Cascadia: A System for Specifying, Detecting, and Managing RFID Events

Evan Welbourne, Nodira Khoussainova, Julie Letchner, Yang Li,
Magdalena Balazinska, Gaetano Borriello, and Dan Suciu

Department of Computer Science and Engineering
University of Washington, Seattle

E-mail: {evan,nodira,letchner,yangli,magda,gaetano,suciu}@cs.washington.edu

Abstract

*Cascadia is a system that provides RFID-based pervasive computing applications with an infrastructure for specifying, extracting and managing meaningful high-level events from raw RFID data. Cascadia provides three important services. First, it allows application developers and even users to specify events using either a declarative query language or an intuitive visual language based on direct manipulation. Second, it provides an API that facilitates the development of applications which rely on RFID-based events. Third, it automatically detects the specified events, forwards them to registered applications and stores them for later use (*e.g.*, for historical queries).*

We present the design and implementation of Cascadia along with an evaluation that includes both a user study and measurements on traces collected in a building-wide RFID deployment. To demonstrate how Cascadia facilitates application development, we built a simple digital diary application in the form of a calendar that populates itself with RFID-based events. Cascadia copes with ambiguous RFID data and limitations in an RFID deployment by transforming RFID readings into probabilistic events. We show that this approach outperforms deterministic event detection techniques while avoiding the need to specify and train sophisticated models.

1. Introduction

Radio Frequency Identification (RFID) technology has become increasingly popular in the last several years. New applications that use this technology are emerging both in industrial settings (*e.g.*, supply-chain management [21, 54]) and pervasive computing environments (*e.g.*, elder-care [44] and hospitals [48]). RFID enables applications to track the movements of objects and people carrying small RFID tags in an environment equipped with RFID readers.

In an RFID system, RFID readers produce streams of *tag-read events (TREs)* of the form `(time, tag_id, antenna_id)` that indicate when and where tags are being detected. The `antenna_id` is a unique identifier for the RFID antenna[1] that detected the tag. RFID applications transform low-level TRE streams into meaningful higher-level events. In supply-chain management, for example, TREs can be used to analyze the efficiency of the supply-chain process by tracking the locations that products visit (*e.g.*, factory, distribution center, store). In a friend-finder application [53], TRE streams can serve to automate sharing of a user's current or historical location, as well as the activities they perform (*e.g.*, having lunch). Finally, in the hospital scenario, TREs can help monitor the location and status of patients, staff members, and equipment [48]. For simplicity, we use a digital diary application as a running example in this paper. This application automatically populates a user's calendar with higher-level events (*e.g.*meetings, encounters, breaks) which are generated as the user moves through an office building with his RFID tags.

We propose Cascadia, a new infrastructure that greatly simplifies the development of pervasive RFID applications such as those described above. Our focus is on large-scale, passive RFID deployments within a single administrative domain such as a hospital, corporate or academic campus. We experiment with our own infrastructure, the *RFID Ecosystem* [53], which includes hundreds of RFID readers and thousands of passive EPC Gen 2 tags throughout the Paul G. Allen Center for Computer Science and Engineering at the University of Washington.

1.1. Motivation

A common way to architect a large-scale RFID application is to use a relational database management system (RDBMS). The TREs are stored directly in an RDBMS table. New events are expressed as queries and alerts are handled with triggers. This approach is perfectly adequate when applications change infrequently and are based primarily on TREs from readers at a few key locations – precisely the context of supply-chain management systems. In fact, all major RDBMS vendors now provide support for

[1] An RFID reader typically has multiple antennas (ours have four) that can be spread several meters away from the reader and from each other.

these types of applications (*e.g.*, [41]). However, in a pervasive computing environment, the mix of applications is much more dynamic and events are based on complex and often user-specific conditions. Moreover, events in a pervasive setting are likely to occur throughout a denser, more diverse set of locations, not all of which are likely to be equipped with an RFID reader.

As an example, a typical event in the supply-chain setting is the transition of a package from its source to its destination [54]. Detecting this event amounts to executing a simple query over TREs and computing a time difference. In contrast, a typical pervasive computing event is the determination that a particular group of individuals were involved in a meeting. This is a more complex event as the individuals may be only a subset of a larger group and may meet in any one of many locations. Furthermore, the chance of incomplete information and missed readings is likely to be much higher in a pervasive environment [56] due to deployment limitations and because the movement of people is less easily regulated and constrained as compared to a package passing through a loading dock door. As a result, the higher-level events used by RFID applications are more difficult to specify, detect and manage.

1.2. Contributions

Cascadia addresses the above challenges by providing an infrastructure for building pervasive RFID applications. In general, Cascadia simplifies application development in three ways: (1) it hides the low-level details of limited deployments and dirty TRE streams by exposing a high-level model of probabilistic entity movements through space; (2) it provides declarative and visual means for specifying sophisticated events on top of location information; and (3) it facilitates management of these high-level events with a simple event-based API. These high-level properties translate into the following detailed services.

1. **Decoupling applications from low-level RFID data**. Because RFID data is incomplete, dirty and often ambiguous, Cascadia uses a probabilistic model for tag movements through an environment and for the resulting events. Applications operate on that model rather than on the raw TREs (Sections 2 and 3.1).
2. **Enabling developers and users to define high-level events**. In Cascadia, applications specify high-level RFID events in a declarative fashion using *PeexL*, a sequence language with a SQL-like syntax but designed to handle dirty data and uncertain events (Section 3.2). Cascadia also provides an intuitive graphical interface, called *Scenic*, for generating event specifications in PeexL (Section 3.3).
3. **Enabling continuous high-level event detection from lower-level event streams**. Cascadia includes *PEEX, a Probabilistic Event EXtractor* that continuously extracts developer and user-defined events (Section 3.2).
4. **Facilitating management of events and metadata**. Cascadia stores all detected events in an RDBMS and simplifies their management with standard event-driven and query-based APIs (Section 3.4).

A variety of access control techniques can be used to enforce privacy in Cascadia. Our baseline privacy policy is *Physical Access Control* (PAC), which allows a user to access events that occurred only when and where she was physically present. Users may extend PAC with additional context-dependent access control policies. We refer the reader to our prior work for details [38, 47].

Overall, Cascadia is intended to support user-oriented pervasive computing applications with services for event specification, notification, and near real-time detection on top of an existing RFID infrastructure within a single administrative domain. In this paper, we present the design, implementation, and evaluation of Cascadia. We demonstrate Cascadia's practicality with results from experiments on (1) the usability of Cascadia's graphical interface, (2) the precision and recall for event detection, and (3) measurements of latency for event notification. All measurements were made on real traces collected in our building-wide RFID deployment [53, 56]. To demonstrate how Cascadia facilitates application development, we build the digital diary application mentioned above.

We present Cascadia's data model in Section 2 and describe its architecture in Section 3. We mention important implementation details in Section 4 before describing the digital diary application in Section 5. Finally, we present an evaluation of Cascadia in Section 6, related work in Section 7, and conclude in Section 8.

2. Cascadia Data Model

In this section we present Cascadia's data model, which comprises a location model, an entity model, and an event model. The data model abstracts away the many technical details and difficulties of an RFID deployment to present applications with data in a form that is easier to work with. The location model hides the details of the RFID infrastructure while capturing an abstract notion of tag location and movement. The entity model allows applications to work with meaningful entities (e.g. people, places, things). Finally, the event model defines how entity movements and relationships can map to high-level events and how these events are represented.

2.1. Location Model

Reasoning about location and movement using raw RFID data is challenging for two reasons. First, RFID antennas often fail to detect tags in their vicinity [16, 30]. Second, due to budgetary constraints or lack of foresight,

Location Model	**Event Model**
`At(time,tagID,loc,prob)`	`EventType(time,`a_1`,...,`a_n`,prob)`
Entity Model	**Event Primitives and Operators**
`People(tagID,name)`	`with` and `without`
`Things(tagID,name,owner)`	`inside` and `outside`
`Places(name,coordinates)`	`near` and `far`
Relations encoding entity lattice	`AND, LASTS, and SEQ` - Conjunctions, Duration, and Sequence

Table 1. Cascadia data model.

a deployment may not have antennas in all locations of interest. These two issues make it impossible for applications to define events on raw RFID data. We address this problem with a probabilistic model of tag location over time that decouples raw RFID data from the application-level view of it. The model must be probabilistic to account for missing sensor information. For example, in our deployment antennas are positioned solely in hallways, leaving us with no sensor data to affirm that a tag has entered a particular room - instead this must be inferred with some uncertainty. Location must be similarly inferred when an antenna fails to detect a nearby tag. This situation is common to most pervasive computing and sensor systems [14].

Cascadia's location model is embodied by the `At` relation which has the schema: `At(time,tagID,loc,prob)`. Here, the location attribute contains not an antenna identifier but a value that is meaningful to applications, which we call a *place* [25]. For example, the tuple `(1:10pm,10,room230,0.75)` indicates that at 1:10pm, the tag with ID 10 was located in room 230 with probability 0.75. For each unique `(tagID,time)` combination, `At` stores the *probability distribution* over the tag's place at the given time. Thus, for `(10,1:10pm)`, the system may store the tuple above but also the tuple `(1:10pm,10,room231,0.25)`, indicating that there was also a 0.25 probability the tag was in the adjacent room 231.

Logical views are as old as databases and have been applied to a variety of domains, including models [13]. Cascadia's contribution is to adopt and support a model that (1) abstracts away the details of missing sensor data but (2) is sufficiently low-level so as to not restrict the types of event an application can define on the data.

2.2. Entity Model

User-oriented applications need to reason about meaningful entities, not RFID tags. As such, we model *people*, *things*, and *places* as relations `People`, `Things`, and `Places` with pre-defined attributes (see Table 1). We also distinguish *mobile entities*, which include people and objects from *static entities* which are places.

Cascadia also allows applications to organize entities into a hierarchy (or lattice) with varying levels of abstraction. For example, a person, "Ana", can also be a member of a group, such as "student". The student group can in turn be part of a larger group such as "person", and so on (hierarchies for things and places are similar). The entity model allows applications to specify this type of hierarchy at runtime by adding separate relations. For example, a relation `Role` might map RFID tag numbers to groups identifying students, staff, and faculty. A relation `TypeWorker` could further group students and faculty as "flexible-schedule" workers, and staff as "fixed-schedule" workers.

Event	*% of Apps*
1) X enters the proximity of an entity	36%
2) X enters a place	21%
3) X leaves the proximity of an entity	17%
5) Object is next to/touching object	14%
6) X leaves a place	11%
7) X stays in proximity of a entity	9%
8) X stays at a place	6%
9) X is not in a place	2%
10) X and Y move to distance D apart	2%

Table 2. The most common RFID events ranked by frequency of use in the literature.

2.3. Event Model

Events are at the heart of Cascadia and the services it provides. As such, the event model must be optimized to support the specification, extraction, and management of common RFID-based events. The model must also be probabilistic so that developers and users can decide how to handle uncertainty in events.

To better understand the type and structure of common RFID events we surveyed over 100 papers from past Ubicomp conferences. For each application scenario we studied the use of any *meaningful events* that were or could be extracted from RFID data. We consider an event to be *meaningful* if it could be understood and directly valued by average users: *e.g.*, Ana entering a room is meaningful but a recent Fourier transform of a sensor signal is not.

We further categorize events as *complex* or *simple*. Complex events are complex relationships among two or more subjects (*e.g.*, "Ana takes a coffee break") and can be decomposed into simpler meaningful events (*e.g.*, "Ana left her office, is in the kitchen and has her mug"). Simple events involve two subjects in some basic relation and cannot be further decomposed. RFID offers two types of simple event: location (*e.g.*, "X is at location L") and proximity (*e.g.*, "X and Y are proximate"). We noted all uses of complex and simple events, we also recursively decomposed all complex events into simple events, noting one use for each sub-event. We then clustered all events into groups of similar events and counted the number of events in each cluster. Table 2 shows the most common events along with the fraction of applications in which they were used.

From Table 2 we abstract six *event primitives* defined as: (1) *with and without*: some mobile entities are next to or touching each other or not. (2) *inside and outside*: some mobile entities are inside or outside a place. (3) *near and far*: some mobile entities are within or beyond a given distance of each other or of a particular place. Each primitive defines a *point event* which occurs at a single point in time.

The analysis of events in the survey also showed that events are typically composed in three ways: conjunction, duration, and sequencing. Conjunction combines events which occur nearly simultaneously into a more complex event. Duration can be used to extend an event primitive in time, for example: "Ying stays in the lab for 10 minutes". Finally, sequencing composes events as a sequence in time. As an alternative to these methods, machine learning techniques have been used to derive higher-level events [43, 44]. However, as we discuss in Section 7, these techniques are typically specialized for identifying events that are pre-defined by a model or by labeled data.

Cascadia's event model is thus implemented on top of the entity relations, the base events in the `At` relation, and using the six event primitives and three operators (conjunction, sequencing, and duration) defined above. In addition, all events have a probability that represents the uncertainty which comes from the `At` relation and propagates to higher-level events. For example, low probability `ENTERED-ROOM` events could be composed to form a `START-STUDYING` event with accordingly lower probability. Events are stored in relations that bear the event's name and which can be queried using the Cascadia API. All events have a timestamp that marks the time *when the event ended*. For example, Ana's `START-STUDYING` event might be stored in a table with schema `START-STUDYING (time, person1, person2, room, prob)`. An example tuple is `(1:10pm, Ana, Bill, room605, 0.4)`, which represents Cascadia's belief that Ana and Bill are having a study session at 1:10 pm with probability 0.4.

Currently, we adopt point semantics for time; the details of interval semantics (*e.g.*, ordering and overlap of events) are much more complicated to support. With point semantics, the time of the last point event in a sequence defines the time of the event, making event ordering easy. It is important to note that the time of each point event in a sequence can be exposed as an attribute of the higher-level event. For example, a `Meeting` can be defined as a `Meeting-Started` followed by a `Meeting-Ended`. The `Meeting-Started` event exposes the start time, is defined as two people entering a room, and occurs when the second person enters. The `Meeting-Ended` event exposes the end time and occurs when one person leaves the room. However, in the remainder of this paper we use an alternate definition of `Meeting` which consists of only a `Meeting-Started` event.

3. Cascadia Architecture

In this section, we present Cascadia's system architecture (see Figure 1). At the lowest level, Cascadia receives and stores raw RFID data from a network of RFID readers. The TREs are processed with a particle filter to populate the `At` relation with smoothed, probabilistic location events.

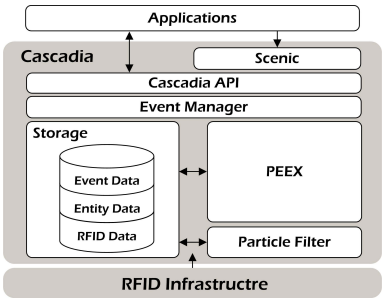


Figure 1. Cascadia system architecture.

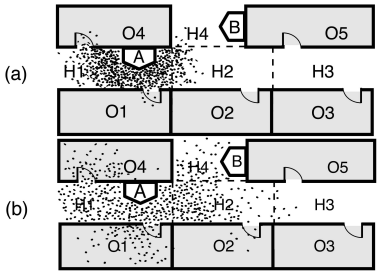


Figure 2. A particle filter's sample-based representation of the distribution over a single tag's place at two timesteps. (a) Antenna A detects the tag, creating a focused distribution. (b) No antennas detect the tag, creating a distribution with more uncertainty.

This base data is then processed by the Probabilistic Event EXtractor (PEEX), which continuously extracts and stores higher-level events. Above PEEX is the Event Manager, which is responsible for managing event definitions, subscriptions, and notifications as well as for executing queries on behalf of applications. Subscription and query services are exposed to applications with an API that supports both declarative queries and event-driven programming. Finally, Cascadia further simplifies the process of event specification with Scenic, a user-level tool that assists non-experts in specifying common higher-level events. We present each of these components in detail below.

3.1. Particle Filter

As discussed in 2.1, the exact location of a tag in an RFID system is low-level and uncertain. The `At` relation thus gives the location of a tag, x, at each timestep, t, as a *distribution* over x's possible place at t. To populate the `At` relation using RFID data requires (1) a definition of place and (2) a method for inferring distributions over place. We discuss our approach to these requirements below.

3.1.1 Defining Place

In Cascadia, the space within a building is discretized into places by an administrator (using mechanisms described in Section 3.4.1). In our experiments, each room is a place and hallways are sliced into non-uniform segments based upon

the adjoining doorways. Places can have varying size, ours are all at least several square feet in size.

3.1.2 Inferring Location

To infer place distributions from RFID readings, Cascadia uses a particle filter [17]. Particle filtering is a standard technique for inferring a "hidden" state (*e.g.*, the location of a tag) from observations (*e.g.*, TREs). This technique lends itself well to the asymmetric, multi-modal distributions typical of location estimates. While our use of particle filtering is not a research contribution, we describe the process below for completeness.

A particle filter represents the distribution over a tag's possible place with a set of samples, or *particles*, such that more likely places are associated with more samples. Each particle is associated with a specific position (*i.e.*location coordinate tuple) within a place. During updates, particles move along the edges of a connectivity graph (also defined by an administrator) that connects these positions. For the purposes of event detection, however, only the containing places (not the positions or graph edges) are considered.

To update its estimate of a tag's place as time progresses, the particle filter first moves each particle forward according to a *motion model*. Cascadia's default motion model (used in our experiments) moves particles straight down hallways at roughly 1.0 meter/second, and chooses direction uniformly at intersections (doorways are considered intersections). The set of moved particles captures the updated distribution over the tag's place. Upon receiving a TRE, the particle filter re-weights the particles using a *sensor model*: particles having coordinates consistent with the TRE are given higher weights than those with inconsistent coordinates. The default sensor model (used in our experiments) assigns a fixed, high weight (e.g. .8) to particles within the read range of the detecting antenna[2] and a low weight (e.g. .01) to particles beyond this range. After re-weighting, a new set of uniformly-weighted particles is produced from the weighted set using importance sampling with replacement, and the process repeats. The default motion and sensor models can be tweaked or replaced by an administrator.

This update process is concretely depicted in Figure 2. In the first timestep, (a), antenna A detects the tag. Particles close to A get high weights and are more likely to be resampled, producing a distribution that is fairly concentrated around A: *i.e.*, the tag's place is fairly certain at that time (in the figure, hallway H1 has roughly 0.85 probability). In the next timestep, (b), no antennas detect the tag. Without any sensor input, the particle filter cannot determine whether the tag entered an office or whether it perhaps remained in the hallway and simply was missed by the readers. The particle distribution reflects this ambiguity–it is more diffuse and covers a larger set of places than the distribution in (a). These particles will continue to disperse with each timestep until another TRE re-focuses the distribution. It is important to note that when another TRE finally occurs, particles within range of the antenna will be more heavily weighted and hence probably resampled, causing many particles with non-viable location coordinates to disappear. However, the particle filter does not "teleport" to the coordinates of the TRE any particles that were not already there.

The `At` relation is populated at each timestep using the corresponding set of particles: each place P containing at least one particle produces a new `At` tuple for P with a `prob` that is the sum of the particle weights inside that place. An important consequence of this construction is that when a particle distribution is diffuse, as in Figure 2(b), the probabilities of *all* `At` tuples for this timestep–including the correct one–are low because the probability mass is spread across many places. Thus low absolute probabilities can still identify meaningful events. As an example consider a distribution in which place P_1 has probability .2 and all other places have probability .01. This distribution suggests that it is twenty times more likely that the tag is in P_1 than anywhere else. In contrast, a distribution where P_1 and P_2 both have probability .5 actually has *less* certainty about the tag's place, despite the high absolute probability values.

Finally, we note that particle filters generally produce distributions in which a small number (1-3) of places have significant probability (*e.g.*$>$ .2), while all remaining places have diminutive probabilities (*e.g.*$<$ .01).

3.2. PEEX

PEEX [36, 37] is Cascadia's event detection subsystem. It takes declarative event specifications as input and continuously extracts the specified events from base data in the `At` relation. For example, in our calendar application, PEEX takes specifications for events like `Meeting` and extracts the events while the particle filter populates the `At` relation.

3.2.1. PeexL Query Language

PeexL is a declarative query language for specifying high-level probabilistic events for PEEX. Event specifications in PeexL have the form:

```
FORALL I1, I2,..., In
[ CTABLE C ]
WHERE Condition
CREATE EVENT E
SET Assignments
```

The arguments to the `FORALL` clause, $I_1, \ldots, I_n$, correspond to `At` events, other composite events, or to regular database tables and may optionally be preceded by a negation `!`. The `CTABLE` clause specifies an optional, developer-defined confidence table, which helps in handling ambiguity as we discuss in the following section. The `WHERE` clause is as in SQL with the addition of the `SEQ` predicate which

[2] An antenna's read range is about nine feet in our RFID deployment.

```
1   FORALL At A1, At A2, At A3, At A4
2   CTABLE MeetingStats C
3   WHERE SEQ(AND(A1, A2), AND(A3, A4))
4      AND A1.tag = 'Bill' AND A2.tag = 'Ana'
5      AND A1.loc <> 'DB Lab' AND A2.loc <> 'DB Lab'
6      AND A3.tag = 'Bill' AND A4.tag = 'Ana'
7      AND A3.loc = 'DB Lab' AND A4.loc = 'DB Lab'
8      AND C.person1= A1.tag AND C.person2= A2.tag AND
9      AND C.room = A3.loc
10  CREATE EVENT MEETING E
11  SET E.person1 = A1.tag,
12     E.person2 = A2.tag,
13     E.room = A3.loc;
```

Figure 3. Meeting Event in PeexL

we borrow from [6, 58]. `SEQ(`I_1`,` I_2`, ...,` I_m`)` states that I_j`.time` $\leq$ I_{j+1}`.time` for $j \in [1, m-1]$. One can also specify that an argument to the `SEQ` operator `LASTS` for a specified time. Finally, the `CREATE EVENT` and `SET` clauses define the name and the attributes of the new event.

Figure 3 illustrates a PeexL query that extracts `MEETING` events. This query specifies that if Ana and Bill are outside the database lab (lines 4-5), and then they are inside the database lab (lines 6-7), then they may be having a meeting in the database lab (lines 10-13). The ordering of events is determined by the `SEQ` construct (line 3) which specifies that both Ana and Bill are outside and then inside. The `CTABLE` clause specifies the confidence table for the event.

3.2.2. PEEX Event Detector

Event extraction in PEEX is performed by an Event Detector that runs periodically. Like base events in the `At` relation, all extracted events are stored persistently using one relation per event specification. The Event Detector assigns a probability to each newly detected event before storing it in the appropriate relation.

The Event Detector operates with two time windows. The first window, Δ, is a longer window (*e.g.*, about one day worth of data). It bounds the time range in which the Event Detector searches for events to ensure constant performance in face of a growing data archive. Δ should be small enough to ensure good system performance while still covering the most common types of events (Table 2). The second window, δ, specifies the frequency at which the Event Detector executes. δ determines the latency of event detection but does not affect what events are being detected. Both time windows are set by an administrator.

Extracting Events. During event extraction, the Event Detector leverages the underlying RDBMS where events are stored. To do so, it transforms PeexL event definitions into SQL queries that it executes every δ seconds. There are six key parts to this transformation. *(1)* All `SEQ(`I_1, I_2`,..)` constructs are transformed into explicit predicates on input event timestamps. *(2)* `LASTS` predicates are translated into *count sub-queries*. For example, if an underlying event must last for 10 seconds, the translated SQL specifies that the event must occur once, then again ten seconds later, and also at the eight distinct timesteps in between (thus the count is 10). *(3)* Negations are re-written into outer-joins, which join two relations but include tuples without matches in the result. *(4)* To avoid repeatedly detecting the same events on successive runs, the Event Detector transforms event definitions into stateful, *incremental* queries. These queries only retrieve combinations of low-level events in which at least one has occurred in the most recent δ window. *(5)* Additionally, the Event Detector inserts a predicate stating that all the underlying events must occur within the larger time window (Δ seconds). *(6)* Finally, the generated SQL includes a calculation that computes the probability of the event as a function of the probabilities of the events on which it depends as well as the appropriate probability from the corresponding confidence table.

Computing Event Probabilities. The intuition behind confidence tables is that many higher-level events are correlated with the attributes of underlying lower-level events. For example, it may be that when Ana and Bill are outside the lab and then inside the lab, they usually start a meeting. In contrast, when Ana and Ying enter the lab, it may be more likely that they are simply crossing paths. Confidence tables help capture such correlations. They take the form: `CONF_TABLE(`$A_1, A_2, \ldots, A_n$`,comb-prob,prob)`.

If confidence tables are to be used, a developer or an administrator must populate them by providing PEEX with training data. In this case, training data amounts to `At` traces that are labeled with complex events. The `At` traces in this data are probabilistic while the labels are deterministic (*i.e.*, have probability 1).

Confidence tables aim to capture the probability that a complex event occurs *given that* the underlying event combination occurs with some probability (*e.g.*, the probability that a specific `Meeting` event occurs given that the underlying `At` event combination occurs with probability 0.2.) For example, consider the `Meeting` event combination *i.e.*, four `At` events, two indicating that Ana and Bill are outside a room followed by two indicating they are both inside the room. If PEEX learns that out of all the times where this combination of At events occurs with probability 0.2, then the Meeting event also occurs 90% of the time, then it will append to the confidence table the tuple `('Ana', 'Bill', 'DB Lab', 0.2, 0.9)`. This rule in the confidence table says that if PEEX sees this combination of `At` events for 'Ana' and 'Bill' with probability 0.2, then it must generate the `Meeting` event with probability 0.9.

The confidence table used in Figure 3 has schema `MeetingStats(person1,person2,room,comb-prob, prob)`. It maps each `(person1, person2, room, comb-prob)` tuple to the probability that `person1` and `person2` are indeed having a meeting in `room` given that the probability they were seen outside and then inside the room is `comb-prob`.

One can think of the confidence table as a set of functions. There is a function for each `(person1, person2, room)` tuple, which maps `comp-prob` (the probability of the underlying event combination) to `prob` (the probability that the composite event occurs). Let's call this function $f_{(person1,person2,room)}$. In practice, it is impossible to learn these functions precisely, especially with a small amount of training data. As such, PEEX assumes that the functions are linear and it only learns the gradient for each function. PEEX can also use the gradient to easily calculate the probability of a complex event - it simply multiplies the probability of the underlying event combination by the gradient value found in the estimated confidence table. PEEX could use any method for estimating the gradient of the function given some training data. In our experiments, we take the slope of the line going through the origin and the centroid of the training data points as the gradient for the function.

As an example, consider the rule above: 90% of the time when PEEX sees the four-`At`-events combination with probability 0.2 for 'Ana' and 'Bill', they are indeed having a meeting. If this is the only rule PEEX learns for 'Ana' and 'Bill', the learned gradient will be 4.5. However, if PEEX sees the underlying event combination in the testing data with probability 0.1, then it will produce a `Meeting` event for 'Ana' and 'Bill' with probability $0.1 * 4.5 = 0.45$. Our linearity assumption means that the result of this calculation can be greater than 1. If this is the case, PEEX assigns probability 1 to the extracted event.

The attributes of a confidence table are defined by the developer or administrator who creates it. For example, an administrator may assert that the probability of a `Meeting` event never depends on which people enter the lab but instead only on the lab and the time of day at which they enter it. To do this, the administrator would specify a confidence table with schema `MeetingStats(room, time-of-day, comb-prob, prob)` and alter the event specification to indicate which attributes of the underlying event combination match which attributes of the confidence table. Another interesting attribute that could be used in a confidence table is the *duration between* the underlying events. For example, when defining the `Meeting-Ended` event (described in Section 2.3), the confidence table can include the duration of the meeting (*i.e.*, difference in time between the `Meeting-Started` event and the time at which the first person left the room).

3.3. Scenic

Authoring PeexL event specifications may be difficult for developers and impossible for users. As such, Cascadia provides Scenic, a tool for visual specification of events. Scenic is primarily intended to allow end-users to quickly and easily create or customize event specifications at runtime. Non-expert developers can also use Scenic at design-time to generate PeexL for their applications. For simplicity, Scenic does not allow specification of confidence tables, this is left to developers and administrators.

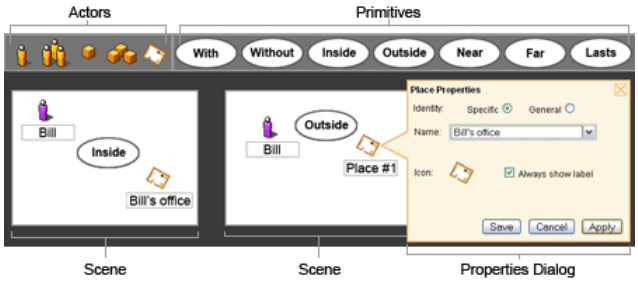


Figure 4. A screenshot of Scenic showing scenes, actors, primitives and a properties dialog for an actor.

Scenic uses an iconic visual language that represents event primitives and entities as icons which can be dragged and dropped onto a storyboard to specify a sequence of point events, or *scenes*. Thus, to specify an event users just "tell the story" of the event, scene by scene. Figure 4 shows the Scenic interface, which consists of a toolbar, below which is a working area called the sequence panel.

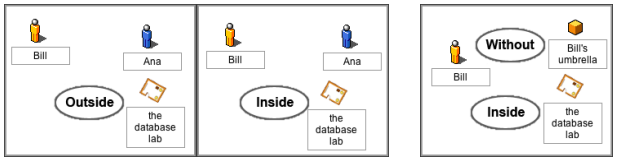
Scenes. Scenes represent point events in a sequence and are displayed as white panels over the grey sequence panel. Users can compose scenes as sequences (*i.e.*, apply the `SEQ` operator) by arranging them horizontally on the sequence panel; time is assumed to flow from left to right, with each scene strictly following the previous one (*i.e.*, not overlapping). Scenes can be inserted and deleted with a few clicks.

Actors. Actors represent entities in a point event and map to relations `People`, `Things`, and `Places` in Cascadia's entity model. Scenic also supports groups of people or things which map to higher levels in the entity lattice (*e.g.*, a group of four "students"). Each type of actor is displayed in the toolbar as a separate icon; by clicking on an icon, users can create and drag a new actor into a scene. Each successively created actor is represented by a different color icon and a distinct, anonymous unique identifier (*i.e.*, "Person #4"). After dropping an actor into a scene a user can right-click and set the actor's identity as either specific (*e.g.*, "Ana") or general (*e.g.*, "student"). Bounds for the size of a group can also be set this way. The list of available identities for an actor is retrieved from the entity tables when Scenic starts.

Primitives. Scenic provides icons for the event primitives and the `LASTS` operator. Conjunctions (`AND`) are indicated by dragging multiple primitives onto the same scene. Primitives are created and used in a similar way to actors. By dragging an event primitive onto a scene, a user can specify a relationship between actors in that scene. The *near*, *far*, and *lasts* primitives also have properties that can be set to indicate how near or far actors must be from each other, or how long a scene must last.

```
⟨EventSpec⟩     ::= ⟨ExtScene⟩+
⟨ExtScene⟩      ::= ⟨Scene⟩ | ⟨Scene⟩ 'lasts' int
⟨Scene⟩         ::= ⟨WithEntity⟩
                  | ⟨WoEntity⟩
                  | ⟨MobileEntity⟩ ('near'|'far') int ⟨MobileEntity⟩
                  | ⟨WithEntity⟩ ('in'|'out'|'near'|'far') Loc
⟨WithEntity⟩    ::= ⟨MobileEntity⟩
                  | ⟨WithEntity⟩ ('w') ⟨MobileEntity⟩
⟨WoEntity⟩      ::= ⟨MobileEntity⟩ 'wo' ⟨MobileEntity⟩
⟨Entity⟩        ::= ⟨MobileEntity⟩ | Loc
⟨MobileEntity⟩  ::= (Person)+ |Group-of-people
                  | (Thing)+ |Group-of-things
```

Figure 5. **Grammar for Scenic.**



(a) (b)

Figure 6. **(a) In the grammar. (b) Not in the grammar.**

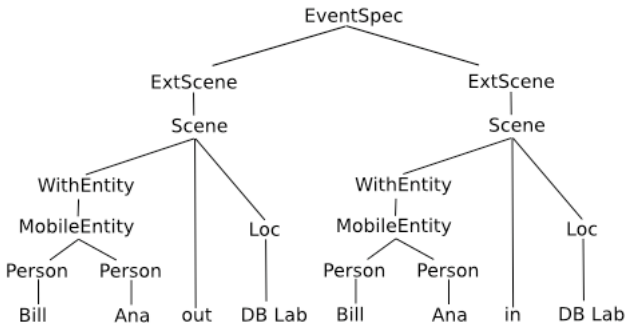


Figure 7. **Parse tree for the example meeting event.**

Scenic construct	PeexL construct
EventSpec	SEQ operator
Scene	one component in SEQ
ExtScene	as Scene but with a lasts clause
with/wo	AND clause (an ! for without) + loc is same
near/far	using auxiliary table Loc2LocDist
in/outside	check if loc is equal (or not) to a room
Person/Obj	one At in the FORALL clause
Group	multiple Ats in the FORALL clause

Table 3. **Translation table**

3.3.1. Scenic Grammar

Figure 5 shows the grammar for Scenic event specifications. Each `EventSpec` consists of one or more `ExtScenes` (extended scenes). Each `ExtScene` is either an instantaneous `Scene` or a `Scene` with a duration. `Scenes` can be: (1) a set of `MobileEntities` in the same unspecified place, (2) a `MobileEntity` without another specified `MobileEntity`, (3) a `MobileEntity` near or far from another specified `MobileEntity`, or (4) a set of `MobileEntities` inside, outside, near, or far from some place.

When an event is specified in Scenic, it either has exactly one translation in the grammar or is rejected by the grammar. Figure 6(a) shows an example of an event that is in the grammar. It is parsed as shown in Figure 7. Figure 6(b) can not be translated using the grammar because it is not clear whether the user means 'Bill is inside the DB lab without his umbrella' or 'Bill's umbrella is inside the DB lab without Bill'.

3.3.2. Converting Scenic Event Specifications into PeexL

After specifying an event in Scenic, a pop-up dialog appears, prompting the user to name the new event and select its attributes from a list. This process specifies the schema of the new event which is then sent to PEEX along with a PeexL translation of the event specification. Table 3 illustrates the correspondence between components of the Scenic grammar and components of a PeexL event specification. Predicates on actor attributes are translated into SQL predicates, possibly including a join with a relation in the entity lattice. For example, Figure 7 is translated into PeexL (Figure 3 without the `CTABLE`) as follows. First, we add an `At` relation in the `FORALL` clause for each `Person`. Second, we add a `SEQ` predicate with two arguments, one for each `Scene`. Because both `Scenes` include `WithEntity`, we use the `AND` predicate, thus producing `SEQ(AND(A1, A2), AND(A3,A4))`. Next we add a predicate for `A1`, `A2` specifying that their location *is not* the 'DB Lab', and one for `A3`,`A4` specifying that their location *is* the DB lab. Finally, we add the remaining predicates that specify the `tag` for each of `A1`, `A2`, `A3` and `A4`.

3.4. Event Manager

The Event Manager serves as the intermediary between applications and Cascadia's services for storage and event detection (Figure 1).

3.4.1. Application Interface

The Event Manager's services are exposed with a Java API that presents entities and events as first-class objects and supports both query-based and event-driven programming models. The API's central class is `CascadiaClient`, which contains methods for creating, deleting, and querying entities and events, and for subscribing to streams of events. Figure 8 lists the methods in `CascadiaClient`.

The `Person`, `Thing`, and `Place` classes can be used to create, update, or delete persistent entities. Each has a set of attributes that match the entity model (see Table 1) and *dynamic attributes* (*e.g.*, `location` for a `Person` and `occupants` for a `Place`) that are continuously updated with a list of K most likely values. The API provides `addPersonGroup` and similar methods to define the lattice of entity values. For applications with admin privileges,

```
// Entity methods (similar for places and things)
boolean      addPerson(Person p);
boolean      addPersonGroup(PersonGroup pg);
boolean      removePerson(Person p);
boolean      removePersonGroup(PersonGroup pg);
List<Person> listPeople();
Person       getPerson(String name);
PersonGroup  getPersonGroup(String name);

// SQL query interface
Query         createQuery();
PreparedQuery prepareQuery(String sql);

// Event methods
RegEvent    addEvent(EventDef definition);
boolean     removeEvent(EventDef definition);
List<Event> getEvent(String name);
List<Event> listEvents();
EventStream subscribeEvent(Subscription subscription);
boolean     unsubscribeEvent(EventStream stream);
List<Event> queryEvent(RegEvent re, long startTime,
                long stopTime, double probThreshold);

// For receiving system-related events
void addListener(CascadiaListener listener);
```

Figure 8. List of methods in the Cascadia client API.

`addLocNode` and `addLocEdge` can be used to define the connectivity graph (*i.e.*, Voronoi diagram) for places.

The API also provides an SQL query interface for querying entity and event relations. This interface is implemented as a wrapper around a read-only JDBC `Connection` with only the `createStatement` and `prepareStatement` methods exposed (as `createQuery` and `prepareQuery` respectively). Queries are currently answered using a standard RDBMS, which is sufficient for simple selections on past events. For more sophisticated queries over probabilistic events, we will eventually replace the RDBMS with a probabilistic database such as MystiQ [10].

Events are defined and handled using a set of classes that encapsulate event-related concepts. Figure 9 illustrates how an application can use these classes to connect to Cascadia, register a PeexL event definition, and subscribe to receive specified events. In addition to the event, a subscription also specifies a minimum probability threshold and the max number K of most likely events the application wants to receive after each time window. To process newly detected events, applications implement an `EventHandler` interface. The `CascadiaListener` interface allows applications to receive system-level events such as the addition of a new entity.

3.4.2. User-Specific Data and Event Templates

To facilitate management of user-specific data, the Event Manager maintains user-specific repositories of events and entities. All events and entities that have been defined for a given user are stored in these tables and may be accessed by any application authorized by that user.

Though users can define and store their own events using Scenic, many applications are developed with particular types of events in mind. As such, the Event Manager allows

```
1  CascadiaClient c =
2           new CascadiaClient(user,password,host,1234);
3
4  String   peexL  = <definition>
5  String   schema = <schema>
6  EventDef ed     = new EventDef(peexL, schema,
7                          "My Event", "Example Event");
8
9  RegistedEvent re     = c.registerEvent(ed);
10 Subscription  sub    = new Subscription(re,3,0.75);
10 EventStream   stream = c.subscribe(sub);
11
12 stream.addEventHandler(this);
```

Figure 9. A code snippet which shows event definition, registration, subscription to an event stream, and addition of event handlers.

event templates in which all entities are variables. Users can load event templates in Scenic to customize their specification. For example, a developer might define an event template, `MEETING` that specifies a meeting between two people in some room. Ana may edit `MEETING` with Scenic to refer to she and Bill in the DB lab. Applications store event templates in the event repository with a special template flag, and Scenic can be invoked with a query string which indicates that it should load a particular event template.

3.4.3. Event Filtering

The Event Manager filters the top-K events provided by `eventOccurred` to exclude those with probability below the subscription's threshold. Entity objects also support entity-based filtering. Applications can implement an entity event handler (*e.g.*, `PersonEventHandler`) to handle events about a given entity. The entity object will then invoke the application's entity event handler with itself and a list of the top-K most likely events (if any) involving that entity for the time window.

4. Cascadia Implementation

Cascadia's Event Manager, PEEX, and Particle Filter comprise 122 classes and over 14,000 lines of Java code, not including comments or parser code generated by ANTLR. We use Microsoft's SQL Server RDBMS with the Java JDBC API. Secure network communications are implemented using Apache's MINA framework. Scenic consists of DHTML with about 9,000 lines of custom Javascript code. AJAX is used to support a streamlined connection to a Java Servlet which has 15 classes, 2,000 lines of code and runs on the Apache Tomcat web server.

5. Example Application

To demonstrate how Cascadia can facilitate application development, we implemented a digital diary application which records occurrences of user and developer defined events in a Google calendar [22]. This application could be a useful tool for analyzing how, where, and with whom one has spent one's time.

The digital diary has two components: a daemon that continuously receives and posts newly extracted events, and a web-based calendar that the user loads to review her diary and edit its settings. The calendar supports three views in which the top-1, top-2, or top-3 detected events for each time period are displayed. The calendar also offers controls for adding, specifying, and customizing the events for the diary. Clicking "create new event" or "customize event" launches Scenic as a floating tool-bar above the calendar in the browser.

The core logic of the diary daemon uses Cascadia's event-driven programming interface to receive and post newly extracted events. The daemon also implements event-specific logic for advanced handling of developer-defined events. For example, the diary comes with the built-in event `ENCOUNTER`, for recording encounters between people, as well as pairs of `MEETING-START` and `MEETING-END` events which record long duration meetings between people. However, the diary filters out redundant `ENCOUNTER` events that occur close in time to `MEETING-START` or `MEETING-END` events as people gather or disperse. An additional filtering thread which processes a queue of recently received events is used to implement this logic. We discuss how Cascadia facilitated the development of the digital diary application in Section 6.3.

6. Evaluation

In this section, we present an evaluation of Cascadia's key components.

6.1. Scenic Evaluation

We evaluated Scenic with a laboratory study that included 11 participants: 6 with computer science (CS) backgrounds and 5 with non-technical or non-programming (non-CS) backgrounds. Participants were offered $20 to participate in a 90 minute study session. After a 10 minute tutorial and practice period, participants were given a series of 22 timed event specification tasks. Each task included a general, high-level English description of an event from the literature that participants were asked to specify as precisely as possible. Of the tasks, 12 were *simple tasks*, which presented descriptions of simple events, likely to be specified in one scene with one primitive. Another 10 were *complex tasks* which presented descriptions of complex events, likely to be specified with multiple scenes and primitives. The study session concluded with a questionnaire that asked participants to rate various aspects of Scenic on expressiveness, precision, and ease-of-use. Finally, we hired 2 coders (both CS) to rate the how well each event specification from each participant captured the original English event description. The coders were trained as experts with Scenic and the basic workings of RFID systems during the first hour of a 2 hour session in which they rated every event specification.

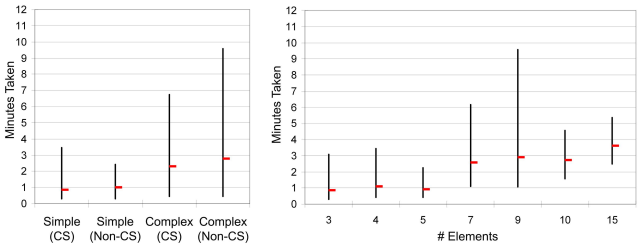


Figure 10. Min, avg, max specification times. (a) For CS and Non-CS. (b) For increasing number of elements.

Overall, CS participants spent 93 seconds per event specification task on average, with a standard deviation of 21 seconds, while non-CS participants spent an average of 111 seconds with a standard deviation of 42 seconds. Figure 10 describes the timing results in detail. The first plot shows that on average, participants could complete simple tasks within 1 minute and complex tasks within 3 minutes. A few participants took significantly longer on complex tasks. In the second plot we clustered tasks by the number of elements in the event (*i.e.*, number of actors + number of relations between actors). This metric directly correlates with the number of operations (*e.g.*, click and drag, dialog interaction) that must be performed to complete the specification. As the number of elements increases, the average time taken per event remains flat for small numbers of elements (< 5) and afterwards increases approximately linearly with the number of elements in the event. The maximum time, however, may increase much more rapidly, showing again that some participants needed to think longer about more complex events. The slight drop in time taken for the events with 10 and 15 elements is due to the fact that these events were logically quite simple but involved many entities - so they required less thought and the time was dominated by manual operations.

Scenic received an overall average rating of 3.9 for expressiveness, 3.4 for precision, and 4.1 for ease of use on a 5 point scale (where 5 is the most and 1 is the least). The lowest ratings received were for precision; participants explained that they were unsure whether the system would correctly interpret their intended meaning for some complex events. For example, in specifying the event "Bill prints his meeting notes, picks them up from the printer room, and returns to his office", one participant did not specify that Bill picked up the meeting notes in the printer room, only that he had them after he returned. The user later expressed concern about whether the system would be able to infer what was intended. The highest ratings were for actors, showing that participants appreciated the direct representation and manipulation of entities. Some participants (both CS and non-CS) really enjoyed using the prototype and even spent extra time to neatly arrange their icons before submitting an

event. Participants requested features to improve the usability such as: copy-and-paste for actors and scenes and some explicit support for representing conjunctions of simultaneous events. We plan to add these extensions in future work.

On average, the coders gave a rating of 6 to specifications for simple tasks and 5.5 to specifications for complex tasks, with standard deviation under 1.5 for both on a 7 point scale (where 7 is the most similar and 1 is the least similar). The mode rating for specifications from each simple task was 7. Events specifications for three of the complex tasks had a mode rating of 6, while specifications for the other complex tasks had mode rating of 7. Coders explained that ratings less that 7 were assigned most often due to missing details in the specification (*e.g.*, exemplified in the above example with Bill). These results show that not only could users quickly create event specifications, but they could create specifications which were arguably correct.

6.2. PEEX Evaluation

In order to evaluate PEEX we collected data for 1 hour with 10 participants in our building-wide RFID deployment. Each participant had with them several tags including their badge, keys, laptop, and mug tags. There were a total of 44 unique tags in our trace. We collected 11585 TREs. The Particle Filter tracked tag movements at a granularity of 1 second using 500 particles per tag, producing a total of 16,141,789 `At` events. For efficiency, PEEX ran only over `At` events with probability greater than 1%, of which there were 2,303,702.

We evaluate PEEX using the `ENTERED-ROOM` event defined as a sequence of two `At` events one outside a particular room and one inside that room. To collect ground truth, we asked participants to label when they traveled between rooms and which objects they carried with them for each trip. We use the labeled data for the first half-hour to populate confidence tables and the remainder to evaluate PEEX' performance.

For this evaluation, we say that an extracted event E correctly captures a labeled event E' if E is within 60 seconds of E'. We use an approximate 60 second window to cope with the varying degrees of inaccuracy in the labeled data. Indeed, participants typically noted when they entered or exited rooms either several seconds before or several seconds after the event actually occurred.

We measure the recall and precision achieved by PEEX. *Recall* is the fraction of labeled events that are captured by some extracted event. *Precision* is the fraction of extracted events that capture a labeled event. Figure 11 shows the results. The x-axis shows the probability threshold, x: for each such threshold, we measure the precision or recall only for those events that were assigned a probability equal to or higher than the threshold. Hence, $x = 0$, shows the precision and recall for all detected events, while $x = 1$ shows the precision and recall only for certain events. For each probability threshold, the graph shows the minimum, first quartile, median, third quartile and maximum across all tags (for the recall graph, the minimum and maximum are 0 and 1 for each threshold). More specifically, we measure recall and precision per tag and the graph summarizes the results across 30 out of 44 tags. The results for the remaining 14 tags were unusable. Either not even one TRE was generated for these tags or the tags failed to be detected for tens of seconds at a time. For the most part, these abnormally high error rates were due to tags being attached to water bottles (water absorbs RF signals) or laptops (metal reflects RF signals). In all cases, tags attached to participants and most of their objects were properly detected and their results are shown in Figure 11.

As the graph shows, PEEX can achieve both a precision and recall of up to 100% with many tags. In general, however, PEEX offers a flexible trade-off between precision and recall unlike a deterministic approach. Indeed, there are two possible deterministic approaches. The *first deterministic approach* is to extract only events that have occurred with certainty. In the graph, this corresponds to $x = 1$. The *second deterministic approach* is to generate all events that have any chance of having occurred ($x = 0$). PEEX can provide a higher recall than the first deterministic approach: the median recall improves from 29% for $x = 1$ to 87% for $x = 0$. PEEX can also deliver higher precision than the second deterministic approach to applications that require it. The median precision increases from 40% for $x = 0$ to 96% for $x = 1$.

Figure 11 shows that, unlike a deterministic approach, PEEX allows applications to choose their desired trade-off between recall and precision by considering only events above some probability threshold. In our example, for $x = 0.4$, the recall for the third quartile is 0.75 while the precision is 0.94. While it seems low, 0.4 is a reasonable probability threshold for two reasons. First, higher-level event probabilities are misleadingly low because of the multiplicative way in which they are computed. For example, an `ENTERED-ROOM` event comprises two lower-level events (outside and inside room) and will be detected with probability equal to the *product* of these event probabilities (*e.g.*, even fairly high probabilities for the lower-level events such as 0.63 and 0.63 or 0.8 and 0.5 will result in a probability of 0.4 for `ENTERED-ROOM`). Second, recall from Section 3.1 that primitive `At` events can be significant even when detected with only low (*e.g.*, 0.2 or higher) probability. Taken together, these two facts indicate that an application should not use a particularly high probability threshold to achieve meaningful results. Overall, Figure 11 shows that PEEX can effectively detect events over data collected in a real RFID deployment, providing good recall and precision in spite of the high uncertainty in the data.

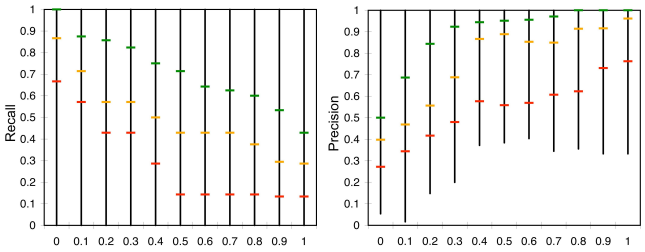


Figure 11. Recall and precision graphs for `Entered-Room` events, x-axis is probability threshold. Results show all quartiles.

Diary Component	Classes	Lines of code
`Parsing config files and startup`	8	400
`Creating events and event streams`	1	50
`Incorporating user-defined events`	1	50
`Event handlers`	1	25
`Entity management`	1	200
`Event management and processing`	3	300
`Populating Diary with Google APIs`	1	150

Table 4. Table showing the approximate break down of code for the digital diary application.

In addition to measuring recall and precision for different probability thresholds we also measured them for different values of k, in a top-K approach. Here, for each time window, we keep only the top K events (ordered by probability). However, the graph is almost identical to Figure 11 and we omit it. The top-K approach does not have a significant benefit in our experiments because there was little ambiguity in event definitions. Participants entered rooms that were far apart from each other; they never entered a room with many adjacent rooms which would cause ambiguous event detections. The main source of ambiguity came from the fact that participants sometimes entered rooms and other times simply passed in front of them. In this case, the top-K approach does not help. In other deployments, however, we expect the combination of top-K and threshold to yield almost the same recall for a better precision than using a threshold alone.

6.3. Application Development

The digital diary implementation shows how Cascadia can greatly simplify application logic in RFID event-based applications. The diary took only 3 days to develop and consists of about 1,200 lines of Java code in 12 classes. The code break down is presented in Table 4. As shown in the table, the bulk of the application logic dealt with the filtering of Cascadia events (*i.e.*, event management).

6.4. System Performance

We characterize Cascadia's performance by measuring the latency introduced by the Particle Filter, PEEX, and the Event Manager while extracting `ENTERED-ROOM` events from the 1 hour trace. Each component was run on a Dual Xeon 3GHz server with 8GB of RAM and 700GB of disk.

The time taken by the Particle Filter per timestep depended on the number of tags being tracked. We found that the Particle Filter could maintain real-time performance (one update per second) for at least 175 tags (using 500 particles each). In measuring the performance of PEEX, we assumed that the `At` events table, along with any other helper tables, had an index on `time` and `tagID`. PEEX took a total of 77.5 s to learn the confidences over the 2,303,700 `At` events, and an average of 48 ms per 5 s time window to extract the `ENTERED-ROOM` events. Finally, the Event Manager took an average of 2.69 ms to retrieve and send each event stream update to a remote application. From these results we conclude that Cascadia can easily run in near real-time for hundreds of users. Additionally, the system could scale even more with parallelization and a few additional servers.

Finally, we note that while we currently store a set of `At` tuples for every tag and every timestep, compression or pruning can easily be applied to reduce storage demands. Most simply, `At` tuples (or detected events) can be compressed or deleted after an expiration window (say, 48 hours for `At` tuples and 1 month for detected events). More sophisticated techniques might compress or prune the data to reduce its size while maintaining a majority of the information (e.g. the `At` tuples over a long interval during which a tag does not move might be compressed into a single distribution).

7. Related Work

Related work for Cascadia spans a variety of areas.

Infrastructures for pervasive computing. Many systems have been built to provide event services for pervasive computing. Many of these systems, such as the Context Toolkit [51], Gaia [50], Aura [52], Solar [7], and ConFab [26] have sought to address issues in addition to event services such as discovery, allocation, and management of resources, potentially heterogeneous sensors, and distributed computing. By contrast, Cascadia only uses RFID data and assumes that an infrastructure for aggregating and centrally storing that data is readily available (as would be the case in a hospital, corporate campus, or smart home). Furthermore, none of these systems has focused on providing Cascadia's combination of user and developer-level support for expressive, declarative event specification, subscription, notification and management on top of a probabilistic data model that targets pervasive computing.

The ParcTab [55], Sentient Computing [2], Event Heap [33], and ConFab systems all proposed data models similar to Cascadia's, yet they did not support uncertain base data and streams of probabilistic events. JCAF [5] presented an event-driven programming API with event sub-

scriptions which is very similar to ours, but requires that developers write custom modules to perform event detection. ParcTab, Stick-e Notes [42], and ConFab provide declarative event specification languages, but they are less expressive and do not leverage RDBMSs. Liquid [24] and the Data Furnace project [19] have provided declarative, expressive event specification languages. Moreover, the Data Furnace project seeks to manage imprecise sensor data and probabilistic events. However, to the best of our knowledge, no algorithmic or system implementation details are published for either system.

RFID data management. Several techniques for compactly representing, summarizing, and efficiently accessing RFID data have been proposed [21, 27]. Most similar to our approach is the Siemens RFID middleware [54], an integrated data management system that includes a rule-based framework to transform RFID observations into business logic. This system is intended for supply-chain management and hence features a data model that emphasizes containment relations (*e.g.*, tagged cases are inside pallets). Additionally, the rules used are deterministic and operate on raw RFID readings.

Event specification. Early applications such as Contextual Reminders for ParcTab, and the more recent SPECs [39] allow expert end-users to write simple, declarative rules to trigger a particular application behavior. This type of specification is infeasible for non-expert users who are not equipped to reason about the structure and logic of rules and triggers. The Solar system provides a visual interface for specifying events, but it requires a knowledge of both the system architecture and the available sensor processing modules. Most similar to Cascadia's Scenic tool is the EventManager [40], which allows end-users to declaratively specify events using a forms-based interface. Yet Scenic supports a larger set of events by allowing sequences and also supports translation into a sophisticated event detection language.

Event detection. Event detection and processing has previously been addressed in three main research areas: active databases [1, 3, 6, 20, 45], publish-subscribe systems [12, 32], and more recently complex event extraction from sensor and RFID data [15, 49, 58]. In all these systems, however, event detection is deterministic: these approaches ignore event ambiguity and possible input data errors.

Activity inference. Previous work [43, 44] proposed dynamic Bayesian networks for inferring human activities from RFID and other sensor data. These systems infer the most likely activity performed by the user and defined by the model. We investigate an alternative technique that allows users to define new events at runtime and where the system reports the entire set of possible activities.

Sensor and RFID data cleaning. Several techniques for cleaning sensor data have been proposed. In these techniques, users declaratively specify either the data cleaning algorithm [30, 18] or a pattern over the data with matching cleaning actions [46, 54]. In contrast, our system operates directly on the dirty data, with no requirement for user-specified cleaning mechanisms. Cascadia, however, can leverage simple low-level cleaning techniques that average measurements within a short time-window [31] and across a group of sensors covering the same area [30]. In previous work [35], we showed that integrity constraints can serve to clean sensor data probabilistically. This technique can also be integrated with Cascadia to improve performance.

Probabilistic databases. There has been much work in this area [4], with our probabilistic events most similar to *maybe-or* tuples in Trio [57], *pc-tables* in Green and Tannen [23], or *disjoint-independent* tuples in Dalvi *et al.*, [10]. Query complexity on such databases has been studied in [10, 9]. Probabilistic temporal databases have been introduced [11], but they use a semantics based on probability intervals, which is different from ours. Recent work has explored query processing over probabilistic data streams [8, 28, 29], which could come from a model such as an HMM or particle filter [34]. These systems focus on selections and aggregation queries, while Cascadia supports a more natural set of operators for defining RFID events: selections, sequences, and negations.

8. Conclusion

In this paper, we presented Cascadia, an infrastructure for the specification, detection, and management of RFID events in pervasive computing applications. Cascadia provides applications with a probabilistic model of RFID data. Cascadia enables developers and users to visually specify events using Scenic. It detects these events continuously using PEEX and provides a convenient API for further managing the detected events and the related entities.

We showed that our approach is practical. Users can easily and effectively express common events with Scenic. The Particle Filter efficiently smoothes dirty RFID data. PEEX achieves good recall and precision on extracted events and allows applications to choose the trade-off between recall and precision. The digital diary application took just a few days to build. Additionally, performance results demonstrate that our current prototype can easily run in near real-time for hundreds of tags. In future work, we plan to further enhance the functionality and usability of Scenic, the event detection performance of PEEX, and the overall system performance.

References

[1] Adaikkalavan, R. and Chakravarthy, S. SnoopIB: Interval-based event specification and detection for active databases. In *Proc. of ADBIS 2003 Conf.*, Sept. 2003.

[2] Addlesee, M. et al. Implementing a sentient computing system. *IEEE Computer*, 34(8):50–56, 2001.

[3] Bacon, J. et al. Event Storage and Federation using ODMG. In *In Proc. of POS9*, pages 265–281, 2000.

[4] Barbara, D. et al. The management of probabilistic data. *IEEE Trans. Knowl. Data Eng.*, 4(5):487–502, Oct. 1992.

[5] Bardram, J. E. The Java Context Awareness Framework (JCAF) — A service infrastructure and programming framework for context-aware applications. *Proc. of the 3rd Pervasive Conf.*, pages 98–115, 2005.

[6] Chakravarthy, S. et al. Composite events for active databases: Semantics, contexts and detection. In *Proc. of the 20th VLDB Conf.*, Sept. 1994.

[7] Chen, G. and Kotz, D. Context Aggregation and Dissemination in Ubiquitous Computing Systems. In *Proc. of WMCSA 2002*, 2002.

[8] Cormode, G. and Garofalakis, M. Sketching probabilistic data streams. In *Proc. of the 2007 SIGMOD Conf.*, June 2007.

[9] Dalvi, N. and Suciu, D. Efficient query evaluation on probabilistic databases. In *Proc. of the 30th VLDB Conf.*, Sept. 2004.

[10] Dalvi, N. et al. Query evaluation on probabilistic databases. *IEEE Data Engineering Bulletin*, 29(1):25–31, 2006.

[11] Dekhtyar, A. et al. Probabilistic temporal databases, I: algebra. *ACM TODS*, 26(1):41–95, March 2001.

[12] Demers, A. et al. Towards expressive publish/subscribe systems. In *Proc. of the 10th EDBT Conf.*, 2006.

[13] Deshpande, A. and Madden, S. MauveDB: Supporting Model-based User Views in Database Systems. In *Proc. of the 2006 SIGMOD Conf.*, pages 73–84, 2006.

[14] Dey, A. et al. Distributed mediation of ambiguous context in aware environments. In *Proc. of the UIST 2002 Conf*, pages 121–130, 2002.

[15] EPCGlobal application level events specification. `http://www.epcglobalinc.org/standards/ale`, Sept. 2005.

[16] Floerkemeier, C. and Lampe, M. Issues with RFID usage in ubiquitous computing applications. In *Proc. of the 2nd Pervasive Conf.*, Apr. 2004.

[17] Fox, D. et al. Bayesian filtering for location estimation. *IEEE Pervasive Computing*, 2(3):24–33, July-September 2003.

[18] Franklin, M. J. et. al. Design considerations for high fan-in systems: The hifi approach. In *Proc. of the 2nd CIDR Conf.*, Jan. 2005.

[19] Garofalakis, M. N. et. al. Probabilistic Data Management for Pervasive Computing: The Data Furnace Project. *IEEE Data Eng. Bull*, 29(1):57–63, 2006.

[20] Gehani, N. H. et al. Composite event specification in active databases: Model & implementation. In *Proc. of the 18th VLDB Conf.*, Aug. 1992.

[21] Gonzalez, H. et al. Warehousing and analyzing massive RFID data sets. In *Proc. of the 22nd ICDE Conf.*, Apr. 2006.

[22] Google. Google Calendar APIs and Tools. `http://code.google.com/apis/calendar`, 2007.

[23] Green, T. J. and Tannen, V. Models for incomplete and probabilistic information. *IEEE Data Engineering Bulletin*, 29(1):17–24, March 2006.

[24] Heer, J. et al. liquid: Context-Aware Distributed Queries. In *Ubicomp 2003*, volume 2864, pages 140–148, 2003.

[25] Hightower, J. et al. Learning and Recognizing the Places We Go. In *Ubicomp 2005*, volume 3660, pages 159–176, 2005.

[26] Hong, J. and Landay, J. A. An Architecture for Privacy-Sensitive Ubiquitous Computing. In *Proc. of the 2nd MobiSys Conf.*, 2004.

[27] Hu, Y. et al. Supporting RFID-based item tracking applications in Oracle DBMS using a bitmap datatype. In *Proc. of the 31st VLDB Conf.*, Sept. 2005.

[28] Jayram, T. S. et al. Efficient aggregation algorithms for probabilistic data. In *ACM-SIAM Symposium on Discrete Algorithms*, Jan. 2007.

[29] Jayram, T. S. et al. Estimating statistical aggregates on probabilistic data streams. In *Proc. of the 26th PODS Conf.*, June 2007.

[30] Jeffery, S. et. al. Declarative support for sensor data cleaning. In *Proc. of the 4th Pervasive Conf.*, Mar. 2006.

[31] Jeffery, S. R. et al. Adaptive cleaning for RFID data streams. In *Proc. of the 32nd VLDB Conf.*, Sept. 2006.

[32] Jin, Y. and Strom. Relational subscription middleware for Internet-scale publish-subscribe. In *Proc of 2nd DEBS Workshop*, June 2003.

[33] Johanson, B. and Fox, A. The Event Heap: A Coordination Infrastructure for Interactive Workspaces. In *WMCSA*, pages 83–93, 2002.

[34] Kanagal, B. and Deshpande, A. Online filtering, smoothing and probabilistic modeling of streaming data. Technical Report CS-TR-4867, University of Maryland, May 2007.

[35] Khoussainova, N. et al. Towards correcting input data errors probabilistically using integrity constraints. In *Proc. of Fifth MobiDE Workshop*, June 2006.

[36] Khoussainova, N. et al. PEEX: Extracting Probabilistic Events from RFID Data. Technical Report UW-CSE-07-11-02, University of Washington, CSE, Nov. 2007.

[37] Khoussainova, N. et al. Probabilistic Event Extraction from RFID Data (poster). In *Proc. of the 24th ICDE Conf.*, Apr. 2008.

[38] Kriplean, T. et. al. Physical access control for captured RFID data. *IEEE Pervasive Computing*, 6(4), Nov. 2007.

[39] Lamming, M. and Bohm, D. SPECs: Another Approach to Human Context and Activity Sensing Research, Using Tiny Peer-to-Peer Wireless Computers. In *Ubicomp 2003*, pages 192–199, 2003.

[40] McCarthy, J. F. and Anagnost, T. D. EVENTMANAGER: Support for the Peripheral Awareness of Events. In *HUC*, volume 1927, pages 227–235, 2000.

[41] Oracle sensor edge server. `http://www.oracle.com/technology/products/sensor_edge_server/index.html`.

[42] Pascoe, J. The Stick-e Note Architecture: Extending the Interface Beyond the User. In *Proc. of 1997 IUI Conf.*, pages 261–264, 1997.

[43] Patterson, D. J. et al. Inferring high-level behavior from low-level sensors. In *Proc. of the 5th Ubicomp Conf.*, Oct. 2003.

[44] Philipose, M. et al. Inferring activities from interactions with objects. *IEEE Pervasive Computing*, 3(4), 2004.

[45] Pietzuch, P. R. and Bacon, J. Hermes: A Distributed Event-Based Middleware Architecture. In *ICDCS Workshops*, pages 611–618. IEEE Computer Society, 2002.

[46] Rao, J. et al. A deferred cleansing method for RFID data analytics. In *Proc. of the 32nd VLDB Conf.*, Sept. 2006.

[47] Rastogi, V. et. al. Authorization Views for Pervasive Sensor Networks. In *UbiPriv 2007*, Sept. 2007.

[48] RFID Journal. Hospital gets ultra-wideband RFID. `http://www.rfidjournal.com/article/view/1088/1/1`, Aug. 2004.

[49] Rizvi et. al. Events on the edge. In *Proc. of the 2005 SIGMOD Conf.*, June 2005. (System demonstration).

[50] Roman, M. et al. Gaia: a middleware platform for active spaces. *Mobile Computing and Communications Review*, 6(4):65–67, 2002.

[51] Salber, D. et al. The Context Toolkit: Aiding the Development of Context-Enabled Applications. In *Proc. of CHI 1999 Conf.*, pages 434–441, 1999.

[52] Sousa, J. P. and Garlan, D. Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. In *WICSA*, volume 224, pages 29–43, 2002.

[53] University of Washington. RFID Ecosystem. `http://rfid.cs.washington.edu/`.

[54] Wang, F. and Liu, P. Temporal management of RFID data. In *Proc. of the 31st VLDB Conf.*, Sept. 2005.

[55] Want, R. et. al. An overview of the PARCTAB ubiquitous computing experiment. *IEEE Personal Communications*, 2(6):28–33, Dec 1995.

[56] Welbourne, E. et. al. Challenges for Pervasive RFID-based Infrastructures. In *PERTEC 2007*, Mar. 2007.

[57] Widom, J. Trio: A system for integrated management of data, accuracy, and lineage. In *Proc. of the 2nd CIDR Conf.*, pages 262–276, Jan. 2005.

[58] Wu, E. et al. High-performance complex event processing over streams. In *Proc. of the 2006 SIGMOD Conf.*, June 2006.