Semantic Streams: a Framework for Declarative Queries and Automatic Data Interpretation

Kamin Whitehouse
kamin@cs.berkeley.edu
Feng Zhao
zhao@microsoft.com
Jie Liu
liuj@microsoft.com

April 2005
Technical Report
MSR-TR-2005-45

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 sensor data, the user can query vehicle speeds; the system decides which sensor data and which operations to use to infer the vehicle speeds. The user can also place constraints on values such as the confidence with which the speed was measured or the amount of energy consumed to measure the speeds. This framework is designed to work in a shared sensor infrastructure, where multiple queries may coexist for extended periods of time, instead of a hand-designed, single purpose sensor network. We propose a semantic service programming model and describe a service description language and a query processor that support the programming model. We demonstrate how this system can be used with a network of video, magnetometer, and infrared break beam sensors deployed in a parking garage.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

1 Introduction

Networks of sensors are ubiquitous in our daily environments even today. For example, most building and office environments have both HVAC and card key sensors, many road intersections and highways have vehicle detection sensors, and a large percentage of new homes have security sensors. Despite their ubiquity and potential for providing information utility, however, these sensors are largely underutilized because the raw data are not readily consumable by end users. A building manager might want to be alerted to excess building activity over the weekends, or a safety engineer might want a histogram of vehicle speeds in the parking garage. Neither the user nor the system can easily interpret raw sensor data from card key readers, motion detectors, and infrared beams into semantic values such as building activity or vehicle speeds. Thus, one of the greatest barriers to the widespread use of sensor networks by non-technical users today is the inability to synthesize semantic values from raw sensor data.

This paper presents a framework called *Semantic Streams* that allows users to program sensor networks with a declarative statement such as, "I want the speeds of vehicles near the entrance of the parking garage." This is different from other approaches in which the user poses queries over raw sensor data [8, 7]. The system allows multiple, independent users to use the same network simultaneously and automatically shares resources and resolves conflicts between their 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 speed estimates to be greater than 90%," or "I want to minimize the total number of radio messages."

Our framework uses a *semantic services* programming model, where each service is a process that infers semantic information about the world and incorporates it into an *event stream*. Each service has a first-order logical description of the semantic information that it needs to be in its input streams and that it adds to its output streams. The input and output streams of services can be wired together. This programming model was designed to allow the processes of interpreting data to be *composed* to create semantically new applications.

Once a set of sensors and services are declared, possibly through libraries or previous applications, the user can pose a query in first-order logic. The query processor employs an inference engine to decide which sensors and services will provide the semantic information that the user requires. The services are converted into a set of rules with *pre-conditions* and *post-conditions* and the inference engine uses a variant of backward-chaining. In other words, it tries to match each element of the query with the post-condition of a service. When successful, the pre-conditions of that service are added to the query. The process terminates when all pre-conditions are matched with declarations of physical sensors, which do not have pre-conditions. The main difference between pure backward-chaining and service composition is that our inference engine actually instantiates each service during the composition process and reuses existing instances whenever possible. This allows *mutual dependence* between services and the ability to check for legal *flow* of event streams, neither of which would be possible with pure backward-chaining.

Semantic Streams is designed for the *sensor infrastructure* domain in which a sensor network may be built by different hardware vendors. It is used repeatedly over long periods of time, for different types of applications, and by independent users, perhaps from different organizations entirely. Sensor infrastructures pose several important problems such as sharing resources between independent applications, resolving conflicts between separate user groups, and coordinate between different users, groups, and hardware vendors. In our semantic service model, all service interfaces are maintained in a central repository (namely a *query server*) along with their complete semantic descriptions, so different groups and hardware vendors can share services without needing to share or understand each other's source code. Because our inference engine reuses existing instances of services whenever possible, it automatically and efficiently reuses resources and operations that are being performed by other users without the need for explicit cooperation. Finally, the semantic markup language used to describe services is designed to give the query processor as much freedom in query execution as possible. This allows the query processor to automatically resolve resource conflicts such as when two applications require different sampling rates from the same sensor.

In general, many combinations of sensors and services will satisfy a given query. The Semantic Streams markup language allows the user to specify constraints on quality of service parameters to help select among otherwise equivalent alternatives. For example the user might specify, "The confidence level on car detections should be above 90%, and latency less than 50 milliseconds." The query engine propagates these constraints through the components in the service graph. If a particular combination does not satisfy the user's constraints, the engine tries the next combination. Allowing the user to specify ranges of constraints instead of specific values is an important component to resource mediation between applications. For example, the system may need to provide one application the largest allowable latency in order to meet the confidence requirements of a second application without increasing overall energy consumption in the network.

The Semantic Streams model and its query processing engine are integral parts of a service-oriented architecture for networked sensor systems, as shown in Figure 1. In the overall architecture, when a user poses a query as an event stream, the query planning engine generates a task graph. The graph is then assigned to a set of physical nodes for execution, a process called *service embedding.* The services are assigned in a way that the assignment preserves the proximity in data flows and optimizes for resource usage, latency, and load. This is an interesting variant of the classic task assignment problem, with the additional sensor net constraints. The service runtime on each node, accepts the task graphs, instantiates services on demand, resolves possible conflict between tasks and resource availability, and executes the query. The focuses of this paper are on service interfaces and automated query planning. The service embedding and execution work are presented in other papers.

To ease our discussion, we explore the interaction of Semantic Streams with

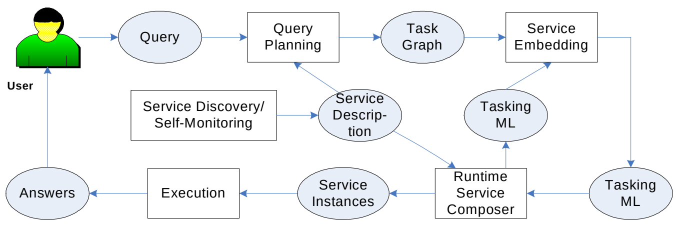


Figure 1: **Planning and Execution** *The user first poses a query to the query processor, which derives an acceptable service graph. That graph is passed to the execution engine along with all variable unification and constraint sets resulting from planning. The execution engine may call back to the query processor during replanning.*

the usage model of sensor infrastructure through a sensor network that we deployed in a parking garage. We demonstrate the semantic descriptions of several services and their use in three queries from different users. We also discuss several limitations and possible extensions of this work, including the ability to give the user actionable error messages. For example, if the query cannot be satisfied, the query processor should be able to suggest where to place new sensors, what operations need to be implemented, or which constraints need to be loosened in order to satisfy the query.

The rest of this paper is organized as follows: Section 2 discusses related work such as macroprogramming and Semantic Web Services. Section 3 describes an parking garage sensor network that we deployed as a motivating example of a tiered sensor network architecture and usage model. Section 4 describes the semantic services programming model. Section 5 describes the first-order logic and CLP(R) semantic markup language used to declare sensors, services, and queries. We explains how to expose quality of service parameters in each service and how to declare constraints or objective functions over them in queries. Section 6 describes the implementation of the query processor, where modifications are made to a standard inference engine to result in valid service graphs. Section 7 describes limitations of this work and possible extensions.

2 Related Work

Semantic Streams adapts work on the composition of Semantic Web Services to the problem of macroprogramming in sensor networks. *Macroprogramming* is a term often used to refer to the process of writing a program that specifies global network behavior as opposed to the behavior of individual nodes. Semantic Web Services (SWS) is a movement to semantically describe modular programs so that they can be automatically composed in different ways to form new services.

2.1 Macroprogramming

Most sensor network programs specify the behavior of individual nodes, with the assumption that these behaviors at the node level will result in the desired emergent behaviors at the network level. Macroprogramming attempts to do the reverse: the user specifies the global network behavior and the macroprogramming framework determines the appropriate local node behaviors. Sensor networks have seen two main classes of macroprogramming: database approaches like TinyDB [8, 2, 6] and functional language approaches such as Regiment [11]. Database approaches allow the user to issue declarative queries over sensor data such as SELECT, JOIN, SUM, or MAX and the system must collect the appropriate sensor data to answer the query. Regiment allows the user to perform more general operations such as MAP, FOLD, and FILTER, which map a function over, aggregate over, or filter all data in a region. The system determines where and when data is stored and operations are performed in the network.

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 less low-level decisions while allowing the system an extra degree of freedom to optimize during execution.

2.2 Composition of Semantic Web Services

No pre-specified language can provide semantic programming without operating on a sophisticated model of the world. To address this problem, Semantic Streams borrows an idea from SWS: the world model is incorporated into each component of code, and the code can be composed in different ways to substitute for a world model. A markup language is used to indicate the semantic meaning of each piece of code, and as more code is added to the system the world model becomes more complete.

Web services are modular programs that are accessible over the web, perhaps from different companies, and may provide services such as credit card authentication or the ability to reserve a plane, hotel or restaurant (see e.g. www.xmethods.com). A main goal of web services is to be easily *composable* to provide, for example, a unified travel agent service that plans a complete vacation through several different companies. To facilitate composability, several languages including WSDL, SOAP, and UDDI describe service interfaces, formalize message protocols, and ease in service discovery. These descriptions, however, are merely syntactic and the composition of services typically requires a human to identify the appropriate services and choreograph them into a workflow. SWS is a movement to semantically describe web services so that they can be composed automatically by computers. Several frameworks such as IRS-II, OWL-S, and WSMF have been proposed for this purpose. See [3] for an thorough overview of this topic.

Semantic Streams adopts the solution provided by SWS by using a markup language to semantically describe pieces of code and automatically compose them into a service graph to answer a query. However, the two approaches are also very different. Web services represent actions on the world while semantic services are stream processors. The execution model is very different: a web services workflow begins with a user-initiated request and continues with a single point of execution through a series of choreographed operations. When it is needed, each web service takes an input, provides an output, and terminates. In contrast, Semantic Streams services operate continuously on a data type called an *event stream*, which is a series of asynchronous events. All services run concurrently. Because of the different execution models, the composition process is also different. The composition of web services usually requires reasoning about execution time; our system does not reason about execution time but does require spatial reasoning.

3 A Motivating Scenario

Most research today focuses on low-power, wireless sensor networks that are deployed in harsh, remote environments and are owned, managed, and programmed by a single group of individuals [9, 16]. However, wired, powered, and stationary sensor networks are much more common today than their more extreme counterparts. In contrast to special purpose networks, these sensor infrastructures could be used repeatedly over long periods of time by many different people and for several different applications. Several new problems arise under this usage model, such as resource mediation between applications of multiple, independent users. New opportunities also arise, such as the ability of Semantic Streams to exploit this usage model by reusing semantic inferences required for older applications to automatically generate solutions to new applications. To better illustrate the challenges and opportunities of this problem domain, we present a concrete sensor network deployment to represent a typical shared, general-purpose sensing infrastructure.

3.1 Example of Sensor Infrastructure

We deployed a sensor network on the second floor of a parking deck on the Tinyware corporate campus. The network consisted of three different types of sensors: a web camera, a magnetometer and infrared break beam sensors. A break beam sensor bounces an infrared beam against a distant reflector. When an object comes between the sensor and the reflector, it detects that the beam has been broken; when the object moves away it detects that the beam has been

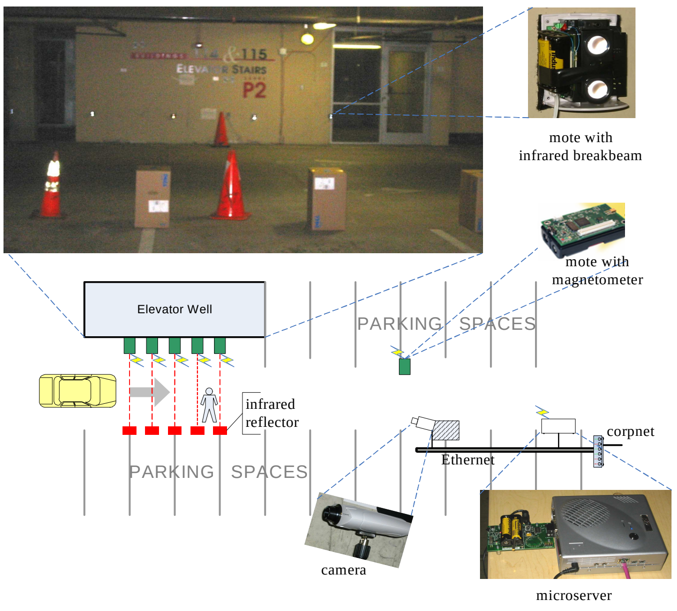


Figure 2: **Sensor Infrastructure** *The break beam sensors were laid out in a row on the wall in the focus area. The digital camera was focused on the same area. The magnetometer was placed several meters downstream near the microserver.*

re-detected. This is the same sensor that might be found at a store entrance to detect customers entering and leaving. 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 2.

3.2 Example Users and Applications

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 and, in combination with data from neighboring locations, even their paths through the parking garage. In this paper, we consider three hypothetical users at Tinyware that might want to use the sensor infrastructure described above:

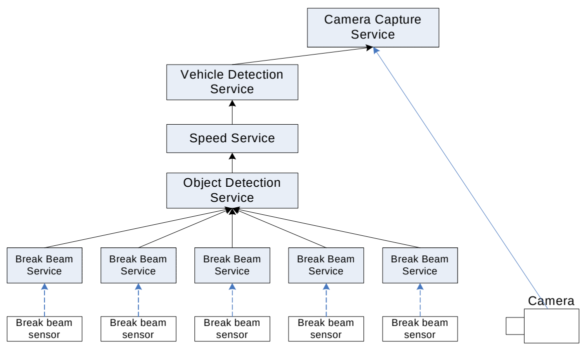
- Police Officer Pat wants a photograph of all vehicles moving faster than 15mph.
- Employee Alex wants to know what time to arrive at work in order to get a parking space on the first floor of the parking deck.
- Safety Engineer Kim wants to know the speeds of cars near the elevator to determine whether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

Pat's application can be solved by using the break beam sensors to infer the speeds of vehicles and triggering the camera to take a photograph. The magnetometer sensor could be used to provide extra confidence that the observed object is indeed a vehicle. Alex's application can be solved simply by observing the distribution of times when cars are observed on the second floor of the parking deck since, presumably, most people do not park on the second floor when there are still spaces on the first floor. Vehicles can be detected by either the break beam sensors or the magnetometer sensors, and the times of their detections can be plotted in a histogram for Alex. Kim's application is a combination of the other two applications. The break beam sensors can be used to infer the speeds of vehicles as in Pat's application, and these speeds can be plotted in a histogram as in Alex's application.

All three applications must run continuously and simultaneously using the same hardware. There are several places where conflicts can arise, such as 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 would not be able to easily coordinate. In this paper, we will show how the Semantic Streams framework avoids the need for coordination between the users. Furthermore, we show how the system is able to reuse functionality from Pat and Alex's applications to automatically compose an application for Kim.

4 The Semantic Services Programming Model

The Semantic Streams programming model contains two fundamental elements: *event streams* and *semantic services*. Event streams are sequences of asynchronous events in time, each of which has a set of associated properties such as time and location. The events can represent objects, such as people or cars, and can have properties such as speeds, directions, or identities. Semantic services are processes that infer semantic information about the world and incorporate it into an event stream. Every event stream originates at a single service and new properties can be *added* to its events as the stream is processed by other services. For example, one service may infer the presence of an object, another service may identify it as a vehicle, and a third service may infer the speed of that vehicle from the sensor data. Semantic services can be composed in new



Break Beam Service
Function: A wrapper service around the break beam sensor.
Inputs: None.
Outputs: A stream of *break* events with two properties: the *rising edge time* at which the beam was broken and the *falling edge time* at which it was redetected.

Object Detection Service
Function: Analyzes the break events to infer the presence or absence of an object.
Inputs: Multiple break streams
Outputs: An *object* stream, where each object event has *time* and *region* properties indicating where and when it was detected.

Speed Service
Function: Compares the rising and falling edges of the break events to infer the speed of the object.
Inputs: An object stream and the break streams that support it
Outputs: An object stream, where each event has a *speed* property.

Vehicle Detection Service
Function: Identifies an event as a car by thresholding the speed of the event [1].
Inputs: An object stream with speed properties
Outputs: An object stream, where each event indicates whether or not the object is a vehicle.

Camera Capture Service
Function: Captures an image from the digital camera when a vehicle is detected with speed greater than 15mph.
Inputs: An object stream with vehicle and speed properties
Outputs: An object stream, where each event has a *photo* property.

Figure 3: **Pat's Application** *requires services that read the break beam sensors, detect objects, identify them as cars, infer their speeds and use these speeds to trigger a camera. a) shows the service composition and b) provides descriptions of the services*

ways with different sensors to enable new types of semantic inference about the world.

Figure 3 and Figure 4 represent the required services, descriptions of how they function, and the service compositions that could be used to provide Pat's and Alex's desired application respectively in our example testbed. Note that, although each event stream originates at a single service, it is not necessarily processed by other services in a linear fashion. For example, a user may want to take pictures of both speeding vehicles and pedestrians. To facilitate the branching and merging of event streams, the service that originates the stream (in this case, the object detector service) assigns each event in the stream a unique ID.

Semantic services are different from both web services and software components like NesC modules [5]. Semantic services can be connected simply by *wiring* them together, similar to NesC modules. However, semantic services communicate through a publish/subscribe mechanism, placing events into an output buffer, where they are read by subscribing services. This is different from the event/command semantics in NesC where a module effectively evokes the function of another module. It is also different from Web Services, which do not usually communicate directly but only through a third entity which orchestrates them into a single workflow.

Semantic services also differ from NesC modules in that their basic function is to infer new information about the world and to explicitly encode it into an event stream. All communication or computational operations are internal to the service. This is different from a NesC module, whose sole function may be to mechanically move data from one node to another; the inference of information about the world is often an emergent behavior from the collaboration of many NesC modules.

Semantic services are thus a higher-level programming abstraction than NesC modules and can in fact be built from NesC modules. Figure 5(a) shows how the breakbeam and object detection services could be implemented as NesC modules. The breakbeam service is conceptually just a single NesC module, with one breakbeam service running on each of the 3 break beam sensors. This implementation of the object detection service, however, is conceptually the combination of 3 distributed NesC modules, which might share their break events using radio packets and elect a leader to analyze them and generate the object detection events. All communication between the NesC modules is internal to the semantic service. Thus, unlike NesC modules, semantic services can be distributed entities. Furthermore, just from the event stream semantics point of view, this implementation is the same as the centralized implementation as shown in Figure 5(b), even though the centralized service can be both easier to program and more efficient (it requires one less radio message). They can only be differentiated by quality of service and resource utilization characteristics.

Obviously, the two applications shown in Figure 3 and Figure 4 can both be written and composed manually by the users. In the following sections, we explain how the users can incorporate their applications into the Semantic Streams framework so that the services can be reused by other applications and the composition can be automated.

5 A Service Markup and Query Language

The Semantic Streams framework requires that the semantics of each service's inputs and outputs be declared, along with the type and location of each sensor. These declarations allow the query processor to compose sensors services in semantically meaningful ways.

5.1 Background on Logic Programming

The Semantic Streams markup and query language is built on 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. In Prolog, words beginning with a capital letter (eg. X) are variables, those beginning with lower case letters (eg. `const`) are constants, and those followed by parenthesis are predicates (eg. *value*(X,`const`)). A Prolog rule consists of a conjunction of antecedents and their consequent, such as the fact that Z is the grandparent of X if Z is the parent of Y and Y is the parent of X.

```
grandparent(Z,X):-parent(Z,Y),parent(Y,X).
```

A fact is simply a rule with no antecedents, such as the facts that Pat is the parent of Alex and Alex is the parent of Kim.

```
parent(pat,alex).
parent(alex,kim).
```

A query is a set of antecendents with no consequent. The solution to a query is all sets of bindings to the query variables that make the query true. For example, the following two queries ask who is a grandparent of whom, and who is a grandparent of Pat, respectively. The answer to the first query is that pat is the grandparent of Kim. The second query evaluates to false, indicating that Pat has no known grandparent.

```
grandparent(X,Y).
   ans:  X=pat, Y=kim.

grandparent(X,pat).
   ans:  false.
```

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. CLP(R) constraints can be combined with Prolog facts, rules and queries by enclosing all CLP(R) statements in brackets. For example, the following rules state that all dates are between 1 and 31 and that the date next week is today's date plus seven.

```
 isDate(X) :- {X=>1,X=<31}.
nextWeek(X,Y) :- {Y=X+7}.
```

Unlike standard Prolog, CLP(R) queries are answered not by bindings on each variable but by the resulting constraint sets on each variable. For example, a statement declaring that Y is *nextWeek* of X results in several constraints on both X and Y.

```
{ isDate(X), isDate(Y), nextWeek(X,Y) }.
   ans:  {X=>1},
         {X=<24},
         {Y=>8},
         {Y=<31}.
```

In this example, if one date is known, the constraint set on the other variable reduces to a singleton.

```
{X=12, isDate(Y), nextWeek(X,Y)}.
   ans:  {Y=19}.
```

For a more complete description of Prolog and CLP(R), see [1, 15]. Our language design takes the advantage of CLP(R) and are implemented using SICStus prolog which has a CLP(R) extension.

5.2 Declaring Sensors and Simple Services

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

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

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 *stream*(), *isa*(), and *property*() predicates describe an event stream and the type and properties of its events. The **service**(), ***needs***(), and ***creates***() predicates describe a service the semantic information that it needs and creates. In query processing, these are treated as *rules* and their *pre-conditions* and *post-conditions*. For example, the Vehicle Detector in Alex's application could be described as a service that uses a magnetometer sensor to detect vehicles and creates an event stream with the time and location in which the vehicles are detected.

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

5.3 Variable Input Streams

The `histogramService` used for Alex's application must plot the arrival times of vehicle detection events. The service could be declared only for this purpose:

```
service( histogramService,
  needs(
     stream(X),
     isa(X,vehicle),
     property(X,T,time),
  creates(
     stream(Y),
     isa(Y,histogram) ) ).
```

However, this description only allows the histogram to plot `time` properties of `vehicle` events, even though the actual service implementation can plot any type of numeric values; this service cannot be *composed* to plot any other event streams or properties. To solve this problem, Alex would define the histogram service to plot any property value of any type of event stream, as follows:

```
service( histogramService,
  needs(
     stream(S),
     property(S,V,P),
  creates(
     stream(Y),
     isa(Y,histogram),
     property(Y,S, plottedStream) ) ).
     property(Y,P, plottedProperty) ) ).
```

The value of S defines the type of stream and the value of P defines the property that is to be plotted. By defining the input stream to be a variable, this re-parameterization allows the user to query for histograms over different types of event streams.

5.4 Querying

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 services could be composed to generate events X that are known to be vehicles. The query interpreter will generate all such possible service 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, object),
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, Alex's query would request a stream of histogram events where the values to be plotted are the arrival times of vehicle events from a different stream. The last line of the query further constrains the plot to only those events detected in a particular region.

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

Queries are solved using backward chaining. For example, the first three predicates in Alex's query can be proved by the post-conditions of the `histogramService`. In order to use the histogram service, however, a stream of events with `time` properties must be available. This can be provided by the post-conditions of the `magVehicleDetectionService`, which in turn requires a `magnetometer` sensor. The last two predicates in Alex's query further constrain the stream X to be a `vehicle` stream originating in a particular region. The steps of the final proof become the application that runs on a sensor network. The execution results of that application are the query answers.

5.5 Reasoning About Space

Sensors have real-world spatial coordinates and, as such, our query processor must be able to reason about space. For example, The declaration of the `magVehicleDetectionService` above uses the same variable R in both the ***needs***() predicate and the ***creates***() predicate to indicate that the region in which vehicles are detected is the same region in which the magnetometer is sensing.

The object detection service used in Pat's application, however, is more complicated. It requires a number of break beam sensors with close proximity to each other and with non-intersecting infrared beams. One way for Pat to declare this is to require three sensors in specific, known locations:

```
service( objectDetectionService,
  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 service description, however, cannot be composed with other sets of break beams. It also cannot be used in any region besides that which has been hard coded. To solve this problem, Pat would use two *logic rules* about spatial relations:

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

The first rule proves that region A is a subregion of region B while the second rule proves that 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 `objectDetectionService` can now be defined to require any three break beams that are within a region *R* and that do not intersect each other.

```
service( objectDetectionService,
   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 the line `\+ intersect( _,R1,R2)` is true if no region is the intersection of regions *R1* and *R2*. Using this semantic description, the service can be used with any three non-intersecting break beam sensors in any region *R*.

5.6 Variable Numbers of Input Streams

While reasoning about space is essential to any query processor that uses real-world sensors, arbitrary reasoning ability is also often convenient. Because the Semantic Streams query processor uses Prolog, the user can add arbitrary reasoning capabilities to it.

For example, the `objectDetectionService` as described requires exactly three break beam sensors. Similar services that use two or four sensors would need to be defined as completely separate services. We could define a recursive logic rule to allow the service to operate over an arbitrary number of break beam sensors. The *breakGroup* predicate is true for any *group* of non-intersecting break beam sensors that are within a specific region.

```
breakGroup( <region>, <initial group>, <group>).
```

For brevity, we do not reproduce the entire definition here. Using this rule, the `objectDetectionService` could then be redefined very simply to require a group of *at least* three break sensors:

```
service( objectDetectionService,
  needs(
     breakGroup(R, [], Group),
     length(Group,Length),
     Length>=3 ),
  creates(
     stream(X),
     isa(X,object),
     property(X,T,time),
     property(X,R,region) ) ).
```

5.7 Quality of Service Constraints

Purely logic queries may be answerable by multiple different service graphs. For example, the query `stream(X), isa(X,vehicle).` could be answered by Alex's `magVehicleDetectionService` or Pat's `vehicleDetectionService`. In general and especially in a network with many sensors, dozens of similar service graphs will provide the same semantic information. In such cases, the query processor should be able to choose between comparable service 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 declare QoS parameters with each service description and to define constraints or objective functions defined in the query that place an ordering on QoS values.

We can associate a confidence parameter C with each event stream by adding a *confidence* property. Each service can derive the value for that parameter from the sensors and other services that it is using. For example, the `objectDetectionService` may be more confident in its detection rate when it is using more than three break beams for redundancy:

```
service( objectDetectionService,
  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 request a specific confidence value and the appropriate number of break beam sensors will be used while the rest will remain off.

```
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, a service that requires 10ms to compute the speed of an object will define its own latency to be the latency of the previous service plus 10ms.

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

The QoS parameters and constraints described in this section are used only at *planning time*, i.e. the time at which the query processor composes sensors and services in response to a query. It is assumed that all quality of service parameters are known. In the next section, we describe how to extract parameter information from planning time and use it at runtime.

5.8 Runtime Parameters & Conflicts

While planning-time values Prolog variables are used to wire the service instantiations, values of CLP(R) variables can also be used at runtime to pass parameters to each service. Instead of using the unification of the variables, each service is passed the resulting constraint sets on each of its parameters. For example, a sensor service that has a `frequency` parameter may be able to use any frequency less than 400Hz. For efficiency reasons, it would like to use the minimum frequency possible. This service may be defined as follows:

```
service( magnetometerService,
  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 services that use this sensor. For example, Alex's `magVehicleDetectionService` might require that the sensor be using a frequency that is a multiple of 5Hz.

```
service( magVehicleDetectionService,
  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 services are composed, the frequency of 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 service at runtime through the execution engine.

Because service parameters are represented as CLP(R) variables, parameter conflicts can often be resolved automatically. For example, if another service 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 is the singular value of 60, which is passed to the magnetometer service at runtime.

The resulting constraint sets on QoS parameters can also be passed to each service at runtime. For example, the `objectDetectionService` above is required by the query to achieve confidence $C>$ 80. At planning time, it estimated a confidence level of 100 given five break beam sensors. However, if one sensor fails or if the nominal confidence values percolating up from the sensors decreases, the `objectDetectionService` may determine that it can not longer meet the required confidence constraints. In this case, it will signal an error to the execution engine which would ask the query processor for another service graph. This process is also known as *execution monitoring* and *replanning* in the artificial intelligence literature [13].

6 Implementation

Incorporating SICStus prolog with CLP(R) extension, our tool processes queries by a variant of backward chaining on service and sensor declarations. Notice that the goal of our query processing is not to show whether a used query can be answers, but to come up with a compact plan for service composition. It is desirable to share services as much as possible among multiple queries.

6.1 Query Processing

In general backward chaining, each unproven element 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. Our system works similarly. The query processor can prove a predicate in the query with the event streams that a service creates. It must then prove everything that the service needs. This procedure recurses until the pre-conditions of all service needs are satisfied by physical sensors definitions.

The main difference between general backward chaining and service composition is that our inference engine actually *instantiates* a virtual representation of each service in the KB every time it is needed. For example, the following query asks for an `object` event stream.

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

When the inference engine processes the first predicate, it searches for any service with a similar post-condition declared in its *creates*clause and finds the *objectDetectionService*. At this point, it actually creates a virtual representation of the service in the KB and adds all of the services preconditions to the query. Once these preconditions are satisfied (by three or more break beam sensors), the inference engine moves on to the second predicate in the query: `stream(X)`. Before matching this predicate to service descriptions in the KB, it matches it to the post-conditions of all existing virtual service instantiations. In this case, the predicate matches a post-condition of the existing *objectDetectionService* instance and is satisfied immediately. The resulting proof is illustrated in Figure 8(a).

There are several advantages to this technique. 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 services 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 services already exist, a query will automatically reuse services that were instantiated in response to other queries. If two users run queries that can both be answered with an object detection service running over three break beam sensors, the service will only be instantiated in response to the first query; the second query will simply reuse the existing services. When the first query terminates, the execution engine removes only those services upon which no other services 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.

The fourth and most important reason for instantiating virtual representations of services during composition is to ensure proper *flow* of event streams, ie. that all event streams originate at a single service. This requires that the query processor reason about the entire existing service graph, which is not possible with a pure backward-chaining approach. For example, the following query is identical to the query above except that the order of the predicates is reversed.

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

If inference engine were to use pure backward-chaining, it could prove the first predicate in the query with any service that has an event stream as a post-condition. In this case, it would initially try the first service listed in the KB, eg. the `magnetometerService`. When the query proves the second predicate, it does not match any post-condition of `magnetometerService` so it matches the predicate with a service in the KB and completes the proof. The resulting proof is shown in Figure 8(b), 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. By creating a virtual representation of each service in the KB, we allow the inference engine to check the entire service graph to verify legal flow after each inference step. If flow is not legal, the inference engine backtracks and tries the next legal step.

6.2 Comparing to Previous Automatic Service Composition Approaches

Our approach 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 [10, 4]. This technique does not easily transfer to semantic services 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). These techniques assume an initial state of the world s_0 and can allow a set of simultaneous actions to take place at time t_i if the state of the world at that time s_i satisfies all of the actions' preconditions. The next state of the world s_{i+1} is the combination of the previous state and the post-conditions of all executed actions. Several studies have used planning techniques for auto-composition of Web Services [14, 17]. The problem with this technique 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, which uses a set of facts in a knowledge base (KB) along with a set of rules to prove a statement. SWORD [12] 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. For example, a address directory service may be described by the rule:

```
person (X),
name (X, N)=> address(X, A), city(X, C)
```

While an internet mapping service that can provide the directions between two places may be described as

```
address(X, XA), city(X, XC),
address(Y, YA), city(Y, YC) =>
  directions(X,Y)
```

These services can be automatically composed to "prove" a query that asks for driving directions between two places, e.g. `directions(X,Y)`, given only the names of two people. The proof itself represents the workflow with which the services should be executed in order to satisfy the query.

SWORD is most similar to our approach. However, the problem with the pure inference-based approach is that all proofs are tree-based while most service graphs are general directed graphs. Because SWORD does not use virtual representations of services during service composition, it cannot accurately represent the flow of event streams, which must always originate at a single service. Moreover, it cannot represent a service graph with mutual dependence.

6.3 Putting It All Together

We revisit our example in section 3 and demonstrate how the system can 1) automatically share and reuse resource between independent users and 2) compose services from two different applications to create a new semantic composition for a third application.

If Pat and Alex are the first users of the system or if existing services do not satisfy their queries, they may need to write services on their own. We are assuming that only the services described in Figures 3 and 4 are available to the Semantic Streams framework.

Each user is presented with a graphical user interface such as the one shown in Figure 9. The interface also shows a 3D rendering of each sensor in our garage testbed and the region that the sensor covers. The post-conditions of all services in the system are listed on the left side of the screen. These post-conditions 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),
```

When Pat's query is executed, the system generates the service graph exactly as shown in Figure 3, which is reproduced in Figure 10(a) for convenience. When Alex's query is executed, a new `histogramService` is first instantiated. However, it does not instantiate a `magVehicleDetectionService` as shown in Figure 4 because another equivalent service already exists. It uses instead the `vehicleDetectionService` instantiated for Pat's application. The resulting composite service graph is shown in Figure 10(b). Alex's application illustrates Semantic Streams automatically sharing resources between independent users.

Kim's query reuses services from both Pat's and Alex's applications. The `histogramService` 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 `speedService` from Alex's application, however, can be reused because it is inferring the speeds of vehicle objects. Kim's application illustrates how a new application can be created without creating any new services; existing services from the other two applications were composed to create a semantically new application. The service graph in Figure 10(c) is then sent to the service embedding engine and get executed on the sensor network.

7 Limitations and Future Work

Like any programming paradigm, the Semantic Streams framework has its strengths and limitations. We discuss some limitations and possible improvements in this section.

7.1 Non-semantics Operations

The semantic services programming abstraction is meant to capture operations that change the semantics of the input and output streams. This abstraction allows the composition of semantic values. However, not all sensor network problems are semantic transformations. For example, routing data from one node to another does not change the semantics of the data. Even if we add the current location where the data is cached to its semantic properties, semantic services cannot easily differentiate between different routing algorithms. In places where fine-grained operational control is of high priority, an imperative programming abstraction like NesC modules would be more appropriate.

7.2 Reasoning About Runtime

A main limitation of the Semantic Streams framework is that the query processor cannot reason about runtime. All runtime processing is expected to be contained within a semantic service and all values outside of a semantic service are time invariant. This becomes a problem with the service graph needs to change at runtime. For example, two applications cannot both have control of the same pan/tilt camera because they might need to point it in different directions. Our inference engine would not allow these two applications to be run simultaneously. However, it may be that the two applications never need to use the camera at the same time, e.g. when vehicles are present one application needs the camera and when they are not present the other application does. While the query processor does currently have enough information to infer this fact, the current implementation does not have the reasoning capability.

One possible solution is to embrace the query processor at runtime. At each step, the query processor can be re-run based on the current information. Obviously, this introduces significant overhead. A more interesting approach is to enhance the query processor to generate plan skeletons rather then concrete plans. The skeleton can be parameterized by time and information obtained at run time. This way, the plans can be efficiently re-instantiated without going through the entire planning process.

7.3 Quantifiers and Scoping

The semantic markup language described in Section 5 is sufficient for the simple examples used in this paper. However, a main limitation to capturing more complex applications is that the language lacks *quantifiers* and *scoping*. For example, we should be able to add parameters to arbitrary services through CLP(R) constraints without actually changing the service implementation, e.g. adding a speed threshold to the camera capture service:

```
service( cameraCaptureService,
needs(
    stream(S),
    property(S,V,speed),
    {V>=15} ),
creates(
    stream(X),
    property(X,F, photo),
    property(X,S, triggerStream) ) ).
```

The reason this constraint cannot be added is that it has global scope; it cannot be differentiated from a case where the same constraint was placed on the input or output of a different service that processes the same stream.

Quantifiers are also necessary for more sophisticated reasoning at planning-time, such as the use of an ontology. Adding a simple ontological rule such as "all vehicles are objects" could be useful.

```
isa(X,object) :- isa(X,vehicle)
```

This rule would allow the query `stream(X),isa(X,object)` to be satisfied by the `vehicleDetectionService`. However, the meaning of this service is not the same under this new context; it detects all vehicles in a region but it only detects some of the objects in a region. This relationship cannot be concisely captured without quantifiers.

7.4 Actionable Error Messages

If the existing sensors and services are not adequate for a particular query, the current implementation of the query processor simply returns failure. However, because we have a goal-oriented query from the user, we should be able to provide actionable error messages. For example, the query processor could provide suggestions like: "To answer this query, you can add a magnetometer sensor to region XYZ." This functionality is possible by allowing the query processor to analyze the failure points in a failed query and present the unproven pre-conditions to the user. The main challenge with this technique is that there may be thousands of unproven pre-conditions and the system must identify those that would be easiest for the user to satisfy. This task may prove feasible if the feedback to the user is limited to, for example, sensor placement.

8 Conclusions

Semantic Streams is a step toward providing a high level abstraction for end users to interact with sensor networks and enabling transparent in-network processing and component reuse. In many domains, the users only need to specify the end goal in terms of what semantic information to collect, and the service composition framework automatically specifies and glues the necessary components to achieve that goal.

The framework presented in this paper provides a declarative language for describing and composing event-based sensor services. There are several benefits to this framework:

- Declarative programming is easier to understand than low-level, distributed programming and allows common people to query high-level information from sensor networks.
- 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.
- 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 service-oriented architecture let non-technical users to quickly extract semantic information from raw sensor data, thus addressing one of the most significant barriers to widespread adoption today.

References

[1] P. Blackburn, J. Bos, and K. Striegnitz. *Learn Prolog Now!* 2001. On-line book: http://www.coli.uni-saarland.de/ kris/learn-prolog-now/.

[2] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. *Lecture Notes in Computer Science*, 1987:3–14, 2001.

[3] L. Cabral, J. Domingue, E. Motta, T. Payne, and F. Hakimpour. Approaches to semantic web services: An overview and comparisons. *Lecture Notes in Computer Science*, 3053:225–239, 2004. Proceedings First European Semantic Web Symposium (ESWS2004), Heraklion, Crete, Greece.

[4] M. Carman, L. Serafini, and P. Traverso. Web service composition as planning. In *Proceeding of ICAPS 2003 Workshop on Planning for Web Services, Trento, Italy*, June 2003.

[5] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Programming Language Design and Implementation (PLDI)*, June 2003.

[6] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. Irisnet: An architecture for a world-wide sensor web. *IEEE Pervasive Computing*, 2(4):22–33, 2003.

[7] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking (MobiCOM '00), Boston, MA*, pages 56–67, Augest 2000.

[8] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *OSDI*, December 2002.

[9] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. *ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, September 2002.

[10] S. McIlraith and T. C. Son. Adapting golog for composition of semantic web services. In *Proceeding of 8th International Conference on Principles of Knowledge Representation and Reasoning (KRR'02), Toulouse, France*, pages 482–496, April 2002.

[11] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. In *Proceedings of the First International Workshop on Data Management for Sensor Networks (DMSN), Toronto, Canada*, Augest 2004.

[12] S. R. Ponnekanti and A. Fox. Sword: A developer toolkit for web service composition. In *Proceeding of The Eleventh World Wide Web Conference, Honolulu, Hawaii*, May 2002.

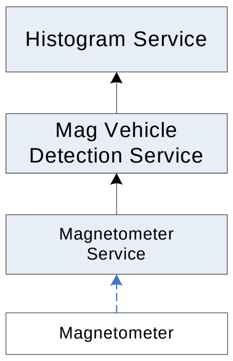
[13] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (Second Edition)*. Prentice Hall, 2004.

[14] M. Sheshagiri, M. desJardins, and T. Finin. A planner for composing services described in daml-s. In *Proceeding of ICAPS 2003 Workshop on Planning for Web Services, Trento, Italy*, June 2003.

[15] SICS AB. SICStus Prolog 3.12.0 user's manual, 2004. http://www.sics.se/isl/sicstuswww/site/documentation.html.

[16] G. Simon, M. Maroti, A. Ledeczi, G. Balogh, B. Kusy, A. Nadas, G. Pap, J. Sallai, and K. Frampton. Sensor network-based countersniper system. In *Proceedings of the 2nd international conference on Embedded networked sensor systems (SenSys04), Baltimore, MD*, pages 1–12, November 2004.

[17] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau. HTN planning for web service composition using SHOP2. *Journal of Web Semantics*, 1(4):377–396, 2004.



Magnetometer Service

Function: A wrapper service around the magnetometer sensor.

Inputs: None.

Outputs: A stream of *magnetometer* events with a single property indicating the magnetic field in the region.

Mag Vehicle Detection Service

Function: Analyzes the magnetometer stream to infer the presence of vehicles.

Inputs: A magnetometer stream

Outputs: An *object* stream, where each object event has *time* and *region* properties indicating where and when it was detected as well as a property indicating that it is a vehicle.

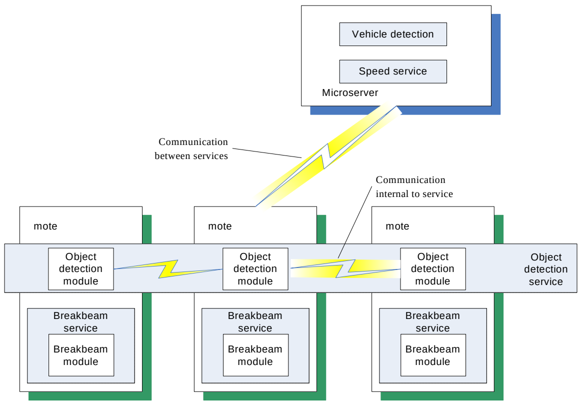
Histogram Service

Function: Plots the time properties of an event stream as a histogram.

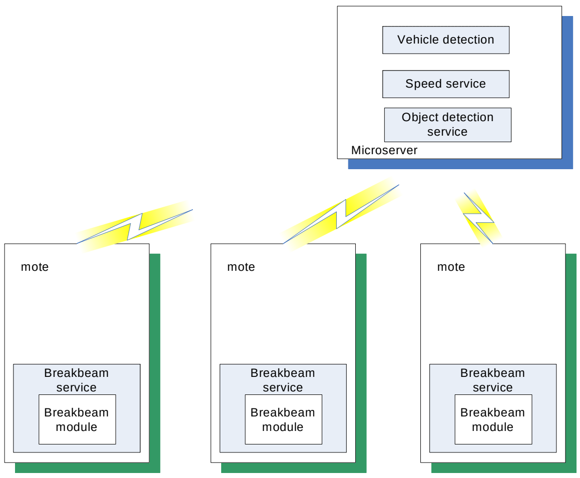
Inputs: An object stream with a vehicle property

Outputs: An *histogram* stream, where each event contains an update to the histogram.

Figure 4: **Alex's Application** *requires services that read the magnetometer sensors, detect objects, and plot their speeds in a histogram. a) shows the service composition and b) provides descriptions of the services*



(a) Distributed Implementation



(b) Centralized Implementation

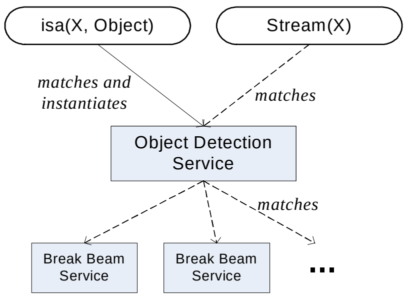
Figure 5: **Execution Model** *Services can be distributed objects or can implemented on a central server. With sensor infrastructure, the centralized implementation is often both more efficient and easier.*

```
service( breakBeamService,
   needs(
      sensor(breakBeam, R),
   creates(
      stream(X),
      isa(X,break),
      property(X,T,time),
      property(X,R,region) ) ).

service( objectDetectionService,
   needs(
      breakGroup(R, [], Group),
      length(Group,Length),
      Length>=3 ),
   creates(
      stream(X),
      isa(X,object),
      property(X,Group, support),           property(X,T,time),
      property(X,R,region) ) ).

service( speedService,
   needs(
      stream(X),
      isa(X,object),
      property(X,Group, support) ),        creates(
      stream(X),
      isa(X,object),
      property(X,S,speed) ) ).

service( vehicleDetectionService,
   needs(
      stream(X),
      isa(X,object),
      property(X,S,speed) ),
   creates(
      stream(X),
      isa(X,vehicle) ) ).

service( cameraCaptureService,
   needs(
      stream(S),
      property(S,V,P) ),
   creates(
      stream(X),
      property(X,F, photo),           property(X,S, triggerStream),
      property(X,P, triggerProperty))).
```

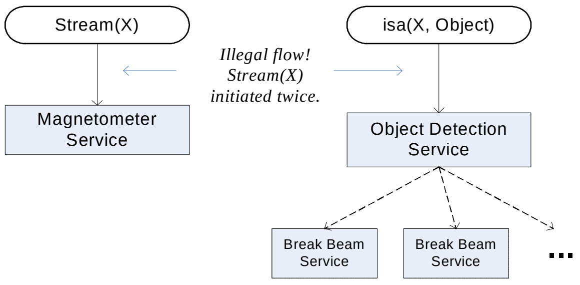
Figure 6: **Pat's Service Markup** *for the services shown in Figure 3*

```
service( magnetometerService,
    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) ) ).

service( magVehicleDetectionService,
    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) ) ).

service( histogramService,
    needs(
        stream(S),
        property(S,V,P),
    creates(
        stream(Y),
        isa(Y,histogram),
        property(Y,S, plottedStream) ) ).           property(Y,P,
plottedProperty) ) ).
```

Figure 7: **Alex's Service Markup** *for the services shown in Figure 4*

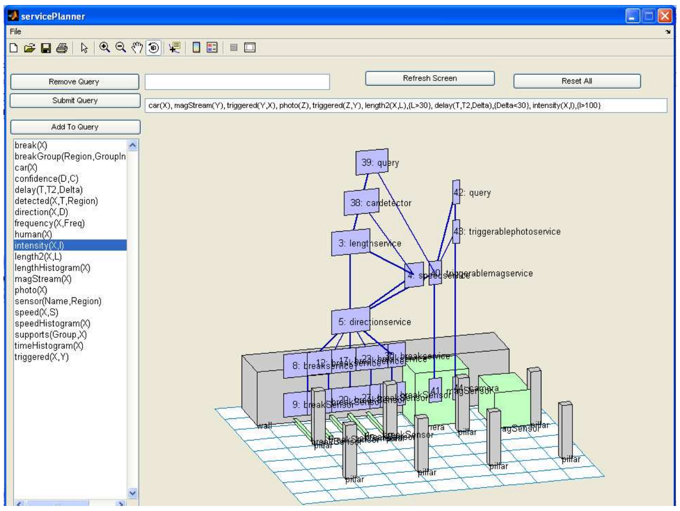


(a) Modified Inference



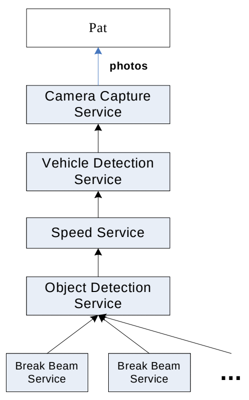
(b) Backward-Chaining

Figure 8: **Service Composition** *The backward chaining algorithm must be slightly modified in order to yield valid service graphs; pure backward-chaining cannot guaranteed valid* flow*.*

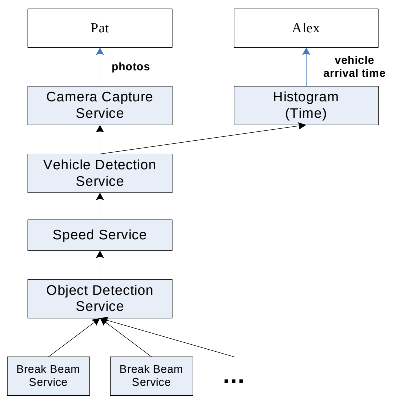


car(X), magStream(Y), triggered(Y,X), photo(Z), triggered(Z,Y), length2(X,L),{L>30}, delay(T,T2,Delta),{Delta<30}, intensity(X,I),{I>100}

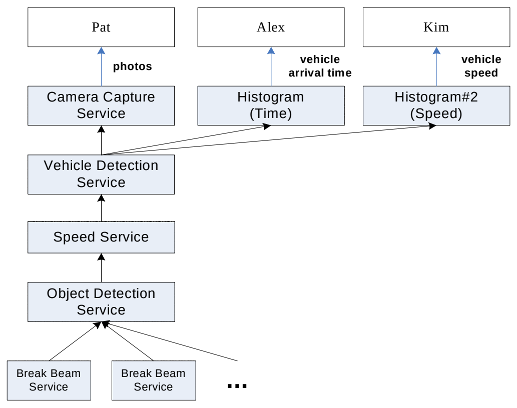
Figure 9: **User Interface** *Each user is presented with a 3D rendering of the sensors in the testbed and, on the left, all predicates that are queryable.*



(a) Step 1



(b) Step 2



(c) Step 3

Figure 10: **Composite Service Graphs** *In step 1, Pat's query produces the expected service graph. In step 2, Alex's query reuses one of the services that is instantiated in response to Pat's query. In step 3, Kim's query composes services from Alex's and Pat's queries to create a new semantic composition.*