Poster Abstract: Automatic Programming with Semantic Streams

Kamin Whitehouse
Computer Science Division
UC Berkeley
Berkeley, CA 94720
kamin@cs.berkeley.edu

Feng Zhao
Microsoft Research
One Microsoft Way
Redmond, WA 98052
zhao@microsoft.com

Jie Liu
Microsoft Research
One Microsoft Way
Redmond, WA 98052
liuj@microsoft.com

Categories and Subject Descriptors

D.1.2 [**PROGRAMMING TECHNIQUES**]: Automatic Programming

General Terms

Language, Design

Keywords

Sensor network, declarative query, service architecture

1. INTRODUCTION

One of the primary barriers to the widespread use of sensor networks by non-technical users today is the inability to automatically synthesize semantic values such as building activity from raw sensor data from for example intrared breakbeams or motion detectors. This paper presents a framework called *Semantic Streams* that allows users to interact with sensor networks with declarative statements 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 [1]. 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 greater than 90%," or "I want to minimize the number of messages."

Our framework uses a *semantic services* programming model [2], where each service is an inference unit that infers semantic information about the world and incorporates it into an *event stream*. Each service has a first-order logic 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.

2. SERVICE AND QUERY LANGUAGES

In our service markup and query language, sensors and services are declared using the special predicates:

- **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]]).

defines a magnetometer that covers a 3D cube defined by a pair of $[x, y, z]$ coordinates.

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, a Vehicle Detector could be described as a service that uses a magnetometer 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))).
```

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. 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 processor will generate all such possible service compositions. To constrain the resulting composition set, one could simply add more predicates to the query. For example,

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

The query processor employs an inference engine to decide which sensors and services will provide the semantic information that the user requires. Using the *pre-conditions* and *post-conditions* of the services, the inference engine performs 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.

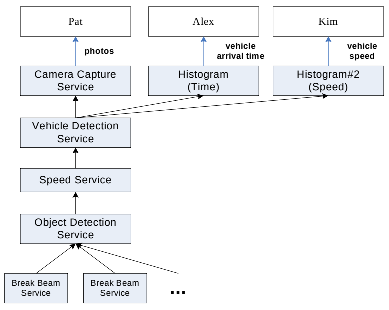

Figure 1: A composed service graph that satisfies Pat's, Alex's, and Kim's queries simultaneously.

As a concrete example, we describe how Semantic Streams is used to automatically synthesize semantic services for three uncoordinated queries through a sensor network that we deployed in a parking garage, 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. The applications are

- 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 garage;
- 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,

with the corresponding queries as

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 a service graph that uses break beam sensors to detect vehicles and to estimate their speed. For Alex's query, a new `histogramService` is first instantiated. However, it does not instantiate another vehicle detection service using e.g. magnetometer, because Pat's application already provides data with equivalent semantics. 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 2 is then sent to the service embedding engine and get executed on the sensor network.

3. DISCUSSIONS

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, possibly from entirely different organizations. Sensor infrastructures pose several important problems such as sharing resources between independent applications, resolving conflicts between separate users, 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 should be above 90%, and latency less than 50 milliseconds." The query engine propagates these constraints through the components in the service graph. 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.

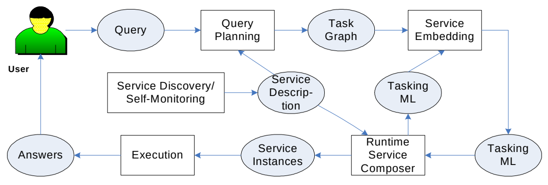

Figure 2: Planning and Execution

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 2. 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.

4. REFERENCES

[1] 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.

[2] K. Whitehouse, F. Zhao, and J. Liu. Semantic Streams: a framework for declarative queries and automatic data interpretation. Technical Report MSR-TR-2005-45, Microsoft Research, 1 Microsoft Way, Redmond, WA 98052, April 2005.