Programming IoT

Rupak Majumdar

Max Planck Institute for Software Systems
Kaiserslautern, Germany

(Joint work with Jeff Fischer)
Estimated U.S. Internet Protocol Traffic, 2000-2017 (Exabytes per Month)

Source: Cisco Visual Networking Index (VNI) and USTelecom Analysis. A DVD is assumed to store a two-hour movie.
Cost of bandwidth: DOWN 40 X
Cost of processing: DOWN 60 X
Cost of sensors: DOWN 2 X

Source: Gartner, IDC
Examples: Consumer analytics, real-time sensing and monitoring
## Information and Analytics

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitoring and profiling user behavior on the Internet</td>
<td>Analytics for business intelligence</td>
<td>Monitoring the behaviors of persons, things, or data through space and time</td>
</tr>
<tr>
<td>Learning user models for targeted ads</td>
<td>Examples: Clickthrough analysis, Location-aware recommendations</td>
<td>Examples: Smart factories, Production trends, Computation &amp; storage trends</td>
</tr>
</tbody>
</table>

Examples: Inventory and supply chain management, Security analytics
Examples: Process automation, Closed loop decision making, Complex autonomous processes
Automation and control

<table>
<thead>
<tr>
<th>1. Process automation</th>
<th>2. Closed-loop decision making</th>
<th>3. Complex autonomous systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controlling the behaviors of persons, things, or data through space and time</td>
<td>Feedback control of consumption for resources</td>
<td>Automatic control in open and uncertain environments</td>
</tr>
</tbody>
</table>

*Examples:
Software-based process control
Smart factories

*Examples:
Networked smart energy management
Smart buildings
Health monitoring

*Examples:
Autonomous cars & traffic networks
Robotic swarms, disaster management
Potential economic impact of IoT in 2025, including consumer surplus, is $3.9 trillion to $11.1 trillion

<table>
<thead>
<tr>
<th>Settings</th>
<th>Size in 2025¹</th>
<th>Major applications</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$ billion, adjusted to 2015 dollars</td>
<td></td>
</tr>
<tr>
<td>Human</td>
<td></td>
<td>Monitoring and managing illness, improving wellness</td>
</tr>
<tr>
<td>Home</td>
<td>200–350</td>
<td>Energy management, safety and security, chore automation, usage-based design of appliances</td>
</tr>
<tr>
<td>Retail environments</td>
<td>410–1,160</td>
<td>Automated checkout, layout optimization, smart CRM, in-store personalized promotions, inventory shrinkage prevention</td>
</tr>
<tr>
<td>Offices</td>
<td>70–150</td>
<td>Organizational redesign and worker monitoring, augmented reality for training, energy monitoring, building security</td>
</tr>
<tr>
<td>Factories</td>
<td>1,210–3,700</td>
<td>Operations optimization, predictive maintenance, inventory optimization, health and safety</td>
</tr>
<tr>
<td>Worksites</td>
<td>160–930</td>
<td>Operations optimization, equipment maintenance, health and safety, IoT-enabled R&amp;D</td>
</tr>
<tr>
<td>Vehicles</td>
<td>210–740</td>
<td>Condition-based maintenance, reduced insurance</td>
</tr>
<tr>
<td>Cities</td>
<td>930–1,660</td>
<td>Public safety and health, traffic control, resource management</td>
</tr>
<tr>
<td>Outside</td>
<td>560–850</td>
<td>Logistics routing, autonomous cars and trucks, navigation</td>
</tr>
</tbody>
</table>

¹ Includes sized applications only.

NOTE: Numbers may not sum due to rounding.

SOURCE: McKinsey Global Institute analysis
This Talk:
Programming abstractions, Models, and Analyses for developing Large-scale IoT Systems

Part I: A language abstraction
Part II: Some verification problems
Dynamics Level | Modeling the world: ODEs, Uncertainty | “Classical” control and signal processing: AD converters, PID controllers
Programming Environment

1. *Streams* and *stateful transformations* of streams
2. *Asynchronous* concurrency, real-time
3. *Uncertainty* as “first-class” object
4. *Heterogeneous* computing platforms
5. *Distributed* infrastructure
Domain-Specific Languages

- **Control**: Simulink/Stateflow
- **Synchronous hardware**: Esterel/Lustre
- **Systems & Networking**: Click
- **Data processing**: Apache Spark Streaming

- This Talk: *ThingFlow*, a DSL for IoT
Example: A Temperature Controller

- Sensor
- Moving avg
- Kalman filter

Interface w/ Cloud infrastructure

Actuator

- Blocking? New thread!
- Multiple sensors? Aggregate!
- Model missing? Learn params!

...
Simple ThingFlow Example

- Periodically sample a light sensor
- Write the sensed value to a file
- Every 5 steps, send the moving average to a message queue

```python
sensor.connect(file_writer('file'))
sensor.transduce(MovingAvg(5)).connect(mqtt_writer)
```
“Traditional” Event-driven Style (Callbacks)

def sample_and_process(sensor, mqtt, xducer, compcb, errcb):
    try:
        sample = sensor.sample()
    except StopIteration:
        final_event = xducer.complete()
    if final_event:
        mqtt.send(final_event,
        lambda:
            mqtt.disc(lamba: compcb(False), errcb),
            errcb)
    else:
        mqtt.disconnect(lamba: compcb(False), errcb)
    return
event = SensorEvent(sensor_id=sensor.sensor_id,
    ts=time.time(), val=sample)
csv_writer(event)
median_event = xducer.step(event)
if median_event:
    mqtt.send(median_event)
else:
    compcb(True)

def loop(event):
    def compcb(more):
        if more:
            event_loop.call_soon(lambda:
                mqtt.send(more)
            lambda: sample)
        else:
            print("all done, no more callbacks to schedule")
def errcb(e):
    print("Got an error: ",
        event_loop.call_later(0.5, loop)
        event_loop.call_soon(lambda:
            mqtt.send(event)
        lambda: sample)

1. Separate connecting streams with handling of runtime situations: distinct control flows for normal, error, and end-of-stream conditions not required

2. Inversion of control avoided: programmer’s view = data flow in the system

3. Scheduling is provided by the infrastructure
With Coroutines

```python
async
def sample_and_process(sensor, mqtt, xducer):
    try:
        sample = sensor.sample()
    except StopIteration:
        final_event = xducer.complete()
        if final_event:
            await mqtt.send(final_event)
        await mqtt.disconnect()
    return False

event = make_event(sensor.sensor_id, sample)
csv_writer(event)
median_event = xducer.step(event)
if median_event:
    await mqtt.send(median_event)
    return True

def loop(event):
    coro = sample_and_process(event)
    task = event_loop.create_task(coro)

    def done_callback(f):
        exc = f.result()  # not necessarily finite
        if exc:
            raise exc
        if f.cancelled() or exc:
            mqtt.disconnect()
        elif f.exception() is not None:
            print(f.result(), event)
        else:
            event_loop.call_later(0.5, loop)

    task.add_done_callback(done_callback)

1. No more callbacks, but interconnection still mixed with control situations

2. Choice to use asynchronous calls propagates through the program: implementation decisions have global effects
```
ThingFlow Features

• Streams of “things”
  – Input things introduce streams of events into the system (e.g., sensors)
  – Output things consume streams of events (e.g., actuators)

• Filters =
  Both input and output things = Stream transformers
ThingFlow Features

- ThingFlow Programs = Graphs of stream transformers connecting input/output ports
  - Basic construct: `A.connect(B, inport=outport)`
  - Syntactic sugar: default ports, chaining filters, combinators
    - `A.map(f)` – map the output stream on A using function f
    - `A.transduce(M)` – transduction by machine M

- Asynchronous, push-semantics
  - explicit scheduling
A basic control loop involving noisy sensors, estimators, control (due to others tasks taking too long), the next scheduled execution is thus, they will run at the same time for each execution of the 60 second intervals. This avoids tasks getting out-of-sync when one misses a deadline. This keeps as if the task had run at the correct time (by making the interval shorter). This avoids tasks getting out-of-sync when one misses a deadline. Finally, if a task is run later than its scheduled time, the new interval to first run on the next execution of the 30 second tasks.

New intervals of 30 and 45 seconds, we will schedule the new 60 second interval. We schedule the new interval to be coordinated with the existing one is run. If there are no tasks with the same interval, we look for the smallest interval that is either a factor or multiple of the existing interval to get a distributed implementation.

Connecting to the Outside World: Adapters

Finally, ThingFlow provides adapters to import and export streams into different data sources or data analysis, learning, and visualization pipelines. It provides readers and writers from csv files, pandas frames, TCP streams, MQTT streams (MQTT is a messaging protocol), etc. Using these adapters, one can write ThingFlow programs which communicate with external message sources or more, we have implemented ThingFlow based control loops. We give specific case studies below.

In ThingFlow, using adapters that talk to network streams, we write:

```python
kalman = Kalman(A, B, C, Q, R)  # Kalman filter
pid = PID(Kp, Ki, Kd)  # PID controller
pid.connect(kalman, port_mapping=('default', 'input'))
_ = Sensor().transduce(kalman)
    .transduce(pid)
    .connect(Actuator())
```

Figure 4. Cascaded filtering example \cite{19}, where the problem of estimating the state of an autonomous vehicle is solved using a cascaded filter model.
A basic control loop involving noisy sensors, estimators, control (due to others tasks taking too long), the next scheduled execution is this corresponds very closely to the control theorist's view. Notice visualization frameworks (Bokeh), etc. Using these adapters, one setting. First, it provides a single interface for sampling and stream computations, and actuation forms an event-processing pipeline: 5.1 A Control Pipeline with Bayesian Filtering 5. Case Studies Our ThingFlow-MicroPython scheduler minimizes wake-ups by deadline. Second interval. Finally, if a task is run later than its scheduled time this one. For example, if we have a new interval 60 seconds and old intervals of 30 and 45 seconds, we will schedule the new 60 second usage. If one runs a workload where the system wakes for half a second at a time once every 100 seconds and stays in deep sleep usage. For example, normal power consumption of an autonomous vehicle is solved using a cascaded filter model. System described in \[19\], the authors discuss two designs: first, the particle filter \( pf \) receives the mean value computed by the Kalman filter \( k_1 \) estimate and the raw camera feed is fed into a particle filter to finally estimate the pose. Figure 4 shows the control theorist's view of the cascaded filtering example \[19\].

\[\begin{align*}
g &= \text{Gyro()} \\
\text{k1} &= \text{Kalman(...)} \\
\text{pf} &= \text{ParticleFilter(I={'gyro', 'camera'})} \\
g.\text{transduce}\text{(k1).delay()}\text{\{.connect(pf, port_mapping=('default','gyro'))} \\
c &= \text{Camera()} \\
c.\text{connect}(pf, port_mapping=('default','camera')) \\
pf.\text{connect}(\text{Controller(...))} \end{align*}\]
A basic control loop involving noisy sensors, estimators, control (due to others tasks taking too long), the next scheduled execution is thus, they will run at the same time for each execution of the 60 second setting. First, it provides a single interface for sampling and stream maintenance, internal state to compute integrations or derivatives. That in addition to the Kalman filter, the controller usually also computes the control signal based on the estimate, and sends the measure a stream of sensor values, estimate state using a filter, computations, and actuation forms an event-processing pipeline:

5.1 A Control Pipeline with Bayesian Filtering

5. Case Studies

As well as write distributed ThingFlow programs. Data sources or data analysis, learning, and visualization pipelines, can write ThingFlow programs which communicate with external col for IoT systems, Spark streams, databases (Postgres, influxDB), frames, TCP streams, MQTT streams (MQTT is a messaging protocol), and is sending the video stream over MQTT (a messaging standard).

Finally, ThingFlow gives specific case studies below. Or more), we have implemented ThingFlow based control loops. We control or building control (where sampling intervals are in minutes compared to the computation requirements, such as in temperature the dynamics is fast. However, when the underlying dynamics is slow an implementation platform for all control systems, especially when to get a distributed implementation. Further optimizations are possible by look for the smallest interval that is either a factor or multiple of the existing one is run. If there are no tasks with the same interval, we matches an older task's interval, it will not be scheduled until the scheduler, we use the following logic. If a new task is added that delays the initial execution of certain tasks and grouping together tasks with the same wakeup interval. When a new task is added to the standard ThingFlow scheduler for Python, we already optimize robustness in the face of tight deadlines or power consumption. In Python's standard asyncio.

For example, normal power consumption of an autonomous vehicle is solved using a cascaded filter model. The gyro system described in the authors discuss two designs: first, the particle filter receives the mean value computed by the Kalman filter as a "true estimate." In ThingFlow, one can change from the first estimate the pose. Figure 4 shows the control theorist's view of the filter is used to provide precise estimates from the gyro. The gyro of an autonomous vehicle is solved using a cascaded filter model.

Since ThingFlow does not provide real-time guarantees, it is not necessary to optimize for minimal power consumption by reducing wake-up intervals of 30 and 45 seconds, we will schedule the new 60 second deadline. This avoids tasks getting out-of-sync when one misses a shorter). This avoids tasks getting out-of-sync when one misses a.

c = Camera()
c.connect(mqtt_writer)

g = Gyro()
k1 = Kalman(...)pf = ParticleFilter(I={'gyro', 'camera'})g.transduce(k1).delay()\ .connect(pf, port_mapping=('default','gyro'))c = Camera()
c.connect(pf, port_mapping=('default','camera'))pf.connect(Controller(...))mqtt_reader(...)
ThingFlow Implementation

• Python3 library
  – CPython: standard Python implementation
  – MicroPython: “bare metal” implementation for embedded systems

https://github.com/mpi-sws-rse/thingflow-python
Semantics = Comm. State Machines

Event streams
In Each Step...

Event streams

Input Thing

Output Thing
In Each Step...

Event streams
In Each Step...

Infinite-state system:

1. Events are infinite-state: 
   events can be chosen from an infinite set (e.g., real-valued signals)

2. Filters are infinite-state: the internal state of filters can be infinite (e.g., a Kalman filter)

3. Queues can be unbounded

4. Probabilistic: The filter transition function can be probabilistic

Semantics: Infinite-state Markov decision process:

- Scheduler picks policy
- State evolves probabilistically based on chosen filter
  (under measureability assumptions)

Core language: Prob streams

Reading from prob streams = sampling from the distribution
The ThingFlow Scheduler

- Responsible for scheduling “things”
  - Periodic observations (sensor sampling)
  - Non-periodic events (e.g. socket readiness)
  - Inter-thing events

- Abstraction over low level details
  - Threading, Order of scheduling

- Different implementations
  - On top of Python’s asyncio scheduler for Cpython
  - Custom, power-saving implementation for ESP8266

- ThingFlow programs must be explicitly scheduled to perform their tasks!
Simple ThingFlow Example

- Periodically sample a light sensor
- Write the sensed value to a file
- Every 5 steps, send the moving average to a message queue

```python
lux_sensor = LuxSensor()
sensor.connect(file_writer('file'))
moving_avg = MovingAvg(5).connect(mqtt_writer)
scheduler = Scheduler(asyncio.get_event_loop())
scheduler.schedule_periodic(sensor, 2)
scheduler.run_forever()
```

Default scheduler: Push an event entirely through the graph before handling the next input

-- Can replace async calls by sync calls
Solar Heater Example

Cooler water → Solar water heater → Hot water

Water temp sensor → Bypass valve → House

Low Pass Filter → Dispatch → Control State Machine

Default → Between → T_{high} → T_{low} → Actuator

Pool
Solar Heater Example: Controller State Machine

Initial → Normal
- $T_{LOW} / \text{OFF}$
- Between / OFF

Normal → Too Hot
- $T_{HIGH} / \text{ON}$
- $T_{LOW} / \text{OF}$
- $T_{LOW} / \varnothing$

Too Hot
- $T_{HIGH} / \varnothing$
Ref.

Solar Heater Example: Code

T_{\text{high}} = 110 \ # \ Upper \ threshold \ (\text{degrees} \ \text{fahrenheit})
T_{\text{low}} = 90 \ # \ Lower \ threshold
sensor = TempSensor(gpio_port=1)

# \ The \ dispatcher \ converts \ a \ sensor \ reading \ into
# \ threshold \ events
dispenser = sensor.transduce(RunningAvg(5)) \ 
        .dispatch([((lambda v: v[2]>=T_{\text{high}}, 't_{\text{high}}'),
                      (lambda v: v[2]<=T_{\text{low}}, 't_{\text{low}}'))]

controller = Controller()
dispenser.connect(controller, port_mapping=('t_{\text{high}}', 't_{\text{high}}'))
dispenser.connect(controller, port_mapping=('t_{\text{low}}', 't_{\text{low}}'))
dispenser.connect(controller, port_mapping=('default', 'between'))

actuator = Actuator()
controller.connect(actuator)
Lighting Project: Motivation

- If out of town for the weekend, don’t want to leave the house dark
- Replay lights “similar” to normal lighting pattern
Lighting Replay Application

Lux Sensors

Data Capture

Analysis and Machine Learning

Player Application

Smart Lights
Lighting Replay Application

Lux Sensors → Data Capture → Analysis and Machine Learning → Player Application

- ESP8266 remote nodes + Raspberry Pi
- Offline analysis and model learning using Jupyter, Pandas, HMMlearn
- Use an HMM model and Phue to control Philips Hue lights

Captured sensor data
HMM state machines

Smart Lights
ESP8266: Wiring Diagram

- SDA
- SCL
- 3V
- GND
Raspberry Pi

- TSL2591 lux sensor
- LED
- Resistor
- Raspberry Pi 2
Raspberry Pi: Wiring Diagram

- **TSL2591 Lux Sensor**
  -pin labels: Vin, 3v0, SDA, SCL
- **Resistor 10k**
- **LED**
  -Anode (long lead)
  -Cathode (short lead)
- **3.3V**
- **GPIO 0**

Additional information:

- Raspberry Pi A+ / B+ and Raspberry Pi 2 physical pin numbers
  -GPIO
  -Ground
  -3.3v
  -5v
  -ID EEPROM
  -Advanced use only!
Lighting Replay Application: Capture

Front Bedroom Sensor Node
- Lux Sensor
- ESP8266

Back Bedroom Sensor Node
- Lux Sensor
- ESP8266

Raspberry Pi (Dining Room)
- Lux Sensor
- MQTT
- Data Capture App
- Influx DB
from thingflow import Scheduler, SensorAsOutputThing
from tsl2591 import Tsl2591
from mqtt_writer import MQTTWriter
from wifi import wifi_connect

import os

# Params to set
WIFI_SID= ...
WIFI_PW= ...
SENSOR_ID="front-room"
BROKER='192.168.11.153'

wifi_connect(WIFI_SID, WIFI_PW)
sensor = SensorAsOutputThing(Tsl2591())
writer = MQTTWriter(SENSOR_ID, BROKER, 1883, 'remote-sensors')
sensor.connect(writer)

sched = Scheduler()
sched.schedule_periodic(sensor, SENSOR_ID, 60)
sched.run_forever()

The MQTT writer is connected to the lux sensor.
Sample at 60 second intervals

See https://github.com/mpi-sws-rse/thingflow-examples/blob/master/lighting_replay_app/capture/esp8266_main.py
Raspberry Pi Code (ThingFlow)

https://github.com/mpi-sws-rse/thingflow-examples/blob/master/lighting_replay_app/capture/sensor_capture.py
Lighting Replay Application: Analysis

Raspberry Pi (Dining Room)

- Flat Files
- HMM definitions

File copy

Laptop

Jupyter Notebook
Preprocessing the Data
(ThingFlow running in a Jupyter Notebook)

reader.fill_in_missing_times()
  .passthrough(raw_series_writer)
  .transduce(SensorSlidingMeanPassNaNs(5))
  .select(round_event_val)
  .passthrough(smoothed_series_writer)
  .passthrough(capture_nan_indexes)
  .output_count()
Data Processing: Raw Data

Front room, last day
Data Processing: Smoothed Data

Front room, last day
Data Processing: K-Means Clustering

Front room, last day
Data Processing: Mapping to on-off values

Front room, last day
Hidden Markov Models (HMMs)

• **Markov process**
  – State machine with probability associated with each outgoing transition
  – Probabilities determined only by the current state, not on history

• **Hidden Markov Model**
  – The states are not visible to the observer, only the outputs (“emissions”).

• In a machine learning context:
  – (Sequence of emissions, # states) => inferred HMM

• The hmmlearn library will do this for us.
  – https://github.com/hmmlearn/hmmlearn

Example Markov process (from Wikipedia)
Slicing Data into Time-based “Zones”

- Sunrise
- 30 Minutes before sunset
- Max(sunset+60m, 9:30 pm)

0 1 2 3 0
HMM Training and Prediction Process

Training
1. Build a list of sample subsequences for each zone
2. Guess a number of states (e.g. 5)
3. For each zone, create an HMM and call `fit()` with the subsequences

Prediction
For each zone of a given day:
- Run the associated HMM to generate N samples for an N minute zone duration
- Associated a computed timestamp with each sample
HMM Predicted Data

Front room, one week predicted data

Front room, one day predicted data
Lighting Replay Application: Replay

Raspberry Pi (Dining Room)

- HMM definitions
- Player Script

WiFi Router and Switch

Philips Hue Bridge

ZigBee

Front Room Smart Light

Back Room Smart Light

- Philips Hue Bridge

- HTTP

- ZigBee
Logic of the Replay Script

• Use `phue` library to control lights
• Reuse time zone logic and HMMs from analysis
• Pseudo-code:
  
  Initial testing of lights
  while True:
    compute predicted values for rest of day
    organize predictions into a time-sorted list of on/off events
    for each event:
      sleep until event time
      send control message for event
  wait until next day

https://github.com/mpi-sws-rse/thingflow-examples/blob/master/lighting_replay_app/player/lux_player.py
ThingFlow: Analysis

- **Bad news**: Communicating finite-state machines + FIFO queues = everything is undecidable!

- Decidable verification in special cases: finite-state events & filters, ordering of messages ignored

- Analyzing a filter: Abstraction & approximation of infinite-state probabilistic processes
  - algorithms with guaranteed error bounds

- **Open**: Tools and analyses for Thingflow programs
  - Asynchrony, Hybrid systems, Uncertainty, Distribution
Analysis of ThingFlow

Two example analyses for subcases:

1. Analyzing event flows: Provenance Analysis
   [Joint work with Roland Meyer & Zilong Wang]

2. Analyzing a filter: Abstracting infinite-state Markov processes
   [Joint work with Sadegh Soudjani and Alessandro Abate]
Provenance

Information about the *source* and *access history* of an object

“All inputs to controller are sanitized”
Provenance for ThingFlow

- Associate principals with filters
- Provenance of a message = Principals who have sent the message chronologically
- Provenance domain = Strings over principal names
Provenance Verification Problem

Given a Thingflow program \( P \), a stream \( x \), and a regular set \( R \) of provenances,
are the provenances of all events in \( x \) always in the set \( R \) along all executions of \( P \)?

Assumptions: Finitely many events, finite-state filters, and ordering of events in queues ignored

Note: The system is still infinite-state!

Example: All inputs to controller have passed through a sanitizer and then a state estimator
Provenance Verification Problem

Given a Thingflow program $P$, a stream $x$, and a regular set $R$ of provenances, are the provenances of all events in $x$ always in the set $R$ along all executions of $P$?

Assumptions: Finitely many events, finite-state filters, and ordering of events in queues ignored

Note: The system is still infinite-state!

Basic abstraction: For each stream, each kind of event, count how many events are currently in the stream
In Each Step...

Event streams

Finite state

Finitely many possibilities

Counting abstraction
Unbounded Events: *Petri Net*

- Finite set of places
- Finite set of transitions
- Places marked with tokens
- State: Marking
- Step: consume tokens from sources, put tokens into targets of a transition
- Defines an infinite state system
The Benefits of Petrification

Petri nets have nice decidable properties:

Coverability problem (is some place markable?) is decidable

Theorem [Rackoff,Lipton] The coverability problem for Petri nets is EXPSPACE-complete.
From ThingFlow to Nets

• A place for each filter state

• A place for each queue and each event type
  – Count how many events of each type in a queue

With provenances, we do not get a Petri net:

Unboundedly many provenances \rightarrow unboundedly many places
Unbounded Provenances: Automata

• Define equivalence classes w.r.t. the states of DFA for the regular set of provenances.

The validity of the provenance property depends on states of the spec automaton, not concrete provenances.

• Define a counter for each queue, event, and state of the spec
Reduction

Program + Provenance DFA $\rightarrow_{\text{poly}}$ Petri net

- Control flow can be modeled by Petri net
- Each counter is a place in the Petri net

Provenance verification problem = *Coverability problem of Petri nets*
Main Theorem

Provenance verification problem for finite-state ThingFlow programs (when ordering is ignored) is EXPSPACE-complete.
Linear Temporal Logic

• Provenance verification = Invariants
• Provenance linear temporal logic:
  “Whenever event in x has provenance R, eventually an event in y has provenance S”

Theorem: ProvLTL decidable for finite-state Thingflow programs (when ordering is ignored)
Analysis of ThingFlow

Two examples of decidability in special cases:

1. Provenance Analysis
   [Joint work with Roland Meyer & Zilong Wang]

   [Joint work with Sadegh Soudjani and Alessandro Abate]
Discrete-Time Markov Process

• State space $S$
• Transition kernel $T(ds' \mid s) = t(s' \mid s) \, ds'$

$$T(C \mid s) = Pr[s' \in C \mid s]$$

• $N$-step safety problem: Given $s_0$, $T$, and a set $A$, find the probability that the system stays in $A$ up to $N$ steps
  – Can formulate as a Bellman iteration (but without any closed form)
Markov Chain Abstraction

- Finite-state Markov chain = Representatives from a partition of the infinite-state space

- Transitions:

\[
P(v_i, v_j) = \int_{A_j} t(s' \mid v_i) \, ds'
\]
Main Result

If $t(. / s)$ is Lipschitz continuous with constant $h$, one can bound the probability of error between the original model and the finite-state abstraction:

$$\left| p_{s_0}(A) - p_{v_0}(A_\delta) \right| \leq N L h \delta$$

Prob of staying in $A$ for $N$ steps

Prob of staying in abstraction of $A$ for $N$ steps

$N = \text{Number of steps}$

$L = \text{Volume of } A$

$h = \text{Lipschitz constant}$

$\delta = \text{Diameter of abstraction}$
Infinite to Finite MDPs

• Bounds are *very* weak!
  – Compared to Monte Carlo simulation

• Open: Better bounds?

• Open: Verification for MDP + asynchronous concurrency?
Analysis of ThingFlow

Two examples of decidability in special cases:

1. Provenance Analysis
   [Joint work with Roland Meyer & Zilong Wang]

   [Joint work with Sadegh Soudjani and Alessandro Abate]

*Open: Analysis of a Thingflow program (combining asynchrony, filters, and probabilities)*
Other Open Problems

1. Parameterized reasoning
2. Real-time control
3. Fault tolerance and distribution
4. Deployment
5. Security, privacy, accountability
Conclusion

• **ThingFlow** = DSL for stream-processing applications for IoT systems
  • Streams & stream transformations
  • Filters & filter composition
  • Uncertainty & infinite-state
  • Asynchrony & explicit scheduling

• Many verification/analysis/tool aspects are open!
Thank You

http://www.mpi-sws.org/~rupak

ThingFlow:
https://github.com/mpi-sws-rse/thingflow-python

ThingFlow Examples:
https://github.com/mpi-sws-rse/thingflow-examples