Efficient Complex Event Processing over RFID Data Stream

Jin Xingyi Lee Xiaodong Kong Ning Yan Baoping
China Internet Network Information Center
jinxingyi@cnnic.cn *lee@cnnic.cn* *nkong@cnnic.cn* *ybp@cnic.cn*

Abstract

RFID technology holds the promise of real-time identifying, locating and monitoring physical objects. To achieve these goals, RFID events need to be collected efficiently and composed expressively. Furthermore, these events have unique characteristics, such as locomotive, temporal and history oriented which should be considered and integrated into an event engine model. The diversity of RFID applications poses further challenges to a generalized framework for RFID events processing. In this paper, the Expressive Stream Language is utilized to collect vast number of primitive events efficiently. Moreover, we introduce a novel semantics to meet requirement of expressive event composition. At last, we use Timed Petri Net to model our newly RFID complex event engine. By introducing typical applications scenarios, we evaluate the validity and effectiveness of our RFID event processing system.

1. Introduction

1.1 Background

The primary differences between the RFID technology and others lie in the real-time events detection and automatically information gathering without people's interfering. However, this technology presents two challenges: 1) RFID readers cannot understand whether a particular event is necessary or not. At the same time, only job that RFID readers can do is to report *any* information they discovered; 2) in light of the RFID reader's expressive limitation, the primitive event generated by readers could only describe like "what time, which tag is read by me". As a result, there is an expression gap between expression of primitive events and the real world requirement.

1.2 Our Solution

There are two phases for data stream processing and abstraction in our RFID data engine model. The first one is to process primitive events, generated by numerous RFID readers, so as to produce predefined basic events. The second phase is to union these basic events into several temporal complex events. The first phase aims at data filtering, which could reduce a system's redundancy dramatically. Moreover, the second phase is used for semantic data composition.

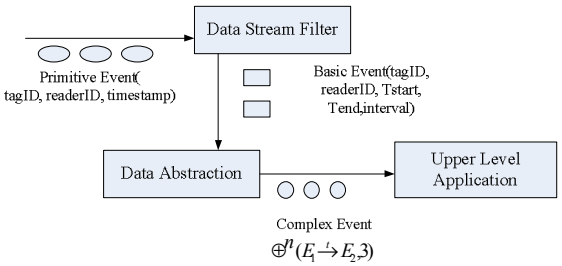

Fig. 1. Framework of our efficient data stream processing.

1.3 Our Contribution

In our paper, we use ESL, presented by [1], to illustrate the basic way of data stream processing at phase 1. Enhanced by the Data Stream Filter, our event engine will take much less storage space than ordinary ones. The newly Complex Event definition semantics is more expressive to meet requirement of RFID applications than [10]. It contains symbols of simple symbolic logic, symbols of temporal logic especially for RFID. We make use of Timed Petri Net [11], [12] to model and to illustrate the realization of our semantics at phase 2.

2. RFID Data Stream Definition

Though we could reap numerous benefits of RFID utilization, it also presents us worrying problems. For example, high volume of primitive events involves serious data duplicate and redundancy. In other words, an urgent challenge of RFID event processing is effective and lossless data filtering. Lots of solutions of data filtering have to balance the data volume against data accuracy and system agility. In the following section, we will introduce a novel solution of raw data

978-0-7695-3131-1/08 $25.00 © 2008 IEEE
DOI 10.1109/ICIS.2008.60

filtering, which aims for both data efficiency and agility.

2.1 primitive events definition

The primitive events of RFID system are the fundamental of application, which are generated by physical RFID readers. The event format is like:

event (*tagID*, *readerID*, *timestamp*)

2.2 basic events definition

Within our event procession solution, we introduce a new predefined event—Basic Event— rather than primitive event, to serve as base unit of event processing and storage. As a result, a large number of primitive events are eliminated from the system by Data Stream Filter. And instead, the concise and lossless predefined event helps us to meet the requirements mentioned above. The Basic Event is defined in the following form:

Basic_Event (*tagID***,** *readerID*, T_{start}, T_{end}, *interval***)**

2.3 non-loss expression of basic event

In reality, the most effective way to trace history of an object is to tell when it in and when it out, but not to record its state in every time point. Similarly, in ALE based system, a specific reader will automatically gather primitive events every time period, namely the read cycle. Taking event stream $\boldsymbol{EV_1}$ for example, R_A automatically captured event every 1 second. So *read_cycle* of R_A is 1 second. As a result, we could utilize one basic event to represent a cluster of continuous events non-destructively, only if let *interval* of basic event equals to system's *read_cycle*. In other words, the basic event (o, R_A, 06:50:25, 06:51:25, 1s) can substitute the R's primitive events stream – (o, R_A, 06:50:25), (o, R_A, 06:50:26), … ,(o, R_A, 06:51:25) non-destructively.

3. Data Stream Processing with ESL

The process engine needs to generate basic events from primitive events. In our system, we transform primitive events into basic ones with Expressive Stream Language (ESL).

ESL is an application language of the Stream Mill system, which supports: 1) continuous queries on data stream, 2) data mining queries and time series queries. It is based on SQL to help users to learn it. We make use of it to define data streams, event transducer, and User Defined Aggregations (UDAs).

3.1 Primitive Events Stream definition

ESL considers primitive events from specific application as ordered sequences of tuples. The example of definition of primitive event with ESL is mentioned below:

Definition 1: Declaring stream of primitive events in ESL

CREATE STEAM primitive-event (
tagID STRING, *readerID* STRING, *timestamp* **timestamp**;)
ORDER BY *timestamp*;
SOURCE "application 1";

The format of the event is the same to event definition above -- **event**(*tagID*, *readerID*, *timestamp*).

3.2 data stream definition

We employ continuous queries on data streams and UDAs to generate basic events. The new basic stream is generated from primitive-event stream by continuous query. The declaration of basic event stream is mentioned below:

Definition 2: Declaring stream of basic events in ESL

CREATE STEAM basic-event AS
SELECT TagID, ReaderID, **timespan** (*timestamp*, *interval*), *interval*
/* *interval* =2000 is interval of basic event, which means 2000 millisecond*/
FROM primitive-event
GROUP BY TagID, ReaderID;

The format of basic event defined by ESL is generally the same to event definition above — **Basic_Event** (*tagID*, *ReaderID*, T_{start}, T_{end}, *interval*). The **timespan** operator is a UDA, which is used to aggregate primitive events which have implicit connection with each others. The "*interval*" parameter is an alias of system's *read_cycle*.

3.3 UDAs definition

According to basic event stream definition, it makes use of *tagID* and *readerID* to compose primitive events. That is to say those events, having same *tagID* and *readerID*, are merged into one cluster. Consequently, we use the **timespan** UDA to merge these events into one or several basic events, judged by adjacent primitive events' interval and system's *read_cycle*. In the following, we will show the elaborate declaration of **timespan** UDA.

Definition 3 Declaring **timespan** UDA of basic-event stream in ESL

AGGREATE **timespan** (*timestamp*, *interval*)
TABLE **state** (T_{start}, T_{end})
INITIATE :{
INSERT INTO **state** (timestamp, timestamp) ;}
ITERATE :{

INSERT INTO RETURN SELECT T_{start}, T_{end} FROM ***state***

WHERE T_{end} < timestamp + interval;

UPDATE state SET T_{end} = timestamp WHERE T_{end} >= timestamp + interval;

UPDATE state SET T_{start} = timestamp, T_{end} = timestamp

WHERE T_{end} <= timestamp + interval ;}

TERMINATE: {

INSERT INTO RETURN SELECT T_{start}, T_{end} FROM ***state*** ;}

The format of return value of **timespan**, which is one or more time span like "T_{start}, T_{end}", is declared by TABLE clause. While in this UDA, state contains only one tuple, which can be queried and update using SQL statement. The three clauses, namely INITIATE, ITERATE and TERMINATE, are the declarations of timespan's different life phases, and the action scope of each phase is determined by the curly braces.

4. complex events definition

The elementary RFID events, such as *primitive* ones or *basic* ones, are far from expressive for practical applications such as business work-flow or real-time monitoring system. Thus RFID processing engine needs to introduce a specific semantics for RFID complex events definition. In this section, we will show our new defined RFID complex event declaring language which is built on the groundwork of [4]. As an extension of previous work [4] and [5], we firstly introduce the quantitative constraint and time-span constraint. Furthermore, we also give a detail description of how to transform events from one complex event type to another, which is not mentioned by [5]. Firstly, we will introduce some functions and expression used in our system.

4.1 definition of logical reader

LR stands for the logical reader, which represents a union set of several physical RFID readers. The ***Logical Reader*** represents the union set of *some RFID physical readers*. Consequently, read by a LR also means the tagged object is detected by one of these physical readers.

4.2 Semantics for Complex Event

In our event engine, a complex event is just defined as a bunch of several basic events, which may have implicit connection with each others. As a result, introduction of complex event will not impose more redundancy and complexity into a system. Namely, the complex events only serve as system's collators, who make a system more organized and concise.

In the following section, we make use of capital letters such as Basic_Event or E_1 to represent an event type, and use small letters, *basic_event* or e_1, to represent an instance of a specific event type. And the function object_type (*e*) is used to retrieve the *type_ID* of a basic event. With this events composition semantics, we could define more complex event types.

Event Composition Operator

$\bigcup$ (UNION): The union symbol is much the same with what it is in set theory. Thus, we could merge two events into one, either complex events or basic ones.

$\xrightarrow{t_1,t_2}$ (SEQ): The sequence operator is a temporal event composition operator, by which we could composite two events, either basic_event or complex_event, into one. Specifically, $E_1 \xrightarrow{t_1,t_2} E_2$ means $e_1 \in E_1, e_2 \in E_2, t_1 < dist(e_1,e_2) < t_2$ (e_i represent an instance of E_i). Especially, the format of $\xrightarrow{t}$ is acceptable, by which we could composite two event where $0 < dist(e_1,e_2) < t$

$\otimes^{t_1,t_2}$ (TLoop): The time loop operator is another event composition operator, by which we will composite one or more events into one. Specifically, $\otimes^{t_1,t_2}(E)$ could bind $\{e_1,e_2,\ldots,e_n\}$ into one TLoop event if $\forall i \in [1,n-1], t_1 < dist(e_i,e_{i+1}) < t_2$ and e_i is a instance of E. Especially, the format of $\otimes^t$ is acceptable, by which we could composite two event where $0 < dist(e_i,e_{i+1}) < t$

$\oplus^n$ (NLoop): The numeric loop operator is a quantitative composition operator, by which we could composite given number of events into one. Specifically, $\oplus^{100}(E)$ could composite 100 events into one NLoop event, only if they are the instances of E.

4.3 User Define Event

In this section, we will provide an example to demonstrate how to define User Define event Type (UDT) with the logical connective and event composition operator.

Example 1:

We also could define a UDT like this:

$E = \oplus^n(E_1 \xrightarrow{t} E_2, 3)$ WHERE type(E_1) ="Basic" and LR(E_1) ="14"

Assuming that the basic_event input stream of the system is like:

$e_1^1(1,14), e_2^1(1,15), e_1^2(2,14), e_2^2(2,15), e_1^3(3,14), e_2^3(3,15)$, $0 < dist(e_2^i, e_1^i) < t, LR(e_1^i) = "14"$. Then, according to the

Composition Operator definition, the return value of E is: $\{\{e_1^1, e_2^1\}, \{e_1^2, e_2^2\}, \{e_1^3, e_2^3\}\}$

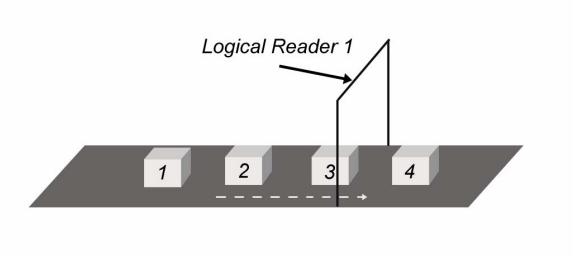

Fig. 2. A scenario of production line in logistics application.

5. Complex Events Detection Engine

While we could use composition operator to define complex event effectively, it is much harder to detect these events defined by the system automatically. Thus, we innovatively introduce a temporal Petri-Net based machine to act as a complex event detection engine. Firstly, we will introduce the Timed Petri-Net (TPN), which is a typical extension of traditional Petri-Net (PN) and essential to our temporal semantics.

5.1 TPN introduction

PN is one mathematical model for discrete distributed system, which is to model general application in non-temporal and discrete system. In light of temporal character of RFID system, we introduce a typical extension of PN, TPN.

TPN is a 5-tuple ***N***=(***S***, ***T***, ***F***, *W*, ***LS***), where ***S*** is a finite set of places, ***T*** is a finite set of transactions, ***F*** associates an arc set known as *flow relation*, ***W*** is a set of arcs weight of each transaction.

LC is represented as a set of static life cycle, a time interval like [t_1, t_2]. Taking Fig. 7 for example, where the life cycle of $buffer_1$ is [t_1, t_2], it suggests that a token can be fireable only if its life span staying in $buffer_1$ is larger than t_1 and less than t_2. And a token will be eliminated from a state when the token has stayed in it longer than t_2.

5.2 TPN representation of universal complex event

In our system, a universal complex event has 4 states, namely **initial**, **execute**, **commit** and **abort** state and has 3 transactions, namely **ex**, **cm**, and **ab** transaction. An event is in the **initial** state when it is initialized by its engine. When the event engine completes some operation, such as redundancy reduction, validation, pattern recognition and etc, the event can be fired by the **ex** transaction from **initial** state to **execute** state. At last, when the event is committable for **cm** transaction, its state will transform from **execute** to **commit** by **cm**. Or, it also could be aborted by system due to some particular reason.

5.3 TPN representation of composition operator

The main purpose of using TPN is to represent temporal composition operator, which are hard to denote by non-temporal mathematical models, such as binary tree in [7]. Since the non-temporal operators—union could easily realized by Petri Net, therefore, in this paper we mainly cast our net on how to model temporal operators, such as SEQ, TLoop, NLoop by TPN.

TPN representation of SEQ

To approach this problem with minimum confusion, I will first introduce a special kind of life cycle, like the $buffer_2$ in Figure 7. The [0, 0] life cycle means "fire or eliminate". In other words, the token will either be fired immediately to another state by transition, or eliminated from the state. For example, if a token is fired to $buffer_2$, the state should checks whether any of its next transitions ($buffer_2\bullet$) is fireable. Then, if it is fireable, the transition is triggered and the token is consumed immediately, or the token will be eliminated from $buffer_2$. Accordingly, in the following, we use a solid marking, like ❶, to represent fireable event, either basic event or complex one; and a hollow marking, like ①, to stand for non-fireable one.

We could use TPN, which is graph-based model, as shown in following, to represent the SEQ operator:

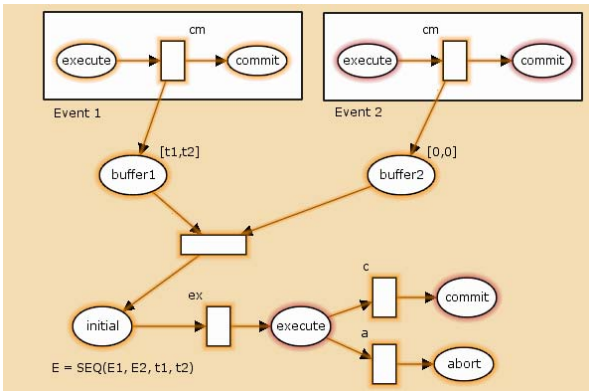

Figure 3. SEQ operator description by TPN

$\xrightarrow{t}$.

In the following example, we could make use of the TPN model to simulate a new defined complex event: **E**= SEQ (*E1*, *E2*, 2s, 2.5s), so as to verify the validation of it. The primary job of it is to monitor the $buffer_1$, and $buffer_2$, which is essence of event composition. Through the event stream, presented in following table, is pseudo, it helps us to comprehend

the TPN clearly.

Define E WHERE $E = E_1 \xrightarrow{0.8s,1.2s} E_2$

read_cycle	1	2	2+	3	3.2	3.2+
buffer 1	①	①,③	①,③	❶,③	❶,③	③
buffer 2		❷			❹	
initial						❶,❹

With the explanation of above table, it is reasonable to conclude that the SEQ's TPN model effectively

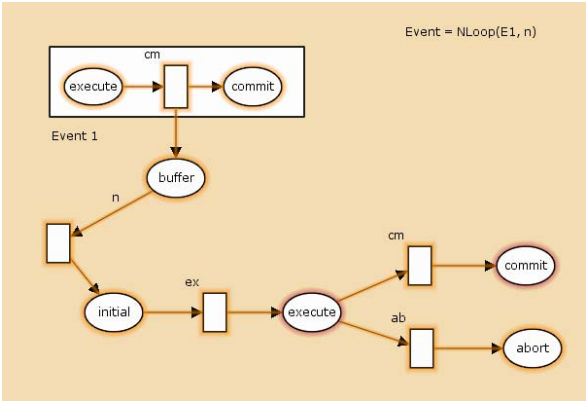

Figure 4. NLoop operator description by TPN

model realizes the definition of SEQ operator.

TPN representation of NLoop

Similarly, we also could use TPN, as shown in following, to represent the NLoop operator: $\oplus^n$. Since there is no temporal constraint to NLoop operator, it can be realized easily by traditional PN with weight arcs. The following table is to verify the validation of the TPN modeling of NLoop.

$E = \oplus^4(E_1)$

read_cycle	1	2	3	4	4+
buffer 1	❶	❶,❷	❶–❸	❶–❹	
initial					❶–❹

In the same way, the TPN with weighed arcs could effectively realize the requirement of NLoop operator definition.

TPN representation of TLoop

We will first introduce a special kind of arc – the inhibitor arc for decreasing the confusion. The Inhibitor, defined in [13] and shown by the following figure, is used to reverse the logic of an input place. With an inhibitor arc, the absence of a token in the input place enables, not the presence. Taking the following figure for example, the state buffer2 is available only if there is no token in it or every token in it is non-fireable (i.e. it has stay in buffer2 less than t1). Definition of the term—fireable refers to the definition of TPN, at the very beginning of the section.

$E = \otimes^{0.9s,1.1s}(E_1)$

Read cycle	1	2	3	4	5	5.2+
buffer 1	❶	❶~❷	❶~❸	❶~❹	❶~❹	
buffer 2	①	❶,②	❷,③	❸,④	❹	
initial						❶~❹

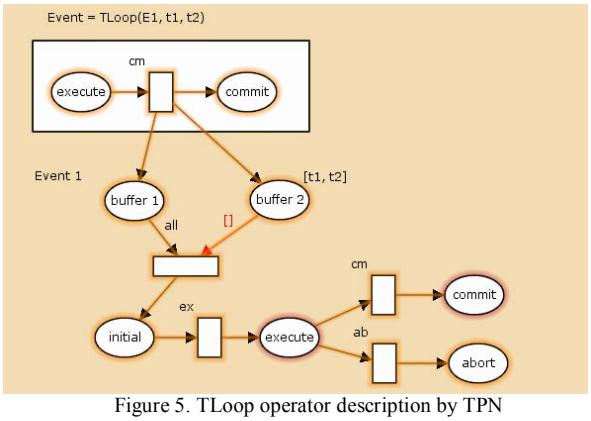

Figure 5. TLoop operator description by TPN

As show, the TPN, with help of inhibitor and life cycle, could be regarded as acceptable way to model the TLoop operator. The model composites events what we want to composite successful, defined by the definition of TLoop.

6. Performance Evaluation

6.1 Expectation of Basic Event

In our system, we have an established RFID system to serve as our data source, a RFID enabled personal management system. Though these RFID data are just experimental, however it is representative for practical application as well. Accordingly, the introduction of basic event could provide a system less duplicity.

Expectation of basic event number is **E**:

$E = \frac{\tau \times N}{W} + (1-\tau)N$ where:

N measures throughput of primitive events stream. Its unit is number per second;

rc is *read_cycle* of the system; T is the upper limit of time span of basic event. Apparently, for every basic event, $T > T_{start} - T_{end}$. Its unit is second and $T>1$; W stands for ratio of T/rc;

τ is a ratio of m/N, where m stands for the number primitive events whose tagged objects are read in current *read_cycle* and were also read in previous *read_cycle*. Apparently, $0 \le \tau \le 1$;

In the extreme case, when $\tau = 0$, namely every tagged object cannot be detected by a same reader twice consecutively, the **E** equals to N, the number of primitive events. On the other hand, when $\tau = 1$,

namely every tagged object is fixed in a particular location, the **E** equals to *N/W*, the minimal value of basic events.

Evaluation of Basic Event

In this section, we provide primitive events as data source which are triggered by both non-movable object, such as devices, and locomotive objects such as people. And it could enable us to compare the effectiveness of basic event for different application scenarios.

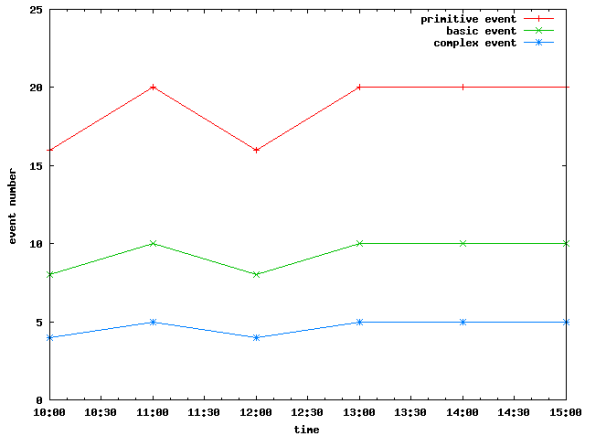

Figure 6. Comparison of the number of 3 kinds of events in fast-move scenario

Scenario 1. People Monitoring where $N \approx 18$, $\tau \approx 0.5$, $T = 60$, $rc = 1$

Define E_1 WHERE ***type*** (E_1) ="Basic" AND ***LR*** (E_1)=1;

Define E_2 WHERE ***type*** (E_2) ="Basic" AND ***LR*** (E_2)=2;

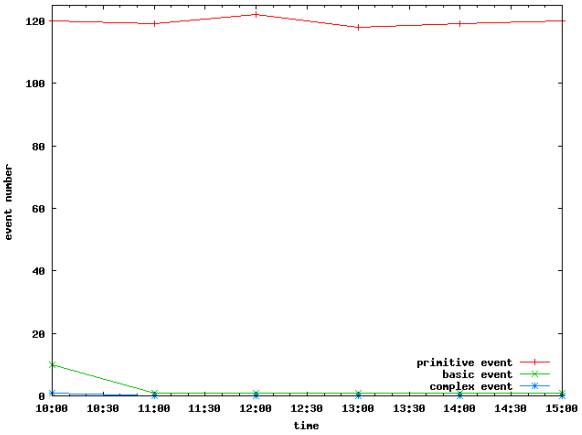

Figure 7. Comparison of the number of 3 kinds of events in slow-move scenario

Define E WHERE $E_1 \xrightarrow{1s,60s} E_2$

Scenario 2. Device Monitoring where $N \approx 120$, $\tau \approx 0$, $T = 60$

Define E_1 WHERE type (E_1) ="Basic" AND LR (E_1)=1;

Define E_2 WHERE type (E_2) ="Basic" AND LR (E_2)=2;

Define E WHERE $E_1 \xrightarrow{1s,60s} E_2$

There are two main conclusions that can be drawn from the above curves: 1) at constant N and W, the effectiveness of basic event varies inversely with the τ. In other words, the more frequent tagged objects move, the more basic event will generated; 2) at constant N and τ, the effectiveness of basic event is directly proportional to W, which represents the information volume of one basic event.

7. References

[1] B. Yijian, T. Hetal, L. Chang, W. Haixun and Z. Carlo, "A Data Stream Language and System Designed for Power and Extensibility," in *Proc. of 15th ACM Conf. on Information and Knowledge Management*, Virginia, 2006, pp.337–346.

[2] R. Cocci, Y. Diao and P. Shenoy. "SPIRE: Scalable Processing of RFID Event Streams," in *Proc. 5th RFID Academic Convocation*, 2007.

[3] D. Gyllstrom, E. Wu, H. Chae, Y. Diao, P. Stahlberg and G. Anderson, "SASE: Complex Event Processing over Streams", in *Proc. of CIDR*, Asilomar, CA, 2007.

[4] E. Wu, Y. Diao and S. Rizvi. "High-PerformanceComplex Event Processing over Streams", In *Proc. of SIGMOD*, Chicago, 2006, pp. 407–418.

[5] H. Gonzalez, J. Han and X. Li, "Mining compressed commodity workflows from massive RFID data sets," in *Proc. of CIKM*, 2006, pp. 162–171.

[6] B. Shivanath, W. Jennifer, "Continuous queries over data streams," in *Proc. of SIGMOD*, 2001, pp. 109–120

[7] F. WANG, S. LIU, "Bridging physical and virtual worlds:complex event processing for RFID data streams," in *Proc. 10th Int. Conf. on Extending Database Technology*, Munich, Germany, 2006, pp. 588–607.

[8] E. Masciari, "RFID data management for effective objects tracking", in *Proc. of 23rd Annu. ACM Symposium on Applied Computing*, 2007, pp. 457-461.

[9] F. WANG, P. LIU. "Temporal management of RFID data," in *Proc. of the 31st VLDB Conf.*, New York , 2005, pp. 1128–1139.

[10] S. CHAKRAVARTHY, V. KRISHNAPRASAD and E. ANWAR, "Composite events for active databases: Semantics contexts and detection," in *Proc. of 20th VLDB Conference*, Santiago Chile, 1994, pp. 1011–1205.

[11] Y. Bai, F. Wang, P. Liu, C. Zaniolo and S. Liu, "RFID Data Processing with a Data Stream Query Language," in *Proc. of ICDE, 2007*, pp. 1184–1193.

[12] R. Mascarenhas, D. Karumuri, U. Buy, R. Kenyon, "Modeling and Analysis of a Virtual Reality System with Time Petri Nets," in *Proc. of the 20th Int. Conf. on Software Engineering*, 1998, Kyoto, Japan, pp. 33–42.

[13] C. Lakos, S. Christensen, "A General Systematic Approach to Arc Extensions for Coloured Petri Nets," in

Proc. of the 15th Int. Conf. on Application and Theory of Petri Nets, 1994, pp. 338–357.