A Novel Engine: Complex Event Processing with LAIPE

Dong Wang[1], Mingquan Zhou[1, 2*], Sajid Ali[3], Yusong Liu[4], Pengbo Zhou[2]

[1] School of Information Science and Technology, Northwest University, Xi'an 710127, China.
[2] College of Information Science and Technology, Beijing Normal University, Beijing 100875, China.
[3] Department of Computer Scinece, University of Education, Lahore 60000, Pakistan.
[4] School of Information, Renmin University of China, Beijing 100872, China.

* Corresponding author. Tel.: +8602988302950; email: mqzhou_nwu@yahoo.com
Manuscript submitted January 22, 2016; accepted April 14, 2016.
doi: 10.17706/jcp.12.4.317-334

Abstract: Most of the Complex Event Processing (CEP) engines are responsible for filtering, selecting and aggregating atomic events to represent high level composite events according to predefined rules are the latest breed of event-oriented software, which is being presented nowadays to better support timely information processing. Introducing CEP engines to the information system domain provides an opportunity to enhance the capabilities of event processing in real-time. To answer this need, we first propose an event algebra that defines the detailed semantics of event models and operators. More specifically, we propose a Lightweight Automata based Incremental Processing Engine (LAIPE) based on a layered architectural design. On the one hand, it adopts a language (LAIPL) and its rule structure which explicitly conceived to natural and easy identification of complex events. On the other hand, it provides efficient algorithms for rule translation and event incremental processing approach based on event automaton model. Finally, our evaluation on LAIPE and existing Esper engine shows that the time cost of LAIPE is averagely reduced 42.3% than Esper in different tests, and our developed engine has better performance in processing different sliding window and selectivity rates with a large number of events.

Key words: Complex event processing, light-weight automata based incremental processing, event automaton, event processing engine.

1. Introduction

In the last decade, the complexity of information systems has increased as well as the systems have evolved from synchronous to asynchronous [1]. It is due to rapid development of information technology, and the business environments become more and more dynamic. As today's integrated systems produce huge amount of information that is often relevant in only a short period of time, the real-time identification of meaningful data from primitive events is becoming more important [2]. Therefore, the need for an efficient, high performance, and timely information processing engine is obvious.

To obtain a global view of a situation based on detected events we use Complex Event Processing (CEP), which is becoming more and more popular because it enables derivation of higher-level information from streams of simple events [3]. The general architecture of such CEP engine is shown in Fig. 1. In general CEP engine usually processes input events according to rules and detects different situations as composite events. Rules and patterns are defined by system administrator in build time, they are components of CEP engine for event process. A CEP engine follows three basic steps, (1) Input events generating from different event sources are captured by CEP engine (Run time); (2) Rules and patterns are defined by user (Build time) which are implemented in CEP engine to process events and detect meaningful data (Run time); (3) Composite events are generated immediately when rules are matched (Run time).

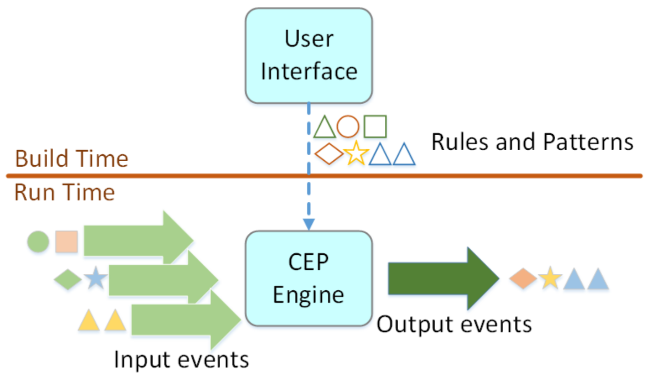


Fig. 1. The general architecture of a CEP engine.

Both from the academia and industry, several CEP engines with accompanying event processing language have been proposed in the last few years [4], such as Esper [5], Cayuga [6], NextCEP [7], T-Rex [8] and so on. However, there is a boundary separated business processes systems on one side while existing CEP engines on the other side [9]. As a matter of fact, the performance of most traditional CEP engines is limited when facing event processing in information system domain. Particularly, they focus on the language semantics and try to embed CEP capabilities into a middleware, ignoring the interoperability, extensibility and modularity requirements as well as high performance.

We aim to build a lightweight, layered CEP engine architecture that can be seamless integrated with information systems. Specially, the need for a high performance CEP engine that can be used to deal with a large number of input events in integrated systems is obvious. Starting from these premises, we developed Lightweight Automata based Incremental Processing Engine (LAIPE), a new CEP engine for processing huge number of events in order to gain high performance. In this paper, we propose an event algebra for event definition, operation and expression in information system domain, as well as Lightweight Automata based Incremental Processing Language (LAIPL) for rule definition. We will also show the design, implementation and evaluation of LAIPE that focus on event processing approach optimization base on traditional methods.

In summary, the main contributions of this paper are:

(1) Our event algebra defines the semantics of both atomic and complex events, which allows easy adaptation to information system needs. It has been used to describe event aggregation in our CEP engine implementation.

(2) A high-level event processing language LAIPL and rule structure based on our event algebra, which have been used to define event composition constraints in our CEP engine.

(3) A Lightweight Automata based Incremental Processing (LAIP) approach has been proposed by using the event automaton model for real-time event processing.

(4) We show the design and implementation of Lightweight Automata based Incremental Processing Engine (LAIPE) as well as its layered architecture.

The remainder of this paper is structured as follows. We give an overview on related work with a discussion of some CEP engines in Section 2. Section 3 first defines basic formal event-related concepts. We briefly introduce event model, operation, and expression. While Section 4 shows the defined SQL-like event processing language LAIPL and its rule structure. Section 5 presents our event processing algorithms, emphasizing its advantages in regard to the event automaton model. In Section 6, we describe how the LAIPE implements such automaton to efficiently process input events. Section 7 compares the performance of our CEP engine to Esper. Finally, Section 8 holds concluding remarks for our work.

2. Related Work

A range of CEP engines have been designed to detect events using different methods. In this section, we briefly show an overview of previous work and some classical engines, particularly focusing on the event processing approach. Automata-based processing method is adopted by some CEP engines. Engines translate rules into automata to provide efficient capability of event detection. Most commonly CEP engines use the automata-based method to achieve CEP technology.

Cayuga [6], [10], [11] is designed and implemented to process event by Cornell Database Group. Cayuga is scalable and expressive to detect a large number of complex patterns in event streams. A Cayuga Query Engine executes state transitions of Cayuga automata which is transformed from event patterns. To improve the performance, Cayuga uses indexing and memory management techniques.

The authors of [7] describe a distributed event processing system called NextCEP. The CEP patterns in the system are detected through an expressive automaton-based approach. To illustrate its scalability and efficiency, NextCEP is used for credit card fraud detection.

T-Rex [8] is a new CEP middleware used to combine efficiency and expressiveness specifically designed to real-time process a large number of events as they flow from outside to center of system, recognizing complex events relevant for the system. It adopts an event processing language (TESLA) explicitly conceived to easily and naturally describe composite events, as well as providing an efficient event detection algorithm based on automata to illustrate TESLA rules.

Unfortunately, each specific CEP engine has its very own development language, query language and functional interfaces. To overcome this problem, some researchers choose Java (the most popular programming language in 2015 [12]) to build open source CEP engine as well as defining some standards in CEP technology.

The Lightweight Stage-based Event Processor (LiSEP) is a general-purpose java-based CEP engine [13]. It is based on a layered architecture, whose design clearly separates the core logic devoted to event processing from low-level thread management handled by the Staged Event-Driven Architecture (SEDA) framework. It is used in a dangerous goods monitoring during maritime transport, called SITMAR, to demonstrate the availability of LiSEP.

Esper [5] is considered the leading open-source CEP provider. It supports the essential features of typical CEP systems like sliding temporal windows, event pattern operators, and external method calls. Epser adopts Event Processing Language (EPL) to express filtering, aggregation and joining, over multiple event streams, while the language is used to define SQL-like rules on different types of events. It has been deployed in some systems such as M2M system [14], failure prediction system [15], and monitoring system [16].

In summary, most CEP engines used in previous work are not to expose the optimization of event processing method in CEP engine, but rather focus on language semantics and performance aspects and often try to embed CEP capabilities into a middleware, thus resulting in a big proprietary package [9]. The work described here is a result of our previous experience in event processing method. In the following sections we discuss how our event processing method has been designed and optimized in order to improve performance of CEP engine in information system domain.

3. Event Algebra

This section devoted to the base event algebra for our event processing method. It first describes the basic event-related concepts in the field of information system. Then the event operation and expression are introduced.

3.1. Event Model

Firstly, we give the semantics of our events, according to the information system domain, and show their characteristics. David Luckham and Roy Schulte [17] gave a definition of an event, it is anything that happens, or is contemplated as happening. We consider a number of objects of interest in an information system, e.g., a user access system, user information submission, CPU usage of server, memory usage of server, and network traffic. In information system, each event is correlated to an object as well as each object has a state at a certain time, e.g., the current access user number or the information of a submission.

In our engine, we distinguish atomic events (ae) and complex events (ce), it is similar with the definition in [18]-[20]. Atomic events are single events which occur at a particular point of time. While complex events are formed by combining atomic and possibly complex events. The event models of atomic and complex events are in the following. Moreover, we define two special events: ε is the empty event (it is an event without value) and the failed-detection event Ø [7]. Finally, we use e to indicate all events include atomic events and complex events.

Definition 1 Atomic Event. The atomic event is represented by a four tuple, where Ŧ is the type of event, S is the source (information system) of event, t is the timestamp of event, while Λ is the value of event.

Atomic event: AE={ Ŧ, s, t, Λ}

As an example, e^{user_number} shows the current access user number is 89 in system A. It has been formed in the atomic event model:

e^{user_number}: { user_number, system A, 20151029 15:08.236, 89}

Definition 2 Complex Event. The complex event is also represented by a four tuple. However, as it is formed by atomic events and possible complex events, t is a time period that starts from t_{begin} and end at t_{end}, while Λ is a set of event value.

Complex event: CE={Ŧ, s, $t\{t_{begin},\ t_{end}\}$, Λ {$\Lambda_1,\ \Lambda_2,\ \cdots$}}

In fact, complex event model is a transformation of atomic event model. t of complex event can also be denoted as t{$t_{begin},\ t_{end}$}, where $t_{begin} = t_{end}$. And $\Lambda = \Lambda_1$ while Λ_2=null, Λ_3=null, $\cdots$.

Definition 3 Event Type. The set of all events with a same type is called the event type Ŧ. Event type shows the object which triggers the event.

$\mathbf{E}^{Ŧ} = \{E^{Ŧ1},\ E^{Ŧ2}, \ldots,\ E^{Ŧn}\} = \bigcup_1^n E^{Ŧi}$

Events in an event type have same attribute of Ŧ. For instance, the type may refer to events regarding memory usage, however, the other event properties may be different (e.g., information system or value). An example is the type of all events that denote the memory usage. E^{mem} means this event type indicates memory usage and constituted by $e_1^{mem},\ e_2^{mem}, \ldots,\ e_n^{mem}$.

Definition 4 Event space. The set of all possible events known from a certain information system is called the event space $\tilde{E}$.

$\tilde{E} = \{E^{Ŧ1},\ E^{Ŧ2}, \ldots,\ E^{Ŧ\tau}\} = \bigcup_1^\tau E^{Ŧx}$

$= \{ae_1, ae_2, \ldots, ae_n, ce_1, ce_2, \ldots ce_m\} = \{\bigcup_1^n ae_i,\ \bigcup_1^m ce_j\}$

The event space is formed by all the sets of event type, as well as all the atomic events and complex events. For example, $\tilde{E}^{portal}$ represents all the events that occurred in portal. It can be illustrated as $\tilde{E}^{portal} = \{e \mid \forall e, e.s = \text{"portal"}\}$.

3.2. Event Operation

Event operation is used to show the relationships among correlated atomic and complex events. This section describes the event composition and basic operators for composite events.

Definition 5 Event Composition. It is denoted by $\rightarrow$. Assume that e is contributed by events $e_1, e_2, \ldots, e_n$. The relation is showed as $\{e_1, e_2, \ldots, e_n\} \rightarrow e$.

Event composition illustrates a new complex event that is composed by all contributing events while inherit the characteristics of them. After compositing, the attributes of the new event is according to the contributing events and composition operator. Different operators lead to different complex events.

Annika Hinze and Agnès Voisard [21] proposed event instance consumption that contains keeping matched events, consuming matched events, consume and repeat of the filtering events. This definition is in consideration of all the domains of event processing. However, in the domain of information system, event consumption is the consuming matched events model, that means each event only takes part in one composite event of an event type.

Adopting the consuming matched events model, we define the following event processing operators that can be used to query a complex event, they are inherited from [22]-[24]. In the following, E^{i} represents an event type.

Definition 6 Disjunction Operator ∨. The disjunction operator selects any event that occurs if either $e_1 \in \mathrm{E}^1$ or $e_2 \in \mathrm{E}^2$ occurs. We define the composite event $e_3 \in \{\mathrm{E}^1 \vee \mathrm{E}^2\}$, the occurrence time of e_3 depends on which event occurs. If $\{e_1\} \rightarrow e_3$, then $e_3.t = e_1.t$, and if $\{e_2\} \rightarrow e_3$, then $e_3.t = e_2.t$. There is an exceptional case that E^1 and E^2 are both occurred, the result will be two composite events, $\{e_1\} \rightarrow e_3$ and $\{e_2\} \rightarrow e_4$.

Definition 7 Conjunction Operator ∧. The conjunction operator outputs the events occur if both input events $e_1 \in \mathrm{E}^1$ and $e_2 \in \mathrm{E}^2$ occur $(e_1 \neq e_2)$. Conjunction operator generates the composite event without order of input events. The composite event $\{e_1 \wedge e_2\} \rightarrow e_3$, its occurrence time is $e_3.t_{begin}=\min\{e_1.t,\ e_2.t\}$ and $e_3.t_{end}=\max\{e_1.t,\ e_2.t\}$.

Definition 8 Negation Operator. The negation operator means that the event does not exist in the input events. ¬E means that no e∈E occurs. It usually needs to be used together with other operators.

Definition 9 Sequence Operator;. The sequence operator selects a given sequence of events from the input event sequence. $E^1;E^2$ is a given sequence that occurs when first $e_1 \in E^1$ occurs and then $e_2 \in E^2$ occurs. The composite event e_3 is represented by $\{e_1;e_2\} \rightarrow e_3$. The time of e_3 is $e_3.\mathrm{t}_{begin}= e_1.\mathrm{t}$ and $e_3.t_{end}= e_2.t$.

Definition 10 Selection Operator $e_{[i]}$. It defines the occurrence sequence of the *ith* event in the event set of $E = \{e_1,\ e_2,\ \ \{\mathrm{nte}_N\}$, where $i \in N$ (N is the set of natural number). The parameter can be changed to first and last. $e_{[first]}$ represents the first event in the event set E, while $e_{[last]}$ denotes the last event in E.

Moreover, we employ temporal and batch operators to provide a spicy sliding window capability in our event algebra, which contains time and batch window.

Definition 11 Temporal Operator ⊙. The temporal operator confines the temporal boundary for a composite event. If we define $\{\mathrm{E}^1;\mathrm{E}^2\} \rightarrow \mathrm{E}^3$and the temporal boundary within t, it is denoted as $\{\mathrm{E}^1;\mathrm{E}^2\} \odot t$, that means $\mathrm{E}^3=\{e_3\ |\ \exists e_1 \in \mathrm{E}^1\ \exists e_2 \in \mathrm{E}^2\ \exists e_3 \in \mathrm{E}^3\ \{e_1;\ e_2\} \rightarrow e_3\ \wedge (e_3.t_{end}\text{-}e_3.t_{begin}) \leqslant t\}$.

Definition 12 Batch Operator @. It defines the event batch size of a selection. We define $\{e_i;\ e_j\} \rightarrow$e and the event batch size is 10, it can be represented by $\{e_i;\ e_j\}$@10, the expression of e means $e=\{e\ |\ \exists e_i,\ \exists e_j\ \{e_i;\ e_j\} \rightarrow e \wedge (j\text{-}i) \leqslant 10\}$.

Since the event operators explanation is similar to Wen Yao illustrates in [25] is not our focus, we only give the definition of base event operators. We consider how to propose an event algebra that can be used to improve CEP capability in information system domain. The following subsection shows how to use these event operators to express event relationships.

3.3. Event Expression

In real integrated systems, it contains hundreds of thousands of web applications with a large number of users. The most common events in the system, such as CPU/memory usage of server, application response time, current access user number, are represented as different event type in our event algebra. Now we show the event expression of the newly proposed event model and operators to the example profiles.

E1: A CPU usage too high event occurred in server S001: Let $E_{S001}^{CPU\_Usage}=\{e_1;\ e_2;\ \dots;\ e_n\}$ be the type of CPU usage events in server S001. Then we have to observe the composite event e that $\{e_1;\ e_2;\ \dots;\ e_n\}\odot$ 5min$\rightarrow e$ and $e.\Lambda=(\sum_1^n e_i.\Lambda)/n$. If $e.\Lambda$ is greater than HIGH_USAGE (a predefined constant), than a CPU usage too high event is created.

E2: User saturation in an application: Let $e_1 \in E_{App}^{CPU\_Usage\_High}$ be the type of CPU usage high events. Let $e_2 \in E_{App}^{User\_Number}$ be the set of current user number event type. The user saturation event occurs when $E_{App}^{CPU\_Usage\_High}$ and $E_{App}^{User\_Number}$ both occur during 300 events. A simplified definition for the composite event e could be expressed as $\{e_1 \wedge e_2\}@300\rightarrow$e.

E3: Application runs slowly: Let $e_1 \in E_{App}^{CPU\_Usage\_High}$ be the type of CPU usage events and $e_2 \in E_{App}^{MEM\_Usage\_High}$ be the type of memory usage events. Let $\{e_3, e_4, \dots, e_{52}\} \in E_{App}^{Response\_Time\_Long}$ be the set of events that means application response time is long. The response time is related to the CPU and memory usage of application server. So the composite event of application response time long is shown as $\{(e_1 \vee e_2);\bigcup_3^{52} e_i\}\odot$30min$\rightarrow e$.

As the examples of event expression described above, different composite events (complex events) are conceivable to illustrate by the proposed event algebra. We will use the algebra to denote the relationship of events, which can leverage CEP to optimize the event processing method in integrated systems.

4. An Event Processing Language and Rule

High-level Event Processing Language (EPL) used to describe event rules and constraints plays a crucial role in event processing engines. An expressive EPL should be able to flexibly denote all the constructs (composition, disjunction, conjunction, negations, sequences, selection, batch and temporal operations) and often grants benefits in definition and executing. The most representative languages are Cayuga Event Language [6], SASE [26], Esper language [27] and so on. However, they are different from each other and adopted by different CEP engines. To optimize the event processing method by developing a new CEP engine, first of all, we need to present a dedicated declarative and specially designed language to meet the requirements of event processing.

The language developed for our LAIPE engine is Lightweight Automata based Incremental Processing Language (LAIPL), it is based on the SQL-like syntax, extensions and customization are made to adapt the language to the specificities of event processing. In LAIPL we assume that events are formatted by the event model (Section 3.1). Leveraging the advantage of LAIPL, we propose LAIPL Rule that defines composite events from atomic and complex ones. Each LAIPL Rule has the following structure:

Rule Name	
Define	Ŧ (Λ_1: type_1, Λ_2: type_2, ..., Λ_n: type_n)
From	$E^{Ŧ\_1}$.Ŧ, $E^{Ŧ\_2}$.Ŧ, ..., $E^{Ŧ\_n}$.Ŧ
Where	$E^{Ŧ\_1}$.Λ = C_1, $E^{Ŧ\_2}$.Λ = C_2, ..., $E^{Ŧ\_n}$.Λ = C_n, SlidingWindowConstraint
Composite	$E^{Ŧ\_1}$.attr = A_1, $E^{Ŧ\_2}$.attr = A_2, ..., $E^{Ŧ\_n}$.attr = A_n

According to the parameters of event model, the rule structure defines a composite event from its component attributes intuitively. The first line is the name of rule. Second line defines the type (Ŧ) and value (Λ_1: type_1, Λ_2: type_2, ..., Λ_n: type_n) of the composite event. The From clause provides the input event types for the types $E^{Ŧ\_1}$.Ŧ, $E^{Ŧ\_2}$.Ŧ, ..., $E^{Ŧ\_n}$.Ŧ. The Where clause defines aggregation constraints of the new event using a set of constraints (C_1, C_2, ..., C_*n*), which depend on the parameters defined in the second line. Finally, the Composite clause shows the composition constraints (A_1, A_2, ..., A_*n*) of the input events attributes ($E^{Ŧ\_1}$.attr, $E^{Ŧ\_2}$.attr, …, $E^{Ŧ\_n}$.attr).

This section informally describes the LAIPL rules through a tutorial. To present the LAIPL rules supported by event algebra in an easy and clear way, we use some definitions of composite event presented above (Section 3.3) and show how they can be encoded in LAIPL rules. Starting with a minimal complexity example, more and more clauses and features are described in the following.

To give a simplest query declarable with the defined LAIPL rule structure, we look at the event expression E1. Indeed, the composite event $E_{S001}^{CPU\_Usage\_High}$ is aggregated from atomic event $E_{S001}^{CPU\_Usage}$ when the average usage of $E_{S001}^{CPU\_Usage}$ is above a predefined constant value in 5 minutes. Assuming the value of HIGH_USAGE is 80, we redefine E1 as Rule 1-1 by LAIPL. The composite event type is CPU_Usage_High and its value is an integer variable. The From clause defines the input event type CPU_Usage and Composite clause limits the source of input event is S001. The condition is defined in Where clause that the average value of input event is above 80 in 5 minutes, that means if the average value below 80, the composite event will not be aggregated. A dedicated effort has been devoted in executing a set of functions enabling users to declare time relations and constrains between events by using the Within keyword in Where clause.

```
Rule 1-1
Define       CPU_Usage_High (Usage: Integer)
From         CPU_Usage
Where        Average(CPU_Usage.value)>80 Within(5 min)
Composite    CPU_Usage.Source=S001
```

Rule 1-1 may have oversight in CPU usage monitor, as the average value can not show the latest CPU usage if the usage is low at the eventual 30 seconds. Similarly, we use Each function to define a composite event to show the real-time CPU usage state in Rule 1-2.

```
Rule 1-2
Define       CPU_Usage_High (Usage: Integer)
From         CPU_Usage
Where        Each(CPU_Usage.value)>80 Within(5 min)
Composite    CPU_Usage.Source=S001
```

According to Rule 1, it is obvious that LAIPL rule allows composite event generated from single event type, however, composite event can also be aggregated from two or more event types. The third example illustrates the use of LAIPL rule structure to express a composite event that is generated from two event types. In Rule 2, the composite event type has been defined with two values, which collected from two input event types. And more complex aggregated constraints are described in Where clause. A new clause *And* is used to combine different aggregated constraints together in rules.

```
Rule 2
Define       User_Saturation(Number: Integer, CPU_Usage: Integer)
From         CPU_Usage_High, User_Number_High
Where        Each(CPU_Usage_High)>80 And Each(User_Number_High)>1000 Batch(300)
Composite    CPU_Usage_High.Source= User_Number_High.Source
```

As a final example, we show how to accommodate more complex needs in LAIPL. Event expression E3 describes a three event types aggregation that includes disjunction, sequence, and temporal operations.

Comparing with Rule 2, Rule 3 contains more composite event values and input event types. In this rule we combine different aggregated operations, taking advantage of LAIPL ability to define composite event generated from other complex ones. In particular, we first present the sequence function indicated by the *Then* clause, which orderly processes events from two constraints. Using this event, we define the application server runs slowly when CPU and memory usage is high as well as the total events of application response time is longer than 30 seconds.

```
Rule 3
Define       Application_Slow(CPU_Usage: Integer, MEM_Usage: Integer, Response_Time: Integer)
From         CPU_Usage_High, MEM_Usage_High, Response_Time_Long
Where        Each(CPU_Usage_High)>80 And Each(MEM_Usage_High)>80 Then
             Count(Response_Time_Long>30)=50 Within(30 min)
Composite    CPU_Usage_High.Source=MEM_Usage_High.Source=Response_Time_Long.Source
```

Consequently, LAIPL provides all the key operators identified above like composition, disjunction, conjunction, negations, sequences, selection, batch and temporal operations through LAIPL rules. It combines these operators with user definable selection and consumption statement to offer good start to write a wide range of different rules. The rest of paper focuses on event processing method that how to execute the rules in our proposed CEP engine.

5. Event Processing Method

In this section, we propose an automata model and two algorithms used in LAIPE. The automata model detects events based on Finite State Machines (FSM) according to LAIPL rules statement. The translation algorithm shows how the engine translates generic LAIPL rule into an automata model. And the LAIP algorithm illustrates the event processing method of LAIP.

5.1. Event Automaton Model

Most CEP engines adopt FSM to automatically derive operational states according to event patterns and rules. Cayuga automaton [6] is based on a variant of a nondeterministic FSM, which is proposed to implement Cayuga algebra expression. Ralf Bruns [28] integrates CEP and FSM to denote the operational states of an ambulance car through intelligent fusion of event data in his research.

Consequently, to optimize the performance of CEP performance, we also combine CEP with FSM to implement our LAIPL rules and represent the operational states of composite events in LAIPE. We propose an automaton model, which is based on traditional FSM and the event-driven FSM defined in [28]. An event automaton model is a tuple {$\dot{S}$, -, +, $\dot{E}^{in}$, $\dot{E}^{out}$, γ} where

$\dot{S}$ is the set of states in the automaton model;

- is the start state, $- \in \dot{S}$;

+ is the end state, $+ \in \dot{S}$;

$\dot{E}^{in}$ is the set of input events;

$\dot{E}^{out}$ is the set of output events;

γ is the state transitions between states, γ represents $\dot{S} \times \dot{E}^{in}$»$\dot{S}$, and $\gamma \subseteq \dot{S} \times \dot{E}^{in}$.

In the event automaton model, - and + are two necessary states in $\dot{S}$. The set of states in a simplest automaton just has – and + as they are the initial and final states of an automaton. Usually, states are depended on some specific events value change in servers and applications. γ defines the transitions triggered by events that change the states from one to another. Considering the state transitions between states as a matrix, γ represents the relationships between $\dot{S}$ and $\dot{E}^{in}$. As + is an end state without none next state, if there are n states in $\dot{S}$, γ is a $(|\dot{S}|-1) \times |\dot{S}|$ matrix. The following example in Fig. 2 shows a complex automaton. In the automaton, $\dot{S}$ = {-, +, $\dot{S}1$, $\dot{S}2$} and $\dot{E}^{in}$ = {$\dot{E}^1$, $\dot{E}^2$, $\dot{E}^3$, $\dot{E}^4$, $\dot{E}^5$}. Its state transition matrix is shown in Table 1. ε is an empty event indicates that there is no transition between two states.

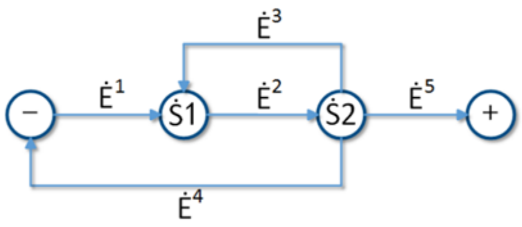
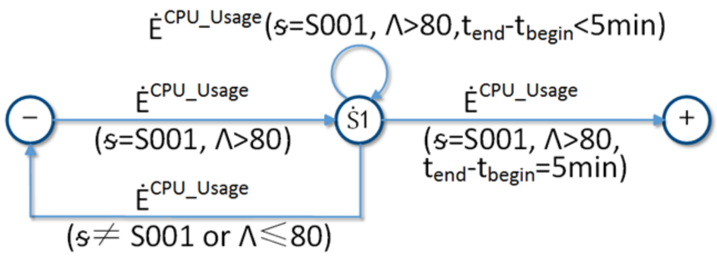


Fig. 2. An example of automaton model.

Table 1. State Transition Table

State	-	$\dot{S}1$	$\dot{S}2$	+
-	ε	$\dot{E}^1$	ε	ε
$\dot{S}1$	ε	ε	$\dot{E}^2$	ε
$\dot{S}2$	$\dot{E}^4$	$\dot{E}^3$	ε	$\dot{E}^5$

The automaton model can implement any LAIPL rule with the proposed event algebra expression. Rule 1-2 defines a composite event of CPU_Usage_High, which can be translated to an automaton instance A1 shown in Fig. 3. The time condition of t_{end} - t_{begin} = 5min represents the state transition occurs within 5 minutes. And the attributes of $\dot{E}^{CPU\_Usage}$ are confined to some certain values, ꞩ=S001 means the source of an event is S001, Λ>80 indicates the value of an event is above 80. In A1, we use "," to show the conjunction relationship of constraints as well as "or" is disjunction relationship. Particularly, the occurrence of transition from $\dot{S}1$ to – generates a failed-detection event Ø, and then A1 will be released.



Ė^CPU_Usage(ꞩ=S001, Λ>80,t_end-t_begin<5min)
Ė^CPU_Usage (ꞩ=S001, Λ>80)
Ṡ1
Ė^CPU_Usage (ꞩ=S001, Λ>80, t_end-t_begin=5min)
−
+
Ė^CPU_Usage (ꞩ≠ S001 or Λ≤80)

Fig. 3. Automaton A1 of Rule 1-2.

In general, we believe our event automaton model, when used appropriately, makes it easier to significantly enhance the reliability of the automaton states as well as implementing the event algebra by rendering explicit event expressions, and thus improves the executability of LAIPL rules.

5.2. Translation of LAIPL Rule

Before entering into the details of the LAIPE processing approach we first introduce the translation algorithm. When we started to design the translation algorithm for LAIPE, we considered two similar approaches, both approaches focus on the decomposition and translation of the rule statement. Extracting constraints by separating key clauses, the first one translates rule statement into a specific event processing model, such as TNCES model [29] and automaton model [8]. The second approach translates rules in a parsing tree [13], which uses the key clauses to create non-leaf nodes as well as generating leaf nodes by constraint details.

LAIPE translation algorithm translates rule into the proposed automaton model, which is efficient for event processing. It contains the initialization of an automaton model, and translations of Define, From, Where and Composite clauses with their particular constraints. These considerations are generalized into the LAIPL rule translation algorithm, which translates a generic LAIPL rule into the proposed event automaton model, is sketched in Algorithm 1.

Algorithm 1. LAIPL rule translation algorithm.
Input: LAIPL rule statement.
Output: Automaton.

```
1    Automaton CreateAutomatonModel(Rule)
2    Automaton A = InitAutomaton(Ṡ, -, +, $Ė^{in}$, $Ė^{out}$, $\gamma$);
3    EventType $Ė^{out}$ = RuleParse(Rule, Define);
4    A. $Ė^{out}$ = $Ė^{out}$;
5    EventType $Ė^{in}$ = RuleParse(Rule, From);
6    A. $Ė^{in}$ = $Ė^{in}$;
7    AggregationConstraint = RuleParse(Rule, Where);
8    CompositionConstraint = RuleParse(Rule, Composite);
9    StateSet Ṡ = CreateState(CompositionConstraint, AggregationConstraint);
10   A.Ṡ = Ṡ;
11   Transition T = GenerateTansitions(Ṡ, AggregationConstraint);
12   A.$\gamma$ = T;
```

At the beginning, the engine generates an automaton and initializes it into a six-tuple automaton model by the InitAutomaton function (line 2). Every automaton has the start and end states, and they are created in initialization. Followed by, the engine parses the rule statement in Define clause by the RuleParse(Rule, Define) function (line 3), and the output events are defined by E^{out} (line 4). The input events are assigned from the From clause in a similar way (line 5 and 6). The states are generated from rule statement in Where and Composite clauses by the CreateState function (line 9). After the states of the automaton captured from rule statement, the engine builds the transitions γ of states by GenerateTansitions function (lines 11 and 12). As an example, the automaton in Fig. 3 shows the model of Rule 1-2.

After the automaton models originating from LAIPL rule are built, they are submitted to the main event processing thread of the engine. The thread instantiates an automaton model to an instance with the start state and holds every automaton instance during its lifecycle.

5.3. Lightweight Automata Based Incremental Processing

From both industry (Esper) [5] and academia (Cayuga) [6], most CEP engines adopt an incremental approach to detect complex events. It recognizes partial sequences in the form of automata and stores the intermediate results derived from the computation of primitive events as soon as composite events are detected. Gianpaolo Cugola [30] gave name of this incremental approach as Automata based Incremental Processing (AIP), and use it as a baseline for his CEP engine. To reduce the time and space cost in CEP engine, AIP approach shares all the events as much as possible and relies on a defined memory management procedure to reclaim the space. It reduces latency in detecting complex events, however, AIP need more memory space for event storage, indeed the memory cost is high when the volume of input events is huge.

In order to improve the efficiency of CEP engine that processes more input events with the same rules than previous AIP algorithms [5], [8], [30], we propose a Lightweight Automata based Incremental Processing (LAIP) approach. LAIP approach is implemented in LAIPE, which processes input events in automatons to capture composite events. As we enhance the availability of the automaton model, our LAIP approach has some advantages than AIP. It better supports sliding window and logic operations as well as failure detection function during event processing. LIAP Algorithm is shown in the following.

Algorithm 2. LAIP algorithm.

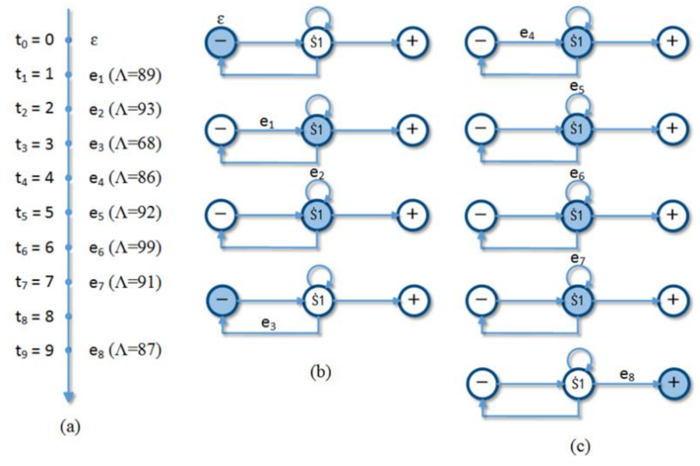
Input: Input event (Ẽ).

Output: Composite event (ce) or failure event (Ø).

```
1   While(!Empty(EventQueue))
2     e = EventQueue.front();
3     for i=0; i<GetAutomatons(e.Ŧ);i++
4       for j=0; j<Automaton(i).GetAutomatonState(e.Ŧ);j++
5         A= Automaton(i).State(j);
6         if (A.WaitTime > ($t_{end}$ - $t_{begin}$))
7           A.Release;
8         else
9           if (A.SatisfyConstraints(e.attribute))
10            A'=A.NextState(e);
11            if (A'.State= EndState)
12              CreateCompositeEvent(A');
13              A'.Release;
14            if (A'.State= StartState)
15              CreateFailureEvent(A');
16              A'.Initialize;
17            Update A by A';
18          else
19            A.WaitTime.Update;
20    EventQueue.Pop();
```

The key role in LAIP approach is played by automaton instances, which are used to detect composite events in LAIP algorithm. EventQueue is a queue that stores undisposed events in LAIPE. When EventQueue is not empty (line 1), the operations of LAIP algorithm have been shown in Algorithm 2. First, it extracts an event e from EventQueue by the Front function, it is nearly real-time event processing (line 2). According to the event type, LAIP gets all the related automaton instances and their states (lines 3 and 4). Second, for each automaton instance in a certain state A (line 5), it is performed as following steps. (i) LAIP checks whether the WaitTime of the certain state A is less than the time of t_{end} - t_{begin}. Otherwise, the automaton of the state is immediately released (line 7). (ii) The attributes of the event are checked whether they satisfy the constraints of A (line 9). If the event is accepted, the state of automaton instance is turned to a next state A' by the NextState function (line 10). If not, the WaitTime of A is updated (line 19). (iii) If the state of A' is the end state (line 11), that means the automaton generates a composite event (line 12), and it will be released (line 13) after the composite event is created. (iv) If the state of A' is the start state (line 14), a failure event Ø is generated (line 15) and the automaton instance A' will be initialized (line 16). (v) When the next state is adopted, the state of automaton instance is updated to A' (line 17). Finally, when an event has been processed by every related automaton, it is removed from EventQueue by the Pop function.

As an example of how to execute an automaton instance through LIAP approach, considering Rule 1-2 and the corresponding automaton instance A1. Fig. 4 shows the event processing steps of a set of input events with automaton A1. We define $E_{S001}^{CPU\_Usage}$= $\{e_1;\ e_2;\ \dots\ ;\ e_8\}$ as input events for the automaton. Fig. 4. (a) is a time shaft that starts form top (t_0) and finishes at the bottom (t_9), where the time segment is 1 minute. The input events are arrived sequentially with different value, and ε as a virtual event used to initialize an automaton instance. Fig. 4. (b) and (c) represent the state transformations of the automaton instance of A1. The last instance state generates a failure event in Fig. 4. (b), while the bottom instance state creates a composite event by accepting e_8 in Fig. 4. (c).



$t_0 = 0$ ε
$t_1 = 1$ e_1 (Λ=89)
$t_2 = 2$ e_2 (Λ=93)
$t_3 = 3$ e_3 (Λ=68)
$t_4 = 4$ e_4 (Λ=86)
$t_5 = 5$ e_5 (Λ=92)
$t_6 = 6$ e_6 (Λ=99)
$t_7 = 7$ e_7 (Λ=91)
$t_8 = 8$
$t_9 = 9$ e_8 (Λ=87)
(a)
(b)
(c)

Fig. 4. An example of automaton model instance A1 is processed by LAIP.

Table 2. The Change of LAIP Variable in Memory

Time	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
Input events	ε	e_1	e_2	e_3	e_4	e_5	e_6	e_7	ε	e_8
Value	Null	89	93	68	86	92	99	91	Null	87
State	-	\$1	\$1	-	\$1	\$1	\$1	\$1	\$1	+
Wait Time	0	0	1	0	0	1	2	3	4	5
Output Events	Null	Null	Null	Ø	Null	Null	Null	Null	Null	ce

The details of state transition change of automaton A1 are introduced in the following. At time t_0, a single automaton A1 is initialized to the start state by ε, waiting in its initial state for input events. At time t_1, an event e_1 enters the engine, which satisfies the constraints of start state transition of A1, and triggers a state transition from start state to \$1. And then, at time t_2, e_2 is accepted and the state of A1 maintains as \$1. However, e_3 occurs and triggers a transition to the start state at time t_3. The state transition generates a failure event and initializes A1. Similarly, from t_4 to t_7, the state transitions are similar with which are at t_1 and t_2. Although no event occurs at t_8, the wait time of A1 is updated to 4. Finally, at time t_9, the arrival of e_8 causes the acceptation of the end state, and the detection of the valid sequence composed by e_4, e_5, e_6, e_7, e_8 has been recognized as a composite event ce.

During the event processing in the above example, the change of LAIP memory variable such as value, state, wait time of automaton A1, are shown in Table 2.

6. Implementation

In this section we represent the layered architecture of Lightweight Automata based Incremental Processing Engine (LAIPE), starting from its framework and modules that contribute to the high performance processing of events. To minimize limitation of development language, we used Java which was the most popular language in the world [12] in implementation of LAIPE. In fact, while our CEP engine optimized event processing method to improve performance by adopting LAIP approach with automaton model.

LAIPE is a lightweight CEP engine designed to provide high performance event processing capability with modularity and extensibility for integrated information systems. As illustrated in Fig. 5 the general LAIPE framework is composed of four layers. The upper engine layer deals with the capture, filter, processing, and storage of input events as well as the translation from LAIPL rule to automation instance, which is similar with the INDCEP engine [31]. It encloses the proposed event processing algorithms of the LAIPE and leverages on event algebra concepts and automaton model to separate particular processing logic from scheduling problems and thread management [13]. The other layers in Fig. 5 shows a base infrastructure of the implementation environment playing as unique essential requirement for the LAIPE deployment.

When a new rule is sent into the engine, it is stored in rule base. The engine creates the rule statement object and forwards it to a thread which is deployed the translation algorithm. The translation algorithm thread parses the rule statement depends on LAIPL rule structure, and generates new automaton instance according to automaton model.

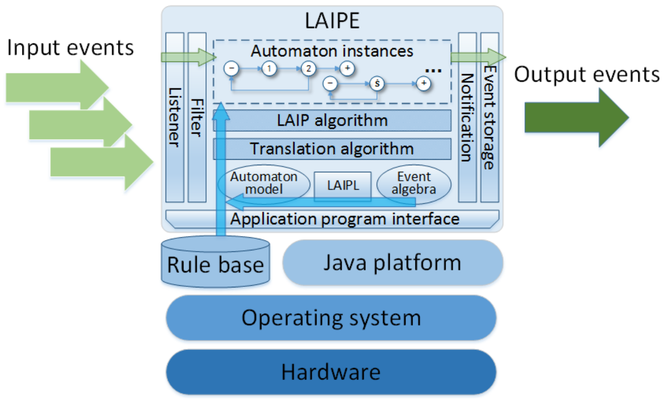


Fig. 5. LAIPE layered architecture.

While a new event enters the system, it is first detected by the Listener module, which collects atomic events from different event sources (information systems and web applications). Listener module sends arrived events to Filter module for filtration of events. The communication between event sources and LAIPE is built through TCP socket, as its efficiency of event transmission is better than web service or http requests. The meaningless and empty events are ignored as well as others are put in a FIFO queue. The LIAP algorithm thread gets each event in the queue to check whether it satisfies a transition in each automaton instance or not, and holds the transition table of each automaton instance. Once a composite event is generated, it is sent to the event storage module and a notification is created to system administrator.

7. Performance

In order to perform functional validation and performance evaluation of the proposed LAIPE architecture, we developed a simple proof-of-concept prototype CEP engine, supporting both the LAIPL rule translation and Lightweight Automata based Incremental Processing (LAIP) algorithms, with the proposed event algebra and Lightweight Automata based Incremental Processing Language (LAIPL) described in the previous sections. Therefore, it makes possible that the event rules designed by domain experts can be deployed in integrated information systems that use CEP engine to generate significant complex events in real-time. Furthermore, we demonstrate that our approach can be adopted in complex event processing engine with high performance.

The performance of a CEP engine is strongly influenced by the workload, which is referred to the complexity and number of rules and events. Therefore, we decided to use a large number of custom workloads, exploring the relationship between engine and workloads as broadly as possible. We used an event provider (local event thread) to generate events with pre-defined types and values at a configurable rate continuously. Then the generated events were sent to the event listener, which is a part of LAIPE. With the event provider, the impact of communication between event sources and CEP engine is eliminated, which allows us to measure the actual performance of the LAIPE.

Our experiments has two main goals: (1) demonstrating the availability and event processing capability of LAIPE with our event processing approach described in Section 5; (2) comparing LAIPE with another common CEP engine that could handle some of the rules introduced in Section 4.

According to our knowledge, we decided to use Esper in the comparison of our experiments for many reasons: it provides efficient processing capability and is widely used in many domains [14]-[16], [32]; its language provides more operators than the others [33]; it is an open-source component for CEP that is available for both Java and .NET, which makes its adoption easier [34]. The source code for the latest version is freely available for download [5].

All the tests were demonstrated on notebook computer having Intel Core i5-3210M CPU 2.50 GHz and 4 GB of Memory, running Windows7 32 bit Professional. Additionally, in order to make the experimental results more convincing, we executed each test 10 times, and took the average value of results. In the following, the experimental results are illustrated in detail.

7.1. Sliding Window

The event processing capability of the two engines is compared in this subsection. This case was based on a simple testing LAIPL rule computing the average value of the selected events and consisting of the two strictly constraints. As sliding window is separated to time window and batch window, rule 4-1 contains the temporal operator as well as rule 4-2 includes the batch operator.

To evaluate the capability of event processing, LAIPE and Esper were respectively deployed rule 4-1 and 4-2. To stress the two engines, each of them was performed with 20 rules, which have the same constrains. In order to simulate input events, the event provider generated atomic events with uniformly distributed value between 1 and 100, while all the input events had same event type and source.

```
Rule 4-1
Define      CPU_Usage_High (Usage: Integer)
From        CPU_Usage
Where       CPU_Usage.value>Selectivity Within(0.1 sec)
```

```
Rule 4-2
Define      CPU_Usage_High (Usage: Integer)
From        CPU_Usage
Where       CPU_Usage.value>Selectivity Batch(10)
```

First of all, rule 4-1 was deployed in the two engines to test the capability of executing time window. Fig. 6 shows how processing time varies in relation to the number of input events. We observe that both engines perform very well, even if the event provider sends a huge number of input events to them in a short time. Even more important from Fig. 6 is that the processing time of LAIPE is about half of Epser's to deal with the same number of input events, that means LAIPE processes events faster than Esper with the same deployed rules and workload.

Second, rule 4-2 was used to evaluate the performance of the two engines in batch window. Fig. 7 shows the processing time of the two engines, adopting batch operator to deal with input events (batch size is 10). It can be observed that the processing time of LAIPE is two-thirds of Esper's. Therefore, the performance of LAIPE is also higher than Esper's with batch window operation.

In general, both engines performed with good performance to process huge number of input events, even if the hardware we used for the experiments was a notebook computer. Our experimental experiences confirmed the proposed LAIPE can process events faster than Esper in most scenarios, deriving the adoption of LAIP approach.

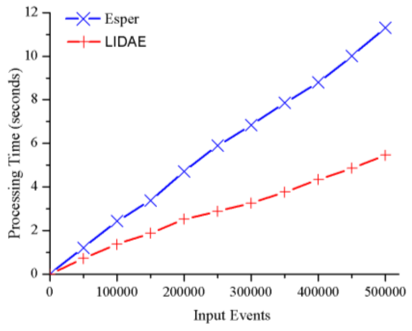


Fig. 6. Comparison Between LAIPE and Esper of Time Window (Adopt Rule 4-1).

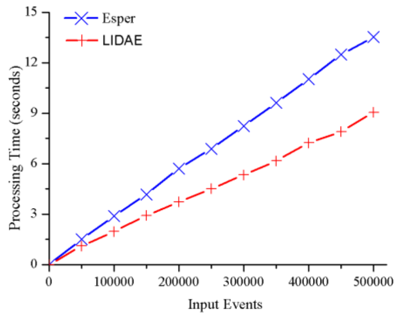
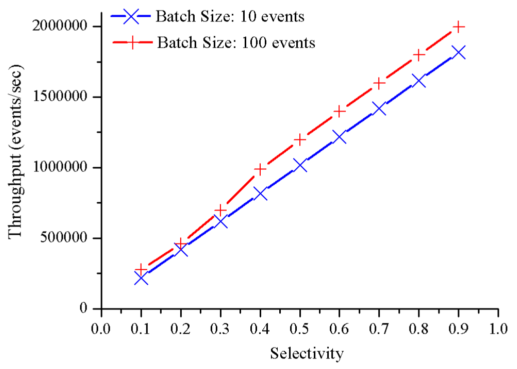
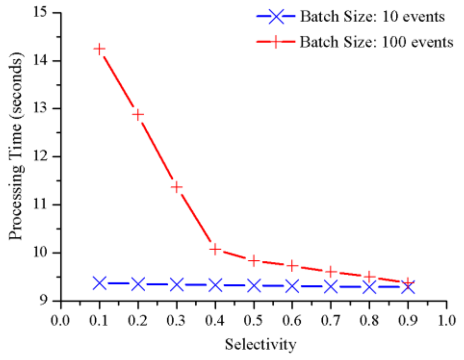


Fig. 7. Comparison Between LAIPE and Esper of Batch Window (Adopt Rule 4-2).



(a) Throughput values of the two batch sizes



(b) Processing time of the two batch sizes

Fig. 8. LAIPE selectivity based testing results.

7.2. Selectivity

The previous experiments compare LAIPE with Esper. In this subsection, we show the performance of LAIPE with different selectivity rates and batch sizes. Fig. 8 shows how throughput and processing time vary in relation to the selectivity rates (from 10% to 90%) with different batch sizes (10 and 100). In particular, the processing time decreases fast when selectivity rate below 40% and batch size is 100. As expected, LAIPE with greater selectivity rate will have better performance, and smaller batch size leads to less processing time.

8. Conclusion

One of the major challenges for integrated information systems is to improve the ability of information processing with CEP. To address this challenge, we described the design and implementation of a Lightweight Automata based Incremental Processing Engine, called LAIPE.

We first proposed an event algebra to denote the information transmission as events in information system domain, including event model, event operations, and event expression. Then we defined an event processing language LAIPL, which provided a user-friendly and expressive querying modeling capability. Adopting LAIPL, LAIPL rule expressed querying constrains for event detection and aggregation. Based on Finite State Machine, we described an event automaton model in order to implement LAIPL rule. According to the automaton model, rule statement could be translated into automaton instance by a translation algorithm. Furthermore, we introduced LAIP approach to improve the performance of CEP engine, leveraging the automaton instance for processing the input events in real-time. Finally, we illustrated a novel layered architecture of LAIPE, whose design clearly separates the core logic devoted to event processing from low-level thread management handled by the proposed algorithms.

The experimental results shows the performance of LAIPE is better than Epser in dealing with a large number of input events. Compare with Esper, LAIPE decreases the processing time in dealing with time and batch windows. Moreover, we test the performance of LAIPE with different selectivity rates and batch sizes to show the engine has efficient capability in event processing.

Acknowledgment

This work was supported by the "Research on key technology of virtual restoration mosaic in Terracotta Army (20136101110019)", "Research on the method of virtual restoration of damaged Terracotta Army based on global optimization (61373117)".

References

[1] Jammes, F., Bony, B., Nappey, P., Colombo, & A. W., Delsing, J. *et al.* (2012). Technologies for SOA-based distributed large scale process monitoring and control systems. *Proceedings of IECON 38th Annual Conference on IEEE Industrial Electronics Society*. Montreal, Canada.

[2] Martin, P., & Matjaz, B. J. (2014). Towards complex event aware services as part of SOA. *IEEE Transactions on Services Computing, 7(3)*, 486-500.

[3] Luckham, D. (2008). The Power of Events: An introduction to complex event processing in distributed enterprise systems. *Lecture Notes in Computer Science, 5321*, 3.

[4] Cugola, G., & Margara, A. (2011). Processing flows of information: From data stream to complex event processing. *Acm Computing Surveys, 44(3)*, 359-360.

[5] ESPERTECH. Event processing with Esper and NEsper. Retrieved 2015, from http://esper.codehaus.org

[6] Demers, A., Gehrke, J., Panda, B., Riedewald, M., Sharma, V., & White, W. (2007). Cayuga: A general purpose event monitoring system. *Proceedings of Innovative Data Systems Research (CIDR)*. CA, USA.

[7] Nicholas, P. S.-M., Matteo, M., & Peter, P. (2009). Distributed complex event processing with query rewriting. *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems,* New York, USA.

[8] Gianpaolo, C., & Alessandro, M. (2012). Complex event processing with T-REX. *The Journal of Systems and Software, 85(8)*, 1709–1728.

[9] Wei, M. Z., Ismail, A., Jun, L., & Mohamed, D. (2007). ReCEPtor: Sensing complex events in data streams for service-oriented architectures. Hp Laboratories, 1-21.

[10] Cayuga Complex Event Processing System. Retrieved 2015, from http://cayuga.sourceforge.net/

[11] Lars, B., Alan, D., Johannes, G., *et al.* (2007). Cayuga: A high-performance event processing engine. *Proceedings of 2007 Special Interest Group on Management Of Data Conference (SIGMOD)*. Beijing, China.

[12] TIOBE Index for October 2015. From http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[13] Ivan, Z., Federica, P., & David, P. (2012). A lightweight and extensible complex event processing system for sense and respond applications. *Expert Systems with Applications, 39(12)*, 10408–10419.

[14] Ralf, B., Jürgen, D., Henrik, M., & Sebastian, S. (2015). Intelligent M2M: Complex event processing for machine-to-machine communication. *Expert Systems with Applications, 42(3)*, 1235–1246.

[15] Roberto, B., Luca, M., & Marco, R. (2015). On-line failure prediction in safety-critical systems. *Future Generation Computer Systems, 45(1),* 123–132.

[16] Deyu, P., & Ruonan, R. (2011). The research on complex event processing in monitoring system. *Proceedings of 2011 International Conference on Computational and Information Sciences*. Chengdu, China.

[17] David, L., & Roy, S. (2008). Event processing glossary – Version 1.1. *Event Processing Technical Society*, 1-19.

[18] Omran, S., Francis, G., Heiko, B., Waseem, M., & Sattler, K.-U. (2013). Monitoring and autoscaling IaaS clouds: A Case for complex event processing on data streams. *Proceedings of 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*. Dresden, Germany.

[19] Li, Q. L., Yan, J., Ting, H., & Xu, H. C. (2013). Smart home services based on event matching. *Proceedings of 2013 10th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD).* Zhangjiajie, China.

[20] Jayasekara, S., Kannangara, S., Dahanayakage, T., *et al.* (2015). Wihidum: Distributed complex event processing. *Journal of Parallel and Distributed Computing*, 42–51.

[21] Annika, H., & Agnès, V. (2015). EVA: An event algebra supporting complex event specification. *Information Systems, 48*, 1–25.

[22] Wang, Y. H., Cao, K., & Zhang, X. M. (2013). Complex event processing over distributed probabilistic event streams. *Computers and Mathematics with Applications, 66(10),* 1808-1821.

[23] Yu, G., Ge, Y., & Li, C. W., (2012). Deadline-aware complex event processing models over distributed monitoring streams. *Mathematical and Computer Modelling, 55(3-4)*, 901-917.

[24] Yan, L., & Dong, W. (2010). Complex event processing engine for large volume of RFID data. *Proceedings of 2010 Second International Workshop on Education Technology and Computer Science.* Wuhan, China.

[25] Wen, Y., Chu, C.-H., & Zang, L. (2011). Leveraging complex event processing for smart hospitals using RFID. *Journal of Network and Computer Applications, 34(3)*, 799–810.

[26] Wang, F. J., Zhang, X. M., Wang, Y. H., & Cao, K. N. (2013). The research on complex event processing method of Internet of Things. *Proceedings of 2013 Fifth Conference on Measuring Technology and Mechatronics Automation*. Hong Kong, China.

[27] Waheed, A., Andrei, L., & Jose, L. M. L. (2012). Formal modelling of complex event processing: A generic algorithm and its application to a manufacturing line. *Proceedings of 2012 10th IEEE International Conference on Industrial Informatics (INDIN)*. Beijing, China.

[28] Ralf, B., Holger, B., *et al.* (2014). Using complex event processing to support data fusion for ambulance coordination. *Proceedings of 17th International Conference on Information Fusion (FUSION)*. Salamanca, Spain.

[29] Waheed, A., Andrei, L., & Jose, L. M. L. (2012). Formal modelling of complex event processing: A generic algorithm and its application to a manufacturing line. *Proceedings of 2012 10th IEEE International Conference on Industrial Informatics (INDIN)*. Beijing, China.

[30] Gianpaolo, C., & Alessandro, M. (2012). Low latency complex event processing on parallel hardware. *Journal of Parallel and Distributed Computing, 72(2)*, 205–218.

[31] Omran, S., & Sattler, K.-U. (2013). Distributed complex event processing in sensor networks. *Proceedings of 2013 IEEE 14th International Conference on Mobile Data Management*. Milan, Italy.

[32] Fernando, T.-S., Mercedes, V.-V., & Antonio, F. S.-G. (2015). A complex event processing approach to detect abnormal behaviours in the marine environment. *Information Systems Frontiers*, 1-16.

[33] Juan, B.-P., Guadalupe, O., & Inmaculada, M.-B. (2014). A model-driven approach for facilitating

user-friendly design of complex event patterns. *Expert Systems with Applications, 41(2)*, 445–456.

[34] Esper Team, EsperTech Inc, Esper Reference. From http://www.espertech.com/esper/release-5.2.0/esper-reference/pdf/esper_reference.pdf

Dong Wang was born in Shaanxi Province, China, in 1987. He is a Ph.D. student at the School of Information and Technology, Northwest University, China. As a part of his Ph.D. research, he is currently exploring solutions for information systems integration and web portals. His main research interests are in distributed systems, and more specifically in the area of complex event processing. He is the leader of an innovative talent training project in Northwest University.

Mingquan Zhou is a professor and doctoral supervisor at the College of Information Science and Technology, Beijing Normal University and director of Key Laboratory Engineering Research Center of Virtual Reality and Application, Ministry of Education, China. His research interests are information processing, computer graphics and 3D visualization.

Sajid Ali received the Postdoctoral & Ph.D. degrees from the College of Information Science and Technology, Key Laboratory Engineering Research Center of Virtual Reality and Application, Ministry of Education, Beijing, China. Beijing Normal University, China in 2013 and 2015 respectively. He is a faculty member of University of Education, Lahore. His current research interests include sensor motion, 3D-human motion, and computer network.

Yusong Liu is a master student at the School of Information, Renmin University of China. His interests are in automatic speech recognition, natural language understanding, and voice analysis. He is also a project manager in Pachira company.

Pengbo Zhou is a Ph.D. student at the College of Information Science and Technology, Beijing Normal University and Beijing Key Laboratory of Digital Preservation and Virtual Reality for Cultural Heritage. His interests are in intelligent information processing, cultural heritage protection, and 3-D model processing.