Semantic Streams: A Framework for Composable Semantic Interpretation of Sensor Data

Kamin Whitehouse[1], Feng Zhao[2], and Jie Liu[2]

[1] UC Berkeley, Berkeley, CA, USA
kamin@cs.berkeley.edu
[2] Microsoft Research, Redmond, WA, USA
{zhao, liuj}@microsoft.com

Abstract. We present a framework called Semantic Streams that allows users to pose declarative queries over semantic interpretations of sensor data. For example, instead of querying raw magnetometer data, the user queries whether vehicles are cars or trucks; the system decides which sensor data and which operations to use to infer the type of vehicle. The user can also place constraints on values such as the the amount of energy consumed or the confidence with which the vehicles are classified. We demonstrate how this system can be used on a network of video, magnetometer, and infrared break beam sensors deployed in a parking garage with three simultaneous and independent users.

1 Introduction

While most sensor network research today focuses on ad-hoc sensor deployments, fixed sensor infrastructure may be much more common and in fact is ubiquitous in our daily environments even today. Homes have security sensors, roads have traffic sensors, office buildings have HVAC and card key sensors, etc. Most of these sensors are powered and wired, or are one hop from a base station. Such sensor infrastructure does not have many of the technical challenges seen with its power-constrained, multi-hop counterpart: it is relatively trivial to collect the data and even to allow a building's occupants to query the building sensors through a web interface. The largest remaining obstacle to more widespread use is that the non-technical user must semantically interpret the otherwise meaningless output of the sensors. For example, the user does not want raw magnetometer or HVAC sensor data; a building manager wants to be alerted to excess building activity over the weekends, or a safety engineer wants to know the ratio of cars to trucks in a parking garage.

Our paper presents a framework called *Semantic Streams* that allows non-technical users to pose queries over semantic interpretations of sensor data, such as "I want the ratio of cars to trucks in the parking garage", without actually writing code to infer the existence of cars or trucks from the sensor data. The key to our system is that previous users will have written applications in terms of *inference units*, which are minimal units of sensor data interpretation. When a new semantic query arrives, existing inference units can then be *composed* in

K. Römer, H. Karl, and F. Mattern (Eds.): EWSN 2006, LNCS 3868, pp. 5–20, 2006.

new ways to generate new interpretations of sensor data. If the query cannot be answered, the system may ask for new sensors to be placed or for new inference units to be created. In this way, the sensor infrastructure and the semantic values it can produce grow organically as it is used for different applications.

The system also allows the user to place constraints or objective functions over quality of service parameters, such as, "I want the confidence of the vehicle classifications to be greater than 90%," or "I want to minimize the total energy consumed." Then, if the system has a choice between using a magnetometer or a motion sensor to detect trucks for example, it may choose to use the motion sensor if the user is optimizing for energy consumption, or the magnetometer if the user is optimizing for confidence. Finally, our system allows multiple, independent users to use the same network simultaneously through their web interface and automatically shares resources and resolves resource conflicts, such as two different requirements for the sampling frequency of a single sensor. Towards the end of the paper, we demonstrate how this system is used on a network of video, magnetometer, and infrared break beam sensors deployed in a parking garage.

2 The Semantic Streams Programming Model

The Semantic Streams programming model contains two fundamental elements: *event streams* and *inference units*. Event streams represent a flow of asynchronous events, each of which represents a world event such as an object, person or car detection and has properties such as the time or location it was detected, its speed, direction, and/or identity.

Inference units are processes that operate on event streams. They infer semantic information about the world from incoming events and either generate new event streams or add the information to existing events as new properties. For example, the *speed inference* unit in Figure 1 creates a new stream of objects and infers their speeds from the output of sensors A and B. The *vehicle inference* unit uses the speeds in combination with raw data from sensor C to label each object as a vehicle or not. As a stream flows from sensors and through different inference units, its events acquire new semantic properties.

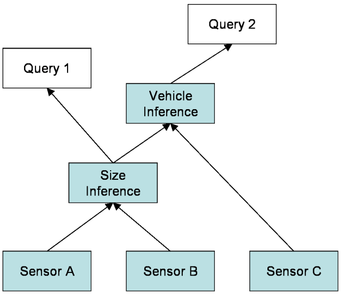


Fig. 1. Programming Model. Events streams feed inference units and accumulate semantic information as they flow through them.

The goal of this programming model is to allow *composable inference*; event streams can flow through new combinations of inference units and still produce valid world interpretations. While composable inference will never infer completely unforeseeable facts, it can be used to answer queries which are slight variations or combinations of previous applications, for example inferring the size of a car using logic that was originally intended to infer the sizes of people. For simplicity, we can assume that all inference units are running on a central server where all sensor data is collected, although it would be straightforward to execute some inference units directly on the sensor nodes.

3 A Logic-Based Markup and Query Language

In order to automatically compose sensors and inference units, we use a markup language to encode a logical description of how they fit together. To ensure that inference units are not composed in ways that produce invalid world interpretations, each inference unit must be fully specified in terms of its input streams and output streams and any required relationships between them. For example, the *vehicle inference* unit in Figure 2 may *create* vehicle events and *need* speed events and sensor *C* events that are co-temporal and co-spatial.

The Semantic Streams markup and query language is built using SICStus Prolog and its constraint logic programming (real) (CLP(R)) extension. Prolog is a logic programming language in which facts and logic rules can be declared and used to prove queries. CLP(R) allows the user to declare numeric constraints on variables. Each declared constraint is added to a constraint set and each new constraint declaration evaluates to true iff it is consistent with the existing constraint set. For a more complete description of Prolog and CLP(R), see [1].

3.1 Declaring Sensors and Simple Inference Units

Semantic Streams defines eight logical predicates that can be used to declare sensor and inference units. The font of each predicate indicates whether it is a top-level or an inner predicate.

- **sensor**(<sensor type>, <region>)
- **inference**(<inference type>, <needs>, <creates>)
- ***needs***(<stream1>, <stream2>, ...)
- ***creates***(<stream1>, <stream2>, ...)
- *stream*(<identifier>)
- *isa*(<identifier>, <event type>)
- *property*(<identifier>, <value>, <property name>)

The **sensor**() predicate defines the type and location of each sensor. For example

```
sensor(magnetometer, [[60,0,0],[70,10,10]]).
sensor(camera,        [[40,0,0],[55,15,15]]).
sensor(breakBeam,    [[10,0,0],[12,10, 2]]).
```

defines three sensors of type `magnetometer`, `camera`, and `breakBeam`. Each sensor is declared to cover a 3D cube defined by a pair of $[x, y, z]$ coordinates. For simplicity, we approximate all regions as 3D cubes, although this restriction does not apply to Semantic Streams in general.

The **inference**(), ***needs***(), and ***creates***() predicates describe an inference unit in terms of the event streams that it needs and creates. The *stream*(), *isa*(), and *property*() predicates describe an event stream and the type and properties of its events. For example, a vehicle detector unit could be described as an inference unit that uses a magnetometer sensor to detect vehicles and creates an event stream with the time and location in which the vehicles are detected.

```
inference( magVehicleDetectionUnit,
   needs(
      sensor(magnetometer, R) ),
   creates(
      stream(X),
      isa(X,vehicle),
      property(X,T,time),
      property(X,R,region) ) ).
```

3.2 Encoding and Reasoning About Space

Sensors have real-world spatial coordinates and, as such, our language and query processor must be able to encode and reason about space. As a simple example, our declaration of the `magVehicleDetectionUnit` above uses the same variable R in both the ***needs***() predicate and the ***creates***() predicate. This encodes the fact that the region in which vehicles are detected is the same region in which the magnetometer is sensing.

A more complicated inference unit may require a number of *break beam* sensors (which detect the breakage of an infrared beam) with close proximity to each other and with non-intersecting detection regions. One way to declare this is to require three sensors in specific, known locations:

```
inference( objectDetectionUnit,
   needs(
      sensor(breakBeam, [[10,0,0],[12,10, 2]]),
      sensor(breakBeam, [[20,0,0],[22,10, 2]]),
      sensor(breakBeam, [[30,0,0],[32,10, 2]]) ),
   creates(
      stream(X),
      isa(X,object),
      property(X,T,time),
      property(X, [[10,0,0],[32,10, 2]]) ), region) ) ).
```

This inference unit description, however, cannot be composed with break beams other than those which have been hard coded. To solve this problem, we could use two *logical rules* about spatial relations:

- *subregion*(<A>, <B>)
- *intersection*(<A>, <B>, <C>)

The first predicate is true if region A is a subregion of region B while the second predicate is true if region A is the intersection of region B and region C. An example of the first rule written in CLP(R) notation is:

```
subregion(
   [ [X1A, Y1A, Z1A],[X2A, Y2A, Z2A] ],
   [ [X1B, Y1B, Z1B],[X2B, Y2B, Z2B] ]):-
      {min(X1A,X2A)>=min(X1B,X2B),
      min(Y1A,Y2A)>=min(Y1B,Y2B),
      min(Z1A,Z2A)>=min(Z1B,Z2B),
      max(X1A,X2A)=<max(X1B,X2B),
      max(Y1A,Y2A)=<max(Y1B,Y2B),
      max(Y1A,Z2A)=<max(Z1B,Z2B)}.
```

The `objectDetectionUnit` can now be defined to require any three break beams that are within a region R and that do not intersect each other.

```
inference( objectDetectionUnit,
   needs(
      sensor(breakBeam, R1),
      sensor(breakBeam, R2),
      sensor(breakBeam, R3) ),
      subregion(R1,R),
      subregion(R2,R),
      subregion(R3,R),
      \+ intersect( _,R1,R2),
      \+ intersect( _,R1,R3),
      \+ intersect( _,R2,R3) ),
   creates(
      stream(X),
      isa(X,object),
      property(X,T,time),
      property(X,R,region) ) ).
```

Where in Prolog `\+ intersect( _,R1,R2)` is true if regions $R1$ and $R2$ do not intersect. With this logical description, the inference unit will function over any three non-intersecting break beam sensors in any region R.

3.3 Declaring Queries

A query is simply a first-order logic description of the event streams and properties desired by the user. For example, a simple query could be:

```
stream(X), isa(X,vehicle).
```

This query would be true iff a set of sensors and inference units could be composed to generate events X that are known to be vehicles. In many cases, the query interpreter will be able to generate many such inference compositions. To constrain the resulting composition set, we could simply add more predicates to the query. For example, we could query only for car events in a certain region:

```
stream (X), isa (X, car),
property (X, [[10,0,0],[30,20,20]], region).
```

A more sophisticated query might require specific relationships between event streams. For example, a histogram unit may update a histogram with incoming events and generate new events each time it is updated. A query could then request a stream of histogram events Y where the values being plotted are the times of vehicle detection events in stream X. The last line of the query further constrains the plot to only those vehicle events detected in a particular region.

```
stream (Y), isa (Y, histogram),
property (Y, X, stream),
property (Y, time, property),
stream (X), isa (X, vehicle),
property (X, [[10,0,0],[32,12,02]], region).
```

4 Query Processing: A Variant of Backward-Chaining

Once the sensors and inference units of a particular sensor infrastructure are defined, our system responds to queries by automatically composing the sensors and inference units using a variant of the standard backward chaining algorithm. In backward chaining, each unproven predicate of the query is matched with the consequent of a rule or fact in the Knowledge Base (KB). If it is matched with a rule, the antecedents of the rule must be proved by matching with another rule or fact. Backward chaining terminates when all antecedents have been matched with facts, and otherwise fails after an exhaustive search of all rules. Inference unit composition is very similar to backward chaining. The query processor matches a predicate in the query with properties of the event streams created by an inference unit. It must then provide everything that the unit needs using either other inference units or physical sensors. This procedure recurses until the requirements of all inference units are satisfied by physical sensors. The sensors and inference units used to prove the query constitute the inference graph that will provide the desired semantic values specified in the query.

The inference composition engine must ensure legal *flow* of event streams:

- all streams with the same variable name in a query or inference unit description are actually the same stream.
- all streams with the different variable names in a query or inference unit description are actually different streams.
- all streams are acyclic and originate only once.

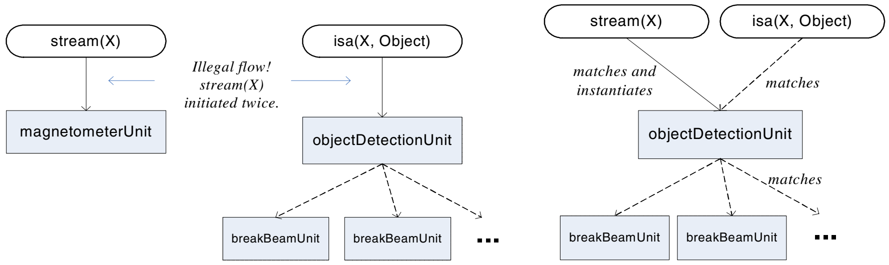
Many inference units require these global properties of all inference graphs in order to guarantee valid interpretations of their input streams.

A pure backward-chaining approach does not guarantee legal flow, as shown with the following example query:

stream(X), *isa*(X, `object`).

Pure backward-chaining would prove the first predicate in the query with any inference unit that has an output event stream. It would initially try the first unit listed in the KB, eg. the `magnetometerUnit`. The second predicate, however, does not match any post-condition of `magnetometerUnit` so the inference engine matches it with any other inference unit in the KB that does, eg. `objectDetectorUnit`, and completes the proof. The resulting proof is shown in Figure 2(a), and clearly is not a valid solution to the query because the event stream X originates in two different places, once in each subtree of the proof, and the streams denoted by X in the query are not actually the same streams. This problem is caused by the fact that backward chaining proves each predicate in the query in isolation.

Our composition engine actually *instantiates* a virtual representation of each inference unit in the KB the first time it is used in the proof, and each new event stream originating at that unit is unified with a known constant value. Subsequent predicates are proved by matching against all existing virtual instantiations before matching with any new inference units. For example, in the example query above the composition engine matches the first predicate to the `magnetometerUnit`, as did standard backward chaining, but this time creates a virtual instance of `magnetometerUnit` and assigns a unique ID to the event stream X. Once its preconditions are satisfied (by a magnetometer sensor), the inference engine moves on to the second predicate in the query: *isa*(X, object). This predicate does not match any properties produced by `magnetometerUnit`, and a match to `objectDetectionUnit` fails because the two different inference unit instantiations create different stream IDs and cannot both unify with the same variable X in the query. Thus, the illegal proof in Figure 2(a) fails. The composition engine then backtracks and matches the first predicate to a different inference unit: `objectDetectionUnit`. It then tries to match the second predicate



(a) Backward-Chaining (b) Inference Composition

Fig. 2. Inference Unit Composition. The backward chaining algorithm must be slightly modified in order to yield valid inference graphs; pure backward-chaining cannot guaranteed legal flow.

to the same virtual instance and this time succeeds because this inference graph satisfies legal flow. The resulting legal proof is illustrated in Figure 2(b).

Besides correctness of flow, there are several other benefits to using this variation of backward chaining. First, it is efficient because results from previous proofs are cached and reused; many predicates in a query are likely to be querying the same subtree in a proof. Second, it allows *mutual dependence*, where two inference units each declare the other as a pre-condition. Mutual dependence cannot occur in a pure backward-chaining approach because it would lead to infinite recursion. A third advantage is that, by causing the inference engine to first check which inference units already exist, a query will automatically reuse inference units that were instantiated in response to other queries. If two users run queries that can both be answered with an object detection unit running over three break beam sensors, the unit will only be instantiated in response to the first query; the second query will simply reuse the existing inference units. When the first query terminates, the execution engine removes only those inference units upon which no other units depend so as to not interrupt execution of the second query. In this way, Semantic Streams allows the automatic sharing of resources and the reuse of processing and bandwidth consumption between independent users without requiring them to coordinate with each other.

5 Adding Constraints to Inference Units and Queries

5.1 Quality of Service Constraints

Pure logic queries may be answerable by multiple different inference graphs. In general and especially in a network with many sensors, dozens of similar inference graphs will provide the same semantic information. In such cases, the query processor should be able to choose between comparable inference graphs based on *quality of service* (QoS) information such as total latency, energy consumption, or the confidence of data quality. In this section, we explain how to use CLP(R) notation to define QoS parameters for each inference unit and to define constraints or objective functions in the query that place an ordering on otherwise equivalent inference graphs.

We can associate for example a *confidence* parameter C with each event stream to denote the confidence of the data in the stream. For simplicity, we will assume that C takes a value between 0 and 100, although more sophisticated representations may be used. Each inference unit can derive the value of that confidence from the sensors and other inference units that it is using. For example, we could define a recursive predicate *breakGroup*(R, [], Group) which is proven by unifying `Group` with a set of break beam sensors. If `objectDetectionUnit` required such a group, it may provide a more confident detection rate when it is using more break beams for redundancy, as encoded in the following declaration:

```
inference( objectDetectionUnit,
   needs(
      breakGroup(R, [], Group),
      length(Group,Length),
      Length>=3,
      {C=>Length*20, C=<100} ),
   creates(
      stream(X),
      isa(X,object),
      property(X,T,time),
      property(X,R,region),
      property(X,C,confidence) ) ).
```

A query can then require a specific confidence value on object detections, as shown below. For this query, the query processor would continually try to prove the query until the inference graph provided a confidence value greater than 80, meaning it must include at least 5 break beam sensors (or an alternate object detection unit). Thus, the user does not need to manually specify an inference graph in order to achieve desired confidence; the programmer's logical definition of the QoS parameter allows the user to declaratively constrain the solution to those inference graphs with sufficiently high confidence.

```
stream(X), isa(X,object), property(X, C, confidence), {C>80}.
```

Similar techniques can be used to constrain latency, power consumption, bandwidth or other QoS parameters. For example, an inference unit that requires 10ms to compute the speed of an object will define its own latency to be the latency of the previous unit plus 10ms.

```
inference( speedDetectorUnit,
   needs(
      stream(X),
      isa(X,object),
      property(X,LS, latency),
      {L=LS+10} ),
   creates(
      stream(X),
      property(X, S, speed),
      property(X, L, latency) ) ).
```

Queries can place constraints on multiple QoS parameters as well as declare objective functions over them, as in the following example which minimizes latency subject to constraints on confidence levels:

```
stream(X), isa(X,object), property(X, C,confidence), {C>80},
   property(X, L,latency), {minimize(L)}.
```

To satisfy such a query, the algorithm finds all possible inference graphs that satisfy the confidence constraints and selects the one with the minimum latency. As with all inference in Prolog, the composition algorithm uses exhaustive search over all combinations of inference units, which can be quite expensive. However, composition is only performed once per query and requires minutes or less.

5.2 Runtime Parameters and Conflicts

The previous section assumes that estimates of all parameters are known at planning-time. However, when estimates are not known at planning-time, constraints on CLP(R) variables can also be used at run-time. For example, a sensor that has a `frequency` parameter will not have a predefined frequency at which it must run. Instead, it may be able to use any frequency less than 400Hz and, for efficiency reasons, it would like to use the minimum frequency possible. This unit may be defined as follows:

```
inference( magnetometerUnit,
   needs(
      sensor(magnetometer, R),
      {F<400},
      minimize{F}),
   creates(
      stream(X),
      isa(X,mag),
      property(X,T,time),
      property(X,R,region),
      property(X,F,frequency) ) ).
```

Where `minimize` is a built in CLP(R) function that sets the variable to the smallest value consistent with all existing constraints. Other constraints on its frequency might come from inference units that use this sensor. For example, the `magVehicleDetectionUnit` might require that the sensor be using a frequency that is a multiple of 5Hz.

```
inference( magVehicleDetectionUnit,
   needs(
      stream(X),
      isa(X,mag),
      property(X,F,frequency) ),
      {F1 = 5 * N, N mod 1=0}),
   creates(
      stream(X),
      isa(X,vehicle),
      property(X,T,time),
      property(X,R,region) ) ).
```

When these two inference units are composed, the frequency of the sensor is constrained to be the minimum value less than 400Hz that is a multiple of 5Hz. The resulting constraint set is singular and the planner determines the sensor frequency to be exactly 5Hz. This constraint set (while singular) is passed to the instantiation of the inference unit at runtime through the execution engine.

Because inference unit parameters are represented as CLP(R) variables, parameter conflicts can often be resolved automatically. For example, if another unit were to require that the magnetometer run at a multiple of 12Hz, the resulting constraint set on the variable F would be

- F is an integer multiple of 5.
- F is an integer multiple of 12.
- F is less than 400.
- F is the minimum value satisfying all of the above.

The constraint set reduces to the singular value of 60 which is passed to the magnetometer unit at runtime, and the sensor runs at 60Hz.

When the constraint set is not a singular value, it can be passed to each unit at runtime for what is known as *execution monitoring* and *replanning* in the artificial intelligence literature [2]. For example, the `objectDetectionUnit` from above can be given the constraint set $\{80 < C < 100\}$. When a sensor fails or the nominal confidence values percolating up from the sensors decrease, it may determine that it can no longer meet the required constraints and it signals an error to the execution engine, which asks the query processor for a new inference graph.

6 An Example of Semantic Streams

To provide an example of how the Semantic Streams framework is used, we deployed a sensor network on the second floor of a parking deck on the Microsoft corporate campus. The network consisted of three different types of sensors: a web camera, a magnetometer, and infrared break beam sensors. Both the break beam and magnetometer sensors were controlled by micaZ motes and communicated wirelessly with our microserver, a headless Upont Cappuccino TX-3 Mini PC. The camera and microserver were both connected to the corporate network by Ethernet.

The focus of the network was a 4x5 meter area directly in front of an elevator. All vehicles entering this floor of the parking deck passed through this area, as did most pedestrians using the elevator. We placed 5 infrared break beam sensors in a row across the area, 1m apart and about .5m from the ground, such that the beams were broken in succession by any passing human or vehicle. The camera was also focused on the area and a magnetometer was placed about 10m downstream. The focus area and the arrangement of the six wireless sensors, camera, and microserver is shown in Figure 3.

Although the number of sensors in our deployment is small, they can be used for many different purposes. For example, they can infer the presence of humans, motorcycles and cars as well as their speeds, directions, sizes, metallic payloads and, in combination with data from neighboring locations, even their paths through the parking garage. In this paper, we consider three hypothetical users at Microsoft that want to use the sensor infrastructure described above:

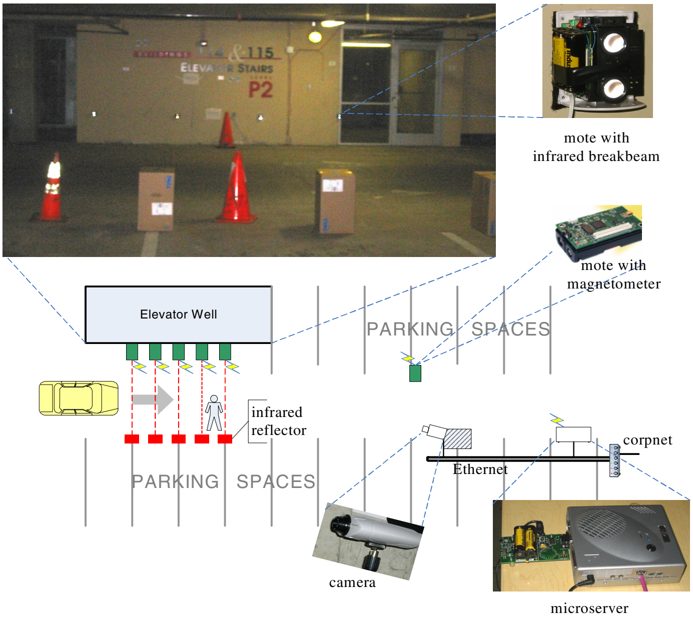
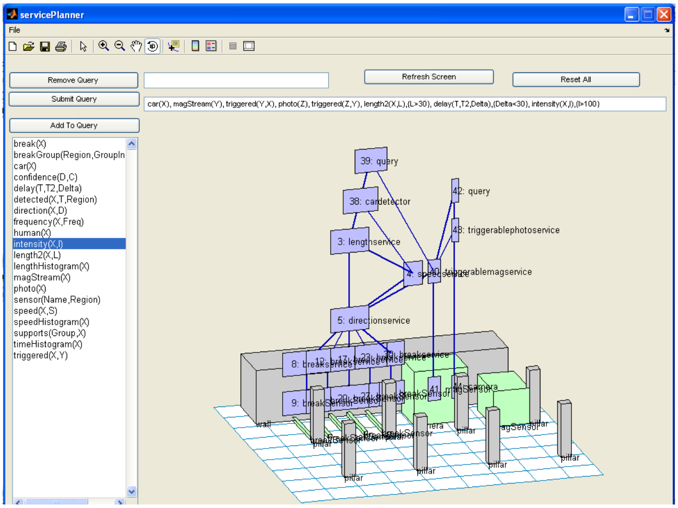


Fig. 3. Sensor Infrastructure. The break beam sensors were laid out in a row on the wall in the focus area. The digital camera was focused on the same area. The magnetometer was placed several meters downstream near the microserver.

- Police Officer Pat wants a photograph of all vehicles moving faster than 15mph.
- Employee Alex wants to know what time to arrive at work in order to get a parking space on the first floor of the parking deck.
- Safety Engineer Kim wants to know the speeds of cars near the elevator to determine whether or not to place a speed bump for pedestrian safety.

All three applications must run continuously and simultaneously using the same hardware. There are several places where conflicts can arise: which nodes are on or off, which program image each node is running, what sampling rates they are using etc. However, all three users are from different organizations within the company and are not be able to easily coordinate. In this example we demonstrate how the system can 1) automatically share and reuse resources between independent users and 2) compose inference units from two different applications to create a new semantic composition for a third application. For brevity, our demonstration does not illustrate how the users optimize QoS parameters.

We assume Pat and Alex are the first users of this sensor infrastructure and must create all of their own inference units. Pat creates units to infer object speeds from break beam sensors, identify them as vehicles, and take pictures of a region triggered by an event. Alex creates a unit to classify objects as vehicles based on magnetometer output and a unit to plot arbitrary values in a histogram. All of these inference units are added to the library associated with



car(X), magStream(Y), triggered(Y,X), photo(Z), triggered(Z,Y), length2(X,L),{L>30}, delay(T,T2,Delta),{Delta<30}, intensity(X,I),{I>100}

break(X)
breakGroup(Region,GroupIn
car(X)
confidence(D,C)
delay(T,T2,Delta)
detected(X,T,Region)
direction(X,D)
frequency(X,Freq)
human(X)
intensity(X,I)
length2(X,L)
lengthHistogram(X)
magStream(X)
photo(X)
sensor(Name,Region)
speed(X,S)
speedHistogram(X)
supports(Group,X)
timeHistogram(X)
triggered(X,Y)

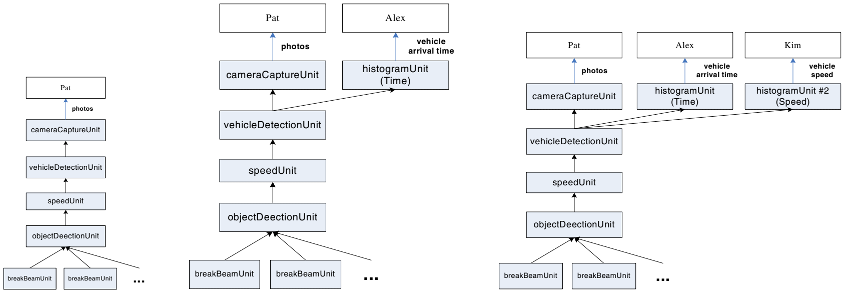
Fig. 4. User Interface. Each user is presented with a 3D rendering of the sensors in the testbed and, on the left, all predicates that are queryable.

the infrastructure and each user is presented with the graphical user interface shown in Figure 4. The interface shows a 3D rendering of each sensor in our garage testbed and the region that the sensor covers. Furthermore, the predicates describing the event streams created by all inference units in the system are listed on the left side of the screen. These stream descriptions are the only predicates that can be used in a query, although variable names may be changed to create new compositions and CLP(R) constraints may be added. Each user selects the appropriate predicates to create their desired queries:

Pat *stream*(X),
property(X,P, photo),
property(X,Y, triggerStream),
property(X,speed, triggerProperty),
stream(Y),
isa(Y,vehicle),

Alex *stream*(X),
property(X,H, histogram),
property(X,Y, plottedStream),
property(X,time, plottedProperty),
stream(Y),
isa(Y,vehicle),

Kim *stream*(X),
property(X,H, histogram),
property(X,Y, plottedStream),
property(X,speed, plottedProperty),
stream(Y),
isa(Y,vehicle),



(a) Step 1 (b) Step 2 (c) Step 3

Fig. 5. Composite Inference Graphs. In step 1, Pat's query produces the expected inference graph. In step 2, Alex's query reuses one of the inference units that is instantiated in response to Pat's query. In step 3, Kim's query composes units from Alex's and Pat's queries to create a new semantic composition.

In our example, Pat executes the query first and the system generates the inference graph shown in Figure 5(a). When Alex's query is executed, a new `histogramUnit` is first instantiated. However, it does not use the magnetometer based vehicle detection because another equivalent unit already exists. It uses instead the `vehicleDetectionUnit` instantiated for Pat's application, which is based on break beams. The resulting composite inference graph is shown in Figure 5(b). Alex's application illustrates Semantic Streams automatically sharing resources between independent users.

Kim's query reuses inference units from both Pat's and Alex's applications. The `histogramUnit` from Alex's application can be reused, although a new instance must be created because the existing instance does not match Kim's query (it plots different values). The existing instance of the `speedUnit` from Alex's application, however, can be reused because it is inferring the speeds of vehicle objects. Kim's application illustrates how existing units from the other two applications were composed to create a semantically new application. The final inference graph with all three applications is illustrated in Figure 5(c) and is also seen in the user interface in Figure 4. These inference units can then be instantiated on the server and fed raw data from the sensors as it is received, producing the semantic values requested by the users.

7 Related Work

Semantic Streams adapts ideas from Semantic Web Services (SWS), a movement to semantically describe and automatically compose web services, to the problem of *macroprogramming*, which is the process of writing a program that specifies global sensor network behavior as opposed to the behavior of individual nodes. Sensor networks have previously seen two main classes of macroprogramming: database approaches like TinyDB [3, 4] and functional language approaches such as Regiment [5]. Semantic Streams is similar to these approaches in that the user issues a query specifying global behavior. One main difference is that, in both systems above, the user is required to understand which operations to run over the raw sensor data and how to interpret the meaning of the results. Semantic Streams allows the user to issue queries over semantic values directly without addressing which data or operations are to be used. The advantages of semantic queries are analogous to those of macroprogramming in general: the user of macroprogramming need not specify the best time and place to execute each operation, while the user of semantic queries need not specify which operations to run or which data to run them over. This allows the user to make fewer low-level decisions and allows the system an extra degree of freedom for automatic optimization during execution.

Our inference unit composition algorithm differs from the three main techniques that have previously been used for the automatic composition of Web Services: agent-based, planning-based, and inference-based approaches. Agent-based approaches perform a heuristic search through the set of all Web Services, either simulating or actually executing each of them to find a path to the desired resultant state [6, 7]. This technique does not easily transfer to Semantic Streams because it explicitly assumes a sequential execution model.

A concurrent execution model can be captured by Artificial Intelligence techniques such as Partial Order Planning (POP) and Hierarchical Task Networks (HTN). The problem with these techniques is that the planner performs a rather mechanical matching of post-conditions provided at time t_i with pre-conditions needed at time t_{i+1}; it cannot perform any *reasoning*, which is needed in our system to deal with spatial relationships, quality of service properties, and parameter conflicts, among other things.

Reasoning can be performed by an inference engine as in SWORD [8], which uses an inference engine to automatically compose Web services by converting each one into a set of logic rules which states that its post-conditions will be true given its pre-conditions. The problem with the pure inference-based approach is that all proofs are tree-based while most inference graphs are general directed graphs. Because SWORD does not use virtual representations of inference units during composition, it cannot guarantee legal flow of event streams. Moreover, it cannot represent an inference graph with mutual dependence.

8 Conclusions

The framework presented in this paper provides a declarative language for describing and composing inference over sensor data. There are several benefits to this framework. First, declarative programming is easier to understand than low-level, distributed programming and allows common people to query high-level information from sensor networks. Second, the declarative language allows the user to specify desired quality of service trade-offs and have the query interpreter execute on them, rather than writing imperative code that must provide the QoS. Finally, the framework allows multiple users to task and re-task the network concurrently, optimizing for reuse of services between applications and automatically resolving resource conflicts. Together, the declarative programming model and the constraint-based planning engine in our framework allow non-technical users to leverage previous applications to quickly extract semantic information from raw sensor data, thus addressing one of the most significant barriers to widespread use of sensor infrastructure today.

Acknowledgements

Special thanks to Prabal Dutta and Elaine Cheong for help with the parking garage deployment, and to Nithya Ramanathan for help with the composition algorithm.

References

1. SICS AB: SICStus Prolog 3.12.0 user's manual (2004) http://www.sics.se/isl/ sicstuswww/site/documentation.html.
2. Russell, S., Norvig, P.: Artificial Intelligence (Second Edition). Prentice Hall (2004)
3. Madden, S.R., Franklin, M.J., Hellerstein, J.M., Hong, W.: TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In: OSDI. (2002)
4. Bonnet, P., Gehrke, J., Seshadri, P.: Towards sensor database systems. Lecture Notes in Computer Science (2001)
5. Newton, R., Welsh, M.: Region streams: Functional macroprogramming for sensor networks. In: DMSN. (2004)
6. McIlraith, S., Son, T.C.: Adapting golog for composition of semantic web services. In: KRR. (2002)
7. Carman, M., Serafini, L., Traverso, P.: Web service composition as planning. In: ICAPS. (2003)
8. Ponnekanti, S.R., Fox, A.: Sword: A developer toolkit for web service composition. In: World Wide Web Conference. (2002)