

STFL: A SpatioTemporal Filtering Language with Applications in Assisted Living

Athanasios Bamis and Andreas Savvides
ENALAB, Yale University
New Haven, CT 06520, USA
{firstname.lastname}@yale.edu

ABSTRACT

In this paper we introduce the *Spatio Temporal Filtering Language (STFL)*, which is a language framework that aims to provide the primitives for easily defining rules and sequences of rules and constraints. These sequences of rules can be used to convert low-level streams of sensor data into higher-level semantics and provide triggers for actuation. Among others *STFL* provides support for heterogeneous types of sensors, composability and code reusability. Special emphasis is given on the support of different categories of users by providing different types of interfaces spanning from a natural-like language aiming at end-users to a regular scripting language aiming at system developers. The expressiveness and power of *STFL* is presented through an assisted living scenario.

Categories and Subject Descriptors

D.3.2 [Programming Languages]: Language Classifications - Data-flow languages, Very high-level languages

General Terms

Design, Languages

Keywords

human activity monitoring, assisted living, spatiotemporal filtering, actuation

1. INTRODUCTION

The recognition of human behavior using sensors to provide assistive services entails the rapid interpretation of sensor data streams to generate actuation decisions. Low-level sensor data needs to be converted to higher level actions through the execution of a wide variety of mechanisms ranging from simple rules to elaborate Finite State Machines, grammars, filters, statistics and other methods. These methods would in turn have to operate over varying spatial and temporal ranges. Such a process should also be able to uti-

lize a heterogeneous set of sensors making system programming a challenging task.

Using our experiences from the BehaviorScope project at Yale [3, 6], we have designed the *Spatio Temporal Filtering Language (STFL)*, that is intended to provide a set of abstractions that makes it easy to compose assistive services by offering mechanisms for specifying how heterogeneous sensor data streams should be processed. In this initial form *STFL* provides a layered set of abstractions targeting different levels of expertise. It starts by putting raw sensor data into name-value pairs, it provides methods for building filters from predicates and then composing them into states and streams up to the level where an inexperienced end user could configure the system using a few commands in human like language (see Figure 1). System developers can implement functionalities inside the *STFL* core to provide new abstractions and services, application developers can integrate new sensing modalities and specify how they can be used. Domain experts and end users can use the higher level interfaces of *STFL* to write scripts or specify how the system should behave through a graphical system interface. The primary goal of our design is to simplify application development and make it more robust and accessible to developers, domain experts and end users.

In this paper we introduce *STFL* by outlining how its different components can be applied to compose a specification for processing data and generating actuation. Its expressive power is illustrated through an example derived from one of our assisted living deployments.

Our presentation is organized as follows: Section 2 lists our design requirements and describes *STFL*'s layered architecture. Section 3 describes the language's core components and actuation primitives, Section 4 presents the higher-level interfaces of *STFL* and Section 5 illustrates through an example how they can be used. Finally, Section 6 presents shortly an overview of the related work and section 7 concludes the paper.

2. DETECTING EVENTS OF INTEREST

The main input of assisted living environments consists of streams of measurements associated with spatiotemporal information. Motion sensors and cameras provide a stream of locations of persons associated with a timestamp, whereas simpler types of sensors as wearable sensors, door/window sensors or thermometers provide events or measurements associated with a location (the location of the sensor) and a timestamp. Processing this information can produce higher-level semantics that can be used for actuation spanning from

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PETRA'09 June 9-13, 2009, Corfu, Greece

Copyright 2009 ACM ISBN 978-1-60558-409-6 ...\$5.00.

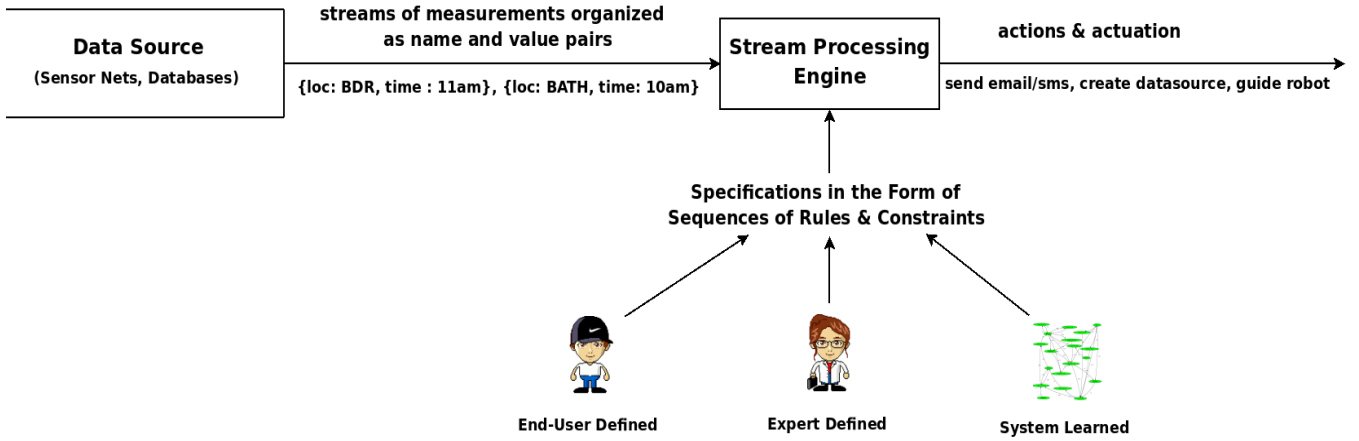


Figure 1: High-level overview of the functionality offered by *STFL*.

simple notifications to guiding persons with cognitive decline to successfully complete their tasks (e.g., see [10]).

Simple high-level activities can usually be decomposed into simple sequences of rules and constraints (which are essentially Finite State Machines) describing the activity in terms of sequences of locations and specific temporal characteristics, expressed as transitions between states, but also including timing and other constraints for these transitions. The *Spatiotemporal Filtering Language (STFL)* can be used by different categories of users to define sequences of rules and constraints describing human activities and provides handles for actuation and statistics extraction upon the detection of these activities.

The basic initial requirements for *STFL* are the following:

- Allow the definition of sequences of rules and constraints including most types of FSMs.
- Easily support new types of sensor measurements.
- Provide different types of APIs aiming at users with different expertise.
- Allow composability, code reusability and the formation of hierarchies (i.e., allow using rules and sequence of rules as part of other sequences of rules).
- Allow fine-grained actuation.

To this end, the basic assumption made by *STFL* is that sensor measurements consist of name-value pairs or attributes, whether the value concerns measurements of actual physical values or the associated metadata. The most common types of metadata are the temporal and spatial properties of these measurements. In most applications that require the detection of human behaviors, the incoming stream of data will include events associated with a location (i.e., the location of the event, maybe implicitly given by the unique id of the sensor that generated it), its start time and its duration (maybe extracted from a sequence of timestamps). Other types of metadata might include RSSI values, remaining battery life and anything else that a sensor node can generate.

STFL is organized in four different layers (see Figure 2). The layer that provides the basic functionality of the language is the *STFL Core* that provides the primitives for

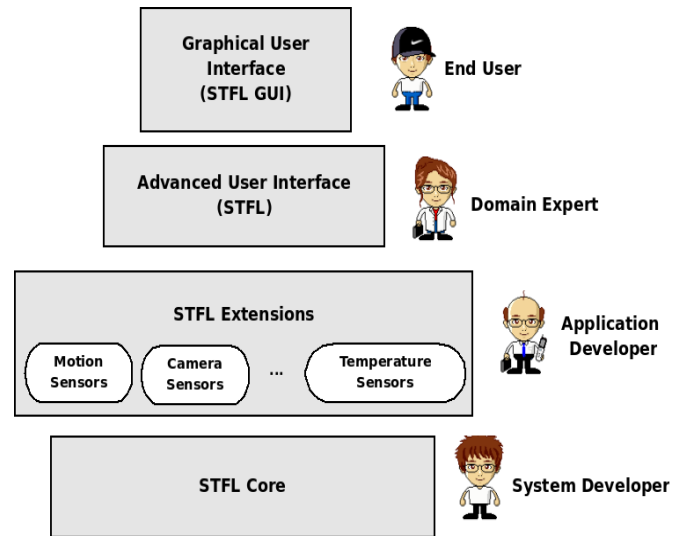


Figure 2: *STFL* provides four different layers aiming at users with different requirements and expertise.

modelling streams of sensor measurements and defining sequences of rules and constraints. *STFL Extensions* are written by the application developers and provide libraries of actuation functions, libraries for extracting statistics, as well as sensor and application specific operators and extensions of the *STFL Core*. The *Advanced User Interface* or simply *STFL* is a close to natural programming language that can be used by experts and advanced users to securely (i.e., without full access to the system) access a significant part of the functionality of the *STFL Core*. Finally, the *Graphical User Interface* or *STFL GUI* provides a web-based graphical environment, where the end-users can be guided through the process of defining their own simple sequences of rules, using a large subset of *STFL*.

The *STFL Core* is essentially the “assembly language” for *STFL* and all the other layers are translated or are extensions of the *STFL Core*. The initial version of the *STFL Core* and the *STFL Extensions* have been written using the Python programming language. This means that the system and application developers have a lot of power and versatility in extending and modifying any part of *STFL*, as well as

access to the large number of modules available in Python. One more significant property of Python is that it can be easily interfaced to most other programming languages and it is part of many web frameworks allowing thus, the easy integration of *STFL* to most existing systems. *STFL* code can be compiled into *STFL Core* code using a compiler provided by the *STFL Core*. The purpose of the *STFL GUI* is to provide a graphical way for end-users to generate *STFL* code without needing to know the syntax of the latter.

3. THE STFL CORE

The *Core* of the language defines the primitives for modelling sensor measurements and allows the definition of sequences of rules and conditions. Moreover, it provides an implementation of the *Stream Processing Engine*. To be as generic as possible and to allow easy addition of extensions and new functionality, the *Core* decomposes every sensor measurement and every sequence of rules into very simple primitives that are organized hierarchically to provide the full functionality of the language. These primitives are represented by Python objects and, consequently, extending or modifying their functionality is an issue of overloading the operators and methods that they provide.

The Core of *STFL* provides the following primitives, whose organization appears graphically in Figure 3:

- *Value*: Is the most primitive type of the language and provides a way for grouping pairs of values and names. Thus, the pair “location, ‘Bedroom’”, “node id, 12”, “temperature, 27” or “datetime, ‘2008-12-09 22:31:27’” are all valid types of *Values*. Of course, there is no reason why more complex types as the locations of persons under the field of view of a camera (see [3]) cannot be represented as *Values*, as long as they can be modeled as python objects. Every operator (logical, relational, arithmetic) that is defined for the type of objects that are represented by a *Value* are also defined for *Values* with the same name (and implicitly the same type of represented objects), which are called *compatible*. To this end, if $X = \text{“temperature, 27”}$, $Y = \text{“temperature, 28”}$ and $Z = \text{“node id, 23”}$ are *Values* the comparison $X < Y$ will return *True*, whereas the comparison $Y > Z$ will raise an exception.
- *Measurement*: Is a collection of *Values*. When *Measurements* are coming directly from the sensors, then they represent essentially the raw measurements that each sensor produces. Thus, a *Measurement* for a PIR sensor will look like “node id, 10”, “timestamp, 1229132955” (including maybe other measured values as the RSSI of the node), where “node id” will essentially give us the location of the sensor (and implicitly that motion was detected there) and the timestamp gives us the time that the motion was detected. Operators are also defined between *Measurements* as well as between *Measurements* and *Values*. Operators can be applied between two *Measurements* when both *Measurements* have all their *Values* compatible or with other words for each *Value* of the first *Measurement* there is a compatible *Value* on the other *Measurement*, or alternatively one of the *Measurements* contains a subset of the *Values* of the other *Measurement*. Hence,
 - if $M_1 = \{\text{“node id, 10”}, \text{“temperature, 27”}\}$, $M_2 = \{\text{“node id, 10”}\}$ and $M_3 = \{\text{“node id, 10”}, \text{“timestamp, 1229132955”}\}$, then $M_1 == M_2$ and $M_2 == M_3$ will both return *True*, whereas $M_1 == M_3$ will raise an exception. Moreover, most basic operators are defined between *Values* and *Measurements*, in which case the *Value* is first casted to a *Measurement* and then the operator is applied.
- *Predicate*: Is the smallest part of a rule that we can define. It consists of a *Value* and a valid binary operator (i.e., an operator which when applied will return a *True* or *False* response as for instance the relational operators) for the type of measurement that the *Value* contains. Every *Predicate* contains a method that accepts as input a *Measurement* and returns a *True* or *False* response indicating whether the *Measurement* satisfies the relation. Thus, the *Predicate* defined by the operator *eq* (*==*) and the *Value* “node id, 12” will return *True* for all measurements coming from node with id 12. As an operator for a predicate can be used any function that accepts as input arguments a *Measurement* and a *Value* and returns a value *True* or *False*.
- *Filter*: Groups one or more *Predicates* in a single logical rule and provides a method which when applied on a *Measurement* will return a *True* or *False* response depending on whether the *Measurement* matches the specifications of the rule. Consequently, *Filters* can express rules of the form “timestamp > 10am AND timestamp < 11am”, “node id == 10 OR node id == 12”.
- *State*: Represents a rule in the sequence of rules that are used to express the patterns of interest. In this sense, it can also represent the states of an FSM, providing also the primitives for compacting many states into a single one that captures all of them. Each *State* has a name and contains also information about the *State* that preceded it (if such a state exists) and the *States* that need to be checked after the current *State*. Every *State* is also related with one or more *Filters* again organized as logical rules. Thus, a *State* can represent rules of the form “(node id == 10 OR node id == 12) AND (timestamp > 10am AND timestamp < 11am)”, which will return *True* only for *Measurements* from nodes 10 and 12 and between 10am and 11am.
- *Sequence*: Represents the sequence of rules defined by the user. It consists of one or more *States* “glued” together using *Transition* objects, which we will describe next. It is identified by a name and its basic property is that it is a “special” type of *State*. This is important because it means that any *Sequence* can be part of another *Sequence*, thus allowing us to define hierarchies of rules consisting of simpler rules.
- *Transition*: Defines when the next *State* will be checked. The default option is to check the next *State* after the current *State* has been satisfied without allowing any non-matching *Measurements* in between. Other valid options are to allow don’t care *Measurements* between two consecutive *States*, and to allow the logical “OR”

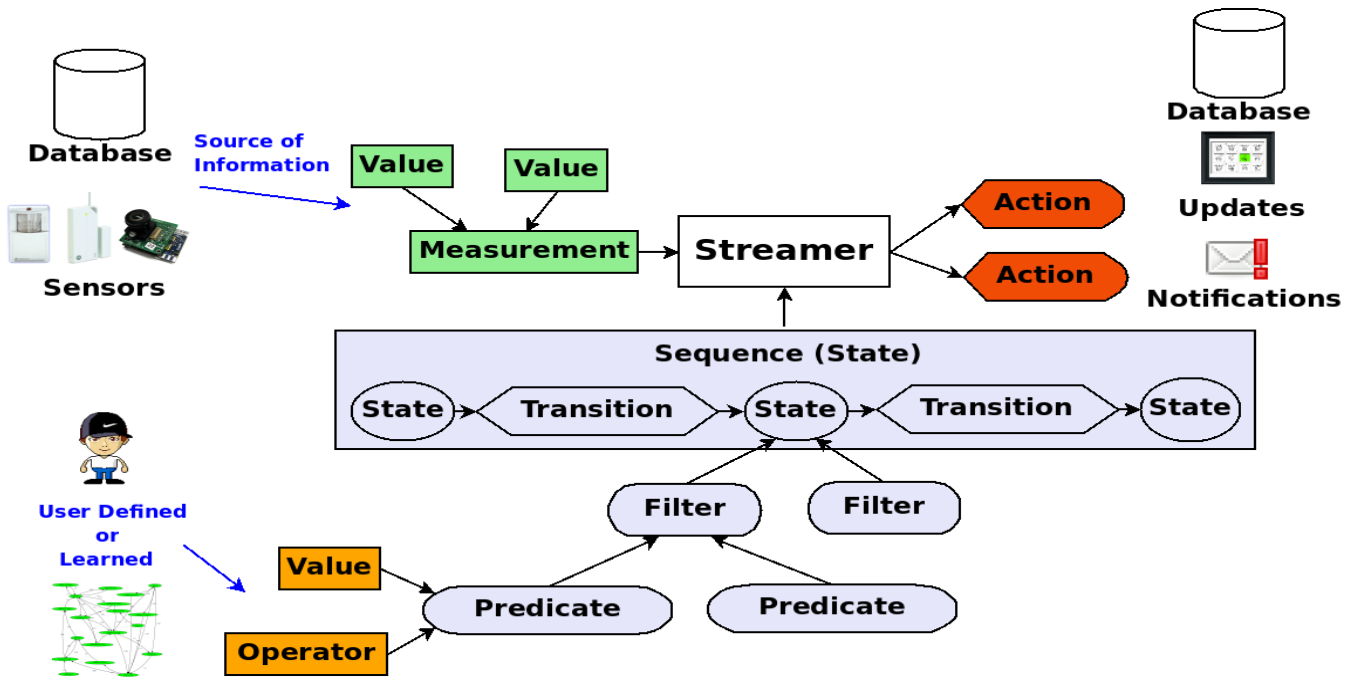


Figure 3: The hierarchy of the primitives provided by the STFL Core for encoding sensor measurements and sequences of rules and conditions.

or “AND” of *States*. In both the “OR” and “AND” cases all the *States* that are connected with an “OR” or an “AND” will be checked in parallel from left to right. In the case of “OR” if any of the *States* matches the stream of *Measurements* the computation will proceed to the next *State* after the last “OR” in the subsequence. In the case of the “AND” all *Measurements* must match before proceeding to the next *State*. Thus, the “AND” can be used to define that some rules need to happen but their relative order is not important. Every *Transition* defines also a maximum (to prevent stale *Sequences*) and optionally a minimum duration within which the “transition” to the next *State* must happen.

- *Action*: It groups one or more actions (in the form of functions) that will be called when the object that the *Action* is associated with will be satisfied. The functions contained in an *Action* will be called with the arguments that the user has specified (so that we can have libraries of function that the user can customize) and with the instance of the object that called the *Action*, as well as the subsequence of the stream of *Measurements* that caused the rules of the object to be satisfied. Valid objects that an *Action* can be associated with are *Predicates*, *Filters*, *States* and *Sequences*. Using *Actions* the user can have very fine grained control of when actuation is happening. Possible actuation types can include communicating messages via e-mail or SMS, storing data in databases, updating status of devices (e.g., digital picture frames), producing audio alerts, guiding robots and anything that would make sense for the particular environment.
- *Streamer*: Has the role of the processing engine. In particular, a *Streamer* is associated with one or more

data sources, which produce streams of data. The *Streamer* groups the data into *Values* and *Measurements* either directly from the program that collects the data from the sensor network or indirectly from the database that the data has been stored, using functions provided by the developers of the sensor network or generic functions for accessing different database engines. Sequences of rules defined by the users or automatically learned by the system (e.g., see [5, 4]) are then attached to the proper *Streamer*.

The *STFL Core* as mentioned provides also a compiler for converting *STFL* code to Python code using the *STFL Core* primitives. Constraints of the data can be expressed as *Predicates* and *Filters*, *STFL* statements separated by the operator “(right) after that” and the logical “or” and “and” can be expressed as *States* and the duration constraints can be defined as *Transitions*.

4. HIGHER-LEVEL INTERFACES

On top of the *STFL Core* our language framework, provides as mentioned three additional layers that provide different functionality and aim at different categories of users. In this section we present shortly an overview of these layers starting with the *Advanced User Interface* or simply *STFL*, and then presenting shortly the *STFL Extensions* and the *STFL GUI*.

4.1 Advanced User Interface (STFL)

The *Advanced User Interface* provides a language named, similarly with the framework, *Spatio Temporal Filtering Language (STFL)* that can be used to express most of the functionality offered by the *STFL Core*. The basic premise of the language is that it attempts to be as close to natural

ENALAB BehaviorScope | Data sources | Triggers | Getting started | Welcome, enalab. | Log out

Please Specify A Detection Rule

Here you can specify an activity that you want detected (see [help](#) for more information and examples).

Going To Bed is

in	Bathroom Up	after	23:00	for at least	00:10	, right after that	
in	Bedroom1	at any time	hh:mm	for at least	00:15	, after that	Remove this rule.
in	Select Room	at any time	hh:mm	for any duration	hh:mm	.	Remove this rule.

when Going To Bed is detected

send email to user bob with message: tom is going to bed

clear advanced mode submit

Figure 4: *STFL* code can be graphically generated using the *STFL GUI*.

language as possible without sacrificing too much of the expressiveness of the underlying *STFL Core*. The language provides basic operators and keywords for specifying spatial and temporal properties of the rules, as well as for organizing the rules into sequences. *STFL* provides a basic syntax that is invariant to all types of sensor measurements in addition to a limited number of sensor specific operators and syntax. *STFL* is compiled into *STFL Core* code by a parser provided by the *STFL Core* layer.

STFL allows to capture events based on their temporal characteristics as the start time, the duration and/or the end time of an event. The user can specify the start and end time of an event relative to an absolute time or a range of times. In particular the user is allowed to define expressions as “before 8 am”, “between 8 am and 9 pm”, “at about 8 am” and more. Similar options are provided for the duration of an event, where the user is allowed to specify expressions as “for 8 min”, “between 20 min and 30 min”, “about 15 min” and more.

STFL provides also primitives for defining sequences of events, which capture the sequential properties of the streams of measurements. In particular *STFL* provides two operators “after that” and “right after that”, which define that the current event must be followed by the event that will be specified next. The keyword “right” specifies that the event on the right of the operator must succeed immediately (i.e., without other events occurring between them) the event on the left of the operator, whereas when this keyword is not given, we can have “don’t care” events separating the events to which the operator is applied. A number of keywords can be used to define timing constraints for the transitions between events.

Another key feature of *STFL* is that user defined rules can become part of other rules, thus allowing the formation of

hierarchies of rules. This is essential for assisted living applications, as it allows the creation of re-usable libraries for detecting common events of interest. Finally, *STFL* allows the definition of actions upon the detection of a sequence of sensor measurements that match a rule.

4.2 *STFL* Extensions

The *STFL Extensions* provide application and sensor specific interfaces for accessing the functionality offered by the *STFL Core*. In particular the *STFL Extensions* can provide application specific operators and interfaces for creating and manipulating specific types of sensor measurements and filters. An example of such an operator is an operator that checks if the location of a person as reported by a localization sensor (e.g., camera, RFID, GPS) is inside a given polygon. This operator accepts the points that define a polygon in the form of a *Value*, as well as the location of a person encoded as a *Measurement* object and decides efficiently if the given location resides inside the given polygon. Programs, consequently, can use this operator to define *Predicates* involving the locations of persons.

In addition libraries of possible actuation types for particular applications can be programmed as extensions. This actuation types can be either generic as “send email”, “generate audio notification” or they can be sensor or application specific. The most important types of extensions concern different types of statistics. Many of the statistics required for processing data collected from sensors can be re-applied to many different types of sensors with similar properties or to the semantics extracted from the data.

4.3 Graphical User Interface (*STFL GUI*)

The *Graphical User Interface (STFL GUI)* provides part of the functionality of *STFL* through a web-based graphical interface that is part of BehaviorScope Web (BScopeWeb)

portal¹ (see Figure 4). The *STFL GUI* guides the users through the process of defining their custom sequences of rules, as well as for selecting one or more responses that the system should generate as responses to the detection of these sequences of rules. Using the *STFL GUI* the user can generate an initial version of STFL code that describes the sequence of rules, which can then be manually edited. Moreover the user is allowed to store the rules and activate or deactivate them at will, as well as share them with other users of the system.

5. DISCUSSION

The basic features of STFL can be demonstrated using an example derived from one of our home deployments [3]. In one of our assisted living deployments, an elder lady living alone needs to receive a particular treatment. The treatment needs to be always performed at most one hour prior to her night sleep, but based on her daily condition she might need to receive her treatment an unspecified number of times during the day (a common example of such a need could be absorbent products for incontinence). The supplies required for her treatment are stored in a cupboard in the Kitchen of the apartment which is considered a “common area” (see [3]) and is equipped with camera-based sensors that can track a persons location and door/window sensors that can sense when a person has opened or closed the cupboard. The “private areas” of the house as the bathroom and the bedroom of the elder are equipped with motion sensors, that can sense the presence of a person in the room, but not her exact location.

The first task of our system is to detect when the person is preparing to go to sleep without having received her treatment and generate a reminder. Ideally, we want to notify the person as soon as possible to avoid having to wake her up, something which would significantly hurt the acceptance of the system by the elder. For the same reasons generating a reminder every time the person is about to sleep is also undesirable. A very simple rule that can significantly decrease the time required before deciding that a person is about to sleep is to check for a bathroom visit, before a bedroom visit with a given minimum duration. We must note here, that longer sequences indicating intention for going to sleep can also be automatically learned by the system [4]. Of course, we can always guess that the person is sleeping if we detect that she is in her bedroom for time longer than a threshold (such a threshold can be automatically and adaptively extracted from the history of the person [5]). Thus, a simple “state machine” that describes the detection of the event where the elder goes to bed without taking her medication appears in Fig 5.

The part of this state machine that describes the “going to bed” activity (namely states S1-S4) can be coded using the *STFL Core* primitives as follows:

```
# Value definitions
Bathroom = Value('location', BTH)
Bedroom = Value('location', BDR1)
Duration5min = Value('duration', 5*60)
Duration30min = Value('duration', 30*60)

# Predicate definitions
```

```
AtBathroom = Predicate(Bathroom, eq)
AtBedroom = Predicate(Bedroom, eq)
After10pm = Predicate(StartTime10pm, gt)
ForAtLeast5min = Predicate(Duration5min, gt)
ForAtLeast30min = Predicate(Duration30min, gt)

# Filter definitions
AtBathroomFilter = Filter(AtBathroom)
AtBedroomForAtLeast5min =
    Filter([AtBedroom, ForAtLeast5min])
AtBedroomForAtLeast30min =
    Filter([AtBedroom, ForAtLeast30min])

# State definitions
s1 = State('@ Bathroom')
s2 = State('@ Bedroom for at least 5min')
s3 = State('@ Bedroom for at least 30min')

# Sequence definition
going2bed = Sequence('Going 2 Bed Sequence')

# Here we define the actual sequence of rules.
# In particular, we attach a filter to each state
# using the [] operator, and between states s1
# and s2 we attach a timing constraint using the
# (10*60) syntax. The operator >> indicates that
# state s2 should succeed s1, but that there can
# be additional "don't care" states in between.
# The | operator indicates that one of the 2
# subsequences should evaluate to True in order
# for the entire sequence to be True.
going2bed =
    ( s1[AtBathroomFilter](10*60) >>
      s2[AtBedroomForAtLeast5min] )
    | s3[AtBedroomForAtLeast30min]
```

The same rules using the Advanced User Interface (or simply STFL) could be expressed in a natural manner using the code that follows:

```
Going 2 Bed is
    (in Bathroom, within 10 min after that
     in Bedroom for at least 5 min.)
    or
    in Bedroom for at least 20 min.
```

and in order to send a notification:

```
When Going 2 Bed
    send always sms to user tom with message '...',
    send once email to bob@gmail.com with
        subject '...' and message '...'.
```

Detecting that a person has received the proper treatment can potentially be detected using different types of sensors and methods. In our case we are employing again a sequence of rules expressed using *STFL* to detect that the person has visited the cupboard (i.e., the cupboard door was opened and closed) as well as the area in front of the sink. To check that the person visited the sink using the stream of *Measurements* coming from the camera-based sensors, we could use the following command provided by the *STFL Extensions*:

```
# PredicateArea is defined by the STFL Extensions.
# It accepts as input a list of points representing
```

¹<http://bscope.eng.yale.edu>

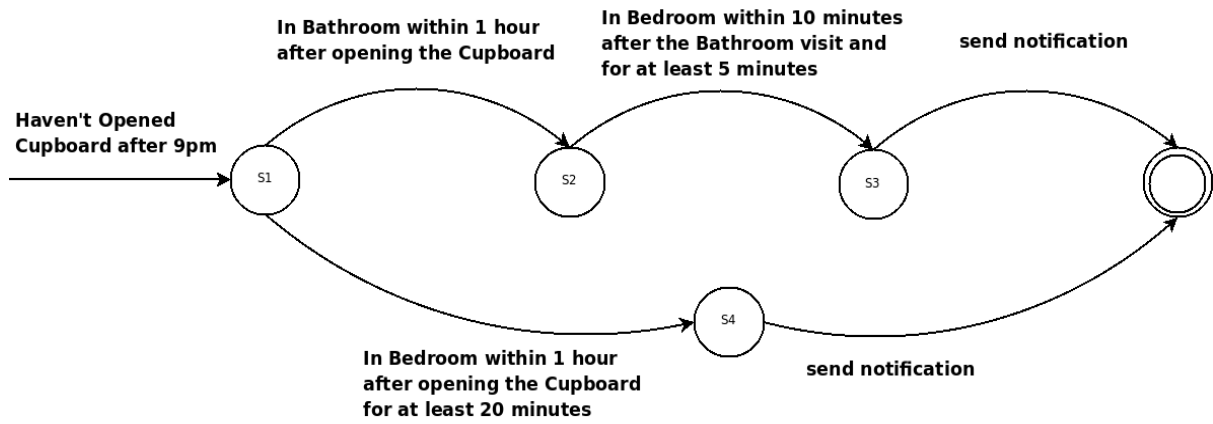


Figure 5: The “state machine” for detecting when a person hasn’t received her treatment at most one hour before going to sleep.

```
# the edges of a polygon and returns True when the
# point contained in the Measurement passed as its
# argument is inside (or by passing the argument
# "outside=True") or outside of the defined area.
inside_sink = PredicateArea(sink_area_edges)
```

Assuming, that the sequence that detects that the elder received her treatment is called “rcvd_treatment”, we can combine it with the “going2bed” sequence to generate notifications when the person forgot to received her treatment before going to bed. In particular, to send an SMS and and email notification we can use the following code:

```
# FilterTimeRange is provided by STFL Extensions and
# allows definitions of time ranges for common temporal
# properties, as the start time of an event.
after9pm = FilterTimeRange(min_start_time = '21:00:00')

# send_sms and send_email are provided by the actuation
# libraries of STFL Extensions and can be customized
# by the user.
sms_action = Action(send_sms, my_sms_params)
email_action = Action(send_email, my_email_params)

forgot_treatment = Sequence('Forgot treatment')

# The new sequence is defined using the previously
# defined sequences and the ~ operator, which evaluates
# to True when the rcvd_treatment sequence times out.
forgot_treatment[sms_action, email_action] =
    ~ rcvd_treatment[after9pm] (60*60) >> going2bed
```

Of course, besides simply sending a notification, we can attempt to express more difficult rules. In particular, we can attempt to determine when the person is about to run out of supplies and send a notification to her caregivers and/or relatives. Apparently, in this example we can reuse the existing definition of when the person received her treatment and we can leverage the fact that all the primitives defined by *STFL Core*, are essentially Python objects that, in addition to the “operators” indicated above, provide also methods with the same functionality. In the following code snippet we can see an example of how one can count the number of times medication was received, using a “for loop” and the “append_right_after” method offered by a *State*:

```
need_refill = Sequence('Supplies Needed')
for i in xrange(AMOUNT,0,-1):
    state = State('Remaining ' + str(i))
    need_refill.append_right_after
        (state = rcvd_treatment,
         max = 24*60*60)
```

As a final comment, instead of simply sending a notification to one or more caregivers/relatives of the person when the person is about to run out of supplies, there is no reason why we cannot instruct the system to send a notification only when some caregiver is near a pharmacy. In particular, assuming that some of the caregivers of the person are equipped with a GPS-enabled cell phone (see [16]) we can use the “*PredicateArea*” to check for presence near some predefined locations of pharmacies only when the elder is about to run out of supplies.

6. RELATED WORK

Because of the interest and difficulty in programming sensor networks a large number of programming methods and models has been proposed. An exhaustive enumeration of the different approaches is out of the scope of this paper, so we are going to mention only some representative examples of each category of solutions and the similarities and differences with our approach. The interested reader can find a survey of some of the most well-known programming models in [11].

The main purpose of *STFL* is to provide a framework for expressing sequences of rules and constraints on data streams in a simple form, as well as converting low-level data into higher-level semantics, based on which it can provide primitives for actuation. On the contrary macroprogramming models as *Abstract Regions* [12] or *Hood* [13] provide primitives for abstracting low-level node-actions and their main purpose is to simplify application development. A category of programming models more closely resembling the capabilities of *STFL* are programming models as *FLASK* [8] and *Vire* [2] that attempt to model the application as a dataflow graph. The main difference of *STFL* and macroprogramming models is that the goal of the former is to allow users to express difficult applications concerning data analysis and data mining, whereas the latter aim at sim-

plifying writing node-level applications. To this end, most macroprogramming approaches can be used only during development and require node-reprogramming, whereas *STFL* can be used for processing the incoming streams of data.

To this sense, *STFL* has certain similarity to the database-based approaches for querying sensor networks like *tinyDB* [7] and *Cougar* [15], which provide SQL-like languages for querying the network for measurements and programming languages that “filter” the streams of data as *Semantic Streams* [14] and *Regiment* [9]. The main problem with these approaches is that they don’t provide a notion of global state or history for the network, which means that the user cannot naturally express sequences of rules.

A different category of languages that have related goals with *STFL* are stream-processing languages as the language of the *Borealis* stream processing engine [1], that aims to develop a system for the efficient processing of large streams of data. The main difference of *STFL* and the query language offered by *Borealis* is the fact that latter provides mostly a language that can be used for querying large databases and manipulate large amounts of data, but has only limited capabilities for expressing in an easy way sequences of rules.

A final crucial difference of *STFL* with all the aforementioned approaches and to the best of our knowledge with any other existing approach is that it aims at providing a method for defining in a natural way sequences of rules and constraints. To this end, any of the aforementioned tools can be used in order to actually implement the low-level engine that queries the network or the incoming stream of data.

7. CONCLUSIONS

In this paper we have presented an initial version of the *SpatioTemporal Filtering Language (STFL)* that is a language framework that aims at providing proper abstractions for defining rules and sequences of rules and constraints for detecting patterns in streams of spatiotemporal sensor data. The initial implementation of the framework consists of four different layers/APIs aiming at different categories of users and allows the definition of simple state machines that can convert low-level streams of data into higher-level semantics, as well as actuation primitives based on these semantics. Future work includes among others the addition of features for statistics extraction from the data, a persistence layer for transparently connecting *STFL* to different databases and a code distribution scheme for in-network processing of the queries. For the latest information and the current release of *STFL*, interested readers can visit <http://enaweb.eng.yale.edu/drupal/stfl>.

Acknowledgments

This work was partially funded by the National Science Foundation under projects CNS 0626802 and CNS 0751513. Any opinions, findings and conclusions or recommendation expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

8. REFERENCES

- [1] D. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryzkina, et al. The Design of the *Borealis* Stream Processing Engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, Asilomar, CA, January, 2005.
- [2] R. Balani, A. Singhanian, S. Han, R. Rengaswamy, and M. B. Srivastava. Vire: Virtual reconfiguration framework for embedded processing in distributed image sensors, January - April 2007. NESL, UCLA Technical Report TR-UCLA-NESL-200701-01.
- [3] A. Bamis, D. Lymberopoulos, T. Teixeira, and A. Savvides. Towards precision monitoring of elders for providing assistive services. In *PETRA '08: Proceedings of the 1st international conference on Pervasive Technologies Related to Assistive Environments*, pages 1–8, New York, NY, USA, 2008. ACM.
- [4] D. Lymberopoulos, A. Bamis, and A. Savvides. Extracting spatiotemporal human activity patterns in assisted living using a home sensor network. In *PETRA '08: Proceedings of the 1st international conference on Pervasive Technologies Related to Assistive Environments*, pages 1–8, New York, NY, USA, 2008. ACM.
- [5] D. Lymberopoulos, A. Bamis, and A. Savvides. A methodology for extracting temporal properties from sensor network data streams. In *Proceedings of the 7th ACM/Usenix International Conference on Mobile Systems, Applications and Services (MobiSys '09)*, 2009.
- [6] D. Lymberopoulos, T. Teixeira, and A. Savvides. Macroscopic human behavior interpretation using distributed imager and other sensors. *Proceedings of the IEEE*, 96(10):1657–1677, Oct. 2008.
- [7] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems (TODS)*, 30(1):122–173, 2005.
- [8] G. Mainland, M. Welsh, and G. Morrisett. Flask: A language for data-driven sensor network programs, May 2006. Harvard University Technical Report TR-13-06.
- [9] R. Newton, G. Morrisett, and M. Welsh. The regiment macroprogramming system. In *Proceedings of the 6th international conference on Information processing in sensor networks*, pages 489–498. ACM Press New York, NY, USA, 2007.
- [10] H. Pigot, A. Mayers, and S. Giroux. The intelligent habitat and everyday life activity support. In *Proc. of the 5th International conference on Simulations in Biomedicine*, April, pages 2–4.
- [11] R. Sugihara and R. Gupta. Programming models for sensor networks: A survey, January 2007. UCSD Technical Report CS2007-0881.
- [12] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.
- [13] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 99–110, New York, NY, USA, 2004. ACM.
- [14] K. Whitehouse, F. Zhao, and J. Liu. Semantic Streams: A Framework for Composable Semantic Interpretation of Sensor Data. *LECTURE NOTES IN COMPUTER SCIENCE*, 3868:5, 2006.
- [15] Y. Yao and J. Gehrke. The Cougar Approach to In-Network Query Processing in Sensor Networks. *SIGMOD RECORD*, 31(3):9–18, 2002.
- [16] A. S. Yu, A. Bamis, D. Lymberopoulos, T. Teixeira, and A. Savvides. Personalized awareness and safety with mobile phones as sources and sinks. In *International Workshop on Urban, Community, and Social Applications of Networked Sensing Systems (UrbanSense08)*, 2008.