Semantics-Based Optimization Across Uncoordinated Tasks in Networked Embedded Systems

Jie Liu
liuj@microsoft.com
Elaine Cheong
celaine@eecs.berkeley.edu
Feng Zhao
zhao@microsoft.com

April 2005
Technical Report
MSR-TR-2005-46

Microservers are networked embedded devices that accept user tasks on demand and execute them on real world information collected by sensors. Sharing intermediate sensing and computing results among these tasks is critical for optimal resource utilization. This paper presents a service-oriented microserver runtime — SERUN and its semantics-based task management design. Event semantics checking and conversion are based on a signal type system (STS) that captures both data values and service triggering. Based on the compatibility of event semantics, redundant computations in uncoordinated tasks are removed. A prototype of SERUN has been experimented in a parking garage sensor network executing three uncoordinated user queries.

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

1 Introduction

Technology advances have greatly changed the landscape of embedded systems. Today, an embedded system may no longer be limited to a single application. Embedded software is no longer developed once for the entire lifetime of a product. The ubiquity of (wireless) networking has opened the gate for embedded systems to participate in the larger digital world. They can be re-tasked, re-configured, and reprogrammed on the fly. They can execute user defined tasks on demand.

In this paper, we consider a particular class of networked embedded devices called *microservers*, which can gather physical information through sensors and can respond to queries sent over the network. Like application servers in enterprise computing, microservers dynamically accept and host user tasks that define the application logic. But, unlike business applications, tasks on microservers are typically long-running and their inputs are physical events and sensor data in real time.

Microservers can take various forms. For example, in a healthcare scenario, the cellphone an elderly person carries can be a microserver. It can accept tasks from family members and healthcare providers sent through the cellular data link, and run them over real-time data gathered from wearable sensors (e.g. location sensing, activity sensing, and vital sign monitoring). The results can go back to the family members and caregivers wirelessly. A telematics computer in a car can be a microserver. The car manufactures, the dealer, and the car maintenance shops may inject different monitoring tasks into the microserver to gather specific events tailored to their interests. The results of those tasks are sent via a cellphone or a satellite link. In a retail warehouse, portable RFID readers with WiFi links, possibly equipped with location, temperature, and humidity sensors, can be microservers that track products and their storage environments on demand. In *ad hoc* sensor networks, a mesh network of microservers can serve as gateway nodes to connect resource constrained embedded sensors to larger network/data infrastructures.

A common challenge for these microservers is that they must host *uncoordinated tasks*, such as user queries, simultaneously. By uncoordinated tasks, we mean that the tasks are injected by different users at unpredictable times. These tasks may have different life time depending on users' interests. Since microservers are not general-purpose computers, the information they collect and process depends highly on their surroundings. It is very likely that there are partial overlaps among simultaneous user tasks. For example, in the healthcare scenario, the caregiver may want to detect certain walking patterns as an early sign of Alzheimer disease. The daughter of the elderly person, seeing that the weather today is nice, wants to set up a reminder that if her parent has not gone out for a walk until 10AM, she will be notified so that she can give her parent a call. The two tasks are independent and have very different time scales — one is long running and the other expires by 10AM. However, parts of the two tasks, namely detecting the behavior of "walking" from various raw sensor data (e.g. from accelerometers or cameras), can be shared.

Many microservers are battery powered mobile devices, they are constrained by the CPU speed, memory size, communication bandwidth, and energy storage. Thus, it is crutial for microservers to find the maximum amount of overlapping sensing and computation in uncoordinated tasks and reduce runtime redundancy. This is an *in-*

termediate information reuse problem, that is, whether a task can reuse intermediate computation results from other tasks. In order to achieve this, a microserver runtime system must:

I. identify overlapping sensing and computation from multiple tasks;
II. suppress parts of a task but keep the rest active;
III. share intermediate results from one task to another.

In this paper, we describe the architecture and task management design of SERUN [1] with an emphasis on enabling intermediate information reuse. SERUN has a component-based architecture. Each task is built using a set of event-driven components, called *services*. The communication between services has a publish/subscribe semantics. A user task is sent to a microserver as a service composition graph (SCG). For each incoming user request, the runtime system examines the existing tasks and tries to find out whether parts of the new request can be fulfilled by existing computations. If so, the redundant services are not instantiated, and their downstream services are subscribed to corresponding existing publishers.

There are several approaches to promote the sharing of intermediate results across tasks. For example, some commonly used services can be manually started and their outputs published in a local tuple space. New tasks can then subscribe to these intermediate results, so that they are computed only once. However, this introduces run-time overhead if those services are not used by any tasks. It is also hard to draw the line between what services should be system provided vs. user defined. Another approach is to compare syntactically the composition of services. This is the most conservative and rigid approach. Two subtasks are considered redundant if they are exactly the same, including service composition topology, service parameters, and their internal states. For example, if task *A* uses a service `vehicleSpeedDetection` to compute both the presence and the speed of a vehicle, and task *B* plans to use `vehicleDetection` to obtain only the vehicle presence information, then *B* cannot use the outputs from `vehicleSpeedDetection` even though its outputs contain all the information *B* needs.

SERUN takes a different and more flexible approach by allowing users to annotate semantics of the data transmitted between services. We rely on a runtime signal type system (STS) to check and automatically convert between data semantics whenever possible. In fact, the input task may not be followed verbatim, if the runtime system can find existing alternative services that provides intermediate results with compatible semantics. This is more sophisticated than simply giving each piece of data a name and matching names at runtime . Since services are event driven, the events passing between services not only carry their value information, but also serve as triggers for service execution. For example, a service that expects temperature sampled at 10Hz cannot be triggered by a 100Hz event source. A big advantage of STS is that it handles this sequencing property naturally within the type system and performs trigger rate conversions as well as data type conversions.

The rest of the paper is organized as following. In Section 2 we present the architecture of SERUN focusing on the features that enables information reuse. We then focus on the signal type system in Section 3, defining signal types and show how

[1] SERUN stands for SErvice-oriented RUNtime.

type checking and type conversion mechanisms can help identify and reduce redundant computation. Section 4 presents a testbed deployment of the runtime system in a parking garage microserver. Section 5 discusses related work. Section 6 concludes the paper and points out future directions.

2 SERUN Architecture

2.1 Service and Composition

In order to facilitate the reuse of intermediate computation results, we must introduce some granularities in build user tasks. In SERUN, an unbreakable piece of computation is called a *service*. A service is an asynchronous piece of computation. Services have input and output *ports*. Input ports accept events, and output ports produce events for other services. A service may have internal state, and its behavior depends on both input events and its internal state. To this extent, services are similar to actors in Ptolemy II [4], element classes in Click [17], actions in UML's action semantics extension [19], among many component-based frameworks.

There is no global state in a task other than the data communicated between services. The communication has a publish/subscribe semantics. Conceptually, all outputs from services go into an event mediator that is visible to all other services. If one of the services is interested in processing an event in the mediator, it *reacts* to a copy of that event. Events are not cached. After it is delivered to every subsciber's input port, it is garbage collected. Since the mediator separates the publishers and subscribers, a service does not care where its inputs come from nor where its outputs go to. Thus intermediate computation results can be easily shared across tasks.

Since all services are local to the microserver, the publish/subscribe semantics is efficiently implemented using an event/delegate mechanism [15] in SERUN. Objects called Relations[2] are introduced to serve as mediators for publishers and subscribers. However, instead of having a single mediator for all publishers and subscribers, there is one relation per event type. The relation connects to one or more output ports (publishers) and zero or more input ports (subscribers). It maintains a list of all connected input ports, and registers itself to every connected output ports as an event handler. Once an event is sent by an output port, the relation sends a copy of the event to every connected input port. The connections between ports and relations are established only once when a user task is first injected to the microserver and remain unchanged throughout the lifetime of the task. Since tasks on microservers are usually long-running, this implementation greatly reduces the overhead of data pattern matching in typical publish/subscribe architectures [6].

In SERUN, service executions are event-driven. Some events may carry time stamps, but service executions are triggered by the presence of events in the input ports. This kind of reactive execution is usually more resource efficient when the inputs are sparse, which is the case for many embedded applications such as wireless sensor networks. In SERUN, each service has its own thread. It reacts to every input event on a first-come-first-serve basis. Once triggered, it performs a finite piece of computation,

[2]The name is influenced by Ptolemy II.

which may change its internal state and may produce outputs, and then go to an inactive mode waiting for the next input event. An input port has an FIFO queue that keeps triggering the inactive service if the queue is not empty.

2.2 Task Management

A user task consists of a set of services and the connections among their ports, called a service composition graph (SCG). It is given to SERUN as an XML document. For most of the paper we use a graphical representation as shown in Figure 1. Services are shown as blocks with their names annotated. Service ports are implicit. Connections between ports are indicated by arrows. A relation is implicit if it connects exactly one output port and one input port, otherwise, it is shown as a diamond.

Once given to the runtime system, the SCG may not be followed verbatim. It can be merged with existing tasks. Figure 1 illustrates the desired optimization result. If `Task1` is already running on a microserver, and `Task2` is later injected, we would like the run time image to be the bottom part of the Figure 1. When `Task1` has finished, not all its elements are garbage collected. Part of it, although may have been started by `Task1`, can be used by other active tasks, as show in Figure 1(b).

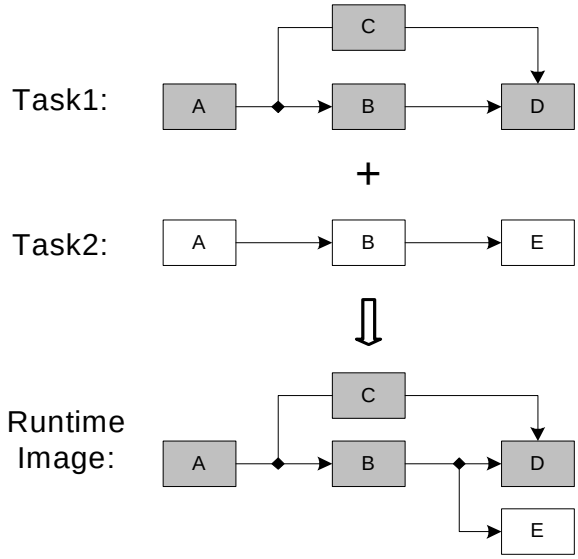
Service lifetime management is achieved by *demand analysis*. Each service is individually started and stopped. A service maintains a list of tasks that demand it. When a service subscribes to a relation, the subscriber's task is propagated to all services backward-reachable from the relation. That is, all services that are used to generate the data that the subscriber needs is part of the task. When a task terminates, its corresponding entry is removed from all services it demands. When the list is empty, the service wraps up and hands itself to the garbage collector. Thus, requirements [II] and [III] introduced in Section 1 can be relatively easy to achieve in our service-oriented architecture. The rest of the paper focuses on achieving requirement [I] — how to maximally identify redundant sensing and computation in an efficient way.

3 A Signal Type System for Information Reuse

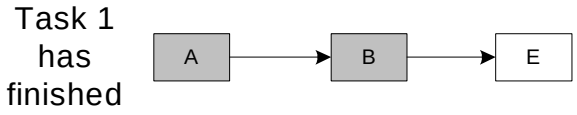
The approach we take to reducing runtime redundancies is based on event semantics. As stated before, the semantics of events in event-driven systems has two parts: the value it carries and the triggering role it plays for services. When reusing events, we must consider both properties. The mechanism we capture and reason about event semantics is a signal type system (STS). The goal of the STS is to capture the property that events produced from an output port "contain all the information needed" by an input port. This section first defines the singal types and then describes how the STS is used to help information reuse.

3.1 Signal types

Inspired by the tagged signal model [14], we define an event as a pair: a tag and a value. A signal is simply a sequence of events. Thus, a signal type consists of two parts: its tag type and its value type.



(a) Merging two overlapping tasks.



(b) Garbage collecting partial tasks.

Figure 1: Task management in SERUN.

Value types: In STS, we treat the values of an event as a record, and its type is a tuple:

$$v = (name, \{(n_1, t_1), (n_2, t_2), ..., (n_k, t_k)\}) \tag{1}$$

where,

- *name* represents the name of the signal;
- (n_i, t_i) is called a *field type*, where n_i is the name of the field and t_i is a primitive data type[3].

A name in STS serves as an identifier for data. It can encode the ID, location, or object identity information, among others. For example, the outputs of a relative humidity(RH)/temperature sensor in room 102 may have value type:

```
("room102", {("RH", float),  ("temp", float)})
```
(2)

We use the set V to represent all value types. The compatibility relation between value types is a simple extension of record types in programming languages (see e.g.

[3]Here, we use the term primitive types loosely. It represents both primitive data types, like *int* and *double*, and objected-oriented classes as well.

[16]). Let $v = (name, \{(n_1,t_1),(n_2,t_2),...,(n_k,t_k)\})$ and $v' = (name', \{(n'_1,t'_1),(n'_2,t'_2),...,(n'_m,t'_m)\}) \in V$, we say v is *compatible* with v', written as $v \leq v'$ if the following holds:

- $name = name'$;
- for each (n',t') of v', there exists (n,t) of v, such that $n' = n'$ and $t \leq t'$. By $t \leq t'$ we mean t is a subclass of t', or for primitive types converting t into t' will not lose data precision.

That is, the fields in v' need to be a subset of that in v subject to primitive type compatibility. For example, if a service requests a temperature reading in room 102 as `("room102", {("temp", double)})`, then the type in (2) is compatible with it.

Tag types: Tags represent timing and ordering relations among the events in a signal. To deal with real world signals, especially sensor outputs, we assume tags take values from $\mathcal{R}$, the set of reals. In particular, we focus on the following subsets of $\mathcal{R}$

- $\mathcal{R}$ is the whole connected set.
- $\mathcal{N} \subset \mathcal{R}$ is the set of natural numbers.
- $\mathcal{D} \subset \mathcal{R}$ is *discrete* if it can be order-preserving and bijectively mapped to a subset of integers [12]. We denote this map $\mathcal{M}_{\mathcal{D}}$, which is unique for every $\mathcal{D}$.
- $\mathcal{P}(t_0,T_s) = \{t | t = t_0 + i \cdot T_s, i \in \mathcal{N}, \text{ and } t_0, T_s \in \mathcal{R}\}$ is the set of integer multiple of T_s starting from t_0. This is a periodic discrete set. We also write $\mathcal{P}(T_s)$ if the start time is understood, say $t_0 = 0$.

The tag sets form the basis of tag types. However, they alone are not sufficient to differentiate sampled continuous signals from periodic discrete event signals such as a clock. For this reason, we extend the tag type system to capture the notion of continuity of underlying signals. We introduce a class of signals called *discrete representation of continuous signals* (DRCSs). A DRCS has a discrete set as its tags, but represents a continuous signal. One implication of the underlying continuity is that we can approximate data values (e.g. through interpolation) even though they are not present in the signal. DRCS is different from timed discrete event signals (DES) even though they may have the same tag set, because DES cannot be interpolated.

We further introduce a base type called *continuous-time signal* (CTS), which can never be instantiated in a digital computer, but serve as the bottom of our type lattice. The CTS is the only signal that has tag type $\mathcal{R}$. We call the untimed signals *sequences*, and their tags (in terms of timing) are $\mathcal{N}$. With these notions, we define the following *coarse-grained* signal types lattice, as shown in Figure 2. Intuitively, a CTS can be sampled to get a DRCS and can generate DESs through event detections. A DRCS can be treated as discrete events by losing the notion of underlying signal continuity. A DES can be converted to a sequence by applying $\mathcal{M}_{\mathcal{D}}$, i.e. losing its timing properties.

The coarse-grained type lattice can be further refined by the containment relations among tag sets, for example, $\mathcal{P}(T_s) \subset \mathcal{P}(2T_s) \subset \mathcal{P}(6T_s)$.... In general, we use C for DRCS tags and D for DES tags. Let $C(p)$ represent a DRCS whose tag set is $\mathcal{P}(p)$, and $D(p)$ be a discrete event signal with the same periodic tags. The fine-grained tag type lattice, denoted as $\mathcal{T}$ is shown in Figure 3, where C_1, D_1, D_2 are aperiodic and C_1 and D_1 have the same tag set. $\mathcal{T}$ is an infinite lattice.

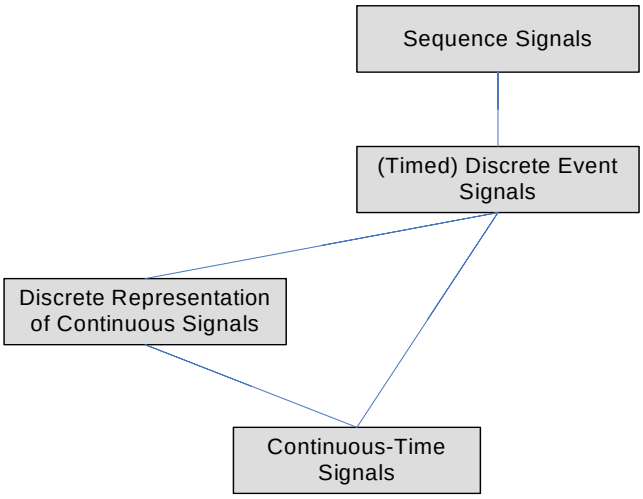


Figure 2: A coarse-grained signal type system based on signal continuity.

Using this lattice, we define compatibility on tag types: let $\tau_1, \tau_2 \in \mathcal{T}$, $\tau_1 \leq \tau_2$ if τ_1 is lower in the lattice than τ_2. For example, according to this type lattice, a continuous-time temperature waveform is compatible with its 1Hz sampling, which is compatible with its 10Hz sampling, which is compatible with a discrete set of temperature events defined on the same time instances, which in turn is compatible with untimed temperature sequences.

We call the overall type system that captures both tag types and value types the *signal type system* (STS). In STS, $s = (\tau, v)$ is compatible with $s' = (\tau', v')$ (i.e. $s \leq s'$) if $\tau \leq \tau'$ and $v \leq v'$.

3.2 Using STS in SERUN

STS is used in SERUN for checking information reusability. Intuitively, if an input port I of service A requests events of type (τ', v'), and an (existing) output port O produces events of compatible type (τ, v), then by connecting I and O, each event A receives will contain all the information A needs for a correct reaction, and A will be triggered *at least* as often as it is expected. For A to be triggered exactly as often as it expects, we must have $\tau = \tau'$.

3.2.1 Signal Types Specification

Task SCGs are sent to microservers in an XML format, called Microserver Tasking Markup Language (MSTML), which is an extension of MoML [4]. A MSTML file has four sections: Sockets, Services, Relations, and Connections. Sockets are interfaces to other microservers or wireless sensors; the services section specifies the services used in this task, including a list of all ports and parameters for each service; the relations section specifies the mediators to connect the ports, and the connections section lists

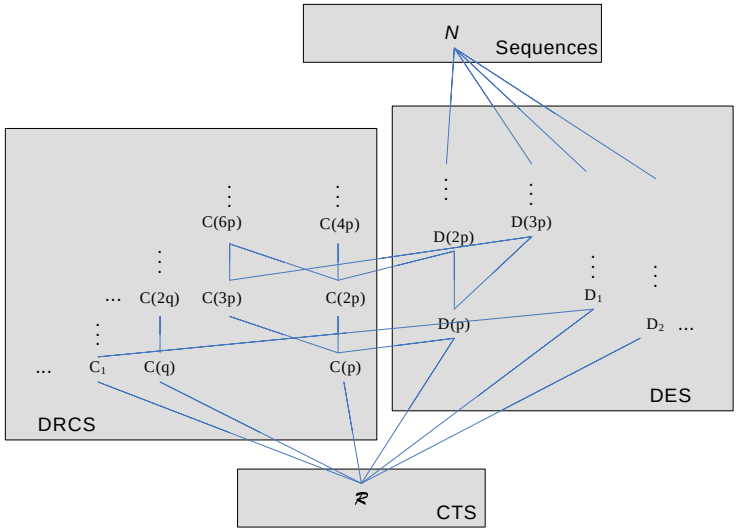


Figure 3: A lattice for tag types. Without loss of generality, we assume $q > p$; q is not a integer multiple of p; C_1, D_1, and D_2 are aperiodic.

which port connects to which relation. The semantics annotation is part of the port declaration in the services section.

Figure 4 shows a segment of the services section in MSTML. Signal types are annotated as properties of ports. For example, the figure shows an instance of `MagVehicleDetection` service that detects the presence of vehicles using magnetometer readings. The service has two ports: "in" and "out." The input port needs a sampled magnetometer signal at 0.1Hz. Each data sample is an integer. The service produces outputs whenever a vehicle is detected. The vehicle detection events are timed discrete events with a field `timeStamp`.

A task can also specify that certain signals should not be substituted by omitting its signal type properties, so that parts of the SCG must be followed verbatim. This feature is important since not all tasks are aimed at getting the final output. For example, one can use the `MagVehicleDetection` service to check whether the magnetometer is working correctly. Replacing it with other vehicle detection mechanisms defeats the purpose.

3.2.2 Type Checking

The goal of reusing services from existing tasks is achieved by type checking the newly injected MSTML specification against existing signals. We assume that each application, when given to SERUN, is already type checked for correctness, and it is self-contained. When an MSTML file is injected, the STS checks for each input port i in the new task and for the collection of output ports O, whether there exists $o \in O$ such that $type(o) \leq type(i)$. This boils down to matching the names of the signals, checking the subset relations among the fields, checking the primary types compatibilities in

```
<service name="Detector" type="MagVehicleDetection">
   <port name="in">
      <property name="input"/>
      <property name="signalType"
         signalName="magnetometer"
         tagType="C(0.1)"
         magneticField="int" />
   </port>
   <port name="out">
      <property name="output"/>
      <property name="signalType"
         signalName="vehicle"
         tagType="D"
         timeStamp="long" />
   </port>
</entity>
```

Figure 4: **Markup of a Detector service in MSTML.** Signal types are annotated as properties of ports.

each field, and most importantly checking the tag type compatibility.

For aperiodic signals, the tag type checking can simply use the coarse-grained type lattice in Figure 2: $C \leq D \leq \mathcal{N}$. For periodic signals, it is possible to perform finer grained checking such as checking the sampling rate. For example, $C(p) \leq C(k \times p)$ for any natural number k, similarly for discrete events.

3.2.3 Type Conversion

Notice that type compatibility does not imply that an input port in one task can connect to an output port in another task. To ensure correct triggering, the tag types must be equal. This is achieved by type conversion of compatible types, based on type checking results. To simplify discussion, we assume that when a service accesses data values from its input events, it always uses the name of the field and cast it to its local type, e.g.

```
int magValue = (int)event.getValue("magneticField");
```

In this section, we focus on the conversion of tag types.

The purpose of type conversion is to provide an input port with exactly what it expects, for both the data values and the triggering times. Analogous to conversions in data type systems, we have developed similar notions of lossless and lossy conversions in STS. When the output port type and the input port type are compatible but not exactly the same, i.e. the input events are a subset of the output events, it is possible for a *lossless* conversion in STS to provide precise events, in terms of both tag and value, to the input port. Here, we only consider conversions of periodic signals.

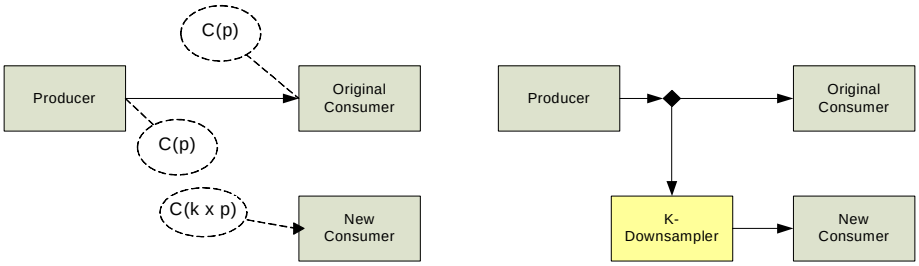


Figure 5: Lossless type conversion in SERUN. Down samplers are automatically inserted to convert tag types.

Let output port o and input port i be type compatible, e.g. $\tau(o) = C(p)$, and $\tau(i) = C(k \times p)$, then the type converter inserts a `k-DownSampler` service, which for every k input events, produces one output event, as shown in Figure 5. Similar conversion can be performed for periodic DESs.

Lossy conversions change event values in order to match tag types. It applies only to DRCSs, taking advantage of underlying continuity of the signal it represents. By using a lossy conversion, the input port will be triggered exactly as requested, but the values of the input events are only approximations to the real values. The accuracy of this approximation depends on the continuity of the underlying signal and the sampling rate at both the output port and the input port. An `Interpolator` service performs lossy conversion. There can be many interpolator services based on different interpolation algorithms. In SERUN, we use the simplest linear interpolator. In order to reduce approximation errors, we perform lossy conversion only when the output signal has a higher sampling rate than the input requirement. Let $\tau(o) = C(p)$, $\tau(i) = C(q)$, and $p < q$. A `LinearInterpolator` starts with an internal counter m set to 0. It is activated by every output event from o. It produces its m-th output when receiving the k-th input, where k satisfies $(k-1)p < mq \leq kp$. The value of the m-th output event is

$$y'_m = y_{k-1} + (y_k - y_{k-1}) \frac{mq - (k-1)p}{p}.$$

where y are the input values and y' are the output values.

Conceptually, an interpolator first converts the DSRC to its underlying continuous-time signal, and then re-samples it according to the frequency required by the downstream input port. Note that the lossy conversion only applies to DRCS but not discrete event signals, even though they may have the same tag set.

3.3 More on signal names

Notice that the compatibility of event values defined so far means that when the service reacts to its input event, it can cast all the expected data fields without runtime exceptions. This does not necessarily mean that the data values are exactly what the

receiver expects when we connect its input to the output of a service in a different task. The only element in STS that ensures the correctness of event values is the name of the signal. But, is this reliable?

For example, a service `counter` in task *A* produces a signal with name `vehicleCount`, and another service in task *B* also expects a signal called `vehicleCount`. Is it correct to connect them together? Although the presence of a vehicle is a physical fact, it is possible that *A* is counting vehicles from 9AM, while *B* expects the counting from 10AM. The encoding "`vehicleCount`" is not a unique identifier for the semantics. On the other hand, names like `vehicleCountFromUTC1-1-2005:09:00:00` and `vehicleCountFromUTC1-1-2005:10:00:00` are able to differentiate them.

In general, in order to uniquely identify events, the names of the signals must be universal and rich. That is, uncoordinated users must first agree on how to name physical and virtual events in the particular application domain, and secondly, the name should reflect runtime information such as time duration, space, units, and accuracy. Very likely, it is a structure rather than a simple string.

Recent movements in semantic web services proposes ontologies for various application domains [3, 2]. In the sensor communities, IEEE 1451.2 [1] and OGC SensorML [20] are both attempts to standardize semantic descriptions of sensor outputs. Currently, these standards do not include high-level information interpretations, but further extensions can be in the scope.

Another approach to prevent users from specifying arbitrary data semantics is to have an automatic service composition engine as the front end for microservers. Instead of having users directly composing services, which would be difficult for nontechnical users, one can provide users with a high-level query interface. Users can specify something like "give me a histogram of vehicle arrival times starting from 9AM today" through a query language; and a query processor generates a SCG with unified signal types annotated. A query language and its processing engine, based on constraint logic programming, has been prototyped for SERUN [24].

4 Example

We built an experimental testbed in a parking garage to prototype our service-oriented networked embedded computing architecture and SERUN. The testbed allows users to run multiple simultaneous queries on real-time parking garage sensor data.

The testbed is located near the entrance of the second floor in a parking garage with one-way traffic, as shown in Figure 6. 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 pass through this area, as do most pedestrians using the elevator.

There are three types of sensors in the system: a web camera, a magnetometer and infrared breakbeam sensors. A breakbeam 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 re-detected. This is the same type of sensor that might be found at a store entrance to detect customers entering and leaving.

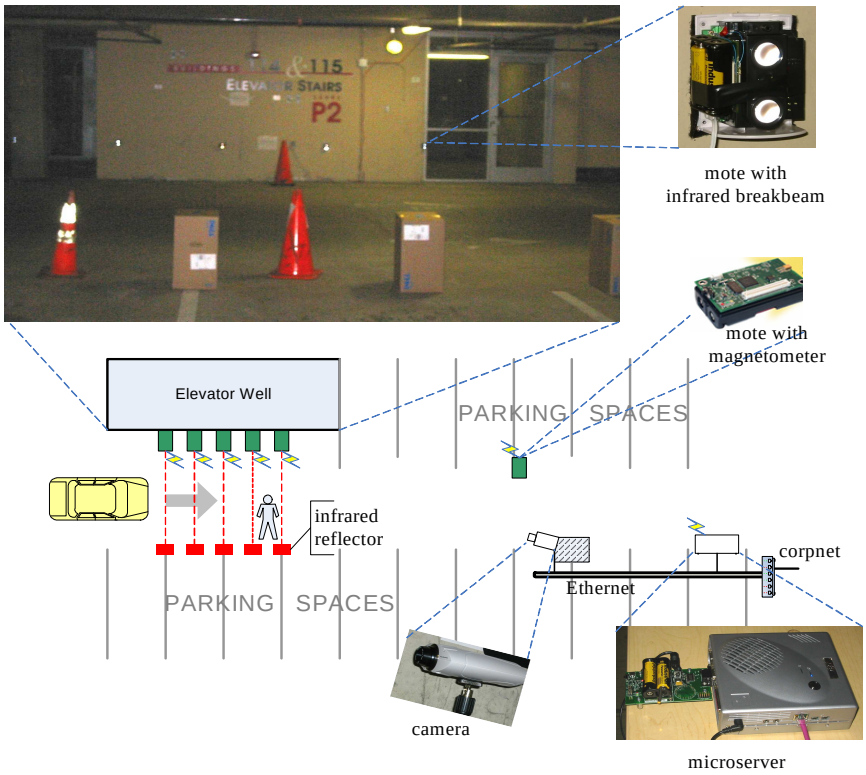


Figure 6: **A garage sensor network deployment.** *The breakbeam sensors were laid out in a row on the wall in the focus area. The digital camera was focused on the same area. The magnetometer was placed several meters downstream near the microserver.*

Both breakbeam sensors and the magnetometer are connected to micaZ motes[4]. Each of these motes is equipped with a 2.4GHz IEEE 802.15.4 (ZigBee) compliant radio. Five infrared breakbeam sensors are placed in a row across the area, 1m apart and about .5m from the ground, such that the beams are broken in succession by any passing human or vehicle. Each blocking or unblocking event generates an interrupt to the mote. Slightly down traffic from the breakbeam sensors, we installed a magnetometer-equipped mote that can detect the changing magnetic field of a moving vehicle. A headless Upont Cappuccino TX-3 Mini PC is used as a microserver that communicates with the micaZ motes via 802.15.4 radio, while connected to the company intranet via wired Ethernet. The web camera, with an embedded web server, is also connected to the Ethernet.

Pretending three corporate users, we run the following three queries in the system:

Task T. Traffic engineer Todd wants photographs of all vehicles moving faster than 25mph.

[4]Available from Crossbow Tech. (www.xbow.com).

Task E. Employee Emma wants to know at what time she should arrive at work in order to get a parking space on the first floor of the parking deck.

Tash C. Corp security officer Cory wants to collect magnetic field signatures whenever there is a moving object (human or vehicles) passing through the section.

There are two ways to detect a vehicle in our system: by using the breakbeam sensor array or by using the magnetometer. The breakbeam sensors can estimate the speed of the vehicle, while the magnetometer cannot. To detect a vehicle, Task E only needs to sample the magnetometer at 16Hz, while to collect the magnetic field signature, the magnetometer needs to be sampled at 256Hz.

As a prototype, we use Ptolemy II as a graphical interface for users to build their tasks and annotate data semantics. Each user builds the task from his/her own standpoint, without considering possible resource sharing. For example, Todd's must detect the vehicles using the breakbeam array since he wants speed information. His application is shown in Figure 7(a). Emma may find that using magnetometers to detect vehicles more straightforward, so she builds an application as shown in Figure 7(b). She sets the magnetometer to sample at 16Hz. Cory turns on the magnetometer at 256Hz and logs the data in a running buffer. The buffer output is triggered by object detections from the breakbeams. It uses the speed estimate to calculate which section of the buffer is sent back to Cory. Cory's application is shown in Figure 7(c).

The semantics of the data are also annotated in the model. For example, using a short notation, in Task T:

$$type(T3) := (\texttt{D, "sortedEdges", \{("timeStamps", long[])\}});$$
$$type(T4) := (\texttt{D, "vehicle",}$$
$$\texttt{\{("timeStamp", long), ("speed", double)\}});$$

in Task E:

$$type(E2) := (\texttt{C(1/16), "magnetometer",}$$
$$\texttt{\{("magneticField", int)\}});$$
$$type(E4) := (\texttt{D, "vehicle", \{("timeStamp", long)\}});$$

in Task C:

$$type(C1) := (\texttt{D, "sortedEdges", \{("timeStamps", long[])\}});$$
$$type(C2) := (\texttt{D, "movingObject",}$$
$$\texttt{\{("timeStamp", long), ("speed", double)\}});$$
$$type(C4) := (\texttt{C(1/256), "magnetometer",}$$
$$\texttt{\{("magneticField", int)\}});$$

STS derives the following type relations, among others:

$$type(T3) = type(C1) \quad (3)$$
$$type(T4) < type(E4) \quad (4)$$
$$type(C4) < type(E2) \quad (5)$$

Note that using a syntax-based approach, only (3) can be detected. Our semantics-based approach gives better information reusa-bility. For example, when Task T is already running and Task E is injected, neither `Magnetometer` nor `MagVehicle Detection` services are instantiated. The input `E4` is subscribed to the output `T4`. If Task C is already running when Task E is injected, a `16-DownSampler` service is automatically inserted between `C4` and `E2`. If the magnetometer in Task C were set to a sampling frequency that is not a multiple of $1/256$, a `LinearInterpolator` service would be inserted to perform a lossy conversion. When Task C, Task E, and Task T are injected in that order, the runtime image looks like the one shown in Figure 8.

5 Related Work

Componentizing computation is a trend in networked systems. It has the benefits of software reuse, information hiding, and some degree of fault tolerance [22]. In Internet and enterprise computing, web services [5] are pre-built software components that can be assembled across organizations and over the network to form new applications. Component-based and service-oriented approaches to organizing embedded software is also emerging [23, 13, 9, 17]. Publish/subscribe models like those in LINDA [8], CORBA event service [21], and JavaSpaces [6], are popular approaches to mediate uncoordinated tasks. Although caching and replication are very common in these architectures, little work has been done on removing redundant publishers.

The IrisNet project [9] studies service composition in resource rich sensor networks. It shares many commonalities with our vision, where a network of sensor enabled embedded devices provide services to end users, and the framework tries to remove redundant sensing and computation at run time. IrisNet uses a caching-based approach for data sharing [18]. In fact, it has a trace-based naming scheme such that the name of the data encodes the sequence of functions that has been applies to get the data. It is essentially an effective way to achieve syntax-based redundancy removal.

Semantics-based type checking and conversion has been seen in unit type systems, where the type system automatically converts measurement units, for example from inches to centimeters, given its understanding of their semantics (i.e. units) [11]. Treating time and sampling rates as part of system semantics is widely seen in engineering design framework. Clock calculus in synchronous languages such as Signal [7] and trigger analysis in time-triggered languages such as Giotto [10] are both capable of reasoning about sampling rates. However, these analyses are performed statically at compile time. We formulate timing properties as part of the data semantics, which allow us to reuse information dynamically at runtime.

6 Conclusion and Future Work

Microservers need to respond to uncoordinated user tasks in a resource efficient way. In this paper, we have presented a service-oriented architecture designed in SERUN, and the effectiveness of using semantics information to help reduce redundant computation. We have defined a signal type system that makes data semantics precise in terms of both

data values and tag types. Using the type system, we can check signal compatibility and perform runtime adaptation using down sampling and interpolation.

A type system is a powerful concept that puts semantics-based optimization into a formal framework. Next we discuss some of the limitations of SERUN and our plans for future work.

Task injection order dependencies. As seen in Section 4, the optimization results depend on the order in which the tasks are injected to the microserver. This is because we only prune the new tasks but keep the existing tasks unchanged. The advantage is that it preserves the continuity of existing tasks. But the disadvantage is that the final computation graphs may be sub-optimal. For example, in our parking garage system, using five breakbeam sensors to detect a car costs more communication energy than using a single magnetometer. It is a good idea to prune Task E if Task T is already running. However, when Task T finishes, it is no longer resource optimal to continue using the breakbeam sensors. We plan to further extend our data annotation to include the resources needed to generate each signal in a SCG. Then SERUN will have the information needed to dynamically optimize resources based on currently running tasks.

Runtime type resolution. Currently, in SERUN all port types are fixed at the time at which tasks are injected. This constrains our capability to further reduce redundancy. For example, two tasks may both need samples from sensor *A*, one with 5Hz sampling frequency and the other with 2Hz sampling frequency. If the first task is started before the second one, the sampling rate will be fixed at 5Hz, and the second task will receive interpolated input. Ideally, when the second task is injected, we should change *A*'s sampling frequency to 10Hz and automatically down sample the outputs for each task, such that no lossy conversion is necessary. This information is readily captured in our STS. However, to effectively use it, we need runtime type resolution capabilities. A service may only specify constraints between its input tag types and output tag types, (e.g. equality), rather than fixing it to a specific set. The runtime system then solves the set of constraints, so that new constraints can propagate through the SCG.

Data fidelity. We described in Section 3.3 the need for enhance naming schemes to embrace richer semantics information. This is particularly important when dealing with real world signals and information processing. Vehicle detections are not crisp values. Different detection schemes may have their own false alarm and miss detection rates. We call this the *data fidelity*. We are interested in extending our type system to capture data fidelity concerns. For example, the breakbeam array may have a higher false alarm rate than the magnetometer in terms of detecting cars (e.g. it may treat two people walking side-by-side as a vehicle.) So the fidelity of the vehicle detection output using breakbeams is lower than that of using the magnetometer. If the required detection fidelity in Task E is higher than what the breakbeam can provide, then Task E cannot be pruned.

These future directions will make the semantics-based task optimization more powerful and practical.

7 Acknowledgments

The authors would like to thank Prabal Dutta and Kamin Whitehouse for their contributions to the parking garage testbed deployment and service implementations.

References

[1] *1451.2: A Standard for a Smart Transducer Interface for Sensors and Actuators - Transducer to Microprocessor Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats*. 1997.

[2] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL web ontology language reference, 2004. W3C, http://www.w3.org/TR/2004/REC-owl-ref-20040210/.

[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] J. Davis, II, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Tsay, B. Vogel, and Y. Xiong. Ptolemy II: Heterogeneous concurrent modeling and design in Java. Technical Memorandum UCB/ERL M01/12, EECS, University of California, Berkeley, Mar. 2001.

[5] T. Erl. *Service-Oriented Architecture : A Field Guide to Integrating XML and Web Services*. Prentice Hall, 2004.

[6] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces: Principles, Patterns, and Practive*. Addison-Wesley, 1999.

[7] T. Gautier, P. L. Guernic, and L. Besnard. Signal: A declarative language for synchronous programming of real-time systems. In *Proc. of a conference on Functional programming languages and computer architecture, Portland, OG*, pages 257 – 277. Springer-Verlag, 1987.

[8] D. Gelertner. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[9] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. IrisNet: An architecture for compute-intensive wide-area sensor network services. *IEEE Pervasive Computing*, 2(4):22–33, October 2003.

[10] T. Henzinger, B. Horowitz, and C. Kirsch. Embedded control systems development with Giotto. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES'01)*, June 2001.

[11] G. S. N. Jr. Conversion of units of measurement. *IEEE Transactions on Software Engineering*, 21(8):651–661, Auguest 1995.

[12] E. A. Lee. Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, 7:25–45, 1999.

[13] E. A. Lee. What's ahead for embedded software? *IEEE Computer*, 33(9):18–26, September 2000.

[14] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on CAD*, 17(12):1217–1229, Dec. 1998.

[15] J. Liberty. *Programming C# (3rd. Ed.)*. O'Reilly, 2003.

[16] J. C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[17] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[18] S. Nath, Y. Ke, P. B. Gibbons, B. Karp, and S. Seshan. A distributed filtering architecture for multimedia sensors. In *First Workshop on Broadband Advanced Sensor Networks (BaseNets)*, October 2004.

[19] Object Management Group. OMG unified modeling language specification (action semantics). November 2002. OMG Document #ptc/02-01-09.

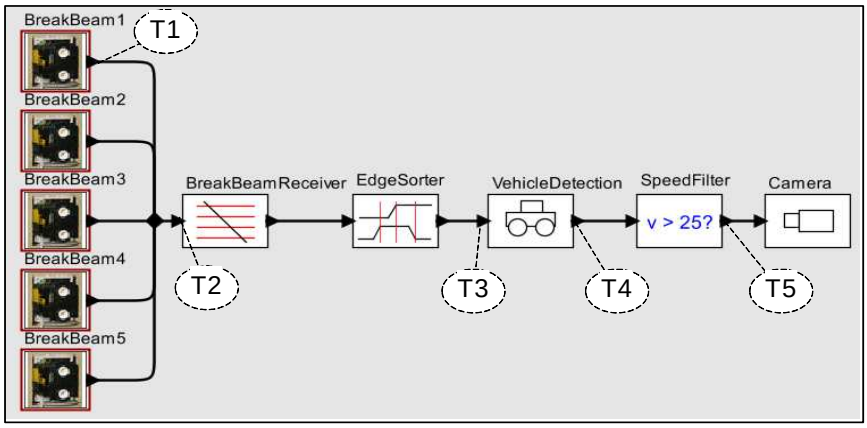
[20] Open Geospatial Consortium, Inc. *Sensor Model Language (SensorML) for Insitu and Remote Sensors (v1.0.0 beta)*. 2004. doc# 04-019r2.

[21] C. O'Ryan, D. C. Schmidt, and J. R. Noseworthy. Patterns and performance of a CORBA event service for large-scale. *International Journal of Computer Systems Science and Engineering, CRL Publishing*, 2001.

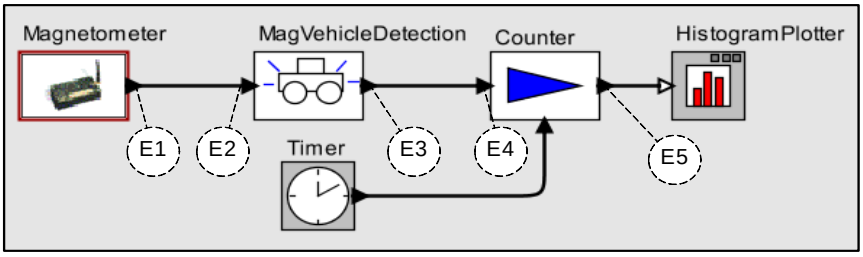
[22] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1997.

[23] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *IEEE Computer*, 333(3):78–85, March 2000.

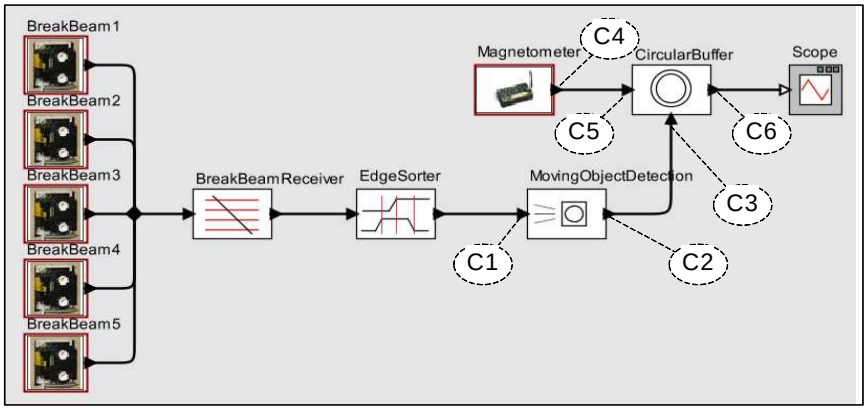
[24] K. Whitehouse, F. Zhao, and J. Liu. Semantic Streams: a framework for declarative queries and automatic data interpretation. Technical Report MSR-TR-2005-45, Microsoft Research, One Microsoft Way, Redmond, WA 98052, April 2005.



(a) Todd's application (Task T)



(b) Emma's application (Task E)



(c) Cory's application (Task C)

Figure 7: **Three tasks are injected to the parking garage microserver. Some port names are labeled in circles.** *Todd uses the breakbeam array to detect speeding vehicles. Emma intends to use magnetometer at a low sampling rate (16Hz) to detect entering vehicles. In Cory's application, high sampling rate magnetometer signals are sent to a circular buffer. Object detections trigger the buffered data be sent to Cory.*

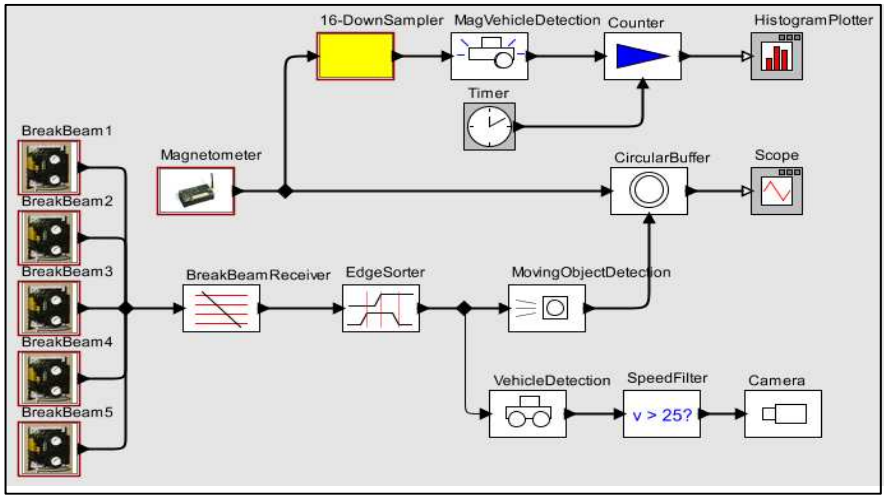


Figure 8: **A conceptual run-time image when all tasks are running.** *The tasks are injected in particular order: Task C, Task E, and Task T. Notice that if they are injected in a different order, the run-time image may be different.*