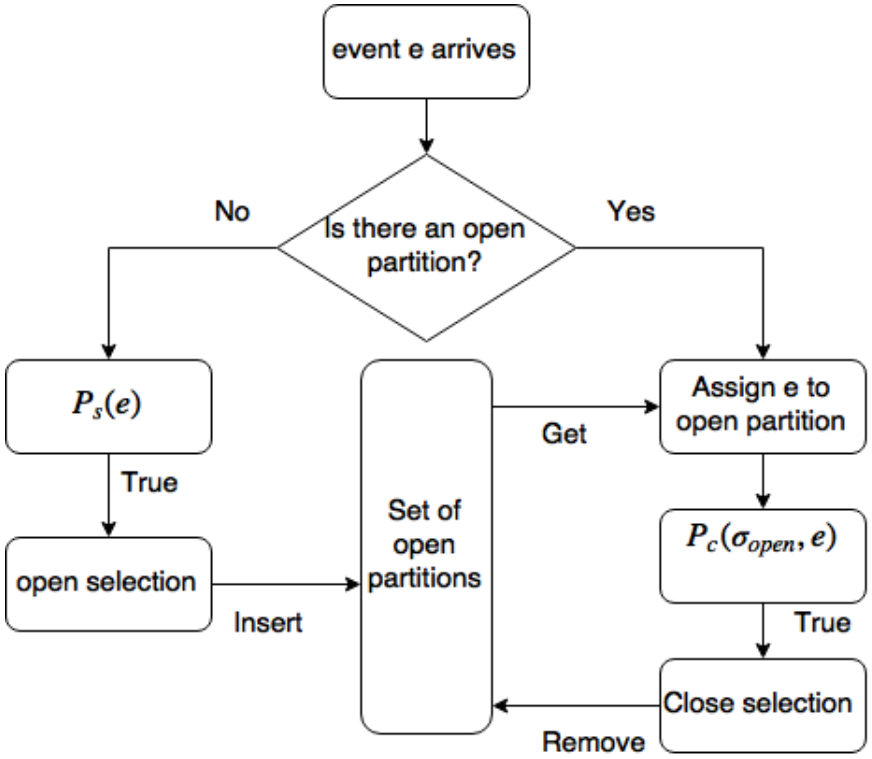

Figure 5.3: Flow chart of stream partitioning for Each Selection - Selected Consumption parameter context

the previous selection and e_{out} with production set D wouldn't be a positive event.

Since e_N^{iN} is the first *terminator* event after e_1^{i1}, according to the proposed partitioning procedure, all the events from e_1^{i1} to e_N^{iN} will be included in one partition. Therefore, e_{out} with production set D will be detected in the instance processing this partition and false negatives do not occur.

Duplicates. A duplicate of an output event is detected, if a selection is processed by more than one instance. In the proposed partitioning procedure, all the partitions are disjoint and a partition is processed by only one instance. This implies that, a selection is processed by only one instance. Therefore, each output event is detected only once and duplicate events do not occur in the output stream.

Ordering of events. The ordering of the events in the output stream of the parallel operator is consistent if the following condition is met: for all i and j, if the output event e_{out}^i occurs before output event e_{out}^j in the output stream of the operator without parallelization, then e_{out}^i occurs before e_{out}^j in the output stream of the operator with

parallelization.

There are two cases of possibility depending on the partition in which e^i_{out} and e^j_{out} are detected:

- Case 1 is where e^i_{out} and e^j_{out} are output events from the same partition. In this case, the instance forwards e^i_{out} before e^j_{out} to the merger. Since the merger does not change the order of the output events from an instance, the order of e^i_{out} and e^j_{out} is maintained in the output stream.
- Case 2 is where e^i_{out} and e^j_{out} are output events from different partitions(and as a result from different selections). In this case, the merger orders the events e^i_{out} and e^j_{out} based on the identifier of the partition in which they are detected. Since e^i_{out} occurs before e^j_{out} in the output stream of operator without parallelization, selection σ_i, in which e^i_{out} is detected, is determined before selection σ_j, in which e^j_{out} is detected. Since σ_i is determined before σ_j, partition p_i, in which σ_i is included, will be opened earlier than partition p_j, in which σ_j is included. As a result, p_i will have smaller identifier than p_j. Therefore, merger orders the event e^i_{out} with lower partition identifier before e^j_{out} with higher partition identifier.

Therefore, the ordering of e^i_{out} and e^j_{out} is maintained in all possible cases.

5.2.3 Example Predicates

For partitioning of this parameter context, the predicates defined in the existing PACE system are used. The opening and closing predicates for sequence operator are given in algorithm 1 of section 3.3.4. The opening and closing predicates for conjunction operator are given in algorithm 2 of the same section 3.3.4.

5.3 Latest Selection - Selected Consumption

In this parameter context, a selection is mapped to only one output event. The latest events of the component event types in a selection are included in the production set of the output event. The events in a selection that are included in the production set of an output event are consumed and are not included in the future selections. Since only latest events of an event type are included in the production set, an event in the incoming stream becomes invalidated when another event of same event type occurs.

5.3.1 Partitioning Model

By changing the partition procedure, in the splitter, of the PACE system, the existing framework can be extended to parallelize an operator with this parameter context. Since the latest events of an event type in a selection are consumed and other events of the same event type are invalided because of the occurrence of the latest event, no events from a selection are included in the future selections. Therefore, similar to *Each selection*

- *Selected consumption* parameter context, at any point of time, there is at most one open selection in the splitter. As a result, there is at most one open partition. As the selections and partitions are similar to the *Each selection - Selected consumption* parameter context, the same partitioning procedure, as described in section 5.2.1, can be used at the splitter.

5.3.2 Proof of Consistent Parallelization

In this section, the proof for consistent paralleization of an operator, with *Latest Selection - Selected Consumption* parameter context, when incoming stream is partitioned according the above proposed partitioning model, is given.

False Positives. Assume that a false positive event e_{out} is detected in selection σ of partition p. Assume that the production set D of the detected false positive event e_{out} is given as $(e_1^{i1}, e_2^{i2}, \ldots, e_N^{iN})$. Since e_out is a false positive event, one of the two following cases must be true:

1. There is an event $e_j^{ij} \in D$, which has already been consumed in a previous selection.
2. There is an event $e_j^{ij} \in D$ where $1 \leq j < N$, such that there exists an event e_j^{ij+1} in incoming stream I_ω, where

$$e_j^{ij+1}.timeStamp > e_j^{ij}.timeStamp$$

$$e_j^{ij+1}.timeStamp < e_N^{iN}.timeStamp$$

.

In case 1, if e_j^{ij} is consumed in two selections, then e_j^{ij} must be included in two partitions. But, according to the proposed partitioning model, any two partitions are disjoint sets of incoming events. Therefore, e_j^{ij} cannot be part of two partitions and the first case is not true.

In case 2, since all the events between the start event and end event of a selection are included in a partition, event e_j^{ij+1} is also included in the partition p. Since $e_j^{ij+1}.timeStamp > e_j^{ij}.timeStamp$, an event processing partition p will detect the output event with production set that contains e_j^{ij+1} and not e_j^{ij}. Therefore, in case 2, e_{out} is not detected by an instance.

Therefore, a false positive event is not detected in all the possible cases.

False Negative. Let e_{out} be a positive event with production set D containing events $(e_1^{i1}, e_2^{i2}, \ldots, e_N^{iN})$ from the incoming queue. Since e_{out} is a positive event, for e_N^{iN-1}, the latest *terminator* event before e_N^{iN}, the condition $e_N^{iN-1}.timeStamp < e_1^{i1}.timeStamp$ must hold. This is because, if $e_N^{iN-1}.timeStamp$ is greater than $e_1^{i1}.timeStamp$,then

all the events e_j^{ij} in D, for which $e_j^{ij}.timeStamp < e_N^{iN-1}.timeStamp$, would have been part of the selection in which e_N^{iN-1} is the *terminator* event. This means that all these events would be consumed in the previous selection and e_{out} with production set D wouldn't be a positive event.

Since e_N^{iN} is the first *terminator* event after e_1^{i1}, according to the proposed partitioning procedure, all the events from e_1^{i1} to e_N^{iN} will be included in one partition. Therefore, e_{out} with production set D will be detected in the instance processing this partition and false negatives do not occur.

Duplicates. A duplicate of an output event is detected, if a selection is processed by more than one instance. In the proposed partitioning procedure, all the partitions are disjoint and a partition is processed by only one instance. This implies that, a selection is processed by only one instance. Therefore, each output event is detected only once and duplicate events do not occur in the output stream.

Ordering of events. The ordering of the events in the output stream of the parallel operator is consistent, if the following condition is met: for all i and j, if the output event e_{out}^i occurs before output event e_{out}^j in the output stream of the operator without parallelization, then e_{out}^i occurs before e_{out}^j in the output stream of the operator with parallelization.

Since e_{out}^i occurs before e_{out}^j in the output stream of operator without parallelization, the selection σ_i, in which e_{out}^i is detected, is determined before σ_j, in which e_{out}^j is determined. Since σ_i is determined before σ_j, the partition p_i, in which σ_i is included, is opened before partition p_j, in which σ_j is included. As a result p_i will have the smaller partition identifier, when compared to p_j. Since the merger orders the output events based on the partition identifier in which the output event is detected, e_{out}^i, whose partition identifier is smaller, occurs before e_{out}^j in the output stream of the parallel operator. Therefore, the ordering of the events is maintained.

5.3.3 Example Predicates

For partitioning of this parameter context, the predicates defined in the existing PACE system are used. The opening and closing predicates for sequence operator are given in algorithm 1 of section 3.3.4. The opening and closing predicates for conjunction operator are given in algorithm 2 of the same section 3.3.4.

5.4 Earliest Selection - Selected Consumption

In this parameter context, the operator correlates each selection to one output event. The earliest events of the component event types in a selection are used in the production set of the output event. The events used in the production set of an output event are consumed and are not used in determining the next selection.

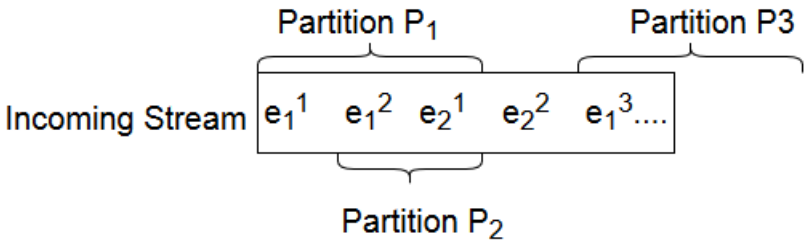

Figure 5.4: Partitioning in a $Sequence(E_1, E_2)$ operator

5.4.1 Partitioning with PACE system.

The idea of the PACE system is to partition the incoming stream by putting all the events from start of a selection to the end of a selection in one partition. To do so the starting predicate $P_s(e)$ determines if an event e starts a new selection and closing predicate $P_c(e, \sigma)$ determines if an event e closes a selection σ.

To determine if an event e closes a selection σ, a state is maintained for each open selection. The decision whether an event e closes selection σ is made based on the state of the selection σ. The state of a selection is determined by the events included in the partition that contains the selection. With *earliest selection - selected consumption* parameter context it is not possible to determine if an event e closes a selection. This is because, when partitions overlap, an event is included in multiple partitions. The closing predicate cannot appropriately maintain the state of an open selection based on the events included in the partition, as it does not have the information if an event is consumed in the earlier selections in which the event is included.

For example, consider the sequence operator $Sequence(E_1, E_2)$. The predicates for this operator are given in algorithm 1 of section 3.3.4. A partition is opened when an event of type E_1 occurs. A partition is closed when an event of type E_2 occurs after opening of a partition. Consider the incoming stream and partitions as shown in figure 5.4. The event e_i^j of the incoming stream is the j^{th} instance of event type E_i. The predicates for this operator gives the first partition as (e_1^1, e_1^2, e_2^1). The selection in this partition is correlated to output event e_{out}^1, with the production set (e_1^1, e_2^1). The second partition is given as (e_1^2, e_2^1). The selection in this partition contains event e_2^1, which is included in production set of output event e_{out}^1 and therefore must have been consumed. This results in false positive event with the production set (e_1^2, e_2^1). As the next partition starts at event e_1^3, the positive output event e_{out}^2 with production set (e_1^2, e_2^2) is not detected, resulting in a false negative event.

Therefore, the predicates of the existing framework does not perform the consistent partitioning of the incoming stream with *earliest selection - selected consumption* parameter context.

Why PACE system cannot be extended?

In order to perform consistent stream partitioning with PACE system for *Earliest Selection - Selected Consumption* parameter context, it is required to determine the *terminator* event of a selection at the splitter. The determination of *terminator* event of a selection, with this parameter context, requires knowledge of consumed events in the previous selections.

For example, consider the *sequence* operator and incoming stream discussed in the previous section (Figure 5.4). With consistent partitioning the first partition is (e_1^1, e_1^2, e_2^1) and the second partition is (e_1^2, e_2^1, e_2^2). This is because, e_1^1 and e_2^1 are the *initiator* and *terminator* events of first selection and e_1^2 and e_2^2 are the *initiator* and *terminator* events of second selection. The splitter can determine that e_2^2 is the *terminator* event of second selection and not e_2^1, only if it has the knowledge that e_2^1 is consumed in the first selection.

The consumed events in a selection are known only after the mapping of selection to output events. Therefore, in order to determine a partition, the splitter has to wait for processing of all the previous partitions. This leads to sequential processing of the incoming stream and no parallelization is acheived. Therefore, it is not possible to modify the partitioning procedure of the existing PACE system to consistently parallelize an operator with *Earliest Selection - Selected Consumption* parameter context.

5.4.2 Merger Filterning

For consistently parallelizing an operator with *Earliest Selection - Selected Consumption* parameter context, *the merger filtering* method is proposed in this section.

5.4.3 Event Processing

In this method, the splitter uses only the starting predicate to determine the opening of a selection. A partition contains only one selection. Therefore, when a selection is opened, a partition is opened and the selection is included in the partition. The opened partition is assigned to an instance for processing. The procedure for splitting is shown in the flowchart in figure 5.5.

The instance processes a partition by detecting all the possible events from the selections in the partition whose *initiator* event is the first event in the partition. The instance forwards the output events to merger, in the order of their detection, along with the production set of the output event. The output event also contains identifier of the partition in which it is detected.

The merger receives the output events from the instances and performs the filtering for positive events. The merger discards the false positive events, and forwards the

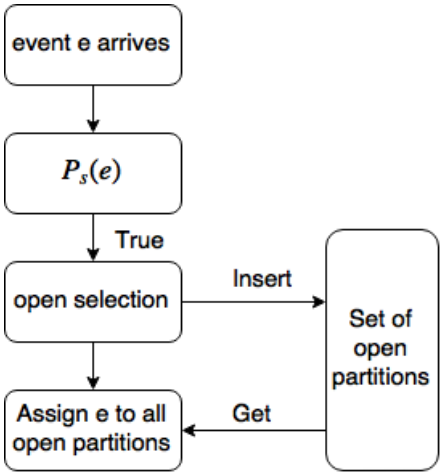

Figure 5.5: Partitioning in a $Sequence(E_1, E_2)$ operator

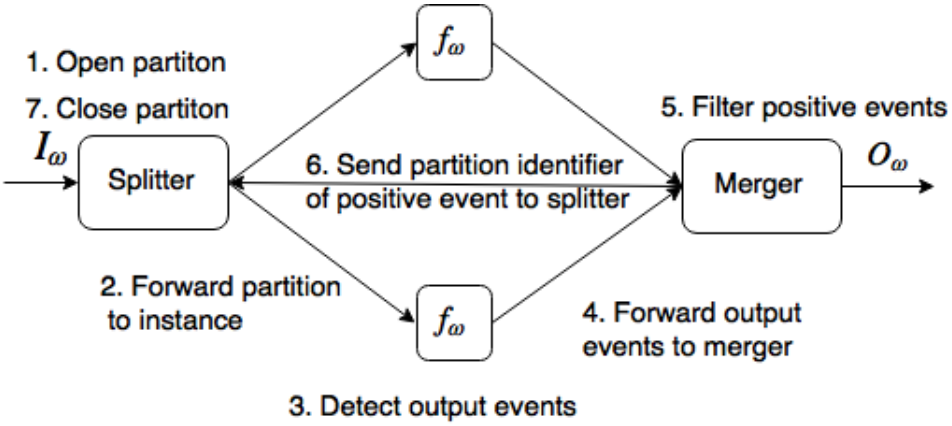

Figure 5.6: Event processing with Merger filtering

positive events to the *consumers* through the output stream. When the merger finds a positive event, it sends a message with the partition identifier of the output event to the splitter. When the splitter receives this message from the merger, the partition with the identifier in the message is closed.

The sequence of steps in event processing is shown in figure 5.6.

Filtering of positive events. To filter positive events from the detected events forwarded by instances, the merger maintains a set of consumed events. As the name suggests, this set contains the events that are included in the production set of output events, i.e. consumed events. The merger orders the detected events from different instances based on the partition identifier of the detected event. The merger processes the

detected event by comparing the production set of the detected event with the set of consumed events. If the production set is disjoint with the set of consumed events, i.e., all the events in the production set are not consumed, the detected event is determined as positive output event. The production of this output event is added to the set of consumed events, as the events in production set are consumed now. If the production set is not disjoint with set of consumed events, the detection is a false positive and therefore, it is discarded.

5.4.4 Proof of Consistent Partitioning

In this section, the proof for consistent parallelization, when an operator is parallelized with merger filtering method, is discussed.

False Positives. Assume that a false positive event e_{out} is detected in selection σ of partition p. Assume that the production set D of the detected false positive event e_{out} is given as $(e_1^{i_1}, e_2^{i_2}, \ldots, e_N^{i_N})$. If e_{out} is a false positive event, then one of the following conditions must hold:

1. There is an event $e_j^{i_j} \in D$, that is already consumed in a previous selection.
2. There is an event $e_j^{k_j} \in \sigma$, which is not consumed in any previous selection, such that,

$$e_j^{k_j}.timeStamp < e_j^{i_j}.timeStamp$$

$$e_j^{k_j}.timeStamp > e_{j-1}^{i_{j-1}}.timeStamp$$

Consider the first case. The detected events from a selection before σ will be from a partition with smaller identifier. the merger orders the events according to the partition identifier. As a result, detected events from selection σ_p will be processed before the detected events from selection σ. Therefore, in the first case, if the event $e_j^{i_j}$ is consumed in a previous selection σ_p, $e_j^{i_j}$ will be in the consumed set of the merger. Since merger discards any detected event whose production set contains an event from the consumed set, e_{out} will be discarded and will not be an output event.

Now consider the second case. In such scenario, the event $e_j^{k_j}$ is also included in the partition p. This is because all the events after the event that opened the partition are included in the partition until the partition is closed. If the second case is true and and there is a detected event e_{out} with production set $(e_1^{i_1}, \ldots, e_j^{i_j}, \ldots, e_N^{i_N})$, then there will be a detected event with production set $(e_1^{i_1}, \ldots, e_j^{k_j}, \ldots, e_N^{i_N})$. This is because, the instance detects the events with *Each Selection - Selected consumption*. Let this detected event be e'_{out}. The production set of e'_{out} and e_{out} is same, except for the event of type E_j. Since the time stamp of $e_j^{k_j}$ is less than the time stamp of $e_j^{i_j}$, e'_{out} will be detected and forwarded to merger before e_{out}. If merger determines e'_{out} as positive

event, then e_{out} will not be a positive event. This is because, the production sets of e'_{out} and e_{out} have events in common. If merger determines e'_{out} as false positive event, then merger determines e_{out} as false positive event. This is because, since e'_{out} is a false positive event and $e_j^{k_j}$ is not consumed, this implies that atleast one of the events common between the production sets of e'_{out} and e_{out} is consumed. Therefore e_{out} is also discarded as false positive.

Therefore in all the possible cases, e_{out} is not an output event and no false positives are detected.

False Negatives. Let e_{out} be a false negative event. Let the production set of e_{out} be D given as $(e_1^{i_1}, e_2^{i_2}, \ldots, e_N^{i_N})$. Since an instance processing a partition detects all the possible events from the partition, e_{out} will be detected by any one of the parallel instances. Since e_{out} is false negative, it must have be discarded in the merger. This implies that there is atleast one event that is common between the production set D and the consumed set of the merger. Since e_{out} is a positive event in the operator without parallelization, the events in D are not consumed in any previous selection. As discussed previously, the false positives are not possible. Therefore, the events in the consumed set of the merger contain only the events consumed in the previous selections. Since, the events in D are not consumed in previous selection, the consumed set of the merger and D are disjoint sets. This contradicts our initial assumption that e_{out} is a false negative event. As a result, merger will not discard the event e_{out} and will determine e_{out} as the positive event. Therefore, false negatives do not occur.

Duplicates. Let e_{out} be the positive event detected from a selection whose *initiator* event is e_i^j. since only one instance correlates the selections in which e_i^j is the initiator event, e_{out} will be detected only once. Therefore, there can be no duplicates of the events in the output stream.

Ordering of Events. The ordering of the events in the output stream of the parallel operator is consistent, if the following condition is met: for all i and j, if the output event e_{out}^i occurs before output event e_{out}^j in the output stream of the operator without parallelization, then e_{out}^i occurs before e_{out}^j in the output stream of the operator with parallelization.

Since e_{out}^i occurs before e_{out}^j in the output stream of operator without parallelization, the selection σ_i, in which e_{out}^i is detected, is determined before σ_j, in which e_{out}^j is determined. Since σ_i is determined before σ_j, the partition p_i, in which σ_i is included, is opened before partition p_j, in which σ_j is included. As a result p_i will have the smaller partition identifier, when compared to p_j. Since the merger processes the output events based on the partition identifier in which the output event is detected, e_{out}^i, whose partition identifier is smaller, occurs before e_{out}^j in the output stream of the parallel operator. Therefore, the ordering of the events is maintained.

5.4.5 Example Predicates

The splitter of an operator with merger filtering parallelization method consists of only the *start* predicate. The *start* predicate of a merger filtering operator is defined in the same way as the start predicate of the operator in the existing PACE system. The start predicates of algorithms 1 and 2 of section 3.3.4 gives the predicates for *sequence* and *conjunction* operator respectively.

6 Parallel Processing with Shared Memory

As discussed in section 5.4, the PACE system cannot be extended to provide consistent parallelization for processing a rule with *earliest selection - selected consumption* parameter context. One method that can be used to parallelize an operator with this parameter context is *merger filtering*, which is discussed in section 5.4.2. In this section, shared memory parallelization, a new framework for parallelization of an operator, executing a rule with *earliest selection - selected consumption* parameter context, is discussed.

Similar to PACE system, this is a data parallelization method. A number of similar operator instances run in parallel. Each instance processes a part of the incoming stream.

6.1 Architecture.

The architecture is similar to PACE system. The nodes in a parallel operator are a splitter, instances, and a merger. The tasks and the procedure of executing the tasks of nodes is different when compared to the PACE system. The instances have a shared memory between them. Figure 6.1 shows an operator with shared memory parallelization.

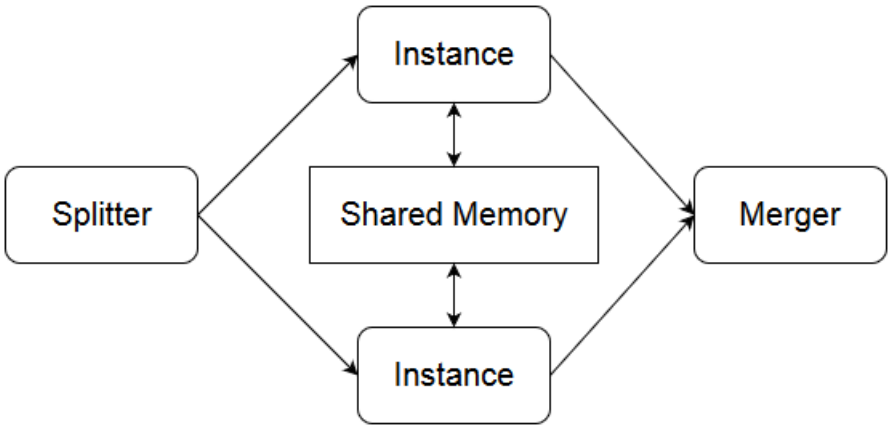

Figure 6.1: Data Parallelization with Shared Memory

Splitter. Splitter connects to all the event sources from which the operator receives events. It receives events from all the event sources and serializes the received events into a single incoming stream I_ω based on the global ordering of the events. Splitter also connects to all the instances that are running in parallel.

Instance. Multiple instances of an operator run in parallel. All the instances have access to a shared memory. An instance accesses the shared memory when processing an incoming event.

Merger. The merger functions in the same way as in the PACE system. It connects to all the operator instances running in parallel. It receives the output events of all the instances and merges them into a single output stream O_ω.

6.2 Event Processing.

An operator, with *earliest selection - selected consumption*, processes the incoming stream I_ω to produce the outgoing stream O_ω in the following way:

1. Determine a selection σ from the incoming stream.
2. Map the incoming events from σ to a single output event as $f_\omega : \sigma \rightarrow e_{out}$.
3. Remove the events in the production set of output event from incoming stream.
4. Goto 1.

As discussed in section 5.4, it is not possible to process the partitions of incoming streams independently in parallel and arrive at consistent output. Therefore, the idea is to process the events from incoming stream in parallel by sharing the state of the instances in a shared memory. Instead of detecting the output event and its production set from a selection of the incoming stream in a single step, the output event is detected by constructing the production set in multiple steps by processing the events from incoming streams one at a time. The state of the partial production sets is saved in the shared memory. Any instance can access a production set from the shared memory.

The production sets in the shared memory are saved in the form of a linked list. The earliest production set in the list is at the head of the list. When a new production set is opened it is added at the end of the list. When an instance accesses a production set it acquires the lock of the production set and releases the lock after the processing of the production set. It is not possible to access a production set in the shared memory without acquiring the lock.

When an instance is free, it sends a request to the splitter for an incoming event. When splitter receives a request from an instance, it sends the earliest event in the incoming stream to the instance and deletes the event from the incoming stream. After receiving an event from the splitter, the instance processes the production sets in the shared memory with the received event. For processing the production sets an instance offers an interface that can be programmed according to the operator functionality. The interface comprises of the following predicates:

- $P_i : (e, \gamma) \rightarrow BOOL$ If the event e is included in the production set γ, then P_i returns true. Else it returns false.
- $P_c : (\gamma) \rightarrow BOOL$ If the production set γ is complete, then P_c returns true. Else it returns false.

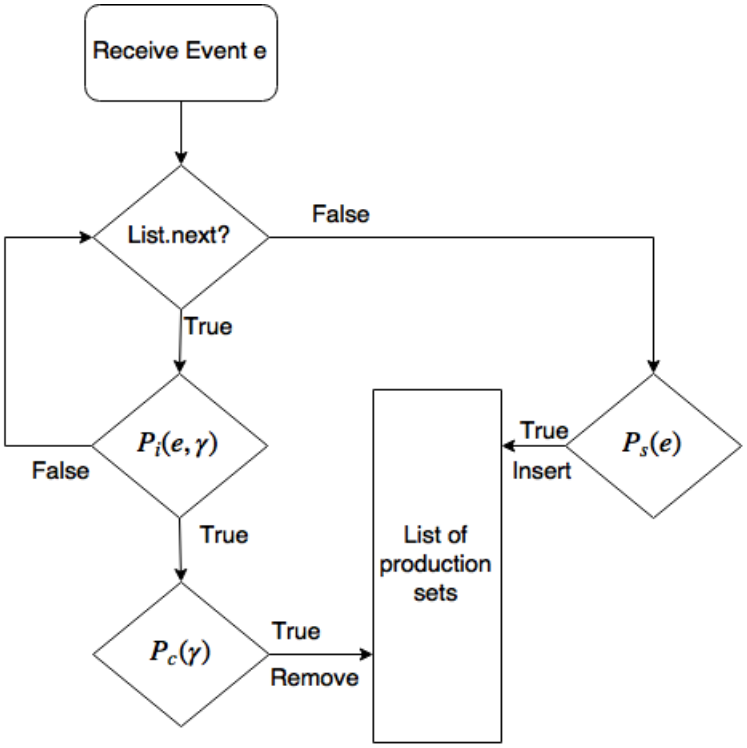

Figure 6.2: Flowchart for event processing with shared memory

- $P_s : e \rightarrow BOOL$ If the event e starts a new production set, then P_s return true. Else it returns false.

The flowchart in figure 6.2 gives the procedure for processing of an event e by an instance.

An instance processes the received event with the production sets in the shared memory, starting from the earliest production set, by executing $P_i(e, \gamma)$. If P_i returns false, the instance executes P_i for the event e with next production set until P_i returns true.

If P_i returns true for a production set, the event e is consumed in that production set and therefore, P_i is not executed for further production sets with this event e. The $P_c(\gamma)$ is executed for the production set γ in which the event e is consumed, to check whether e is the last event of the production set. If P_c returns true for a production set γ, then the output event is detected. The production set γ is removed from the list of production sets in the shared memory. The output event is detected from the production set γ and is forwarded to the merger. The output event forwarded to the merger is attached with the time stamp of the last event in the production set of the output event. The merger uses this time stamp in the output event for ordering of the

events in the output stream.

If P_i, for an event e, does not return true for any of the production sets in the shared memory, $P_s(e)$ is executed. If P_s returns true for an event e, then e starts a new production set. The instance adds this new production set to the end of the list of production sets in the shared memory.

6.3 Synchronization

For consistent detection of output events by shared memory parallelization, synchronization among all the instances is required while accessing the production sets in the shared memory. For proper synchronization the following conditions have to be met:

- When an instance accesses a production set it acquires the lock of the production set and releases the lock after the processing of the production set. It is not possible to access a production set in the shared memory without acquiring the lock.
- If e_1 occurs before e_2 in the incoming stream, then instance I_2, processing event e_2, must access any production set γ_i in the shared memory only after the happening of one of the following two things:
  1. Instance I_1, processing event e_1, has accessed γ_i.
  2. Event e_1 is consumed in a production set.

6.4 Proof of Consistent parallelization

In this section, the proof of consistent parallelization of an operator, with *earliest selection - selected consumption* parameter context, with shared memory parallelization is discussed.

False Positives. Let e_{out} be an output event with production set $(e_1^{i_1}, e_2^{i_2}, \ldots, e_N^{i_N})$, where $e_j^{i_j}$ is the i_j^{th} event of type E_j . For an operator with *earliest selection - selected consumption* parameter context, e_{out} is a positive event, if the following two conditions are met:

1. $\forall 1 \leq j < N \; \forall 1 \leq x_j < i_j \; \exists \; e_{j+1}^{x_{j+1}}$ *such that*

$$e_j^{x_j}.timeStamp < e_{j+1}^{x_{j+1}}.timeStamp < e_{j+1}^{i_{j+1}}.timeStamp$$

2. $\forall 1 < j \leq N$, $\forall x_j < i_j$, *and* $e_j^{x_j}.timeStamp > e_{j-1}^{i_{j-1}}.timeStamp$: the event $e_j^{x_j}$ is consumed in a previous selection.

Let e_{out} is a false positive output event. This implies that, atleast one of the above two conditions is not obeyed. If the first condition is not obeyed, then according to the event processing procedure described in section 6.2, e_{out} cannot be a positive event.

This is because, for the j on which the first condition is violated, $e_{j+1}^{x_{j+1}}$ would have been consumed in a previous production set and hence could not have been part of the production set that resulted in the output event e_{out}. This violates the initial assumption, that e_{out} is an output event. If the second condition is not obeyed, then, for the value of j on which the second condition is violated, $e_j^{x_j}$ could have been part of the production set instead of $e_j^{i_j}$. This violates the initial assumption, that e_{out} is an output event. Therefore, false positive events are not detected.

False Negatives. Let e_{out}, with production set $(e_1^{i_1}, e_2^{i_2}, \ldots, e_N^{i_N})$, be the false negative event. Since it is proved that false positive events are not detected and the events from incoming stream are included in at most one production set, the events in the production set of the false negative event are not consumed in any other production set. This is because, such a consumed event will result in a production set that is not equivalent to production set of any of the positive output events. According to the event processing procedure discussed in section 6.2, if an event e in the incoming stream is not consumed in any of the production sets in the shared memory, $P_s(e)$ is executed. Since e_{out} is a positive event, the first event of its production set $e_1^{i_1}$ returns true for $P_s(e_1^{i_1})$. Therefore, a new production set γ is opened in the shared memory. For all other events $e_j^{i_j}$, the predicate $P_i(e_j^{i_j}, \gamma)$ returns true and $e_j^{i_j}$ is included in γ resulting in the output of event e_{out}. Therefore, false negative events do not occur.

Duplicates. A duplicate output event occurs when there are two output events with the same productions set. Since every event e in the incoming stream is consumed in only one production set, there cannot be two production sets in the shared memory with same set of events. Therefore, duplicate events are not detected.

Ordering. For an operator without parallelization, if selection σ_i is determined before selection σ_j, then output event e_{out}^i mapped from σ_i occurs in the output stream before the output event e_{out}^j mapped from σ_j. if σ_i is determined before σ_j, then the time stamp of the *terminator* event of σ_i is less than the time stamp of the *terminator* event of σ_j. Therefore, the output events are ordered according the time stamp of the *terminator* event of the selection from which the output event is mapped.

As discussed in section 6.2, in the operator with shared memory parallelization, the merger orders the output events according to the time stamps of the last event in the production set. Since the last event in the production set is the *terminator* event of the selection, the output events in the output streams of both the operators, with and without parallelization, are in the same order. Therefore, the ordering of the events in the output stream is preserved.

Algorithm 5 Predicates for *sequence* operator with shared memory

```
1: procedure P_s(Event e)
2:     if e.type == E_1 then
3:         return TRUE
4:     else
5:         return FALSE
6:     end if
7: end procedure
8:
9: procedure P_i(Event e, ProductionSet γ)
10:    if γ.prevType == E_1 then
11:        if e.type == E_2 then
12:            γ.prevType = E_2
13:            return TRUE
14:        end if
15:    else if γ.prevType == E_2 then
16:        if e.type == E_3 then
17:            γ.prevType = E_3
18:            return TRUE
19:        end if
20:    end if
21:    return FALSE
22: end procedure
23:
24: procedure P_c( ProductionSet γ)
25:    if γ.prevType == E_3 then
26:        return TRUE
27:    else
28:        return FALSE
29:    end if
30: end procedure
```

6.5 Example Predicates.

Sequence. Algorithm 5 gives the predicates for a sequence operator $Sequence(E_1, E_2, E_3)$ with three event types E_1, E_2, and E_3. Since E_1 is the initiator event type, the starting predicate P_s returns *true* for an event of type E_1. The predicate P_i returns *true*, if event of type E_2 occurs after event of type E_1 or event of type E_3 occurs after event of type E_2. The predicate P_c returns *true*, if the event of type E_3 is included in the production set.

Conjunction. Algorithm 6 gives the predicates for a conjunction operator $Conjunction(E_1, E_2, E_3)$ with three event types E_1, E_2, and E_3. Since any of the

Algorithm 6 Predicates for *conjunction* operator with shared memory

```
1: procedure P_s(Event e)
2:     if e.type == E_1 OR e.type == E_2 OR e.type == E_3 then
3:         return TRUE
4:     else
5:         return FALSE
6:     end if
7: end procedure
8:
9: procedure P_i(Event e, ProductionSet γ)
10:    if e.type ∈ conjunction.types then
11:        if e.type ∉ γ.prevTypes then
12:            γ.prevTypes = γ.prevTypes ⋃ {e.type}
13:            return TRUE
14:        end if
15:    end if
16:    return FALSE
17: end procedure
18:
19: procedure P_c( ProductionSet γ)
20:    if γ.prevTypes == conjunction.types  then
21:        return TRUE
22:    else
23:        return FALSE
24:    end if
25: end procedure
```

component event types is the initiator event type, the starting predicate P_s returns *true* for an event of type E_1, E_2, or E_3. The predicate P_i returns *true*, if event of type that is not already included in the production set occurs. The predicate P_c returns *true*, if the events of all the component event types are included in the production set.

7 Analysis and Results

In this section, the implementation and the results of evaluation for the parallelization methods proposed in sections 5 and 6 are discussed. The main aim of this evaluation is performance analysis of splitter in partitioning the incoming streams.

7.1 Experimental Set up

Test Environment. The evaluations have been carried out on a cluster of computing nodes with homogeneous capacity. Each node consists of 8 cores(Intel(R) Xenon(R) CPU E5620 @ 2.40Ghz) and 24 GB memory. They are connected by 10 Gigabit Ethernet connections.

Implementation. The components of the parallel CEP, splitter, instances, and merger, are implemented in Java. The other nodes related to the CEP system, sources and consumers, are also implemented in Java.

Experiment. The experiments are performed with a *sequence* operator. The number of component events of the *sequence* operator are varied to see how the performance of the splitter varies with it. The incoming stream of the operator consists of events belonging to the component event types. The *source* sends events at uniform rate to the operator, choosing the type of event randomly from the component event types. For all the results presented in this section, if not specified explicitly, the experiments are conducted with a parallelization degree of four.

7.2 Results

7.2.1 Latest Selection - Zero consumption

Figure 7.1 gives the splitter throughput of an operator with this parameter context. In splitter, the number of opening and closing of partitions per unit time affects the performance. When the number of component events is low, the percentage of *initiator* and *terminator* events in the incoming streams is high. As a result, the number of partitions opened and closed per unit time is high and the throughput is less.

Figure 7.2 gives the instance throughput. In instance, the throughput is affected by the number of output events detected. As mentioned above, the percentage of *initiator* and *terminator* events is high, when the number of component events is low. As a result, high number of output events are detected when the number of component events is low and the instance throughput is less.

7.2.2 Latest Selection - Selected consumption

Figure 7.3 gives the splitter throughput of an operator with this parameter context. Similar to the splitter throughput of the *Latest Selection - Zero consumption*, the splitter

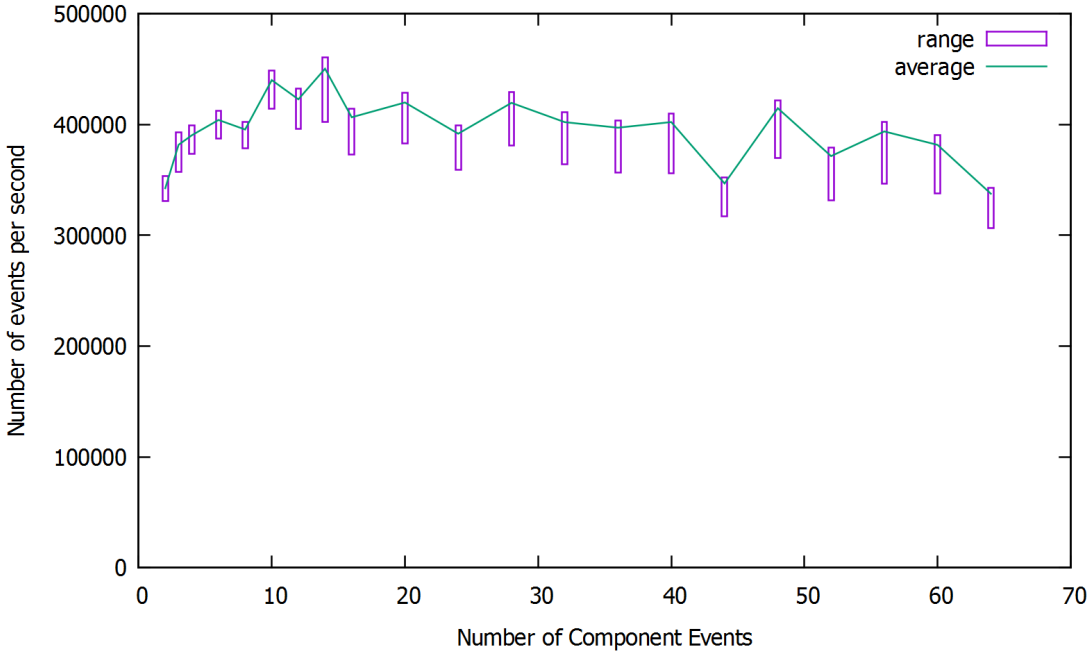

Figure 7.1: Splitter Throughput for Latest Selection - Zero consumption Parameter Context

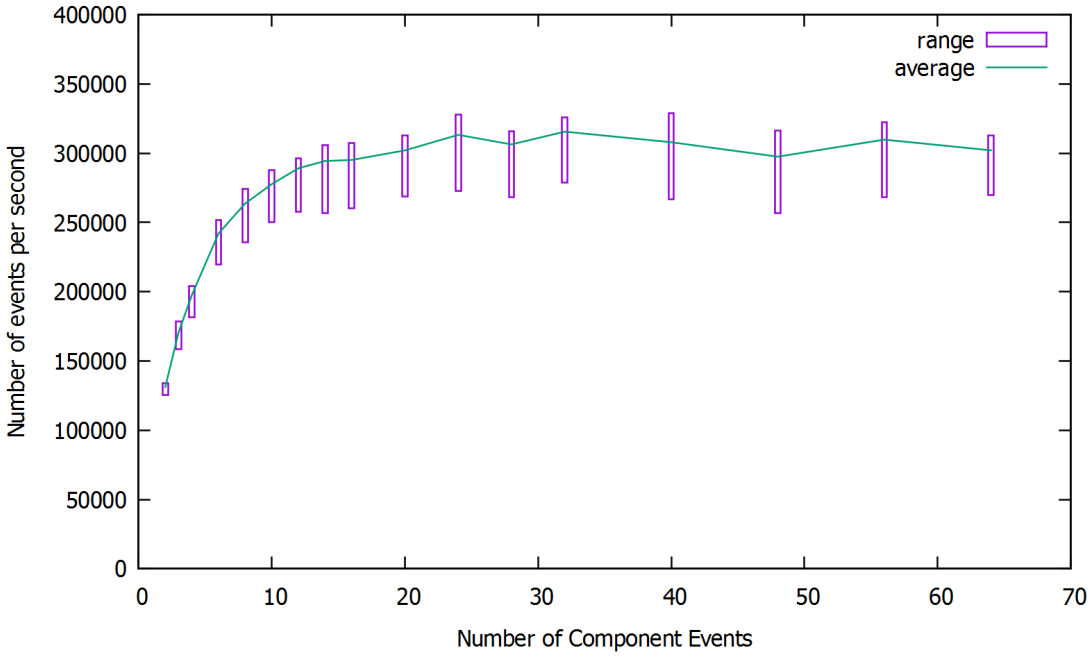

Figure 7.2: Instance Throughput for Latest Selection - Zero consumption Parameter Context

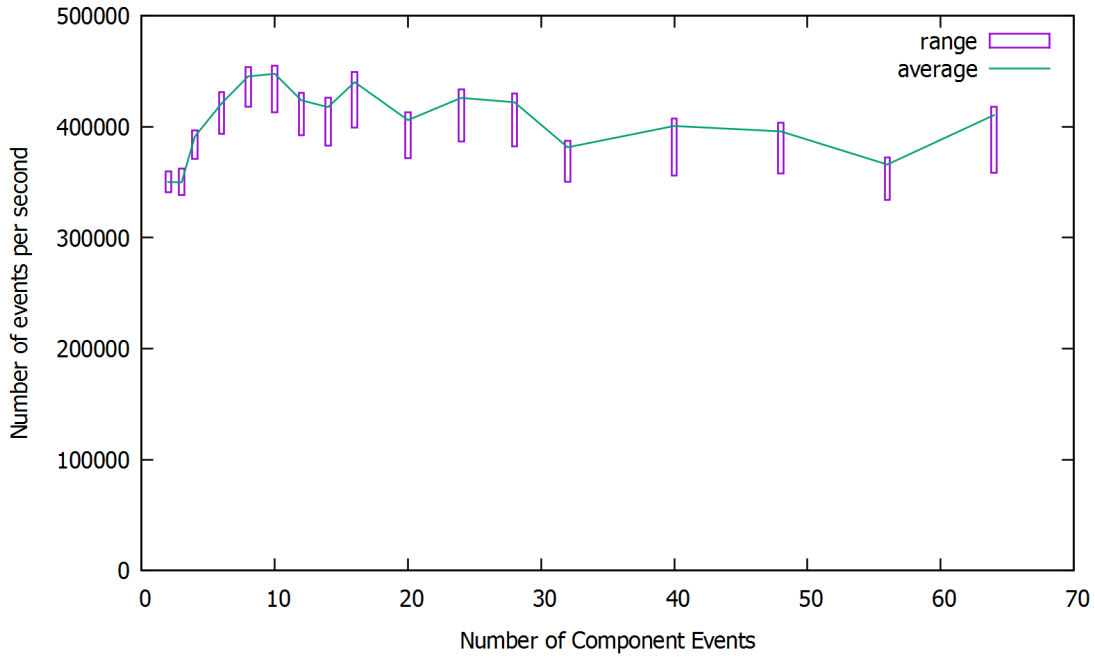

Figure 7.3: Splitter Throughput for Latest Selection - Selected consumption Parameter Context

throughput of this parameter context is low, when the number of component events are less, because of the high number of opening and closing of partitions. Also, similar to the instance throughput of the *Latest Selection - Zero consumption*, the instance throughput of this parameter context is low, when the number of component events are less, because of the high number of detected events.

7.2.3 Each Selection - Selected consumption

As the partitioning procedure and the predicate logic in the splitter with this parameter context is similar to that of the *Latest Selection - Selected consumption* parameter context, the splitter throughput is also similar and is as shown in figure 7.3. Figure 7.5 shows the instance throughput with this parameter context. Since, in this parameter context, all the possible events in a selection are detected, the number of output events detected grows exponentially as the number of component events increases. As a result, the instance throughput decreases exponentially as the number of component events increases. Therefore, for input streams with high event rate, an operator with this parameter context, can be practically implemented only for low number of component events.

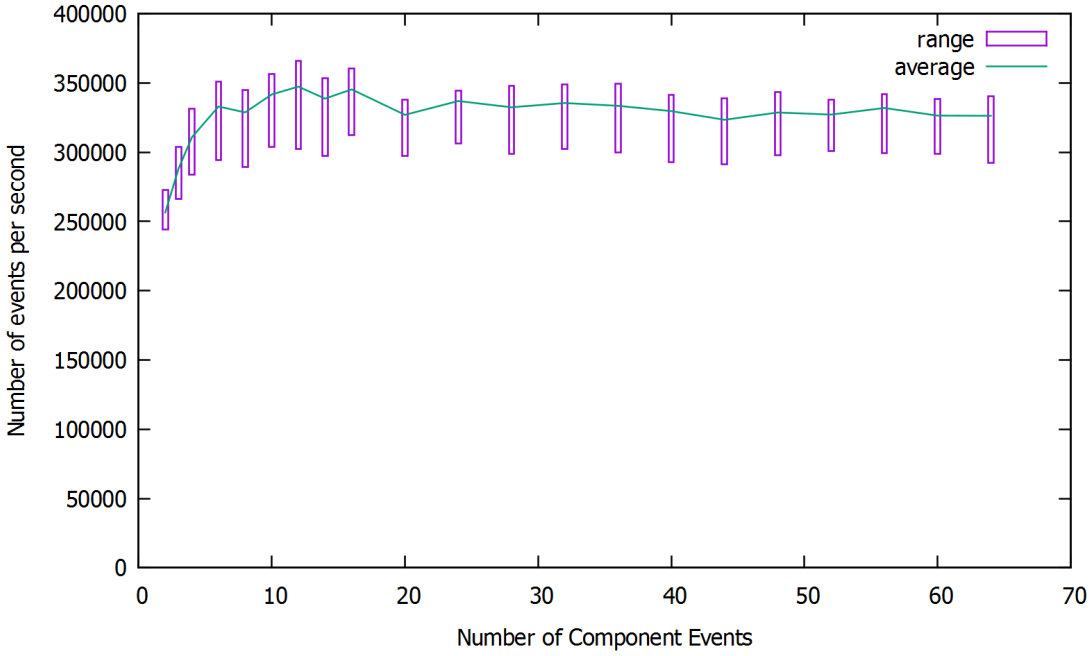

Figure 7.4: Instance Throughput for Latest Selection - Selected consumption Parameter Context

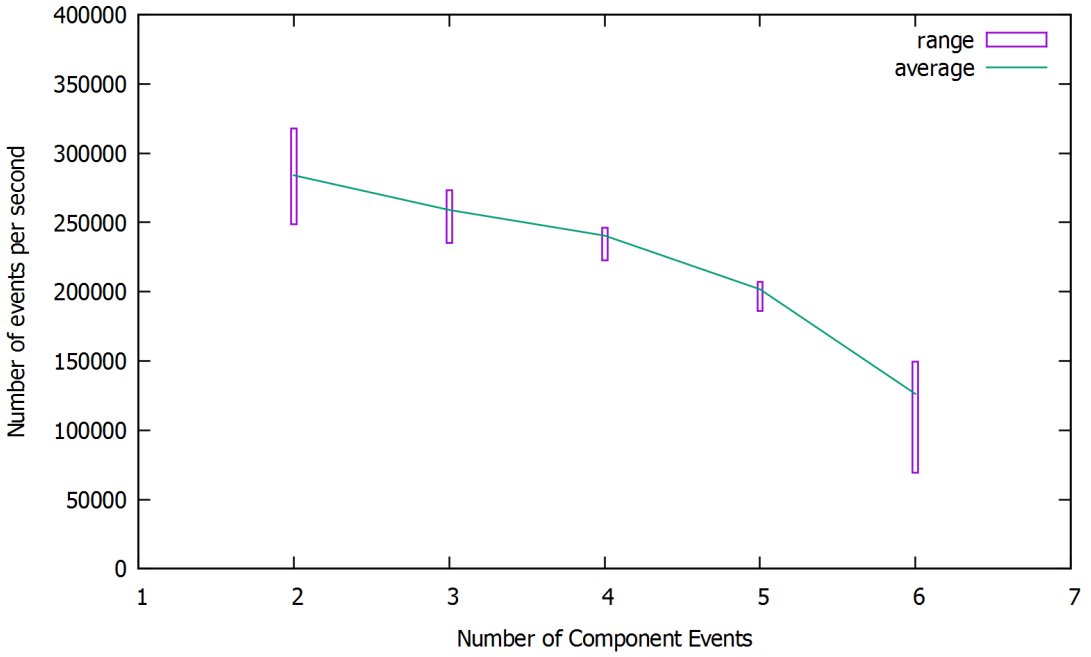

Figure 7.5: Instance Throughput for Each Selection - Selected consumption Parameter Context

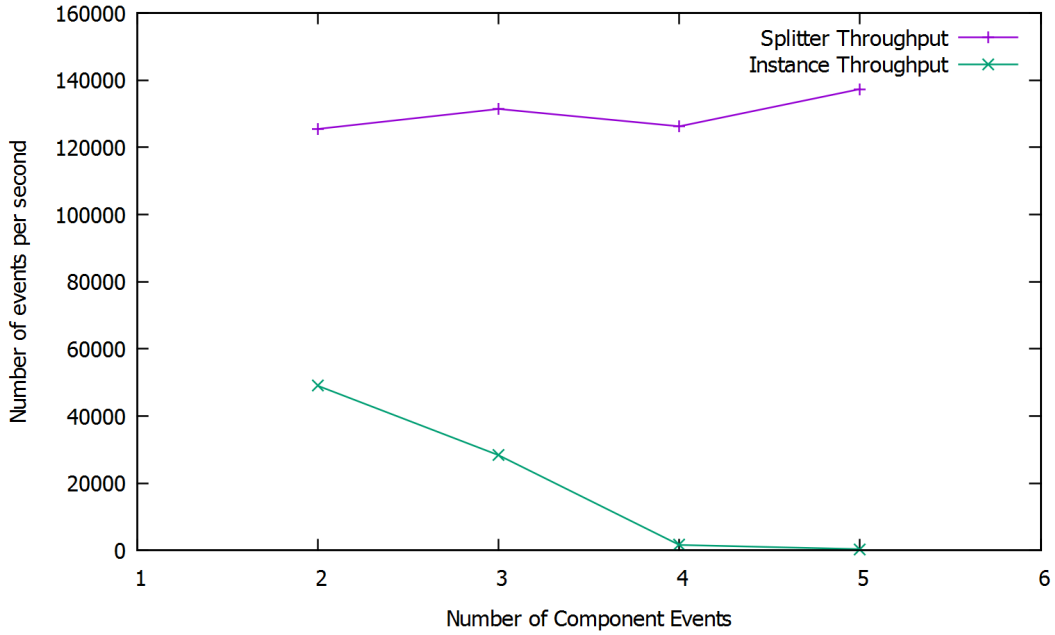

Figure 7.6: Throughput of the Instance and Splitter with Merger Filtering

7.3 Earliest Selection - Selected Consumption

7.3.1 Merger Filtering

Figure 7.6 shows the throughput of splitter and instance with merger filtering. As the number of component events increases, the splitter throughput decreases exponentially. This is because of the exponentially increasing number of events detected, as the number of component events increases. Figure 7.8 shows the percentage of the positive events that are forwarded on the output stream out of the total events detected by the instances. The percentage of the positive events is close to zero for operators with more than 3 component events. this low percentage of positive events results in the low rate of positive output events at the merger. This can seen in figure 7.7. In figure 7.7 the throughput of merger for all the detected events increases with the number of component events. This is because, the processing of detected events at the merger involves set comparison operation between consumed events set and production set of detected events. The set comparison operator is expensive when the two sets are disjoint. When the number of component events is low, the percentage of the positive events is high. This means, with low number of component events, the result of high number of set comparisons is disjoint. Therefore, the throughput is low.

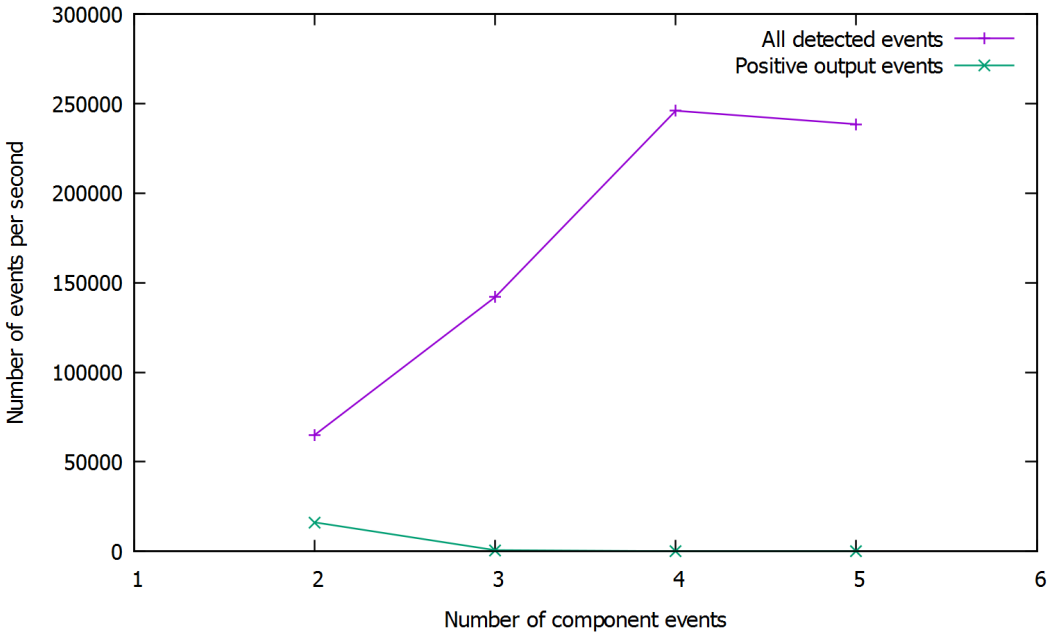

Figure 7.7: Throughput of the Merger with Merger Filtering

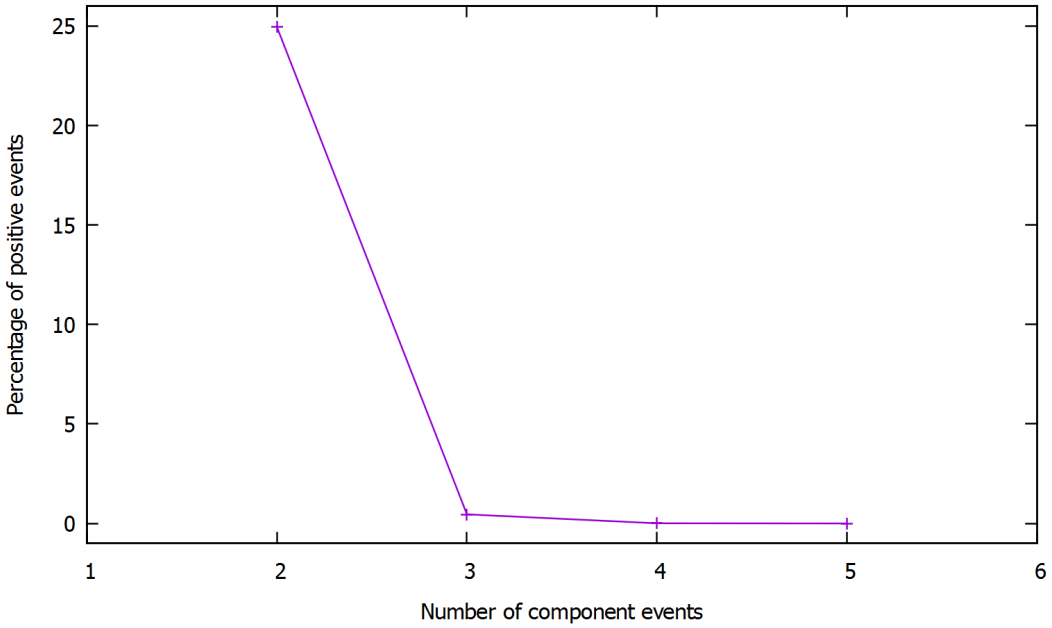

Figure 7.8: Percentage of Positive events at the Merger with Merger Filtering

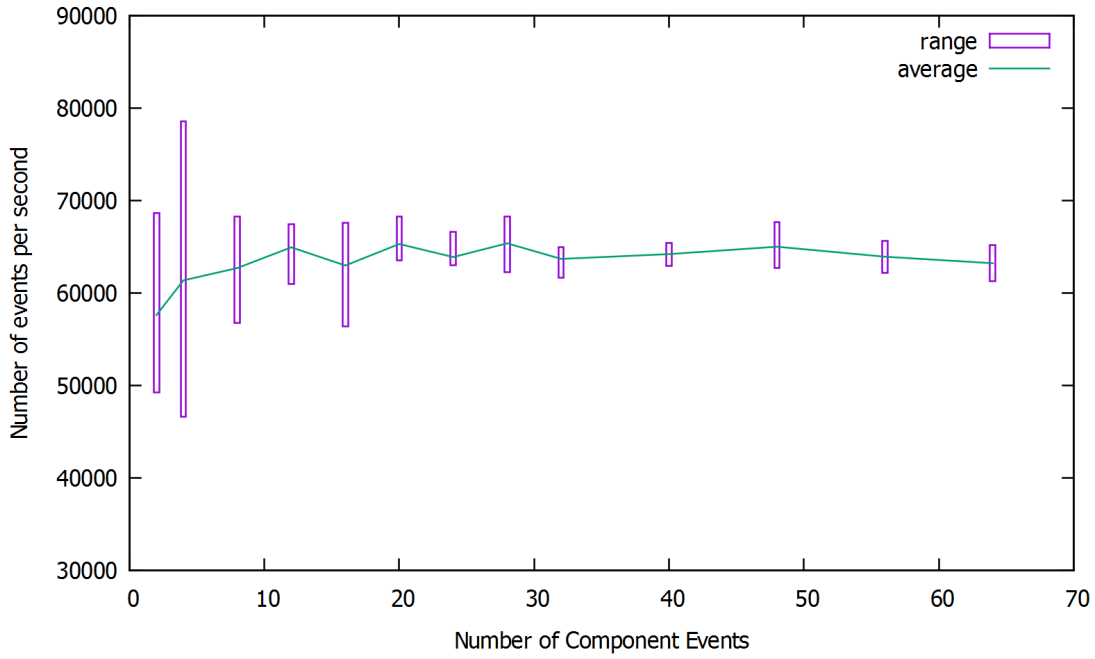

Figure 7.9: Throughput of the Sequence operator with shared memory parallelization with parallelization degree eight

7.3.2 Shared Memory

Figure 7.9 shows the throughput of the *sequence* operator with shared memory parallelization with parallelization degree of eight. The throughput increases slightly initially and is stable later as the number of component events in the operator increases. The throughput is low when the number of component event is less. This is because, when the number of component events are less, the percentage of *initiator* and *terminator* events in the input stream is high. As a result, the production sets are opened and closed with in a short duration of time and there will be less number of open partitions in the shared memory at any point of time. This results in the blocking of processes as the processes try to access the production sets in the shared memory. Therefore, this blocking results in low throughput. As the number of component events increases, the number of open production sets in the shared memory increases. Therefore, the probability for blocking of processes decreases and throughput is stable. Figure 7.10 shows the throughput of the *sequence* operator with various parallelization degrees of two, four, six, and eight. The throughput of the operator increases steadily as the number of operator instances increases. It can be seen that, when the number of component events is two, the throughput for operator with a parallelization degree eight is less than the throughput of operator with parallelization degree four and six. This is because of the high probability of blocking of the instances, when parallelization degree is high and number of component events is small, in accessing the shared memory.

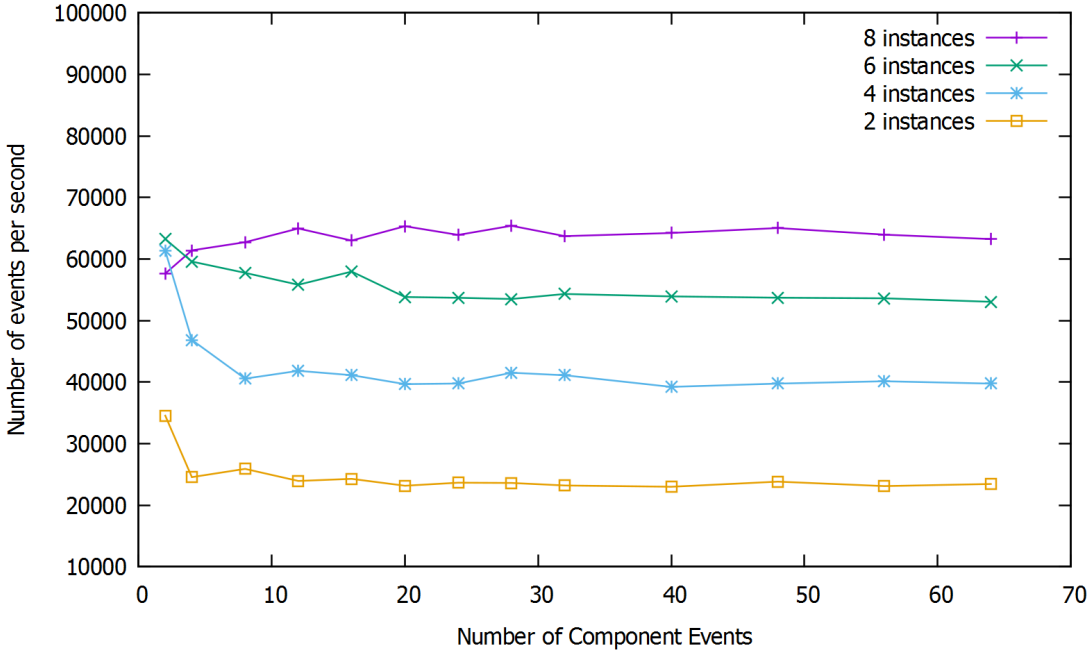

Figure 7.10: Throughput of the operator with shared memory parallelization with various parallelization degrees

8 Conclusion

8.1 Summary

The focus of this thesis is on prallelization of an operator in a distributed CEP system. Specifically, the thesis presented techniques for operator parallelization, when the rules are specified with a parameter context. The form of parallelization discussed in this thesis is data parallelization. In data parallelization approach the incoming events of an operator are divided into partitions, which are processed by instances of the operator running in parallel. There is an existing framework for such an operator parallelization, called PACE system. The PACE system identifies the partitions by using a couple of predicates to determine the start and end of partitions. These partitions can programmed according to the operator, allowing the PACE system to support a wide class of operators.

This thesis analysed the parameter context of the PACE system to be *Each Selection - Zero consumption*. It analysed, if the PACE system can be adopted to parallelize an operator with other parameter contexts. This analysis is done for four general parameter contexts : *Latest Selection - Zero consumption*, *Latest Selection - Selected consumption*, *Each Selection - Selected consumption*, and *Earliest Selection - Selected consumption*.

For the first three of the four above mentioned parameter contexts, it is found that the PACE system can be adopted with certain modifications. The modifications to the PACE system, the logic to program the predicates, and the proofs that such modifications provide consistent operator parallelizaiton are discussed. It is also analysed, by experimental evaluaiton, how the throughput of such a parallelization technique varies, when the number of component events in the operator are increased.

In the case of the *Earliest Selection - Selected consumption* parameter context, it is shown that the predicate logic cannot be successfully adopted to partition the incoming stream. For operator parallelization with such a parameter context, two methods are proposed: merger filtering and shared memory approach.

In the merger filtering method, the architecture of the PACE system is retained. In this approach, the instances of the operator detect events in such a way that no false negatives occur. Note that the false positives may occur. The detected events from all the parallel instances are merged into a single stream and the false positives are filtered and discarded. The proof that the merger filtering approach provides a consistent parallelization is discussed. In the experimental evaluation of the merger filtering approach it is found that, the operator throughput is poor and does not scale well with the increase in the number of component events of the operator.

In the shared memory approach, all the instances running in parallel have access to

a shared memory, where the state of the event processing is stored. In this approach, instead of processing the partitions of the incoming streams, each instance processes one event of the incoming stream at a time updating the state in the shared memory after processing each event. The experimental evaluation with the *sequence* operator shows that the throughput is stable when the number of component events in the operator are increased. The operator throughput also increases steadily with the increase in the number of parallel instances.

8.2 Future Work

In this thesis, the work on operator parallelization is done only for the general parameter contexts. Various other parameter contexts can be defined with selection and consumption policies that are customized for a specific real world application. For example, a new selection policy can be defined with the combination of the two or more selection policies discussed in this thesis. Similarly, a new consumption policy can be defined, where only a subset of the events included in the production set are consumed. Work can be done in the future for operator parallelization with such parameter contexts. In this thesis, the discussion and analysis of shared memory approach is done only for parallelizing operators with *Each Selection - Selected consumption* parameter context. In the future, work can be done on extending the usage of this approach for other parameter contexts. The evaluation of merger filtering approach for operator parallelization with *Each Selection - Selected consumption* parameter context showed, that it is not suitable for practical application because of the poor throughput rate. Work can be done in the future to see if this approach gives a practically viable performance for operator parallelization with other parameter contexts. All the experimental evaluations are conducted with a single *sequence* operator and a synthetic input stream. In the future, experiments can be conducted by implementing a network of operators that depicts a real world situation and an input stream with data acquired from real world scenario.

References

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

[2] Asaf Adi and Opher Etzion. Amit - the situation manager. *The VLDB Journal*, 13(2):177–203, May 2004.

[3] Yijian Bai, Hetal Thakkar, Haixun Wang, Chang Luo, and Carlo Zaniolo. A data stream language and system designed for power and extensibility. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management*, CIKM '06, pages 337–346, New York, NY, USA, 2006. ACM.

[4] Cagri Balkesen, Nihal Dindar, Matthias Wetter, and Nesime 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, pages 3–14, New York, NY, USA, 2013. ACM.

[5] Krysia Broda, Keith Clark, Rob Miller, and Alessandra Russo. Sage: A logical agent-based environment monitoring and control system. In Manfred Tscheligi, Boris de Ruyter, Panos Markopoulus, Reiner Wichert, Thomas Mirlacher, Alexander Meschterjakov, and Wolfgang Reitberger, editors, *Ambient Intelligence*, volume 5859 of *Lecture Notes in Computer Science*, pages 112–117. Springer Berlin Heidelberg, 2009.

[6] S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data Knowl. Eng.*, 14(1):1–26, November 1994.

[7] Gianpaolo Cugola and Alessandro Margara. Tesla: A formally defined event specification language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, pages 50–61, New York, NY, USA, 2010. ACM.

[8] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.

[9] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. Towards expressive publish/subscribe systems. In *Proceedings of the 10th International Conference on Advances in Database Technology*, EDBT'06, pages 627–644, Berlin, Heidelberg, 2006. Springer-Verlag.

[10] Daniel Jobst and Gerald Preissler. Mapping clouds of soa- and business-related events for an enterprise cockpit in a java-based environment. In *Proceedings of the 4th International Symposium on Principles and Practice of Programming in Java*, PPPJ '06, pages 230–236, New York, NY, USA, 2006. ACM.

[11] Boris Koldehofe, Ruben Mayer, Umakishore Ramachandran, Kurt Rothermel, and Marco Völz. Rollback-recovery without checkpoints in distributed event processing systems. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 27–38, New York, NY, USA, 2013. ACM.

[12] Xiangsheng Kong. The study of rfid system based on cep. In *Consumer Electronics, Communications and Networks (CECNet), 2012 2nd International Conference on*, pages 1477–1480, April 2012.

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

[14] R. Mayer, B. Koldehofe, and K. Rothermel. Predictable low-latency event detection with parallel complex event processing. *Internet of Things Journal, IEEE*, 2(4):274–286, Aug 2015.

[15] Roy Schulte. An overview of event processing software. http://www.complexevents.com/2014/08/25/an-overview-of-event-processing-software/.

[16] S. Srinivasagopalan, S. Mukhopadhyay, and R. Bharadwaj. A complex-event-processing framework for smart-grid management. In *Cognitive Methods in Situation Awareness and Decision Support (CogSIMA), 2012 IEEE International Multi-Disciplinary Conference on*, pages 272–278, March 2012.

[17] Sriskandarajah Suhothayan, Kasun Gajasinghe, Isuru Loku Narangoda, Subash Chaturanga, Srinath Perera, and Vishaka Nanayakkara. Siddhi: A second look at complex event processing architectures. In *Proceedings of the 2011 ACM Workshop on Gateway Computing Environments*, GCE '11, pages 43–50, New York, NY, USA, 2011. ACM.

Declaration

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

---

place, date, signature