Systematically exploring control programs (Lecture I)

Ratul Mahajan
Microsoft Research

Joint work with Jason Croft,
Matt Caesar, and Madan Musuvathi
Control programs are everywhere

From the smallest of networks to the largest
Control programs are everywhere

From the smallest of networks to the largest
The nature of control programs

Collection of rules with triggers and actions

motionPorch.Detected:
    if (Now - tLastMotion < 1s
        && lightLevel < 20)
        porchLight.Set(On)
    tLastMotion = Now

@6:00:00 PM:
    porchLight.Set(On)

@6:00:00 AM:
    porchLight.Set(Off)

packetIn:
    entry = new Entry(inPkt.src,
                      inPkt.dst)
    if (!cache.Contains(entry))
        cache.Insert(entry, Now)

CleanupTimer:
    foreach entry in cache
        if (Now - cache[entry] < 5s)
            cache.Remove(entry)
Buggy control programs wreak havoc.

One nice morning in the summer.
“I had a rule that would turn on the heat, disarm the alarm, turn on some lights, etc. at 8am ..... I came home from vacation to find a warm, inviting, insecure, well lit house that had been that way for a week......
That’s just one example, but the point is that it has taken me literally YEARS of these types of mistakes to iron out all the kinks.”
Control programs are hard to reason about

motionPorch.Detected:
  if (Now - timeLastMotion < 1 secs && lightMeter.Level < 20)
    porchLight.Set(On);
  timeLastMotion = Now;

porchLight.StateChange:
  if (porchLight.State == On)
    timerPorchLight.Reset(5 mins);

timerPorchLight.Fired:
  if (Now.Hour > 6AM && Now.Hour < 6PM)
    porchLight.Set(Off);
Desirable properties for bug finders

Sound

Complete

Fast
Two bug finding methods

Testing

Model checking
Two threads in model checking

Check models

Check code
Model checking code

FSM is the most popular abstraction
Touched by his noodly appendage
Model checking code

FSM is the most popular abstraction
  – Decide what are “states” and “transitions”
Example

**motionPorch:**
- porchLight.Set(On)
- timer.Start(5 mins)

**porchLight.On:**
- timer.Start(5 mins)

**timer.Fired:**
- porchLight.Set(Off)
Exploring input space

motionPorch:
  if (lightLevel < 20)
    porchLight.Set(On)
  timer.Start(10 mins)

porchLight.On:
  timer.Start(5 mins)

timer.Fired:
  porchLight.Set(Off)

To explore comprehensively, must consider all possible values of input parameters

[PorchrLight, Timer] [Off, Off]
  LtLvl=0
  ....
  LtLvl=99

[...]

[Off, Off]

[PorchrLight, Timer] [Off, Off]
  LtLvl=0

[...]

[Off, Off]

[On, On]

LtLvl=19
LtLvl=20
LtLvl=99
Symbolic execution

if (x < 2)
    if (y > 5)
        p = 1;
    else
        p = 2;
else
    if (y > 10)
        p = 3;
    else
        p = 4;

(x, y, p) = (∑x, ∑y, ∑p)

(σ↓x < 2, ∑y, σ↓p)

(σ↓y > 5, σ↓x ≥ 2, ∑p = 1)

(σ↓y > 5, σ↓x < 2, ∑p = 2)

(σ↓y ≤ 5, σ↓x ≥ 2, ∑p = 3)

(σ↓y > 10, σ↓x ≥ 2, ∑p = 4)

(σ↓y ≤ 10, σ↓x ≥ 2, ∑p = 4)
Finding equivalent inputs using symbolic execution

1. Symbolically execute each trigger
2. Find input ranges that lead to same state

\textbf{motionPorch:}
\begin{verbatim}
if (lightMeter.level < 20)
    porchLight.Set(On)
    timer.Start(5 mins)
\end{verbatim}

\textbf{porchLight.On:}
\begin{verbatim}
timer.Start(5 mins)
\end{verbatim}

\textbf{timer.Fired:}
\begin{verbatim}
porchLight.Set(Off)
\end{verbatim}
Finding equivalent inputs using symbolic execution

1. Symbolically execute each trigger
2. Find input ranges that lead to same state

```
motionPorch:
    x = lightMeter.Level  
    porchLight.On:
        timer.Start(5 mins)  

porchLight.Off:
    timer.Fired:
        porchLight.Set(Off)
```

\[ LtLvI = 0 \quad \ldots \quad LtLvI = 9 \]
Efficiently exploring the input space

Pick random values in equivalent classes

**motionPorch:**

if (lightMeter.level < 20)  
porchLight.Set(On)  
timer.Start(5 mins)

**porchLight.On:**

timer.Start(5 mins)

**timer.Fired:**

porchLight.Set(Off)
Use symbolic execution alone?

Symbolic, path-based

Concrete, state-based

Trigger0
Trigger1
Trigger2
Exploring temporal behavior: soundness

**motionPorch:**

- porchLight.Set(On)
- timerDim.Start(5 mins)
- timerOff.Start(10 mins)

**porchLight.On:**

- timerDim.Start(5 mins)
- timerOff.Start(10 mins)

**timerDim.Fired:**

- porchLight.Set(Dim)

**timerOff.Fired:**

- porchLight.Set(Off)
  
  if timerDim.On()
  
  Abort();
Exploring temporal behavior: completeness

```
motionPorch:
  if (Now - tLastMotion < 60)
    porchLight.Set(On)
    timer.Start(600)
  tLastMotion = Now

porchLight.On:
  timer.Start(600)

timer.Fired:
  porchLight.Set(Off)
```

To explore comprehensively, must fire all possible events at all possible times
**Trigger0:**
- tTrigger1 = Now
- tTrigger2 = Now
- trigger1Seen = false

**Trigger1:**
- if (Now – tTrigger1 < 5)
  - trigger1Seen = true
- tTrigger1 = Now

**Trigger2:**
- if (trigger1Seen)
  - if (Now – tTrigger2 < 2)
    - DoSomething()
  - else
    - DoSomethingElse()
**Trigger0:**
  
tTrigger1 = Now  
tTrigger2 = Now  
trigger1Seen = false

**Trigger1:**
  
if (Now – tTrigger1 < 5)  
  trigger1Seen = true  
tTrigger1 = Now

**Trigger2:**
  
if (trigger1Seen)  
  if (Now – tTrigger2 < 2)  
    DoSomething()  
  else  
    DoSomethingElse()
The tyranny of “all possible times”
Timed automata

FSM (states, transitions) + the following:

- Finite number of real-values clocks (VCs)
- All VCs progress at the same rate, except that one or more VCs may reset on a transition
- VC constraints gate transitions
**Trigger0:**
  
tTrigger1 = Now  
tTrigger2 = Now  
trigger1Seen = false

**Trigger1:**
  
if (Now – tTrigger1 < 5)  
  trigger1Seen = true  
tTrigger1 = Now

**Trigger2:**
  
if (trigger1Seen)  
  if (Now – tTrigger2 < 2)  
    DoSomething()  
  else  
    DoSomethingElse()
Properties of timed automata

If VC constraints are such that:

No arithmetic operation involving two VCs
No multiplication operation involving a VC
No irrational constants in constraints

Time can be partitioned into equivalence regions

- 28 regions
  - Corner points (6)
  - Line segments (14)
  - Spaces (8)
**Trigger0:**

\[ t\text{Trigger1} = \text{Now} \]
\[ t\text{Trigger2} = \text{Now} \]
\[ \text{trigger1Seen} = \text{false} \]

**Trigger1:**

\[ \text{if} \ (\text{Now} - t\text{Trigger1} < 5) \]
\[ \quad \text{trigger1Seen} = \text{true} \]
\[ t\text{Trigger1} = \text{Now} \]

**Trigger2:**

\[ \text{if} \ (\text{trigger1Seen}) \]
\[ \quad \text{if} \ (\text{Now} - t\text{Trigger2} < 2) \]
\[ \quad \text{DoSomething}() \]
\[ \text{else} \]
\[ \quad \text{DoSomethingElse}() \]
Why regions are fine-grained

- t1: (x<2) [x]
- t2: (y<1) [y]
- t3: (x<2, y>1)

Diagram showing regions and transitions.
Region construction

If integer constants and simple constraints (e.g., $x < c$)

Straight lines
\[
\forall x: \{x = c \mid c = 0, 1, \ldots, c \downarrow x\}
\]

Diagonals lines
\[
\forall x, y: \{\text{fract}(x) = \text{fract}(y) \mid x < c \downarrow x, y < c \downarrow y\}
\]

\[x_2 < x_1 + 2\]
Why this construction works

1. $X_1 < 5$
2. $X_2 < 2$
3. $X_1 < 5 \land X_2 > 2$
Why this construction works

1. $X_1 < 5$
2. $X_2 < 2$
3. $X_1 < 5 \land X_2 > 2$
Exploring a TA

```
Trigger0() [x1, x2]
Trigger1(x1 >= 5) [x1]
Trigger1(x1 < 5) [x1]
Trigger2() [x1, x2]
Trigger2(x2 < 2) [] {DoSomething}
Trigger2(x2 > 2) [] {DoSomethingElse}
```

```
[false]
[true]
```
Exploring a TA

Trigger0 (x1,x2)

Trigger1 (x1 >= 5)

Trigger1 (x1 < 5)

Trigger2 (x2 < 2)

Trigger2 (x2 > 2)

[false]

[true]

[false] x1=0, x2=0

[true] x1=0, x2=0

[true] x1=0.5, x2=0.5

[false] x1=0.5, x2=0.5

[false] x1=1, x2=1

x1=0,

x2=0

x1=0.5,

x2=0.5

x1=1,

x2=1

DoSomething

DoSomethingElse

{DoSomething}

{DoSomethingElse}
Systematically exploring control programs (Lecture II)

Ratul Mahajan
Microsoft Research

Joint work with Jason Croft,
Matt Caesar, and Madan Musuvathi
Recap: The nature of control programs

Collection of rules with triggers and actions

**motionPorch.Detected:**
if (Now - tLastMotion < 1s && lightLevel < 20)
    porchLight.Set(On)
tLastMotion = Now

@6:00:00 PM:
    porchLight.Set(On)

@6:00:00 AM:
    porchLight.Set(Off)

**packetIn:**
entry = new Entry(inPkt.src, inPkt.dst)
if (!cache.Contains(entry))
    cache.Insert(entry, Now)

**CleanupTimer:**
foreach entry in cache
    if (Now - cache[entry] < 5s)
        cache.Remove(entry)
Recap: Timed automata

FSM (states, transitions) + the following:

- Finite number of real-values clocks (VCs)
- All VCs progress at the same rate, except that one or more VCs may reset on a transition
- VC constraints gate transitions
Recap: Properties of timed automata

If VC constraints are such that:

- No arithmetic operation involving two VCs
- No multiplication operation involving a VC
- No irrational constants in constraints

Time can be partitioned into equivalence regions

28 regions
- Corner points (6)
- Line segments (14)
- Spaces (8)
Recap: Region construction

If integer constants and simple constraints (e.g., \( x < c \))

Straight lines

\[ \forall x: \{ x = c \mid c = 0, 1, \ldots, c \downarrow x \} \]

Diagonals lines

\[ \forall x,y: \{ \text{fract}(x) = \text{fract}(y) \mid x < c \downarrow x, \ y < c \downarrow y \} \]

\[ x_2 < x_1 + 2 \]
Recap: Exploring a TA

Trigger0 (\(x1, x2\))
- \([\text{false}]\)
- \([\text{true}]\)

Trigger1 (\(x1 \geq 5\)) [\(x1\)]
- \([\text{false}]\)
- \([\text{true}]\)

Trigger2 (\(x2 < 2\)) [\(x2\)]
- \([\text{false}]\)
- \([\text{true}]\)

\(\delta\)
- \([\text{false}]\)
- \([\text{true}]\)

\(x1=0, x2=0\)
- \([\text{false}]\)
- \([\text{true}]\)

\(x1=0.5, x2=0.5\)
- \([\text{false}]\)
- \([\text{true}]\)

\(x1=1, x2=1\)
- \([\text{false}]\)
- \([\text{true}]\)

\(x1=0, x2=0\)
- \([\text{false}]\)
- \([\text{true}]\)

\(x1=0.5, x2=0.5\)
- \([\text{false}]\)
- \([\text{true}]\)

\(x1=1, x2=1\)
- \([\text{false}]\)
- \([\text{true}]\)
Exploring control programs with TAs

1. Mapping time-related activity to VCs
2. Model devices
3. Construct time regions
4. Compute equivalent classes for inputs
5. Explore states
Mapping to VCs (1/4): Delay measurers

**Trigger1:**

... 
\[ t_{\text{Last}} = \text{Now} \] 
... 

**Trigger2:**

... 
\[ \text{if (Now} - t_{\text{Last}} < 60) \] 
... 

**Trigger1:**

... 
\[ \text{VC}_t_{\text{Last}} = 0 \] 
... 

**Trigger2:**

... 
\[ \text{if (VC}_t_{\text{Last}} < 60) \] 
...
Mapping to VCs (2/4): Periodic timers

timer1.Period = 600
timer1.Event += Timer1Fired
...

Timer1Fired:
  ...

VC_timer1 = 0
...

VC_timer1 == 600:
  ...
  VC_timer1 = 0
Mapping to VCs (2/4): Delayed actions

Trigger1:
...
timer1.Start(600)
...
timer1.Fired:
...

Trigger1:
...
VC_timer1 = 0
...
VC_timer1 == 600:
Mapping to VCs (4/4): Sleep calls

Trigger:

...  
Sleep(10)  
...

Trigger:

...  // pre-sleep actions
VC_sleeper = 0

VC_sleeper == 10:

...  // post-sleep actions
Reducing the number of VCs: Combining periodic timers

timer1.Period = 600
timer1.Event += Timer1Fired
timer2.Period = 800
timer2.Event += Timer2Fired

... 

Timer1Fired:
   ...

Timer2Fired:
   ...

VC_timer = 0

...

VC_timer == 600:
   ...

VC_timer == 800:
   ...
   VC_timer = 0
Reducing the number of VCs: Combining sleep calls

**Trigger:**

- Act1()
- Sleep(5)
- Act2()
- Sleep(10)
- Act3()

**Trigger:**

- Act1()
  
  VC_sleeper = 0
  
  sleep_counter = 1;
  
VC_sleeper == 5:
  
  Act2()

VC_sleeper == 15:
  
  Act3()
Modeling devices

Model a device using one of more key value pairs

- Motion sensor: Single key with binary value
- Dimmer: Single key with values in range [0..99]
- Thermostat: Multiple keys

Keys can be notifying or non-notifying

- Triggers are used for notifying keys

Queries for values are treated as program inputs
Limitations of device modeling

Values can change arbitrarily

Key value pairs of a device are independent

Different devices are independent
Constructing time regions

1. Extract VC constraints using symbolic execution

2. Construct time regions using the constraints

```
Trigger0:
    tTrigger1 = Now
    tTrigger2 = Now
    trigger1Seen = false

Trigger1:
    if (Now - tTrigger1 < 5)
        trigger1Seen = true
    tTrigger1 = Now

Trigger2:
    if (trigger1Seen)
        if (Now - tTrigger2 < 2)
            DoSomething()
        else
            DoSomethingElse()
```
Exploration using TA

Region state = Variables values + VC region + ready timers

1. exploredStates = {}
2. unexploredStates = \{S↓initial\}
3. While (unexploredStates \neq \emptyset)
4. \[S↓i\] = PickNext(UnexploredStates)
5. foreach event in Events, \[S↓i\ . \textit{ReadyTimers}\]
6. foreach input in Inputs
7. \[S↓o\] = Compute(\[S↓i\], event, input)
8. if \((S↓o \notin \text{exploredStates})\) unexploredStates.Add(\[S↓o\])
9. if \((S↓i . \text{ReadyTimers}=\emptyset)\)
10. \[S↓o\] = AdvanceRegion(\[S↓i\])    //also marks ReadyTimers
Optimization: Predicting successor states

**Observation:** Multiple region states can have identical response to a trigger

**Trigger1:**

```java
if (x1 < 5)
    trigger1Seen = true
x1 = 0
```

**Trigger2:**

```java
if (trigger1Seen)
    if (x2 < 2)
        DoSomething()
    else
        DoSomethingElse()
```
Optimization: Predicting successor states

Observation: Multiple region states can have identical response to a trigger

Clock personality: region’s evaluation of clock constraints

Same variable values and ready timers

Different regions but same personality

Compute

Predict
Optimization: Independent control loops

Observation: Control programs tend to have multiple, independent control loops

1. Determine independent sets of variables
2. Explore independent sets independently
DeLorean

Control program
Safety invariants

Front end

Program with virtualized devices

Program analyzer

Clock constraints
Input space classes
Control loops

Explorer

Region states Paths
Demo
Evaluation on ten real home automation programs

<table>
<thead>
<tr>
<th>type</th>
<th>#rules</th>
<th>#devs</th>
<th>SLoC</th>
<th>#VCs</th>
<th>GCD (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1 OmniPro</td>
<td>6</td>
<td>3</td>
<td>59</td>
<td>2</td>
<td>7200</td>
</tr>
<tr>
<td>P2 Elk</td>
<td>3</td>
<td>3</td>
<td>75</td>
<td>2</td>
<td>1800</td>
</tr>
<tr>
<td>P3 MiCasaVerde</td>
<td>6</td>
<td>29</td>
<td>143</td>
<td>2</td>
<td>300</td>
</tr>
<tr>
<td>P4 Elk</td>
<td>13</td>
<td>20</td>
<td>193</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>P5 ActiveHome</td>
<td>35</td>
<td>6</td>
<td>216</td>
<td>14</td>
<td>5</td>
</tr>
<tr>
<td>P6 mControl</td>
<td>10</td>
<td>19</td>
<td>221</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>P7 OmniIIE</td>
<td>15</td>
<td>27</td>
<td>277</td>
<td>6</td>
<td>60</td>
</tr>
<tr>
<td>P8 HomeSeer</td>
<td>21</td>
<td>28</td>
<td>393</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>P9 ISY</td>
<td>25</td>
<td>51</td>
<td>462</td>
<td>6</td>
<td>60</td>
</tr>
<tr>
<td>P10 ISY</td>
<td>90</td>
<td>39</td>
<td>867</td>
<td>6</td>
<td>10</td>
</tr>
</tbody>
</table>
Example bugs

P9-1: Lights turned on even in the absence of motion
   – Bug in conditional clause: used OR instead of AND

P9-2: Lights turned off between sunset and 2AM
   – Interaction between rules that turned lights on and off

P10-1: Dimmer wouldn’t turn on despite motion
   – No rule to cover a small time window

P10-2: One device in a group behaved differently
   – Missing reference to the device in one of the rules
Performance of exploration

Time to “fast forward” the home by one hour
Benefit of successor prediction

Successor prediction yields significant advantage
Comparison with untimed model checking

Untimed model checking reaches many invalid states
Comparison with randomized testing

Random testing misses many valid states
### Exploring OpenFlow programs

<table>
<thead>
<tr>
<th>Program</th>
<th>#devs</th>
<th>SLoC</th>
<th>#VCs</th>
<th>GCD</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAC-Learning Switch (PySwitch)</td>
<td>2 hosts, 2 sw, 1 ctrl</td>
<td>128</td>
<td>&gt;= 6</td>
<td>1</td>
</tr>
<tr>
<td>Web Server Load Balancer</td>
<td>3 hosts, 1 sw, 1 ctrl</td>
<td>1307</td>
<td>&gt;= 4</td>
<td>1</td>
</tr>
<tr>
<td>Energy-Efficient Traffic Engineering</td>
<td>3 hosts, 3 sw, 1 ctrl</td>
<td>342</td>
<td>&gt;= 8</td>
<td>2</td>
</tr>
</tbody>
</table>
Additional challenges in OF programs

packetIn:
    timer = new Timer(5s)
    Insert(timer, inPkt.src, inPkt.dst)

Dynamically created VCs
Variable number of VCs along different paths
Open problems

Handling communicating control programs

Exploring all possible topologies
Summary

Control programs are tricky to debug
  – Interaction between rules
  – Large space of inputs
  – Intimate dependence on time

These challenges can be tackled using
  – Systematic exploration (model checking)
  – Symbolic execution to find equivalent input classes
  – Timed automata based exploration (equivalent times)