Bridging Physical and Virtual Worlds: Complex Event Processing for RFID Data Streams

Fusheng Wang[1], Shaorong Liu[2], Peiya Liu[1], and Yijian Bai[2]

[1] Integrated Data Systems Department
Siemens Corporate Research
Princeton, NJ 08540, USA
{fusheng.wang, peiya.liu}@siemens.com
[2] ⋆ Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095, USA
{sliu, bai}@cs.ucla.edu

Abstract. Advances of sensor and RFID technology provide significant new power for humans to sense, understand and manage the world. RFID provides fast data collection with precise identification of objects with unique IDs without line of sight, thus it can be used for identifying, locating, tracking and monitoring physical objects. Despite these benefits, RFID poses many challenges for data processing and management: i) RFID observations contain duplicates, which have to be filtered; ii) RFID observations have implicit meanings, which have to be transformed and aggregated into semantic data represented in their data models; and iii) RFID data are temporal, streaming, and in high volume, and have to be processed on the fly. Thus, a general RFID data processing framework is needed to automate the transformation of physical RFID observations into the virtual counterparts in the virtual world linked to business applications. In this paper, we take an event-oriented approach to process RFID data, by devising RFID application logic into complex events. We then formalize the specification and semantics of RFID events and rules. We demonstrate that traditional ECA event engine cannot be used to support highly temporally constrained RFID events, and develop an RFID event detection engine that can effectively process complex RFID events. The declarative event-based approach greatly simplifies the work of RFID data processing, and significantly reduces the cost of RFID data integration.

1 Introduction and Motivation

Background

An RFID (radio frequency identification) system consists of a host computer, RFID reader, antenna (which is often integrated into readers), transponders or RF tags. An RFID tag is always uniquely identified by a tag ID stored in its memory, and can be attached to almost anything. The EPC (electronic product code) standard [1] defines such unique IDs around the world. Readers can be mounted at entrance/exit, point of sale, warehouse, and so on. When a tag is in the vicinity of a reader, the reader sends

⋆ Work done while visiting Siemens Corporate Research

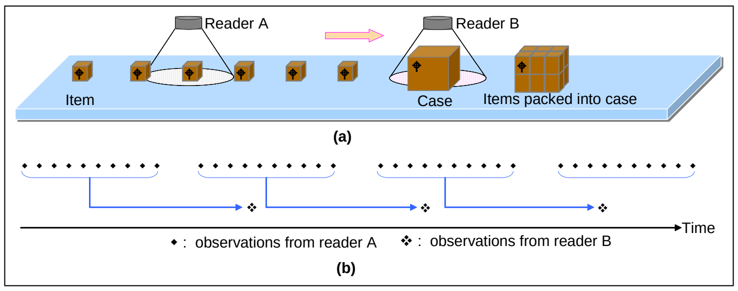
Reader A
Reader B
Item
Case
Items packed into case
(a)
Time
◆ : observations from reader A
❖ : observations from reader B
(b)

Fig. 1. Sample aggregation in RFID. a) Packing of items into its container case; b) Complex events used for aggregation.

energy through RF signal to the tag for power, and the tag sends back modulated signal with ID and data. The reader then decodes and sends the data to the host computer.

With RFID technology, it is possible to create a physically linked world in which every object is numbered, identified, cataloged, and tracked. RFID is automatic and fast, and does not require line of sight or contact between readers and tagged objects. With the significant advantages of RFID technology, RFID is being gradually adopted and deployed in a wide area of applications, such as access control, library checkin and checkout, document tracking, smart box, highway tolls, logistics and supply chain, security and healthcare.

To achieve these, the first task for RFID applications is to map objects and their behaviors in the physical world into the virtual counterparts and their virtual behaviors in the applications by semantically interpreting and transforming RFID data.

RFID Data Transformation and Aggregation

There are generally two types of RFID applications: i) history-oriented object tracking and ii) real-time oriented monitoring. Both need to transform RFID observations into logic data.

History-oriented object tracking. In this type of RFID applications, RFID data streams are collected from multiple RFID readers at distributed locations, and transformed into semantic data stored in RFID data store. The semantics of the data include:

– *Location*, which can be either a geographic location or a symbolic location such as a warehouse, a shipping route, a surgery room, or a smart box. A change of location of an EPC-tagged object is often signaled by certain RFID readers. The location histories of RFID objects are then transformed automatically from these RFID readings, and stored in a location history relation in an RFID data store [2];
– *Aggregation*, i.e., formation of relationship among objects. A common case is the containment relationship, e.g., containment relationship as shown next in Example 1. How to associate relationship among RFID objects in an Auto-ID environment has been identified as a difficult issue for RFID applications [3]. To our best knowledge, no work has been published on solving this problem.

  Example 1: Data Aggregation. In Fig. 1a, on a packing conveyer, a sequence of tagged items move through Reader A and are observed by the reader as a sequence of observations, and then a tagged case is read by Reader B as another observation. After that, all items of this sequence are packed into the case.

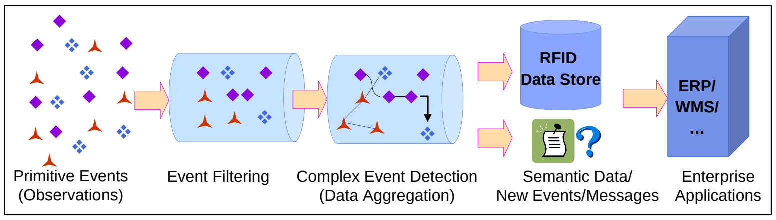
RFID Data Store
ERP/ WMS/ ...
Primitive Events (Observations)
Event Filtering
Complex Event Detection (Data Aggregation)
Semantic Data/ New Events/Messages
Enterprise Applications

Fig. 2. RFID Events Processing

- *Temporal.* RFID observations and their collected data are highly temporal, as studied in [2]. The RFID data store essentially preserves the history of the movement and behaviors of objects.

Real-time Monitoring. RFID is also widely used for real-time applications, where patterns of RFID observations implying special application logic can trigger real-time response. An example is discussed as follows.

Example 2. A company uses RFID tags to identify asset items and employees in the building, and only authorized users (superusers) can move the asset items out of the building. When an unauthorized employee or a criminal takes a laptop (with an embedded RFID tag) out of the building, the system will send an alert to the security personnel for response.

Event-Oriented Processing of RFID Data Streams

Indeed, automatic RFID data transformation can be achieved by first devising application logic as complex events, and then detecting such complex events (Fig. 2). After the detection of these complex events, the semantics are interpreted and can be easily integrated into business applications. RFID reader observations are the only primitive events, which then form complex events. Next we show how to devise complex events for data transformation.

- *Data Aggregation Event.* For Example 1, indeed, the items in the conveyer can be arranged as a sequence of events TSEQ$_A$ with certain temporal constraint (Fig. 1). Then the packing step becomes a sequence event from Reader A, followed by a primitive event O$_B$, an observation of case B from Reader B. Then, the containment relationship is detected and transformed into a containment relation inside the RFID data store.
- *Real-time Monitoring Event.* Example 2 can be simplified by a complex event: the system detects an event A – observation of an object of type “laptop”, and within certain interval τ, e.g., 5 seconds, it does not detect any occurrence of event B – observation of a superuser, i.e., a negated event, then the event triggers an alert action.

RFID events, however, have their own characteristics and cannot be supported by traditional event systems. The two examples above show that RFID events are temporal constrained: both the temporal distance between two events and the interval of a single event are critical for event detection. Such temporal constraints, however, are not well supported by traditional ECA rules detection systems. In addition, non-spontaneous events, including negated events and temporal constrained events, are important for many RFID applications but difficult to support in past event detection engines. Moreover, the actions from RFID events are quite different: they neither trigger new primitive

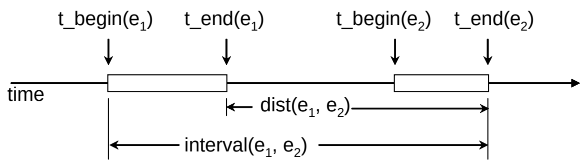

Fig. 3. An illustration of functions used in event expressions

events for the system, nor lead to a cascade of rule firings as in active databases. Thus, there is an opportunity to build a scalable rule-based system to process complex RFID events.

Our Contribution

In this paper, we formulate a declarative rule based approach to provide powerful support of automatic RFID data transformation between the physical world and the virtual world. We develop a graph-based RFID complex event detection engine – RCEDA, where temporal constraint is taken as a first class object in event detection. We introduce pseudo events in event detection to process non-spontaneous events, which are difficult to support in traditional event detection systems. We show that our approach can support RFID applications effectively, and the performance of our event detection engine is quite scalable as well.

The paper is organized as follows. We first give a formal definition of RFID events in Section 2, and then discuss the declarative RFID rules language in Section 3. Event detection engine is discussed in Section 4, and performance is studied in Section 5. Related work is discussed in Section 6, followed by conclusion.

2 RFID Events

In this section, we will formalize the semantics and specification for RFID events. In particular, we will discuss temporal RFID events, which are highly temporally constrained and cannot be well supported by traditional ECA (Event-Condition-Action) rule systems.

An *event* is defined to be an occurrence of interest in time, which could be either a *primitive* event or a *complex* event. Primitive events occur at a point in time, while complex events are patterns of primitive events and happen over a period of time.

In the following discussion, we use `E` to represent an event type, and `e` to represent an event instance.

We first define several functions used in our event expressions (Fig. 3). `t_begin(e)` returns the starting time of an event instance `e`, and `t_end(e)` returns the ending time. `interval(e)` returns the interval of an event instance: `t_end(e) - t_begin(e)`; `dist(`e_1`, `e_2`)` returns the distance between two event instances e_1 and e_2, which is equal to `t_end(`e_2`) - t_end(`e_1`)`; `interval(`e_1`, `e_2`)` returns the interval between two event instances e_1 and e_2, which is equal to `max`$\{$`t_end(`e_2`), t_end(`e_1`)`$\}$ `- min`$\{$`t_begin(`e_2`), t_begin(`e_1`)`$\}$.

2.1 Primitive Event

Primitive events in RFID applications are events generated during the interaction between readers and tagged objects. That is, a primitive event is a reader observation, in the format of `observation(r, o, t)`, where `r` represents the reader EPC, `o` represents the object EPC and `t` represents the timestamp when the observation is made. For example, `observation('r1', o, t)` represents events generated from a reader with EPC `'r1'`. Primitive events are instantaneous. That is, given any primitive event instance `e`, `t_begin(e) = t_end(e)`. Primitive events are also atomic: a primitive event either happens completely or does not happen at all.

Definition of Primitive Event Types. While primitive events are all from observations, they can be of different types, according to the reader EPC, or tag EPC. We first present two user-defined functions on primitive event attributes used to define primitive event types.

- `group(r)` – the group which the reader `r` belongs to. Readers are often deployed into groups in which readers perform the same functionality.
- `type(o)` – the type of the object with EPC `o`. The type can be extracted from its EPC value with a user-defined extraction function, or specified by a user with a mapping function. For example, `type('8E5YUK691I0J60KDN')='laptop'` while `type ('UH7JEFU63MAW6I610') = 'pallet'`.

With above functions, we can define primitive event types. For example, the primitive event type `E` is defined as:

`E = observation(r, o, t), group(r)='g1', type(o) ='case'`

That is, observations of `'case'` by readers in group `'g1'` are of type E.

If `group()` and `type()` functions are not explicitly specified, the default primitive event type is a group with the reader itself.

`E = observation('r', o, t)` $\Longleftrightarrow$
`E = observation('r', o, t),group(r)='r'`

2.2 Complex Event

A complex event is usually defined by applying event *constructors* to its constituent events, which are either primitive events or other complex events. There are two types of RFID event constructors: *non-temporal* and *temporal*, and the latter contains order, temporal constraints, or both. While complex events defined with non-temporal event constructors can be detected without considering the orders among constituent events, complex events defined with temporal event constructors cannot be detected without checking the orders and/or other temporal constraints (e.g., distance or interval) among constituent events.

Basic Non-Temporal Complex Event Constructors

- `OR` ($\vee$): Disjunction of two events E_1 and E_2, $E_1 \vee E_2$, occurs when either E_1 or E_2 occurs.
- `AND` ($\wedge$): Conjunction of two events E_1 and E_2, $E_1 \wedge E_2$, occurs when both E_1 and E_2 occur disregarding their occurrence orders.
- `NOT`($\neg$): Negation of an event `E`, $\neg$`E`, occurs if no instance of `E` ever occurs. Negated events themselves are non-spontaneous and they are usually combined with other events and/or with some temporal constraints.

In this paper, we only consider the above three basic non-temporal complex event constructors, which are in fact sufficient for expressing any complex event patterns without temporal constraints. For example, a complex event $E = ALL(E_1, E_2, ..., E_n)$, which occurs if all E_1, E_2, ..., E_n occur irrespective of their orders, is equivalent to $E = E_1 \wedge E_2 \wedge ... \wedge E_n$.

Temporal Complex Event Constructors

– `SEQ(;)`: Sequence of two events E_1 and E_2, denoted by $E_1;E_2$, occurs when E_2 occurs given that E_1 has already occurred. (Here we assume that E_1 ends before E_2 starts.)
– `TSEQ(:)`: Distance-constrained sequence of two events E_1 and E_2, $\mathtt{TSEQ}(E_1;E_2, \tau_l, \tau_u)$, occurs when E_2 occurs given that E_1 has already occurred and that the temporal distance between the occurrences of E_1 and E_2 is bounded by $[\tau_l, \tau_u]$. That is, $\tau_l \leq \mathtt{dist}(TE_1, E_2) \leq \tau_u$.
– $\mathtt{SEQ}^+(;^+)$: The aperiodic sequence operator, $\mathtt{SEQ}^+(E)$, allows one to express one or more occurrences of an event E.
– $\mathtt{TSEQ}^+(:^+)$: The distance-constrained aperiodic sequence operator, $\mathtt{TSEQ}^+(E, \tau_l, \tau_u)$, allows one to express one or more occurrences of an event E such that the temporal distance between any two adjacent occurrences of E are bounded by $[\tau_l, \tau_u]$.
– `WITHIN`: An interval-constrained event, $\mathtt{WITHIN}(E, \tau)$, occurs if an instance of E, e.g., e, occurs and $\mathtt{interval(e)} \leq \tau$.

Temporal Constraints. While non-temporal event constructors above were discussed in the past [4, 5], the new temporal event constructors that we propose are essential for RFID applications. As shown above, most temporal event constructors use temporal constraints to specify temporal complex events. These include *distance constraint*: minimal distance (τ_l) between two events in a temporal sequence `TSEQ` and maximal distance (τ_u) between two events in a temporal sequence `TSEQ`; and *interval constraint*: maximal interval size (τ) of a complex event as in the `WITHIN` constructor. These temporal constraints are not supported in past event systems.

Examples of Complex Events. In Example 1, the complex event is:

$\mathtt{TSEQ}(\mathtt{TSEQ}^+(E_1, \tau_{l1}, \tau_{u2}); E_2, \tau_{l2}, \tau_{u2})$,

where event types $E_1 = \mathtt{observation}(r_1, o_1, t_1), \mathtt{group}(r_1) = 'r_1'$ and $E_2 = \mathtt{observation}(r_2, o_2, t_2), \mathtt{group}(r_2) = 'r_2'$.

In Example 2, the complex event is:

$\mathtt{WITHIN}(E_1 \wedge \neg E_2, 5sec)$,

where $E_1 = \mathtt{observation}('r_2', o_1, t_1), \mathtt{type}(o_1) = \text{'laptop'}$ and $E_2 = \mathtt{observation}('r_2', o_2, t_2), \mathtt{type}(o_2) = \text{'superuser'}$.

3 RFID Rules

Based on event specification described above, we now define RFID rules. We first introduce the syntax of RFID rules as follows:

```
CREATE RULE rule_id, rule_name
ON event
IF condition
DO action_1; action_2; ...; action_n
```

where rule_id and rule_name stand for the unique id and name for a rule; event is the event part of the rule, condition is a boolean combination of user-defined boolean functions and SQL queries; and action_1; action_2; ...; action_n is an ordered list of actions, where each action is either a SQL statement or a user-defined procedure, e.g., to send out alarms.

An alias of an event can be defined for reuse in the following form:

```
DEFINE event_name = event specification
```

Next, we show that with declarative RFID rules, we can provide powerful support for RFID data processing, including data filtering, data transformation and aggregation, and real-time monitoring.

3.1 RFID Data Filtering

Before RFID data are further processed, they need to be filtered first. There are two types of data filtering for RFID data: *low level data filtering*, and *semantic data filtering*. The low level data filtering cleans raw RFID data, and semantic data filtering extracts data on demand or interprets semantics from RFID data.

Low Level Data Filtering: Duplicate Detection

Duplicate observations are common in RFID applications. This can be caused by several reasons: i) tags in the scope of a reader for a long time (in multiple reading frames) are read by the reader multiple times; ii) multiple readers are installed to cover larger area or distance, and tags in the overlapped areas are read by multiple readers; and iii) to enhance reading accuracy, multiple tags with same EPCs are attached to the same object.

Rule 1. If the same reader observes the same object multiple times within a short interval, e.g., 5 seconds, then mark the previous event as a duplicate.

```
CREATE RULE r2, duplicate_detection_rule
ON WITHIN(observation(r, o, t1); observation(r, o, t2), 5sec)
IF true
DO
   send_duplicate_msg(observation(r, o, t1))
```

Similarly, we can filter duplicates from multiple readers (e.g., `r1` and `r2`), by defining a reader group containing these readers.

Semantic Data Filtering: Infield/Outfield Filtering

RFID rules can also be used to perform effective semantical data filtering. For example, *infield* and *outfield* events are used in smart shelf applications [6]. Although tagged objects on a smart shelf are read all the times, applications may only be interested in when an object is put on the shelf (infield) and when an object is taken off the shelf (outfield) in order to update inventory automatically. The following example illustrates how to use an RFID rule to express infield events and perform the corresponding actions.

Rule 2. If an object is observed by a reader `r` on a smart shelf for the first time, then the rule will insert the observation into the OBSERVATION table. (We assume that the reader is scheduled to bulk-read all objects every 30 seconds in the following example.)

```
CREATE RULE r2, infield_filtering
ON WITHIN(¬observation(r, o, t1); observation(r, o, t2), 30sec)
```

```
IF true
DO
   INSERT INTO OBSERVATION
   VALUES (r, o, t2)
```

Outfield filtering can be defined similarly by switching the order of the negated event.

3.2 Data Transformation and Aggregation

One significant benefit of RFID rules is that data transformation and aggregation is simplified in a declarative way. With a set of data transformation and aggregation rules, RFID observations are automatically interpreted and mapped into their data models and stored in RFID data store.

In the following, we show two examples of how to devise data transformation and aggregation rules, and detect such rules to generate semantic data in a fully automatic environment. We assume that object containment relationships are stored in table OBJECTCONTAINMENT(object_epc, parent_epc, tstart, tend), where object_epc stands for the EPC of the object being contained, parent_epc stands for the EPC of the container object, and [tstart, tend] stands for the period of the containment relationship.

Location Transformation

RFID observations may imply location changes and business movements. For example, an observation by a reader `r` of an object `o` at time `t` implies that the object has entered the location where the reader resides in starting from time `t`.

In the following, we assume that object location information is stored in table OBJECTLOCATION (object_epc, loc_id, tstart, tend), with the EPC of an object, location ID of the object, and the period during which the object stayed.

Rule 3. Any observation by a reader `r` will change the location of the observed object `o`: updating the object's current location by changing its `tend` from "Until Changed" (UC) to `t` and inserting a new location for this object, i.e., the reader's new location with its starting timestamp `t` and ending timestamp "UC."

```
CREATE RULE r3, location_change_rule
ON observation(r, o, t)
IF true
DO
     UPDATE OBJECTLOCATION
     SET tend = t
     WHERE object_epc = o and tend = “UC”;
     INSERT INTO OBJECTLOCATION VALUES(o, “loc2”, t, “UC”);
```

Containment Relationship Aggregation

Automatic data aggregation, a difficult task for RFID applications [3], can now be greatly simplified with RFID rules. (RFID applications need to be engineered accordingly to generate proper patterns.)

Rule 4. If a distance-constrained aperiodic sequence of readings from reader "r_1" is observed followed by a distinct reading from a reader "r_2," it implies that objects observed by "r_1" are being packed in the object observed by "r_2." Then the rule will insert new containment relationships into the OBJECTCONTAINMENT table (Fig. 1).

```
DEFINE E1 = observation("r1", o1, t1)
DEFINE E2 = observation("r2", o2, t2)
CREATE RULE r4, containment_rule
ON TSEQ(TSEQ+(E1, 0.1sec, 1sec); E2, 10sec, 20sec)
IF true
DO
  BULK INSERT INTO CONTAINMENT
  VALUES (o2, o1, t2, "UC")
```

The keyword "BULK" will enforce a bulk insertion of all contained objects into the container.

3.3 Real-Time Monitoring

RFID rules can also provide effective support of real-time monitoring, as shown in the following asset monitoring example.

Rule 5. As shown in Example 2, if the reader mounted at a building exit, "r_4," detects a tagged laptop but does not detect any tagged superuser (who is authorized to move asset items out of the building) within certain time threshold, e.g., 5 seconds, then it implies that the laptop is being taken out illegally, and an alert is sent to a security personnel.

```
DEFINE E4 = observation("r4", o4, t4), type(o4) = "laptop"
DEFINE E5 = observation("r4", o5, t5), type(o5) = "superuser"
CREATE RULE r5, asset_monitoring_rule
ON WITHIN(E4 ∧ ¬E5, 5sec)
IF true
DO send_alarm
```

4 RCEDA: RFID Complex Event Detection

While RFID rules provide powerful support for data transformation and monitoring, the detection of complex RFID events is quite challenging. We next discuss the differences between RFID event detection and traditional ECA event detection.

4.1 RFID Event Detection versus Traditional ECA Event Detection

First, many RFID events (e.g., events containing constructors of TSEQ, TSEQ^+ and WITHIN) contain temporal constraints at instance level, which are not supported by traditional ECA rules. In traditional ECA rule systems [7, 8, 4, 9], event detection is performed at type level, but instance level constraints (such as temporal constraints) are not supported. (Snoop supports interval for periodic events, which have to be between two events.) Thus, in such systems, instance-level constraint checking has to be performed as condition checking. In RFID events, temporal constraints, however, are inherent to the events and highly essential to the correctness of event detection. Thus, RFID temporal constraints cannot be simply taken as conditions. Next we show an example that traditional ECA event detection will not work properly for temporal RFID events. Suppose that we have the following complex event to detect the packing of items into cases in an assembly line (Fig. 1):

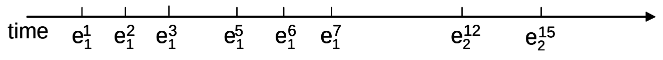

Fig. 4. Sample event history for complex event `E = TSEQ(TSEQ`$^+$`(E`$_1$`, 0sec, 1sec); E`$_2$`, 5sec, 10sec)`

$$\texttt{E = TSEQ(TSEQ}^{+}\texttt{(E}_1\texttt{, 0sec, 1sec); E}_2\texttt{, 5sec, 10sec)}$$

where $\texttt{E}_1$ represents an observation of an item and $\texttt{E}_2$ represents an observation of a case.

If the event detection is done through ECA systems, where instance level temporal constraints are checked as conditions, we will first detect the following instances for complex event `E`, given the event history in Fig. 4.

$$\{e_1^1,\ e_1^2,\ e_1^3,\ e_1^5,\ e_1^6,\ e_1^7\};\ e_2^{12}$$

where e_i^j denotes an instance of event type $\texttt{E}_i$ at time j. The instances $\{e_1^1,\ e_1^2,\ e_1^3,\ e_1^5,\ e_1^6,\ e_1^7\}$, however, do not satisfy the temporal constraints in `TSEQ`$^+$`(E`$_1$`, 0sec, 1sec)` because the distance between e_1^3 and e_1^5 is larger than the upper bound, 1sec. With such an event processing approach, no instances for complex event `E` will be generated, which, however, is not correct. Therefore, for proper processing of RFID events, we must consider temporal constraints as an integral part of the event detection step. Thus, existing ECA-based event systems cannot be used for detecting RFID events.

Second, RFID events by constructors such as `SEQ`$^+$ and `NOT` are non-spontaneous or induced: they cannot detect their occurrences by themselves unless they either get expired or are explicitly queried. Most existing event systems, however, only detect spontaneous events, i.e., events that can detect their occurrences by themselves. For example, while Snoop [4] supports aperiodic sequence and negation constructors, these constructors, however, must always start with an initiator event and end with a terminator event, which is not general enough. The non-spontaneous nature of many RFID event constructors demands a new approach for RFID event processing and detection.

To this end, in this paper, we develop a general RFID Complex Event Detection Algorithm (RCEDA). In our approach, temporal constraints become the first class objects in the event detection phase. To support detection of non-spontaneous events, the system automatically generates *pseudo events* to actively trigger the querying of the occurrences of these non-spontaneous events.

Next, we first discuss the parameter context applicable to RFID applications, then present in detail how to effectively detect RFID complex events under such parameter context.

4.2 Parameter Context for RFID Event Detection

Parameter contexts define which instances of a complex event are actually pulled out of a history of multiple constituent events. Events can always be detected using unrestricted (or general) context, in which all combinations of instances of constituent events are returned as instances of a complex event. The unrestricted parameter context usually produces a large number of event instances. Only some of these combinations, however, are meaningful for an application. Thus, four different restricted parameter contexts have been proposed in [4], including *recent, continuous, cumulative and chronicle*.

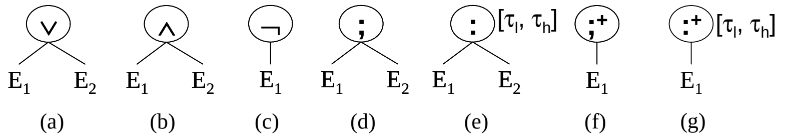

Fig. 5. Graphical representations of complex event constructors: (a) $\mathtt{E = E_1 \vee E_2}$, (b) $\mathtt{E = E_1 \wedge E_2}$, (c) $\mathtt{E = \neg E_1}$, (d) $\mathtt{E = E_1 ; E_2}$, (e) $\mathtt{E = TSEQ(E_1 ; E_2,\ \tau_l,\ \tau_u)}$, (f) $\mathtt{E = SEQ^{+}(E_1)}$ and (g) $\mathtt{E = TSEQ^{+}(E_1,\ \tau_l,\ \tau_u)}$

Among the four types of contexts, only the *chronicle* context will work for RFID events. This is because that complex RFID events often overlap with each other (e.g., Fig. 1b), since multiple readers (often deployed in a sequence of locations) produce observations simultaneously and these observations are collected and processed together. Under the other three types of contexts, there are often events matched from overlapped events which lead to incorrect detection. The *chronicle* context detects complex events in chronicle order of occurrence: the oldest initiator is paired with the oldest terminator. Thus it works properly even when instances for a complex event overlaps. For example, instances for event E in Fig. 4 under chronicle context will include $\{e_1^1,\ e_1^2,\ e_1^3,\ e_2^{12}\}$, $\{e_1^5,\ e_1^6,\ e_1^7,\ e_2^{15}\}$, which are as intended. Thus, we use chronicle context for detecting complex events in RFID applications.

4.3 Graphical Representation of Complex Events

Our event detection uses a graph-based computation model. We first introduce the graphical representation for each complex event constructor and then present how to construct event graphs for complex events in RFID rules.

Fig. 5 illustrates the graphical representation of each event constructor discussed in Section 2.2, where constituent events are represented as child nodes, and the constructed events are represented as parent nodes. We denote a node that represents an event E as $\mathtt{v}_E$. Note that the temporal sequence events are also associated with their distance constraints.

An exception is the WITHIN constructor, which is represented as an interval constraint of the constituent node. For example, Fig. 6a shows the graphical representation of an interval-constrained event $\mathtt{E = WITHIN(E_1 \wedge E_2,\ 10sec)}$. As another example, Fig. 6b shows the graphical representation of a complex event with both interval-constraint and distance-constraint: $\mathtt{E = WITHIN(TSEQ^{+}(E_1,\ 0.1sec,\ 1sec),\ 100sec)}$.

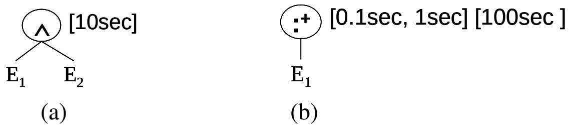

Fig. 6. Graphical representations of interval-constrained complex events: (a) $\mathtt{WITHIN(E_1 \wedge E_2,\ 10sec)}$ and (b) $\mathtt{WITHIN(\ TSEQ^{+}(E_1,\ 0.1sec,\ 1sec),\ 100sec)}$

Given a set of RFID rules $\mathtt{R = \{r_1,\ r_2,\ \ldots,\ r_n\}}$, we construct a graph representing the events for these rules in the following steps.

- *First, build an event graph for each rule's event.* For each rule $\mathtt{r}_i$ in R, we build an event graph $\mathtt{T}_i$ with leaf nodes representing primitive events, internal nodes repre-

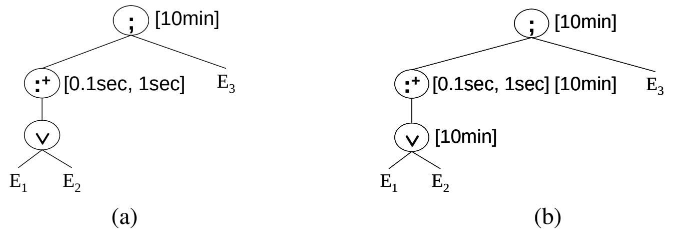

Fig. 7. Graphical representations of an interval-constrained complex event $\texttt{E = WITHIN(TSEQ}^{+}\texttt{(E}_1 \vee \texttt{E}_2\texttt{, 0.1sec, 1sec) ; E}_3\texttt{, 10min)}$: (a) before propagating the interval constraint; and (b) after propagating the interval constraint.

senting complex events and edges linking constituent events with parent complex events. The root node of $\texttt{T}_i$ represents the event part of the rule $\texttt{r}_i$.

– *Second, propagate interval constraints.* For each event graph $\texttt{T}_i$, if there is any interval constraint defined on an event node $\texttt{v}_E \in \texttt{T}_i$, propagate $\texttt{v}_E$'s interval constraint to all the descendant nodes of $\texttt{v}_E$. This is because that a complex event always has a longer interval than its constituent events. Interval constraints are propagated top-down in the event graph: given any event node $\texttt{v}_E$, its interval constraint is set to be the minimum of the current interval constraint of E (if any) and that of its parent event node, if any. For example, Fig. 7b illustrates the graphical representation of a complex event $\texttt{E = WITHIN(TSEQ}^{+}\texttt{(E}_1 \vee \texttt{E}_2\texttt{, 0.1sec, 1sec) ; E}_3\texttt{, 10min)}$ after interval propagating from Fig. 7a. We use $\texttt{v}_E\texttt{.within}$ to represent the interval constraint on event E.
– *Finally, merge common sub-graphs.* We can combine any common sub-graphs in $\{\texttt{T}_1\texttt{, T}_2\texttt{, ..., T}_n\}$ to form an event graph G, thus avoid detecting common sub-events multiple times to improve efficiency and reduce space requirements. For convenience, we use $\texttt{p(v}_E\texttt{)}$ to represent the set of nodes that are parents of $\texttt{v}_E$ in G; and we use $\texttt{r(v}_E\texttt{)}$ to represent a rule whose event part is represented by $\texttt{v}_E$.

By integrating temporal constraints into event graphs, temporal constraints become first class constructs in event detection, and are checked during the detection process, as discussed later.

4.4 RFID Event Detection Mode

Traditional graph-based event processing systems detect complex events in a bottom-up fashion: occurrences of primitive events are injected at the leaves and flow upwards to trigger parent complex events. Such a bottom-up event detection approach, however, is inapplicable to detecting RFID events. In fact, many RFID events (such as those generated from $\texttt{SEQ}^{+}$ and NOT constructors) are non-spontaneous: they cannot detect their occurrences by themselves unless they either get expired – if they are associated with interval constraints – or are explicitly queried about their occurrences from their parent nodes.

Next, we generalize three RFID event detection modes for each node $\texttt{v}_E$ in G.

– *Push*($\uparrow$): An event node $\texttt{v}_E$'s detection mode is push if E is a spontaneous event such that any occurrence of E will trigger $\texttt{v}_E$ to automatically detect the occurrences and propagate them to their parents. For example, primitive events will always automatically propagate their instances to their parents, thus are always in push mode.

- *Pull*($\downarrow$): An event node $\mathtt{v}_E$'s detection mode is pull if E is a non-spontaneous event such that $\mathtt{v}_E$ cannot determine whether instances of E have occurred or not unless being explicitly queried by $\mathtt{v}_E$'s parent node. For example, the detection mode for a NOT event is always pull.
- *Mixed*($\updownarrow$): An event node $\mathtt{v}_E$'s detection mode is mixed if its detection mode is neither *push* nor *pull*. Such event nodes are usually associated with temporal constraints. For example, the detection mode for a complex event $\mathtt{E = TSEQ^+(E_1, \tau_l, \tau_u)}$ is mixed if $\mathtt{E_1}$ is a spontaneous event. When an instance of $\mathtt{E_1}$ arrives at time timestamp, $\mathtt{v}_E$ cannot determine whether the sequence has ended or not unless there is no arrival of other instance of $\mathtt{E_1}$ during the period of [timestamp, timestamp + τ_u].

We can compute the event detection modes for the nodes in an event graph G recursively by starting from primitive event nodes on the leaf level. While the detection mode for a primitive event node is always push, the detection mode for a complex event depends on the event constructor type and the modes of its constituent sub-events.

An RFID rule r is *valid* only if the detection mode for its event E is in either push mode or mixed mode. In this paper, we propose a method to detect mixed mode events by the introduction of *pseudo events*. (If the detection mode for r's event E is pull, then occurrences of E can never be detected and thus r will never be triggered. We call such events invalid events, and corresponding rules invalid rules.)

4.5 Pseudo Events

Existence of non-spontaneous RFID events causes mixed detection mode. Mixed mode RFID event nodes cannot be supported in traditional graph-based event detection systems, which propagate event occurrences bottom up. To address this challenge, we propose to generate *pseudo events* when necessary to trigger explicit queries about the occurrences of these non-spontaneous events, i.e., in a top-down way.

A pseudo event is a special artificial event used for querying the occurrences of non-spontaneous events during a specific period, and is scheduled to happen at an event node's expiration time. We represent a pseudo event instance as $e_i'^{[t_c,t_e]}$, with its target event id i, creation time $\mathtt{t}_c$ and execution time $\mathtt{t}_e$. A pseudo event $e_i'^{[t_c,t_e]}$ will query the occurrences of event i during the period $[\mathtt{t}_c, \mathtt{t}_e]$, or non-occurrences of event i during the period $[\mathtt{t}_c, \mathtt{t}_e]$ if the constructor for event i is NOT.

For a rule r with a push mode event r.E, there is no need to generate pseudo events even though r.E contains non-spontaneous sub-events.

For example, suppose that the event of rule r is WITHIN($\neg$E$_1$; E$_2$, τ) where $\mathtt{E_1}$ and $\mathtt{E_2}$ are primitive events, any occurrence of $\mathtt{E_2}$ (e.g., $\mathtt{e_2}$) will trigger the querying about the non-occurrences of $\mathtt{E_1}$ during the period $[\mathtt{t\_end(e_2)} - \tau, \mathtt{t\_end(e_2)}]$. Thus, there is no need to generate pseudo events in this case.

For a mixed mode event r.E, however, we need to generate pseudo events to trigger the querying about the occurrences of non-spontaneous sub-events. For example, for an interval-constrained complex event $\mathtt{E = WITHIN(E_1 \wedge \neg E_2, \tau)}$ where $\mathtt{E_1}$ and $\mathtt{E_2}$ are primitive events; if $\mathtt{E_1}$ happens first, we need to make sure that there is no occurrence

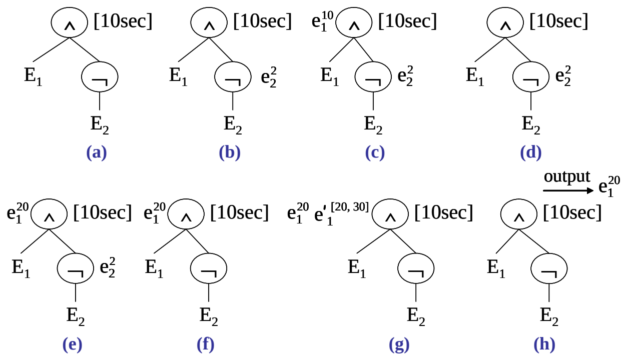

Fig. 8. An example of detecting a complex event $\texttt{E = WITHIN(E}_1 \wedge \neg\ \texttt{E}_2\texttt{, 10sec)}$ with event history $\{e_2^2,\ e_1^{10},\ e_1^{20}\}$: (a) graphical representation for E; (b) on arrival of e_2^2; (c) on arrival of e_1^{10}; (d) after processing of (c); (e) on arrival of e_1^{20}; (f) after processing of (e); (g) after arrival of pseudo event $\texttt{e}_3'^{[20,30]}$, where the event id for $\neg$ E$_2$ is 3; and (h) after processing of (g).

of $\texttt{E}_2$ within τ. Therefore, if there is no occurrence of $\texttt{E}_2$ during the interval of $\texttt{E}_1$'s instance, occurrence of an $\texttt{E}_1$ instance $\texttt{e}_1$ will create a pseudo event with the target event $\neg\texttt{E}_2$, creation time $\texttt{t\_end(e}_1\texttt{)}$ and execution time $\texttt{t\_begin(e}_1\texttt{) + }\tau$. This pseudo event will query about the non-occurrence of event $\texttt{E}_2$ during the period $\texttt{[t\_end(e}_1\texttt{),}$ $\texttt{t\_begin(e}_1\texttt{) + }\tau\texttt{]}$.

For a mixed mode event, we can determine whether a node v in the event graph G needs to generate pseudo events in a top-down way.

The notations used here include: i) $\texttt{v}_E$*.mode*: the detection mode for $\texttt{v}_E$; ii) $\texttt{v}_E\texttt{.pseudo}$: $\texttt{v}_E$'s pseudo event generation flag; and iii) $\texttt{v}_E\texttt{.pseudo\_target}$: the target event of a pseudo event from $\texttt{v}_E$.

Implementation of Pseudo Events. When pseudo events are created, they are put into a sorted pseudo queue (pseudo_queue) according to their scheduled execution timestamps. The incoming RFID event queue (event_queue) is ordered by their observation timestamps. When the event engine fetches an event, it always fetches the earliest event from the two queues.

An Example of Detecting Complex Events Using Pseudo Events

Fig. 8 illustrates an example of detecting a complex event $\texttt{E = WITHIN(E}_1 \wedge$ $\neg\texttt{E}_2\texttt{, 10sec)}$ with pseudo events. We assume an event history $\{e_2^2,\ e_1^{10},\ e_1^{20}\}$, where e_i^j represents an occurrence of event $\texttt{E}_i$ at time j. The steps are described as follows:

1. On arrival of $\texttt{e}_2^2$, $\texttt{v}_{E_2}$ propagates $\texttt{e}_2^2$ to its parent node. Since the parent node is non-spontaneous, it will not further propagate the occurrence (Fig. 8b);
2. On the arrival of $\texttt{e}_1^{10}$, $\texttt{v}_{E_1}$ propagates its occurrence to $\texttt{v}_E$, which triggers the querying about the non-occurrence of $\texttt{E}_2$ during the period $\texttt{[t\_end(e}_1^{10}\texttt{) - 10sec,}$ $\texttt{t\_end(e}_1^{10}\texttt{)]}$, i.e., [0sec, 10sec] (Fig. 8c);
3. Since there is an occurrence $\texttt{e}_2^2$ of $\texttt{E}_2$ during the period of [0sec, 10sec], $\texttt{e}_1^{10}$ cannot be a constituent instance of an E's occurrence. Thus, $\texttt{e}_1^{10}$ is deleted (Fig. 8d);
4. Similarly, on the arrival of $\texttt{e}_1^{20}$, $\texttt{v}_{E_1}$ propagates its occurrence to $\texttt{v}_E$, which triggers the querying about the non-occurrence of $\texttt{E}_2$ during the period $\texttt{[t\_end(e}_1^{20}\texttt{) -}$ $\texttt{10sec, t\_end(e}_1^{20}\texttt{)]}$, i.e., [10sec, 20sec] (Fig. 8e);

5. Since there is no occurrence of $\mathtt{E}_2$, $\mathtt{v}_E$ cannot detect its occurrence unless there is no occurrence of $\mathtt{E}_2$ during the period `[t_end(`$\mathtt{e}_1^{20}$`), t_begin(`$\mathtt{e}_1^{20}$`) + 10sec])`, i.e., `[20sec, 30sec]` (Fig. 8f). Thus, a pseudo event $\mathtt{e}_3'^{[20,30]}$ is scheduled to be generated at time 30sec to query the event node $\mathtt{v}_{\neg E_2}$. We assume that the event id for $\neg\mathtt{E}_2$ is 3;
6. The arrival of $\mathtt{e}_3'^{[20,30]}$ will trigger the querying about the non-occurrence of event $\mathtt{E}_2$ during the period `[20sec, 30sec]` (Fig. 8g). Since there is no occurrence of $\mathtt{E}_2$ during that period, occurrence of `E` is detected (Fig. 8h).

4.6 RFID Complex Event Detection Algorithm (RCEDA)

In this subsection, we discuss how to efficiently detect RFID complex events under chronicle parameter context (Algorithm RFID_COMPLEX_EVENT_DETECTION).

```
RFID_COMPLEX_EVENT_DETECTION($\mathtt{R} = \{\mathtt{r}_1, \mathtt{r}_2, ..., \mathtt{r}_n\}$)
 1   Construct an event graph G representing the rules in R (Section 4.3)
 2   //begin of initializing event graph
 3   Propagate interval constraints starting from the root node of G
 4   Assign an event detection mode for each node in G
 5   Assign pseudo event flag and target for each node in G
 6   //end of initializing event graph
 7   for each incoming event $e_1$
 8       do if $\mathtt{e}_1$ is an instance of a primitive event $\mathtt{E}_1$
 9             then for each parent node $\mathtt{v}_E$ of $\mathtt{v}_{E_1}$
10                       do ACTIVATE_PARENT_NODE($\mathtt{v}_\mathtt{E}, \mathtt{e}_1$)
11                          if $\mathtt{v}_{\mathtt{E}_1}$.pseudo
12                             then GENERATE_PSEUDO_EVENT($\mathtt{v}_{\mathtt{E}_1}, \mathtt{v}_\mathtt{E}, \mathtt{e}_1$)
13                  for each rule r whose event part is represented by $\mathtt{v}_{E_1}$
14                       do trigger the rule r
15          if $\mathtt{e}_1$ is a pseudo event
16             then let E be the target event of $\mathtt{e}_1$
17                  let tstart be the creation timestamp of $\mathtt{e}_1$
18                  let tend be the execution timestamp of $\mathtt{e}_1$
19                  EList ← QUERY_INTERVAL_NODE($\mathtt{v}_E$, tstart, tend)
20                  for each event instance e in EList
21                       do for each parent node, v, of $\mathtt{v}_E$
22                              do ACTIVATE_PARENT_NODE(v, e)
```

Given an event graph G, we first initialize G by: i) propagating interval constraints in a top-down way (Algorithm PROPAGATE_INTERVAL_CONSTRAINT); ii) assigning event detection modes bottom-up based on event constructors and interval constraints (Section 4.4); and iii) assigning pseudo event generation flags top-down based on the event detection modes (Algorithm ASSIGN_PSEUDO_EVENT_FLAG). Then, we can use this event graph to monitor the occurrences of events based on the algorithm RCEDA. The algorithm has three main functions:

– ACTIVATE_PARENT_NODE($\mathtt{v}_E$, $\mathtt{e}_1$): This recursive function propagates an event instance $\mathtt{e}_1$ from one sub-event $\mathtt{E}_1$ of `E` to $\mathtt{v}_E$ and detects whether any instance of `E` has occurred or not. If yes, $\mathtt{v}_E$ will recursively propagate its occurrence to its parent node (if any), i.e. call the ACTIVATE_PARENT_NODE function again, or trigger a rule r whose event part is represented by v_E. If the pseudo flag of v_{E1} is set to true during the event graph initialization, this function will also generate a pseudo event from v_{E1}, $e_i'^{[t_s,t_e]}$, where i is the id of v_E's pseudo event target, i.e., v_E.pseudo_target, t_s and t_e are set based on t_begin(e_1), t_end(e_1) and the temporal constraints on E.

– QUERY_INTERVAL_NODE(v_E, tstart, tend): This function queries about occurrences of the event E during the period [tstart, tend] and outputs such occurrences if any.
– GENERATE_PSEUDO_EVENT(v_{E_1}, v_E, e_2): This function will generate a pseudo event for the target event v_{E_1} on the occurrence of an event instance e_2, where e_2 is an instance of one of v_E's sub-events; v_{E_1} is either the same as v_E or a child node of v_E. The creation time and execution time for the pseudo event will depend on the temporal constraints on v_E, t_begin(e_2) and t_end(e_2).

The algorithm RCEDA works as follows:

– On each occurrence of a primitive event e_1 (of type E_1) attached to a leaf node v_{E1}, the algorithm will propagate e_1 to all the internal event nodes v_E where E_1 is a sub-event of E. That is, the occurrence of e_1 will call the function ACTIVATE_PARENT_NODE(v_E, e_1). Also, the occurrence of e_1 will also trigger all the rules whose events are represented by v_{E_1}.
– On each occurrence of a pseudo event $e_i'^{[t_s,t_e]}$, the algorithm will query about the occurrences of the target event with id i during the period [t_s, t_e], with the function query_internal_node(v_{E_i}, t_s, t_e). The algorithm will recursively propagate each occurrence, e_i, in the query results to event i's parent node v, with the function ACTIVATE_PARENT_NODE(v, e_i).
– On each occurrence e of an event E, either primitive or complex, if the pseudo flag of v_E is set to true during the event graph initialization, the algorithm will generate a pseudo event for v_E.pseudo_target. The creation and execution timestamps of the pseudo event are set based on the t_begin(e), t_end(e) and the temporal constraints between E and the target event. This is done with the function generate_pseudo_event(v_E, v, e), where v is the common parent node between v_E and v_E's pseudo target event node.

5 Performance Study

To evaluate the performance of our approach, we developed a simulator of an RFID-enabled supply chain system with warehouses, shipping, retail stores and sale to customers. Rules are defined for the system to automatically transform and aggregate data. The machine used is a Dell Latitude D610, with 2GHz Pentium M CPU and 1GB memory, installed with Windows XP. We implemented our event detection algorithm RCEDA in C#.

We tested the total event processing time versus the number of primitive events and versus the number of rules, with event arrival rate of 1000 events per second. (To simplify the test, action cost such as database update cost is not counted in the processing time.) The experiment result shown in Fig. 9 demonstrates that the cost increases almost linearly versus the number of events, and that the performance versus number of rules is also quite scalable.

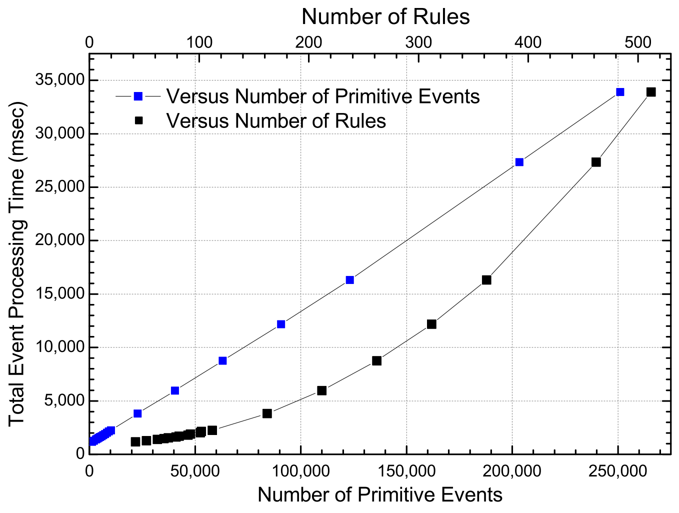

Fig. 9. Event processing time versus number of events and number of rules

6 Related Work

RFID technology has emerged for years and poses new challenges for data processing and management. The importance of event processing is pointed out in [10], but methodology is not provided. In [2], a temporal-based data model is developed for RFID data, and how to use rules to transform RFID data from observations into the data model is also discussed; however, it lacks a complete framework and implementation.

Recently, major IT vendors are providing sophisticated RFID platforms, including the Sun EPC Network [11], SAP Auto-ID Infrastructure [12], Oracle Sensor Edge Server [13], IBM WebSphere RFID Premises Server [14], Sybase RFID Solutions [15], and UCLA's WinRFID Middleware[16]. These platforms provide a general interface to collect RFID data from readers, and then forward the data to applications. These systems, however, only support limited RFID rules: in fact they only support primitive events or their simple combinations. Thus it is up to users' applications to detect complex events. RFID event processing is also discussed in [17, 18], where no formal method is proposed.

Event processing has been studied extensively in the past [19, 9, 7], in the context of active databases. These systems normally use Event-Condition-Action (ECA) rules for event processing. RFID events differ from traditional events in several ways, including the high temporal nature and existence of non-spontaneous events. Thus it is difficult for traditional event detection systems to support RFID event detection.

Temporal constraints are considered in [20, 21], which, however, cannot be used to support the special RFID events such as temporal sequence and temporal negation. Event negation is discussed in [7], where a negated event must have an initiator event and a terminate event. Motakis et al [5] provide a formal discussion of active rules including negated events, but the implementation approach is not provided.

7 Conclusions

One of the major challenges for RFID applications is to bridge the physical world represented with EPC tags, and the virtual world represented with application logic. To address this challenge, we develop an event-oriented framework that can effectively transform and aggregate raw RFID data into semantic data, by i) declarative event specification with temporal constraints; ii) declarative rules definition to support data transformation and real-time monitoring; and iii) an RFID complex event detection engine that supports temporal constraints by integrating instance level constraint checking into the detection process, and uses pseudo events to actively detect non-spontaneous events. The event framework provides comprehensive support of RFID applications, including object tracking and real-time monitoring. For the latter, the difficulty of data aggregation can now be solved soundly through complex event generation and detection. The performance study shows that our system is efficient and scalable. The technology developed in this paper is now integrated into Siemens RFID Middleware [2] to provide integrated RFID solutions for RFID-enabled business applications.

References

1. EPC Tag Data Standards Version 1.1. Technical report, EPCGlobal Inc, April 2004.
2. F. Wang and P. Liu. Temporal Management of RFID Data. In *VLDB*, 2005.
3. RFID 2004 FORUM Report. http://www.wireless.ucla.edu/techreports2/RFID- 2004-Forum.pdf.
4. S. Chakravarthy and D. Mishra. Snoop: an Expressive Event Specification Language for Active Databases. *Data Knowl. Eng.*, 14(1):1–26, 1994.
5. I. Motakis and C. Zaniolo. Formal Semantics for Composite Temporal Events in Active Database Rules. *Journal of Systems Integration*, 7(3/4):291–325, 1997.
6. The METRO Group Future Store Initiative. http://www.future-store.org.
7. S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. In *VLDB*, pages 606–617, 1994.
8. Jennifer Widom and Stefano Ceri. *Active Database Systems: Triggers and Rules For Advanced Database Processing.* Morgan Kaufmann, 1996.
9. N. H. Gehani, H. V. Jagadish, , and O. Shmueli. Composite Event Specification in Active Databases: Model & Implementation. In *VLDB*, 1992.
10. M. Palmer. Seven Principles of Effective RFID Data Management. www.objectstore.com/docs/ articles/7principles_rfid_mgmnt.pdf, Aug. 2004.
11. A. Gupta and M. Srivastava. Developing Auto-ID Solutions using Sun Java System RFID Software. http://java.sun.com/developer /technicalArticles/Ecommerce/rfid/sjsrfid/RFID.html, Oct 2004.
12. C. Bornhoevd, T. Lin, S. Haller, and J. Schaper. Integrating Automatic Data Acquisition with Business Processes - Experiences with SAP's Auto-ID Infrastructure. In *VLDB*, pages 1182–1188, 2004.
13. Oracle Sensor Edge Server. http://www.oracle.com /technology/products/iaswe/edge_server.
14. WebSphere RFID Premises Server. http://www-306.ibm.com/software/pervasive/ws_rfid_premises_server/, December 2004.
15. Sybase RFID Solutions. http://www.sybase.com/rfid, 2005.
16. UCLA WinRFID Middleware. http://www.wireless. ucla.edu/rfid/winrfid/.
17. M. J. Franklin, S. R. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O. Cooper, A. Edakkunni, and W. Hong. Design Considerations for High Fan-In Systems: The HiFi Approach. In *CIDR*, pages 290–304, 2005.
18. S. Rizvi, S. R. Jeffery, S. Krishnamurthy, M. J. Franklin, N. Burkhart, A. Edakkunni, and L. Liang. Events on the Edge. In *SIGMOD*, pages 885–887, 2005.
19. S. Gatziu and K. R. Dirtrich. Detecting Composite Events in Active Databases Using Petri Nets. In *Workshop on Research Issues in Data Engineering: Active Database Systems*, 1994.
20. M. Mansouri-Samani and M. Sloman. GEM: a Generalized Event Monitoring Language for Distributed Systems. *Distributed Systems Engineering*, 4(2):96–108, 1997.
21. G. Liu, A. Mok, and P. Konana. A Unified Approach for Specifying Timing Constraints and Composite Events in Active Real-Time Database Systems. In *RTAS*, 1998.