# LOC8: A Location Model and Extensible Framework for Programming with Location

*Using a location model supporting a range of expressive representations for spaces, spatial relationships, and positioning systems, the authors created LOC8, a programming framework for exploring location data's multifaceted representations and uses.*

**Graeme Stevenson and Juan Ye**
*University College Dublin*

**Simon Dobson**
*University of St. Andrews*

**Paddy Nixon**
*University College Dublin*

Location is a core concept in most pervasive computing systems. Beyond simple uses such as pinpointing an individual's position or identifying a region's occupants, location is a key index for richer querying of an individual's or environment's context.

Although at first glance a simple concept, location information's representation has many forms and subtleties, each suited to particular application classes.[1] To provide application developers with easy access to location information, we must support different positioning systems with varying data formats as well as fusion algorithms to estimate position from multiple readings. We also need a data access approach that hides this complexity and heterogeneity from the developer. This problem has no general solution, necessitating specific frameworks for working with specific kinds of data.

To meet the needs of location-based applications, we've developed lightweight space and sensing models and a set of extensible components that support customization and emerging technologies. The space model supports a range of geometric and relative-spatial-positioning descriptions found in the literature. The sensing model abstracts over various types of positioning systems and incorporates the capture of uncertainty, serving as a foundation on which developers can apply sensor-fusion techniques. Our programming framework, LOC8, sits atop the space and sensing models, providing a rich API for querying location data and exploring its many representations.

## Requirements

A location model should support location data representations from different positioning technologies and extensible metadata descriptions. Many well-known systems can report an entity's coordinate or symbolic position, from GPS and Active Badge to more recent systems such as Ubisense and the fingerprint-based positioning system.[2] Beyond these are less conventional and less expensive methods of reporting an entity's location. For example, a Bluetooth spotter, which can detect the presence of mobile phones, PDAs, and laptops, might position a device within 10 meters of a known point. We can use this information to infer the device owner's position.



Published by the IEEE CS ■ 1536-1268/10/$26.00 © 2010 IEEE

Environments frequently contain multiple positioning systems, so translating readings into a common language of location-centric primitives is important. Because no positioning technology claims to provide perfect accuracy, this language must also provide quality measures to support sensor-fusion techniques for uncertain data. Quantifying uncertainty associated with positioning systems has proved a hot topic in recent years.[3,4]

A space model provides a set of primitives that allow descriptions of regions of space and the relationships between them. Such primitives must support the mapping of positioning systems' different data formats while being expressive enough to support common application queries.

Christian Becker and Frank Durr divide these queries into four categories: positioning, range, nearest neighbor (spatial relation), and navigation.[5] These queries have led to more complex uses of location. Such higher-level queries require flexible conversion between different location representations—for example,

- building a relative spatial relationship between two mobile entities, or
- translating a track of an entity's physical positions to a summary of its movement pattern or to its speed and heading.[6]

Increasingly, researchers don't treat location information as independent but rather as tightly bound to user activity, intention, and interaction. The extension of semantics in location is a popular research topic, underpinning many potential context-aware applications.[1]

## Realizing the Space Model

To represent our space model,[7] we chose the Web Ontology Language (OWL) because of its high-quality tool support and the applicability of reasoners and rules to help identify relationships between points and regions. At the heart of our ontology are the **SymbolicRepresentation**, **GeometricRegion**, and **RelativeLocation** classes, which model the human-friendly names and geometric extents of regions (such as rooms and buildings) and their physical relationships to other parts of the model.

Coordinates have an associated coordinate reference system (CRS), which can be a global standard or locally defined to simplify a region's spatial representation. For example, if an application is bound to a single building, it makes more sense to define a local coordinate system than to use the WGS 84 coordinate system. Translations from one CRS to another are described by an origin point and a rotation matrix. The origin is the displacement of the new CRS relative to its reference CRS, and the rotation matrix describes changes to rotation of the $x$-, $y$-, and $z$-axes. We used Chianghao Jiang and Peter Steenkiste's model[8] to convert coordinates from one CRS to another. Geometric regions consist of one or more 2D or 3D geometric shapes, each defined by a set of coordinates; symbolic representations take the form of an individual associated with a descriptive label.

Developers must assign each space a granularity (the **granularity** property), whose possible values, such as coordinate, room, and city, are defined in the model and are customizable. This allows flexibility in that developers can redefine granularities to suit different applications. The querying process uses granularities to request an entity's location at a particular resolution.

Our model supports four spatial relationships: containment, adjacency, connectedness, and overlap. Containment, adjacency, and overlap are what their names suggest. Connectedness is a particular case of adjacency, in which an entity can pass from a space to its adjacent space. The relationship might specify a third location, such as an exit or elevator, that enables the transition. Connectedness is a rich relationship, implying both the passage's direction and the notion of an accessible (rather than a straight-line) distance between the related locations.

We also provide two types of relative representation: center and compass.[7] In the center representation, a target location is a geometric area, such as a circle or a cuboid, whose center is a coordinate—or a reference location's center point—and whose edge or diameter length is twice a specified distance. The compass representation involves building a CRS whose origin is a coordinate—or a reference location's center point—and whose rotation matrix follows the standard compass directions. In this CRS, the target location's description contains a distance to its origin; the horizontal angle to the target location, measured clockwise from north; and the angle of elevation from the horizontal plane.

For some maps, specifying a set of symbolic regions and their spatial relationships will suffice. Maps that define region geometry let reasoners infer some symbolic relationships, such as containment and adjacency, and estimate missing geometry.

A space model provides a set of primitives that allow descriptions of regions of space and the relationships between them.

## Realizing the Sensing Model

The sensing model maps the reported positions of entities—for example, a person, locatable tag, or wireless device—to points and regions in our space model. (For an overview of the high-level ontologies we use to describe our applications, see "Ontonym: A

```
example:CASLUbisense
 a sensor:Sensor ;
 sensor:coverage map:3f , map:4f ;
 sensor:frequency [...] ;
 sensor:granularity map:coordinateGranularity ;
 sensor:precisionAccuracy
  [ a sensor:PrecisionAccuracy ;
    sensor:accuracy "0.7" ;
    sensor:precision
     [ a muo:QualityValue ;
       muo:measuredIn ucum:meter ;
       muo:numericalValue "2"
     ]
  ];
 sensor:precisionAccuracy [...]
 sensor:rateOfChange [...] .
```

**Figure 1. An abridged description of University College Dublin's Complex and Adaptive Systems Laboratory (CASL) Ubisense sensor and its metadata. The sensor covers the third and fourth floors of the CASL building, and is accurate to within two meters of a Ubitag's true position 70 percent of the time. Distance descriptions use the Measurement Units Ontology (MUO), giving a basis for transforming between different representations.**

```
example:reading
 a sensor:Observation ;
 sensor:about ubitag:010131789 ;
 sensor:observedAt [...] ;
 sensor:temporalDimension [...] ;
 sensor:observedBy example:CASLUbisense ;
 sensor:value
  [ a location:Coordinate ;
    location:referenceCoordinateSystem
      example:ubisenseCoordinateSystem ;
    location:x "1.15" ;
    location:y "3.67" ;
    location:z "21.35"
  ].
```

**Figure 2. An abridged reading produced by the CASL Ubisense sensor described in Figure 1. A Ubitag is related to a 3D coordinate position within the Ubisense coordinate system. The observation time and the source of the reading are also indicated.**

Collection of Upper Ontologies for Developing Pervasive Systems."[9]) Again, the essential part of this process is capturing metadata associated with the sensing process.

We adopt a standard approach to representing the sensed data's characteristics and imperfections by using a quality matrix, which satisfies the sensing model's uncertainty measure requirement. The quality matrix consists of granularity, frequency, coverage, and a list of accuracy and precision pairs.[10] Granularity is the smallest spatial element perceivable. Frequency is the sample rate—how often a sensor generates readings. The sensor manufacturers' technical specifications determine these properties' values. Coverage is the extent of space in which an entity's position can be sensed; the accuracy and precision pairs, which might be multiply defined, describe the probability that an entity's true position is within a given distance of the reported value. For example, with our Ubisense installation in University College Dublin's Complex and Adaptive Systems Laboratory (CASL) building, we achieve 70 percent accuracy with two meters' precision. Although our general quality matrix works with most positioning sensors, it isn't definitive, and we encourage its extension.

All data that a sensor adds to the model references the sensor's quality matrix. Figure 1 describes the granularity, frequency, coverage, and precision-accuracy pairs associated with our Ubisense sensor; Figure 2 describes a sample reading. Both figures use Notation 3 (www.w3.org/DesignIssues/Notation3), a compact Resource Description Framework (RDF) syntax.

The **about** property relates the reading to a particular entity, and the **observedBy** property relates the reading to the sensor that provided it. The **value** property indicates the position at which the sensor located the entity—in this case, a 3D coordinate. Finally, the **temporalDimension** property specifies the time span over which developers should regard the reading's value as reflecting the entity's true position.

Developers can easily add a positioning system to the model, which requires only that they define its metamodel and write a software adapter to transform sensor-reported positions to our model. Optionally, if the positioning system reports coordinates, developers can specify the necessary information to translate points from its CRS to another CRS.

## Related Work in Location Modeling

Nexus is an early open platform providing a foundation that makes developing location-aware applications easier.[1] The Nexus platform's core is a common augmented-world model that supports representation of the location of static real-world entities, such as buildings or trains, and virtual entities with which the real world is augmented, such as virtual billboards. Its query language, Augmented World Query Language (AWQL), supports basic spatial queries including **inside**, **overlaps**, **includes**, **excludes**, and **closest**. In LOC8 (see the main article), we use a loosely coupled modeling technique that treats location information independently from other forms of context. This lets us treat all locatable objects in the same way, irrespective of their property structure, real or virtual status, and use by applications.

The Location Stack is a successful software engineering model that structures location-aware services components into a layered system architecture with robust separation of concerns.[2,3] Our model is based on the Location Stack but differs in four main respects. First, the Location Stack's measurements layer reports data from sensors at a lower level than we support, including distance, angle, and proximity. We decided to deal only with observations at the position level—for example, coordinate and symbolic—because most technologies tend to perform this calculation/abstraction in the sensing system.

Second, we cleanly separate the space model from the sensing model, letting us treat the data in each independently. For example, the sensing model's implementation decides the length of time to retain readings. In contrast, the space model remains relatively static, but the LOC8 framework applies reasoning to its contents to infer additional spatial relationships from available geometric data.

Third, we've taken a cross-layered approach to LOC8's design, recognizing that context and space information can play a role before and during the point of fusion. In contrast, the Location Stack introduces contextual fusion only in its highest layers.

Fourth, the Location Stack architecture includes an arrangements layer, which uses information about the current probabilistic location estimates of multiple objects to identify relationships between them, such as proximity or formation. This form of querying isn't part of our framework's core but is an extension that developers could build.

The Aura project's space model combines hierarchical and coordinate space models.[4] Its interface extends traditional database SQL queries with spatial queries on the PostgreSQL database system, which improves performance and increases flexibility for location-aware applications. This location model supports flexible conversion between different coordinate systems. We've extended this idea in our location model. Although we don't consider our implementation's performance in the main article, we support the queries the Aura model identifies and extend them to support other forms, such as relative positioning.

Finally, MiddleWhere is a distributed middleware infrastructure for location that separates applications from location-sensing technologies.[5] Similar to LOC8, it can add sensing technologies dynamically and transparently from an application perspective. However, the two approaches differ in uncertainty management. MiddleWhere provides Bayes-based probabilistic reasoning to fuse multiple sensor readings, whereas we focus on the generic representation of different sensor data and its quality. We expose this through the programming framework, providing an interface to accommodate different sensor-fusion approaches.

### REFERENCES


1. F. Hohl et al., "Next Century Challenges: Nexus—an Open Global Infrastructure for Spatial-Aware Applications," *Proc. 5th Ann. ACM/IEEE Int'l Conf. Mobile Computing and Networking* (MobiCom 99), ACM Press, 1999, pp. 249–255.
2. J. Hightower, B. Brumitt, and G. Borriello, "The Location Stack: A Layered Model for Location in Ubiquitous Computing," *Proc. 4th Workshop Mobile Computing Systems and Applications* (WMCSA 02), IEEE CS Press, 2002, pp. 22–28.
3. A. LaMarca et al., "Delivering Real-World Ubiquitous Location Systems," *Comm. ACM*, vol. 48, no. 3, pp. 36–41.
4. C. Jiang and P. Steenkiste, "A Hybrid Location Model with a Computable Location Identifier for Ubiquitous Computing," *Proc. 4th Int'l Conf. Ubiquitous Computing* (UbiComp 02), LNCS 2498, Springer, 2002, pp. 246–263.
5. A. Ranganathan et al., "MiddleWhere: A Middleware for Location Awareness in Ubiquitous Computing Applications," *Proc. 5th ACM/IFIP/Usenix Int'l Conf. Middleware*, Springer, 2004, pp. 397–416.


## Using the Programming Framework

On the basis of Jeffrey Hightower and his colleagues' Location Stack architecture[11] (see the "Related Work in Location Modeling" sidebar), we developed the LOC8 framework in Java to support querying of the space and sensing models we constructed. Figure 3 shows LOC8's architecture. The sensing layer reports positioning data as coordinates, symbolic locations, or relative positions; the abstraction layer converts raw sensor data into the OWL representation. As part of this process, the abstraction layer can query the context and space models to find the correct references for resources representing particular regions, people, or locatable objects. The context, sensing, and space models provide standard APIs for

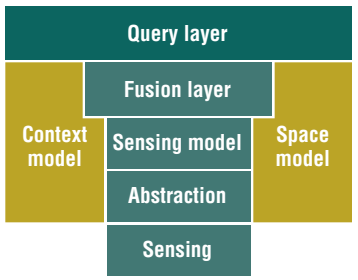



Figure 3. LOC8's architecture. Sensors provide raw data, which is translated to our model and mapped to entity and space descriptions. An interface for performing sensor fusion is exposed, while the top layer of the framework provides application developers with an API for common query types.

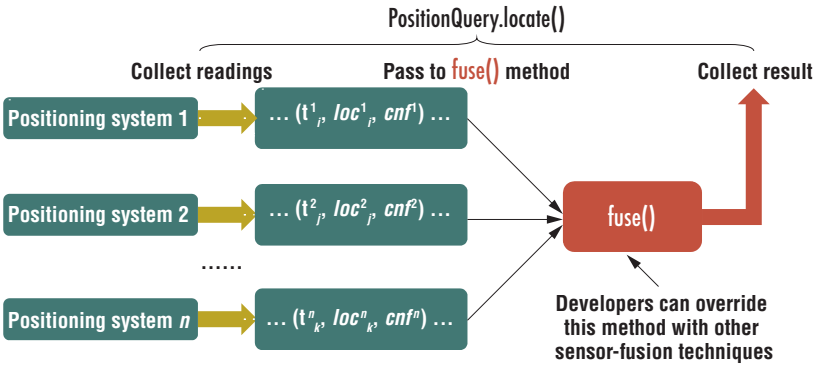



Figure 4. Sensor fusion in the positioning-query module. All entity observations that satisfy a query's constraints are collected and passed to the fuse() method. Developers can implement customized fuse() methods to integrate the readings' confidence levels and resolve an entity's position.

querying their contents. The fusion layer uses these APIs to calculate probabilities for an entity's position and provides a set of calls to invoke this functionality. Finally, the model's top layer supports application querying, providing modules for each of the four query categories.

### Positioning Queries

The most common query is for locating an entity within a space model using available positioning data. LOC8 supports this through its positioning-query module. We can configure the query using six parameters:

- entity specifies the entity's identifier.
- finestGranularity and coarsestGranularity constrain the result's granularity (for example, coordinate, cubicle, or building).
- precision specifies a proximate distance the querying application requires, which affects the confidence value calculation.
- startTime and endTime specify the temporal interval of interest.

The position query's expanded interface is as follows:

```
1. List<PositionResult> locate(Entity entity,
2.   Granularity finestGranularity,
3.   Granularity coarsestGranularity,
4.   Distance requiredPrecision,
     DateTime startTime,
5.   DateTime endTime);
```

The API provides more compact variants using parameters' default values; for example, the time parameter defaults to the current time. Consider the following code for the positioning query, "Where is Bob?":

```
1. Entity bob = sensorModel.getEntity(ENTITY_
   URI + "Bob");
2. List<PositionResult> results = positionQuery.
   locate(bob);
3. for (PositionResult result : results) {
4.   if (result.getLocation().
     hasSymbolicRepresentation()) {
5.     System.out.printf("%s - %s\n",
6.     result.getLocation().
       asSymbolicRepresentation(),
7.     result.getConfidence());
8.   }
9. }
```

In this code, we use the query API to obtain a list of candidates for Bob's current position (lines 1 and 2). We then check whether each result has an associated symbolic representation (line 4). If so, we print that representation's name to the console, along with its associated confidence value (lines 5 through 7).

If the environment contains multiple positioning systems, we apply sensor fusion within the locate() method. This is a three-step procedure (see Figure 4):

1. Get all observations that satisfy the query's input requirements, including entity, time span, granularity, and precision. Transform them into to a triple consisting of the reading's starting time, a position ordered from finest granularity to required granularity, and a confidence value that's the sensor's accuracy at the query's required precision.
2. Pass the triples to the sensor-fusion method fuse(). Developers can apply customized sensor-fusion techniques to fuse(). We've implemented a simple fuzzy-based fuse() that organizes the collected readings in a tree structure according to their granularity and that uses fuzzy logic to update and integrate the confidences on this location data.
3. Order the results according to their

granularity of location, confidence, and time.

If we're interested in Bob's coordinate position rather than the symbolic name associated with his position, we must address two issues:

- how to translate results into a target coordinate system, and
- how to deal with situations in which a symbolic location has no explicit geometry.

To address the first issue, we translate the coordinate system using the approach described earlier. For the second issue, we estimate geometry by assuming a space's boundary is the composite of all its subspaces. If no such information is available, we approximate by inheriting the geometry of a space's superspace. Clearly, this process's success depends on the amount of geometric information available and might not be suitable for all applications. So, developers can use `hasExplicitGeometricRegion()` and `estimateGeometricRegion()` at their discretion.

The following code illustrates the task of plotting Bob's position on a map:

```
1. Entity bob = sensorModel.getEntity(ENTITY_
   URI + "Bob");
2. PositionResult result = positionQuery.
   locateMax(bob);
3. Coordinate centerPoint = result.getLocation()
4.    .asGeometricRegion().centerPoint();
5.    Coordinate translatedPoint = CoordinateUtils
6.    .convert(centerPoint, mapCRS);
7. map.plot(bob, translatedPoint);
```

We calculate Bob's position as we did in the previous example, this time using `locateMax()` to return only the result with the highest confidence at the finest granularity (lines 1 and 2). Assuming Bob's location has an associated geometric region, we use his position as that region's center point (lines 3 and 4), transform it to the map's coordinate system (lines 5 and 6), and call the application plot method (line 7).

### Range Queries

Essentially the inverse of a position query, a range query identifies all entities in a location that match certain criteria. There are four input parameters: `space`, the region whose contents we're interested in; `startTime` and `endTime`, the time span we're interested in; and `entityType`, the entity class to locate. Each result consists of a reference to a located entity and a confidence value representing the likelihood that the entity is in the location at the given time. The interface for the range query is as follows:

```
1. List<RangeResult> in(Space space,
   Class entityType,
2.    DateTime startTime, DateTime endTime);
```

As with position queries, this method has several variants. The query defaults to returning all locatable entities currently in the specified location if the querier omits time and entity type parameters.

To compute a result for a range query, we first query the entity model for the entities matching the specified type. We then use positioning queries to locate each entity. Finally, we check whether each positioning query's result matches, or is a subspace of, the specified location.

> Essentially the inverse of a position query, a range query identifies all entities in a location that match certain criteria.

We code the range question, "Who is in the CASL building?" as follows:

```
1. Space casl = spaceModel.getSpace(MAP_URI +
   "CASL");
2. List<RangeResult> results = rangeQuery.
   in(casl,Person.class);
3. for (RangeResult result : results) {
4.    System.out.printf("%s - %s\n",
5.    result.getEntity(),
6.    result.getConfidence());
7. }
```

We first obtain a reference to the space, which we use to execute a query to find all people in the region (lines 1 and 2). An iterator over the results prints the set of entities along with the confidence in each result (lines 3 through 7).

### Spatial-Relation Queries

The spatial-relation module provides a set of methods for applications in which relations between locations are important. The API's most basic method, `relationship()`, accepts two locations as parameters and checks for containment, adjacency, overlap, and connectedness. Containment, adjacency, and overlap relationships are either expressed directly in the model or calculated in preprocessing at runtime by comparing the geometric regions' boundaries. Map designers must explicitly express connectedness. If none of these relationships exist between the spaces, the next step is to calculate the compass relative position between the two locations. If the locations don't share a common CRS and can't be translated to a common CRS, the query is unanswerable.

We use several variants of the `closest()` method to find proximate entities. Its input parameters follow the same pattern as the previous queries, with the `space` parameter providing an outer boundary for the search. We then calculate the results' positions relative to the target entity and order them from closest to farthest. The method signature is as follows:

```
1. List<ProximityResult> closest(Entity entity,
2.    Space boundary, Class entityType,
3.    DateTime startTime, DateTime endTime);
```

Consider a spatial-relation query in which we want to find the shop nearest

Bob. To represent the answer symbolically or as a coordinate, we use the approaches we just described. However, the final representation we identify is a relative position—for instance, 100 meters northwest. We achieve this by computing the distance and bearing between the points:

```
1. Entity bob = sensorModel.getEntity(ENTITY_
   URI + "Bob");
2. Space campus = spaceModel.getSpace(MAP_
   URI + "UCD");
3. List<ProximityResult> results = relationQuery
4.    .closest(bob, campus, Shop.class);
5. for (ProximityResult result : results) {
6.    CompassLocation rel = result.getLocation()
7.    .asCompassRelative(bob);
8.    System.out.printf("%s relative to %s:
      (%d %d)\n",
9.    bob, result.getEntity(), rel.getDistance(),
10.   rel.getHorizontalAngle());
11.}
```

We look up the objects for Bob and for the University College Dublin campus, which we use to limit the search space (lines 1 and 2). We then execute the query, limiting the search to entities that are shops (lines 3 and 4). The result is an ordered list of shops, from closest to farthest away. We iterate through the results, displaying the distance and horizontal angle between Bob and the target shop for each (lines 5 through 11).

### Navigation Queries

The navigation-query module supports pathfinding between the model's different regions using a selection of `path()` methods that takes two parameters—`source` and `destination`—which can be locations or entities. Methods that accept entities as parameters first calculate the entities' positions using the position-query module before proceeding in the same manner as if the developers had passed a location. The basic pathfinding algorithm works as follows:

> We can also extend the query model by integrating additional context into the query process.

1. Check whether the source and destination locations are the same.
2. If the locations are the same, return the answer.
3. If the locations aren't the same, recursively call the pathfinding algorithm using each location connected to the source location as the new source, keeping track of paths to avoid cycles.

The algorithm has two versions—one that terminates after finding a path and another that searches all paths. This algorithm's current implementation is suited only for evaluating paths through small space models. Considering source and destination locations with different granularities—for example, from Bob's desk to the coffee area—increases the complexity. Improving this approach is a possible area of future research.

The navigation-query module can also calculate the distance between two locations. It determines the point-to-point Euclidian distance by first evaluating each location's center point. If the connection relationship metadata provides the accessible distance, it can also calculate the path-accessible distance.

Extending the previous example, the following code calculates the path between Bob and the nearest shop:

```
1. Path path = navigationQuery.path(bob,
   nearestShop)
2. for (Step step : path.steps()) {
3.    System.out.printf("from %s to %s (%d)\n",
4.    step.getSource(),
5.    step.getDestination(),
6.    step.pathDistance());
7. }
```

After the query executes (line 1), we iterate through each step in the path, printing out the details and the path-accessible distance for each (lines 2 through 7).

### Combination Queries

Constructing more sophisticated queries that use the core queries we just described can simplify application development. Consider a scenario from the Cooperative Object Detection and Ranging (Codar) system demonstration[12] in which two cars are heading for a collision (see Figure 5). We want to construct a service that predicts potential collisions and calculates their time and location. Here's an outline of a simple implementation of this service:

```
1. PositionResult locA = positionQuery.
   locateMax(carA);
2. PositionResult locB = positionQuery.
   locateMax(carB);
3. CompassLocation rel = locB.
   asCompassRelative(locA);
4. …
5. Double degree = rel.getHorizontalAngle();
6. if ((velocity(carB)/velocity(carA))
7.    == Math.abs(Math.tan(degree))) {
8.    Double collisionTime = rel.getDistance()
9.      / relativeVelocity(carB, carA);
10.   ...
11.}
```

Treating car A as a base location, we can deduce car B's position relative to A using our relative-positioning queries (lines 1 through 3). We can then calculate the rate at which the cars are approaching using the distance measure between them over a set of time instances (not shown). We estimate the time when the distances between the two cars will reach zero using B's velocity relative to A (lines 5 through 9).

### Semantic Queries

We can also extend the query model

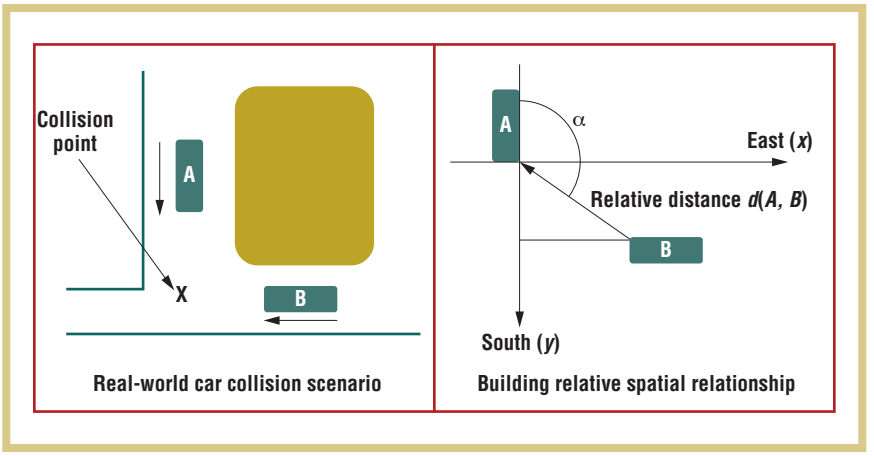



Figure 5. A dynamic-location example: car collision prediction. We repeatedly evaluate the relative position of car B to car A over time to calculate the rate at which the cars are approaching. We can predict the estimated collision time using their relative velocity.

by integrating additional context into the query process. Consider building a query to report whether a person is at home. We achieve this using the following code:

```
1.  public boolean atHome(Entity person) {
2.    boolean result = false;
3.    Property residesIn = ResourceFactory
4.      .createProperty("http://example.com/
        residesIn");
5.    PositionResult locA = positionQuery.
      locateMax(person);
6.    if(person.hasProperty(residesIn, result.
      getLocation()) {
7.      result = true;
8.    }
9.    for(Space space: result.getLocation().
      containedBy()) {
10.       if(person.hasProperty(residesIn, space)) {
11.         result = true;
12      }
13.   }
14.   return result;
15. }
```

After we obtain a reference to the residesIn property (defined externally) (lines 2 and 3), we use the positioning-query module to find the entity's position (lines 3 and 4). We then check to see whether this location, or any location that contains it, is associated with the person by the residesIn property (lines 6 through 12). Finally, the result is returned (line 14).

## Discussion

We developed core space and sensing models from our original requirements set and constructed a rich query model to support common application uses of location. Consequently, most pervasive computing systems that need to model or work with location can use LOC8.

### Engineering Effort

LOC8 provides developers a well-structured, simplified approach for working with what's essentially highly enriched sensor data. This requires engineering effort in terms of constructing a space map, integrating a new positioning system, and designing applications.

For the early adopter, using OWL involves a significant learning curve; an editing tool such as Protégé[13] can ease the process. The language has several complexities, and its serializations are visually unappealing and can be difficult to work with. To ease map construction, designers can apply translations to our model to existing map-drawing tools' output format. Although we developed only a simple prototype of this feature using ArchiCAD, it demonstrates that designers can construct maps without getting their hands dirty. This also opens up the possibility of deriving maps directly from professional architectural drawings.

Mapping positioning systems to the sensing model also falls to early adopters, and is essentially free to other developers. Beyond interfacing directly with each positioning system—a requirement for creating a stand-alone application—developers must use OWL to describe the sensor, its CRS, and its readings. This incurs a one-time cost for each positioning system.

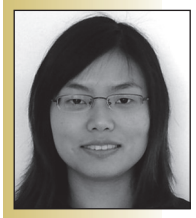
Subsequent application developers will rarely use OWL—perhaps only when tagging locatable entities or defining a local CRS if an existing one doesn't suit. Most cases won't require either of these steps. The framework provides a fully featured API for traversing the space model, and the built-in query modules support the execution of the core query types to meet most applications' needs, as Christian Becker and Frank Durr identified.[5]

### Flexibility and Extensibility

The space model's loose coupling with other aspects of the model and development process has clear benefits. Modelers needn't be concerned with how the application reads, interprets, or accesses the model, so they're less likely to take shortcuts in the mapping process. Using the established Measurement Units Ontology (MUO; http://idi.fundacionctic.org/muo) to represent units of measurement mitigates potential encoding bias from the modeling process, and the choice of OWL means that the space model is naturally distributed. Developers can partition the responsibility for creating the model and integrate the results. This implies straightforward evolution of space models over time. In most cases, the developer needs to build the space model only once for any particular region. Once this initial cost is out of

## the AUTHORS

**Graeme Stevenson** is a PhD candidate at University College Dublin. His research interests include programming languages, middleware for smart spaces, and the Semantic Web. Stevenson has an MPhil in computer science from the University of Strathclyde. Contact him at graeme.stevenson@ucd.ie.



**Juan Ye** is a postdoctoral researcher at the University College Dublin Clarity research center. Her research interests are pervasive and ubiquitous computing and wireless sensor networks, specializing in ontology, context modeling and reasoning, and uncertainty-resolving techniques. Ye has a PhD in computer science from University College Dublin. Contact her at juan.ye@ucd.ie.

**Simon Dobson** is a professor of computer science at the University of St. Andrews. His research centers on adaptive pervasive computing and novel programming techniques, addressing both theory and practice. Dobson has a DPhil in computer science from the University of York. He's a member of the BCS, the IEEE, and the ACM. Contact him at sd@cs.st-andrews.ac.uk.

**Paddy Nixon** is the Science Foundation Ireland research professor in distributed systems at University College Dublin and a principal investigator at Clarity. His research interests include pervasive and autonomic computing, with emphasis on infrastructure aspects of context-adaptive systems. He's an IBM faculty fellow and director of the TRIL (Technology Research for Independent Living) Centre. Nixon has a PhD in computer science from the University of Sheffield. Contact him at paddy.nixon@ucd.ie.


the way, sharing the model across all applications requires zero effort.

### Expressiveness

Because location has many meanings,[1] our model's expressiveness is key to letting location-aware services leverage its subtleties. Because our framework already supports positioning, range, spatial-relation, and navigation queries, developers don't need much code to perform them. Developers can derive more complex location-based scenarios by combining these queries or incorporating additional semantics, as the car-collision prevention and residential-query scenarios show.

### Consistency Checking

We check the consistency of developer-declared spatial relationships when the maps are loaded. As we mentioned earlier, we also use our ontological engines to infer spatial relationships, which we use to validate and complement the declared relationships.

### Future Improvements

We intend to further improve and refine our approach. Currently, our implementation supports only the modeling of Cartesian CRSs, although the addition of polar CRSs is straightforward. We also don't yet support the modeling of elliptical CRSs. The exception to this is WGS 84, which, because of its ubiquity (through GPS), we implemented directly into the Java model. Performing an accurate mapping from a Cartesian CRS to an elliptical one using the rotation matrix and offset technique is impossible. To overcome this, we assume that over short distances we can treat WGS 84 as a linear system, which lets us perform the conversion. However, the greater the distance from the origin, the greater the error introduced.

Our sensing model assumes that a particular CRS's axes share the same unit of measurement, which isn't always true. We also model the sensor-provided precision levels as a single value, which is another simplification. Depending on factors affecting the installation, as we've found from evaluating our Ubisense installation, different precision levels are available on each axis. We could go even further and model precision at different points in the sensing system's coverage area, but we remain unconvinced that the benefits would outweigh the added complexity.

The key to supporting application developers who work with location is by separating the concerns of mapping space, working with positioning systems, and querying data. Our goal in developing LOC8 was to construct a framework that glued these three elements together. Application developers don't need an understanding of sensor system operation and can model spaces without concern for how the data will later be accessed.

We're focused on optimizing the core query modules' implementation and evaluating their performance. Beyond this, we intend to further explore the semantic queries to investigate the integration of additional context types into the querying process.

Our space and sensing ontologies are available under an open source license to promote our model's adoption and practical use by other researchers and developers in the community (http://ontonym.org). We plan to release the query framework code in the near future. P

## ACKNOWLEDGMENTS

Science Foundation Ireland partially supported this work under grant 07/CE/I1147, "Clarity, the Centre for Sensor Web Technologies."

## REFERENCES

1. S. Dobson, "Leveraging the Subtleties of Location," *Proc. Smart Objects and Ambient Intelligence Conf.* (sOc-EUSAI 05), ACM Press, 2005, pp. 189–193.
2. T. King, T. Haenselmann, and W. Effelsberg, "Deployment, Calibration, and Measurement Factors for Position Errors in 802.11-Based Indoor Positioning Systems," *Proc. Int'l Workshop Location and Context Awareness* (LoCA), LNCS 4718, Springer, 2007, pp. 17–34.
3. W.T. Niu and J. Kay, "Location Conflict Resolution with an Ontology," *Proc. 6th Int'l Conf. Pervasive Computing*, LNCS 5013, Springer, 2008, pp. 162–179.
4. A. Ranganathan et al., "MiddleWhere: A Middleware for Location Awareness in Ubiquitous Computing Applications," *Proc. 5th ACM/IFIP/Usenix Int'l Conf. Middleware*, Springer, 2004, pp. 397–416.
5. C. Becker and F. Durr, "On Location Models for Ubiquitous Computing," *Personal and Ubiquitous Computing*, vol. 9, no. 1, pp. 20–31.
6. G. Ananthanarayanan et al., "Startrack: A Framework for Enabling Track-Based Applications," *Proc. 7th Int'l Conf. Mobile Systems, Applications, and Services* (MobiSys 09), ACM Press, 2009, pp. 207–220.
7. J. Ye et al., "A Unified Semantics Space Model," *Location and Context Awareness*, LCNS 4718, Springer, 2007, pp. 103–120.
8. C. Jiang and P. Steenkiste, "A Hybrid Location Model with a Computable Location Identifier for Ubiquitous Computing," *Proc. 4th Int'l Conf. Ubiquitous Computing* (UbiComp 02), LNCS 2498, Springer, 2002, pp. 246–263.
9. G. Stevenson et al., "Ontonym: A Collection of Upper Ontologies for Developing Pervasive Systems," *Proc. 1st Workshop Context, Information, and Ontologies* (CIAO 09), ACM Press, 2009, article no. 9.
10. G. Judd and P. Steenkiste, "Providing Contextual Information to Pervasive Computing Applications," *Proc. 1st Int'l Conf. Pervasive Computing and Communications* (Percom 03), IEEE CS Press, 2003, pp. 133–142.
11. J. Hightower, B. Brumitt, and G. Borriello, "The Location Stack: A Layered Model for Location in Ubiquitous Computing," *Proc. 4th Workshop Mobile Computing Systems and Applications* (WMCSA 02), IEEE CS Press, 2002, pp. 22–28.
12. M. Kranz et al., "Codar Viewer—a V2V Communication Awareness Display," *Pervasive 2008, Advances in Pervasive Computing, Late-Breaking Results*, vol. 236, Austrian Computer Society, 2008, pp. 79–82.
13. N.F. Noy et al., "Creating Semantic Web Contents with Protégé-2000," *IEEE Intelligent Systems*, vol. 16, no. 2, 2001, pp. 60–71.