

# From propositional to first-order monitoring

Jan-Christoph Küster

Joint work with Andreas Bauer and Gil Vegliach



NICTA Funding and Supporting Members and Partners



Australian Government  
Department of Broadband,  
Communications and the Digital Economy  
Australian Research Council



Australian  
National  
University

UNSW  
THE UNIVERSITY OF NEW SOUTH WALES



Trade &  
Investment



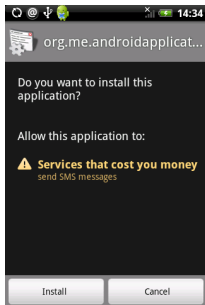
THE UNIVERSITY OF  
SYDNEY



THE UNIVERSITY  
OF QUEENSLAND  
AUSTRALIA

## Why first-order monitoring?

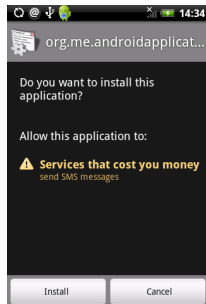
Checking security properties of **systems with data**, e.g., smartphones.



- ▶ Trojan-SMS.AndroidOS.FakePlayer.A
- ▶ While displaying “*Wait, sought access to video library. . .*”, app sends string “798657” in SMS to Russian premium numbers.

# Why first-order monitoring?

Checking security properties of **systems with data**, e.g., smartphones.



- ▶ Trojan-SMS.AndroidOS.FakePlayer.A
- ▶ While displaying “*Wait, sought access to video library...*”, app sends string “798657” in SMS to Russian premium numbers.

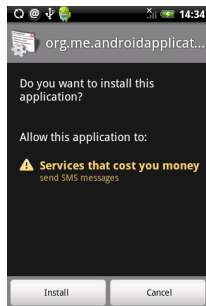
## Security property (syntax)

“Don’t send an SMS to **anyone** not in my contact book”:

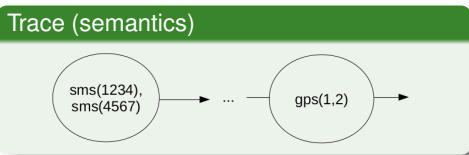
$$\mathbf{G}\forall x : \text{sms. contact}(x)$$

# Why first-order monitoring?

Checking security properties of **systems with data**, e.g., smartphones.



- ▶ Trojan-SMS.AndroidOS.FakePlayer.A
- ▶ While displaying “*Wait, sought access to video library...*”, app sends string “798657” in SMS to Russian premium numbers.



# Spawning automaton

$$\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$$

for  $\varphi \in \text{LTL}^{\text{FO}}$

# Spawning automaton

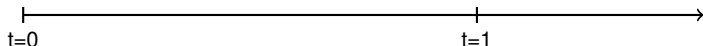
## Example

Once **user**  $u$  has **logged in** to the system from **IP-address**  $ip$ , she must **not send** anything from an IP-address **other than**  $ip$  until **logged out**.

**G**  $\forall(u, ip) : login. \forall(u', ip') : send.$   
 $(eq(u, u') \Rightarrow eq(ip, ip'))$  **U**  $logout(u, ip)$

{  $login(1, 74.125.237.39),$   
 $login(2, 221.199.217.18),$   
 $send(3, 173.252.110.27),$   
 $send(1, 74.125.237.39)$  }

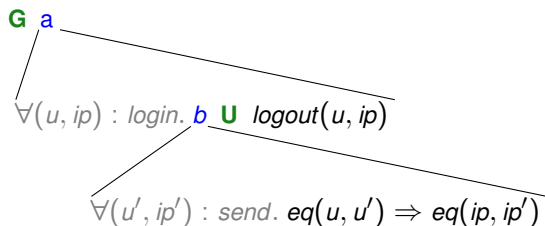
{  $login(6, 82.166.32.78),$   
 $send(1, 74.125.237.39),$   
 $logout(2, 221.199.217.18)$  }



# Spawning automaton

## Example

Once **user**  $u$  has **logged in** to the system from **IP-address**  $ip$ , she must **not send** anything from an IP-address **other than ip** until **logged out**.



# Spawning automaton

## Example



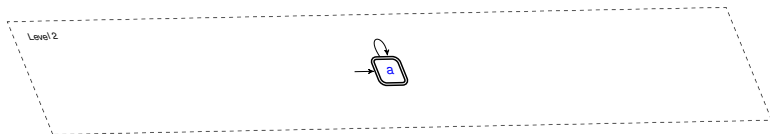
```
{ login(1, 74.125.237.39),  
  login(2, 221.199.217.18),  
  send(3, 173.252.110.27),  
  send(1, 74.125.237.39) }
```





# Spawning automaton

## Example

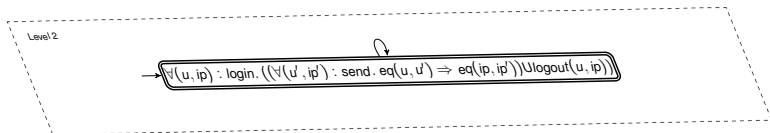


```
{ login(1, 74.125.237.39),  
  login(2, 221.199.217.18),  
  send(3, 173.252.110.27),  
  send(1, 74.125.237.39) }
```



# Spawning automaton

## Example

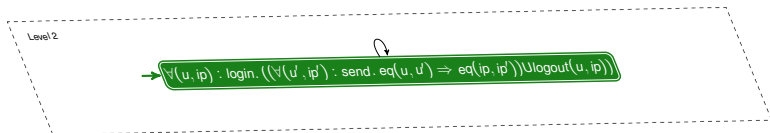


```
{ login(1, 74.125.237.39),  
  login(2, 221.199.217.18),  
  send(3, 173.252.110.27),  
  send(1, 74.125.237.39) }
```



# Spawning automaton

## Example

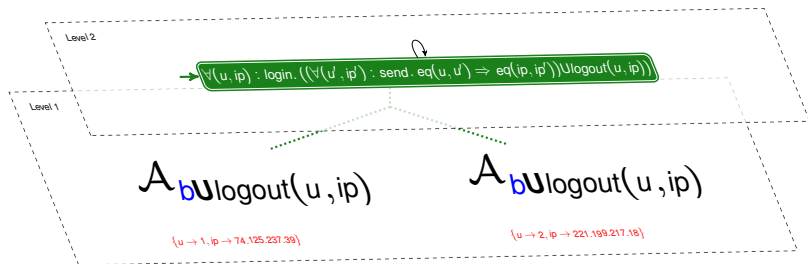


```
{ login(1, 74.125.237.39),  
  login(2, 221.199.217.18),  
  send(3, 173.252.110.27),  
  send(1, 74.125.237.39) }
```



# Spawning automaton

## Example

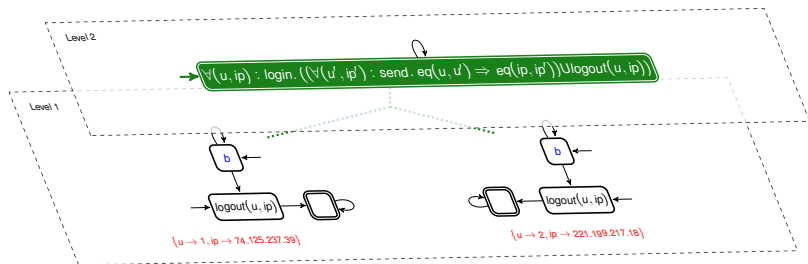


```
{ login(1, 74.125.237.39),  
  login(2, 221.199.217.18),  
  send(3, 173.252.110.27),  
  send(1, 74.125.237.39) }
```

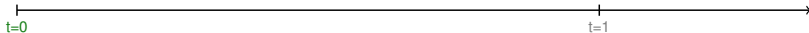


# Spawning automaton

## Example

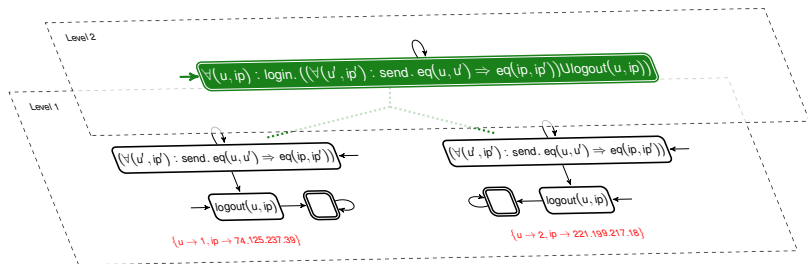


```
{ login(1, 74.125.237.39),  
  login(2, 221.199.217.18),  
  send(3, 173.252.110.27),  
  send(1, 74.125.237.39) }
```



# Spawning automaton

## Example

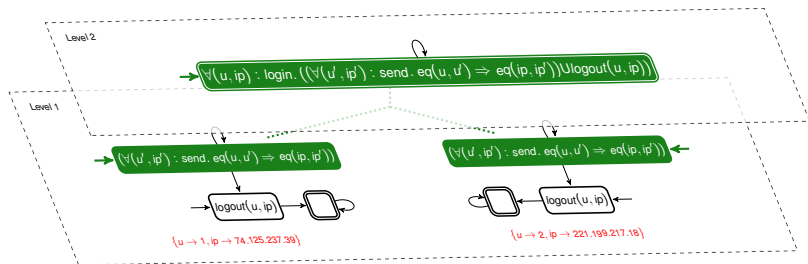


```
{ login(1, 74.125.237.39),  
  login(2, 221.199.217.18),  
  send(3, 173.252.110.27),  
  send(1, 74.125.237.39) }
```



# Spawning automaton

## Example

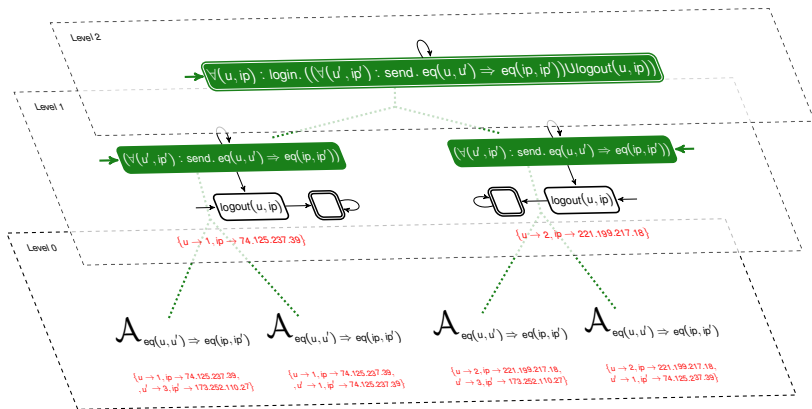


```
{ login(1, 74.125.237.39),  
  login(2, 221.199.217.18),  
  send(3, 173.252.110.27),  
  send(1, 74.125.237.39) }
```



# Spawning automaton

## Example



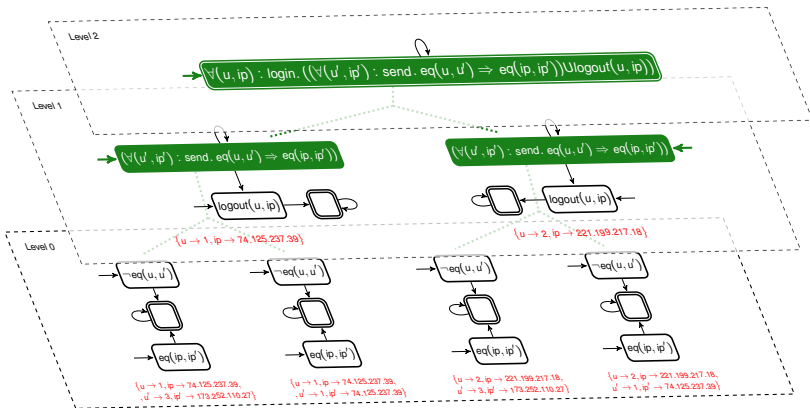
```
{ login(1, 74.125.237.39),
  login(2, 221.199.217.18),
  send(3, 173.252.110.27),
  send(1, 74.125.237.39) }
```





# Spawning automaton

## Example

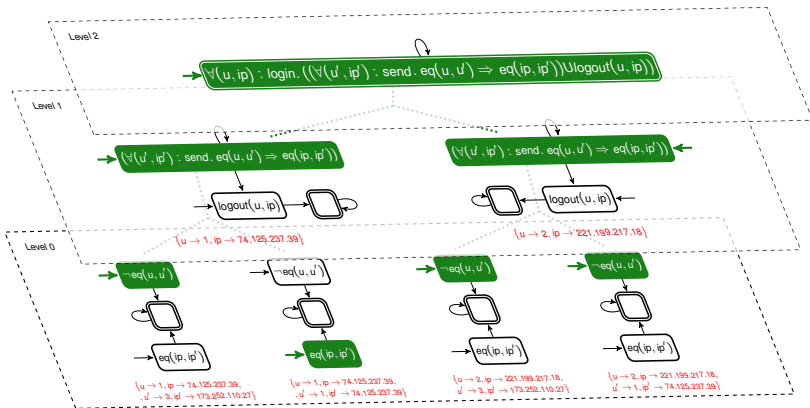


```
{ login(1, 74.125.237.39),
  login(2, 221.199.217.18),
  send(3, 173.252.110.27),
  send(1, 74.125.237.39) }
```



# Spawning automaton

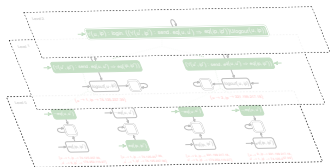
## Example



$\{$  login(1, 74.125.237.39),  
 login(2, 221.199.217.18),  
 send(3, 173.252.110.27),  
 send(1, 74.125.237.39)  $\}$

# Spawning automaton

## Example



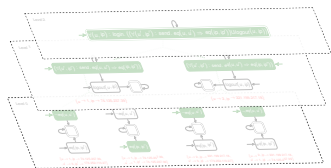
```
{ login(1, 74.125.237.39),  
  login(2, 221.199.217.18),  
  send(3, 173.252.110.27),  
  send(1, 74.125.237.39) }
```

t=0

t=1

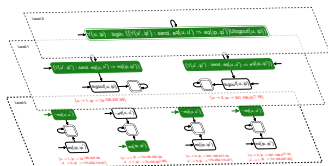
# Spawning automaton

## Example



```
{ login(1, 74.125.237.39),  
  login(2, 221.199.217.18),  
  send(3, 173.252.110.27),  
  send(1, 74.125.237.39) }
```

t=0

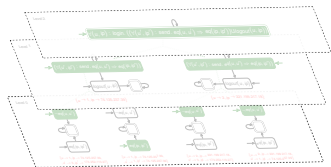


```
{ login(6, 82.166.32.78),  
  logout(2, 221.199.217.18),  
  send(1, 74.125.237.39) }
```

t=1

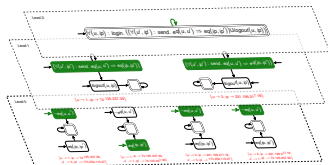
# Spawning automaton

## Example



```
{ login(1, 74.125.237.39),  
  login(2, 221.199.217.18),  
  send(3, 173.252.110.27),  
  send(1, 74.125.237.39) }
```

t=0

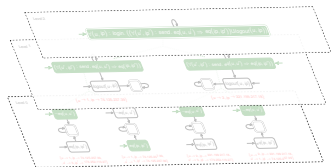


```
{ login(6, 82.166.32.78),  
  logout(2, 221.199.217.18),  
  send(1, 74.125.237.39) }
```

t=1

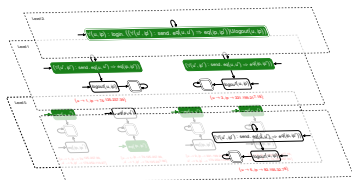
# Spawning automaton

## Example



```
{ login(1, 74.125.237.39),  
  login(2, 221.199.217.18),  
  send(3, 173.252.110.27),  
  send(1, 74.125.237.39) }
```

t=0

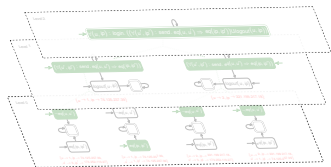


```
{ login(6, 82.166.32.78),  
  logout(2, 221.199.217.18),  
  send(1, 74.125.237.39) }
```

t=1

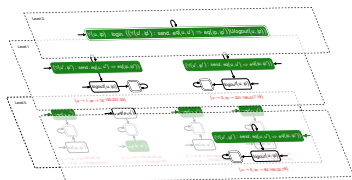
# Spawning automaton

## Example



```
{ login(1, 74.125.237.39),  
  login(2, 221.199.217.18),  
  send(3, 173.252.110.27),  
  send(1, 74.125.237.39) }
```

t=0

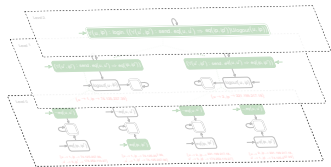


```
{ login(6, 82.166.32.78),  
  logout(2, 221.199.217.18),  
  send(1, 74.125.237.39) }
```

t=1

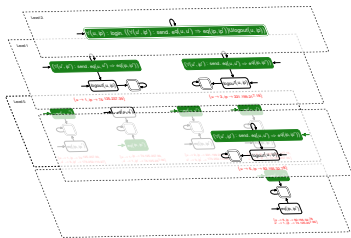
# Spawning automaton

## Example



```
{ login(1, 74.125.237.39),  
  login(2, 221.199.217.18),  
  send(3, 173.252.110.27),  
  send(1, 74.125.237.39) }
```

t=0



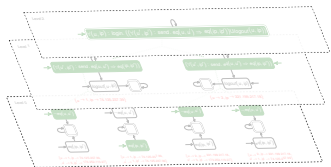
```
{ login(6, 82.166.32.78),  
  logout(2, 221.199.217.18),  
  send(1, 74.125.237.39) }
```

t=1



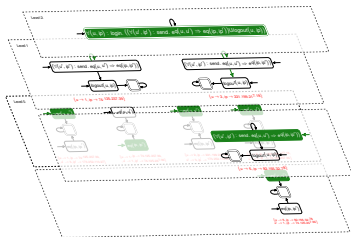
# Spawning automaton

## Example



```
{ login(1, 74.125.237.39),  
  login(2, 221.199.217.18),  
  send(3, 173.252.110.27),  
  send(1, 74.125.237.39) }
```

t=0

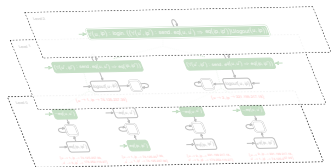


```
{ login(6, 82.166.32.78),  
  logout(2, 221.199.217.18),  
  send(1, 74.125.237.39) }
```

t=1

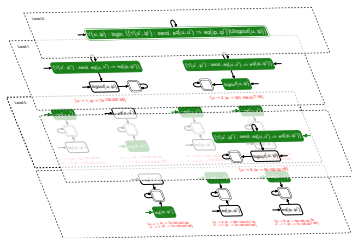
# Spawning automaton

## Example



```
{ login(1, 74.125.237.39),  
  login(2, 221.199.217.18),  
  send(3, 173.252.110.27),  
  send(1, 74.125.237.39) }
```

t=0

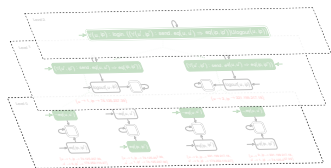


```
{ login(6, 82.166.32.78),  
  logout(2, 221.199.217.18),  
  send(1, 74.125.237.39) }
```

t=1

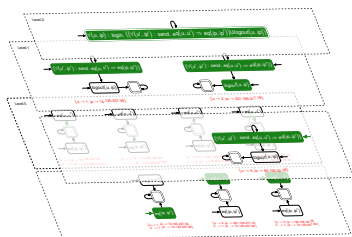
# Spawning automaton

## Example



```
{ login(1, 74.125.237.39),  
  login(2, 221.199.217.18),  
  send(3, 173.252.110.27),  
  send(1, 74.125.237.39) }
```

t=0

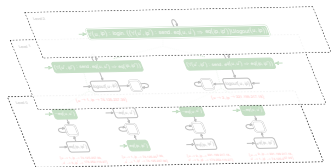


```
{ login(6, 82.166.32.78),  
  logout(2, 221.199.217.18),  
  send(1, 74.125.237.39) }
```

t=1

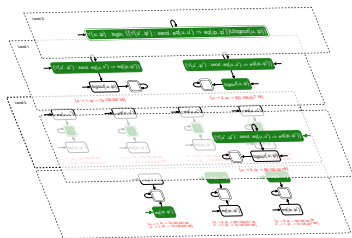
# Spawning automaton

## Example



```
{ login(1, 74.125.237.39),  
  login(2, 221.199.217.18),  
  send(3, 173.252.110.27),  
  send(1, 74.125.237.39) }
```

t=0

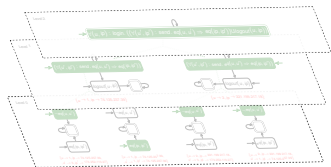
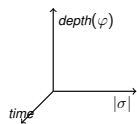


```
{ login(6, 82.166.32.78),  
  logout(2, 221.199.217.18),  
  send(1, 74.125.237.39) }
```

t=1

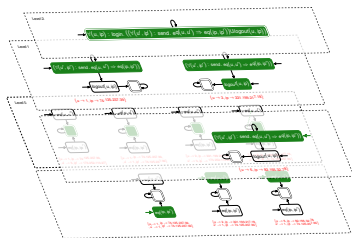
# Spawning automaton

## Example



```
{ login(1, 74.125.237.39),  
  login(2, 221.199.217.18),  
  send(3, 173.252.110.27),  
  send(1, 74.125.237.39) }
```

$t=0$



```
{ login(6, 82.166.32.78),  
  logout(2, 221.199.217.18),  
  send(1, 74.125.237.39) }
```

$t=1$

# Spawning automaton

## Definition

$$\mathcal{A} = (\Sigma, Q, Q_0, \delta_{\rightarrow}, \mathcal{F}, \quad )$$

- ▶  $\Sigma$  an alphabet
- ▶  $Q$  a finite set of states.
- ▶  $Q_0 \subseteq Q$  a set of distinguished initial states.
- ▶  $\delta_{\rightarrow} : Q \times \Sigma \longrightarrow 2^Q$  a transition relation.
- ▶  $\mathcal{F} = \{F_1, \dots, F_n \mid F_i \subseteq Q\}$  an acceptance condition.

# Spawning automaton

## Definition

$$\mathcal{A} = (\Sigma, Q, Q_0, \delta_{\rightarrow}, \mathcal{F}, \ell, \delta_{\downarrow})$$

- ▶  $\Sigma$  an alphabet (**countable set**).
- ▶  $Q$  a finite set of states.
- ▶  $Q_0 \subseteq Q$  a set of distinguished initial states.
- ▶  $\delta_{\rightarrow} : Q \times \Sigma \longrightarrow 2^Q$  a transition relation.
- ▶  $\mathcal{F} = \{F_1, \dots, F_n \mid F_i \subseteq Q\}$  an acceptance condition.
- ▶  $\ell \in \mathbb{N}_0$  is the **level** of  $\mathcal{A}$ .
- ▶  $\delta_{\downarrow} : Q \times \Sigma \longrightarrow \mathcal{B}^+(\mathcal{A}^{<\ell})$  a **spawning function**.

# Spawning automaton

## Definition

$$\mathcal{A} = (\Sigma, Q, Q_0, \delta_{\rightarrow}, \mathcal{F}, \ell, \delta_{\downarrow})$$

- ▶  $\Sigma$  an alphabet (**countable set**).
- ▶  $Q$  a finite set of states.
- ▶  $Q_0 \subseteq Q$  a set of distinguished initial states.
- ▶  $\delta_{\rightarrow} : Q \times \Sigma \rightarrow 2^Q$  a transition relation.
- ▶  $\mathcal{F} = \{F_1, \dots, F_n \mid F_i \subseteq Q\}$  an acceptance condition.
- ▶  $\ell \in \mathbb{N}_0$  is the **level** of  $\mathcal{A}$ .
- ▶  $\delta_{\downarrow} : Q \times \Sigma \rightarrow \mathcal{B}^+(\mathcal{A}^{<\ell})$  a **spawning function**.

## Constructing $\delta_{\downarrow}^{\varphi}$ for $\varphi \in \text{LTL}^{\text{FO}}$

$$\delta_{\downarrow}(q, (\mathfrak{A}, \sigma)) = \left( \bigwedge_{\forall \mathbf{x}: p. \psi \in q} \left( \bigwedge_{(p, \mathbf{d}) \in \sigma} \mathcal{A}_{\psi, v(\mathbf{x}, \mathbf{d})} \right) \right) \wedge \left( \bigwedge_{\neg \forall \mathbf{x}: p. \psi \in q} \left( \bigvee_{(p, \mathbf{d}) \in \sigma} \mathcal{A}_{\neg \psi, v(\mathbf{x}, \mathbf{d})} \right) \right)$$



# Spawning automaton

## When is a run accepting?

### Local acceptance (= Büchi acceptance)

$\mathcal{A}$  accepts exactly those runs in which the set of infinitely often occurring states contains at least a state from each  $F_1, \dots, F_n$ .

### (Global) acceptance

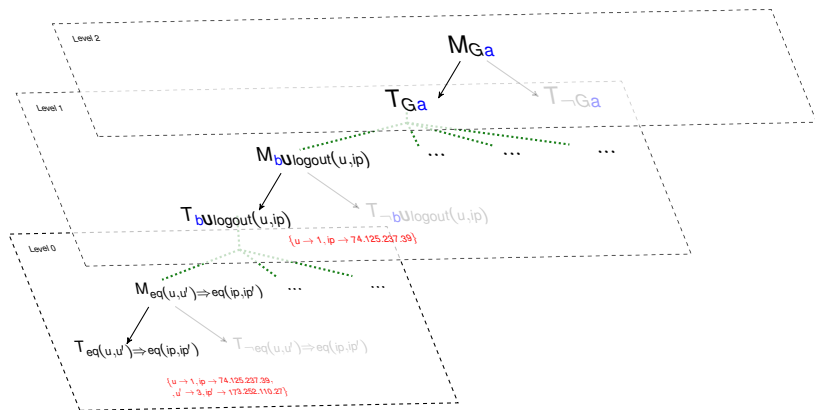
- ▶ if  $\ell = 0$ : local acceptance = acceptance
- ▶ if  $\ell > 0$ : run is local accepting and for all  $i \in \mathbb{N}_0$  there is a set  $Y \subseteq \mathcal{A}^{<\ell}$ , s.t.  $Y \models \delta_{\downarrow}(\rho(i), w_i)$  and all  $\mathcal{A} \in Y$  have an accepting run over  $w^i$ .

# Monitor

$$M_\varphi(\bar{\mathcal{A}}, u) = \top \Rightarrow (\bar{\mathcal{A}}, u) \in \text{good}(\varphi)$$

(resp. for  $\perp$  and  $\text{bad}(\varphi)$ )

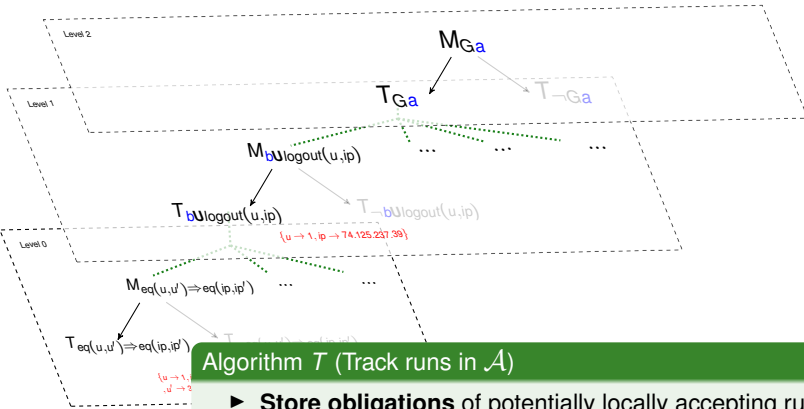
# A monitor for LTL<sup>FO</sup>



## Algorithm $M$ (Monitor)

- ▶ Create two instances of **Algorithm T** (for  $\varphi$  and  $\neg\varphi$ )
- ▶ Forward event to  $T_{\varphi,v}$  and  $T_{\neg\varphi,v}$
- ▶ Return  $\perp$  (resp.  $\top$ ) if  $T_{\varphi,v}$  (resp.  $T_{\neg\varphi,v}$ ) has “no runs”, otherwise print “?”

# A monitor for LTL<sup>FO</sup>



## Algorithm $T$ (Track runs in $\mathcal{A}$ )

- ▶ **Store obligations** of potentially locally accepting runs:  
 $\{[\dots, \delta_{\downarrow}(\rho(i), \sigma_i), \dots], \dots\}$
- ▶ **Execute  $M$**  for each  $\mathcal{A}$  in  $\delta_{\downarrow}(\rho(i), \sigma_i)$  and **wait for result**
- ▶ **Remove runs** if  $\delta_{\downarrow}(\rho(i), \sigma_i) = \perp$
- ▶ **Remove obligations** if  $\delta_{\downarrow}(\rho(i), \sigma_i) = \top$

# Monitor properties

## Trace-length independence

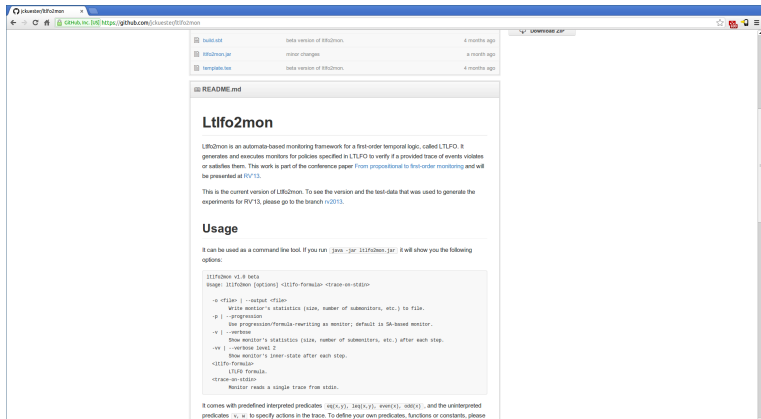
- ▶ If a formula is not trace-length dependent, then the monitor is trace-length independent.
- ▶ Efficiency does not decline with an increasing number of observations.

## Monotonic

- ▶ Once the monitor **returns**  $\perp$  to the user, i.e., has **found a bad prefix**, additional observations do not lead to it **returning**  $\top$  (and vice versa).

# Tool: Ltfo2Mon

<https://github.com/jckuester/ltfo2mon>



The screenshot shows the GitHub repository page for Ltfo2Mon. At the top, there are three release assets:

build.txt	beta version of Ltfo2Mon.	4 months ago
Ltfo2mon.jar	minor changes	3 months ago
templates.txt	beta version of Ltfo2Mon.	4 months ago

Below the assets is the README.md file, which contains the following text:

## Ltfo2mon

Ltfo2mon is an automatic-based monitoring framework for a first-order temporal logic, called LtLFO. It generates and executes monitors for policies specified in LtLFO to verify if a provided trace of events violates or satisfies them. This work is part of the conference paper [From propositional to first-order monitoring and will be presented at RV'13](#).

This is the current version of Ltfo2mon. To see the version and the test-data that was used to generate the experiments for RV'13, please go to the branch [rv2013](#).

### Usage

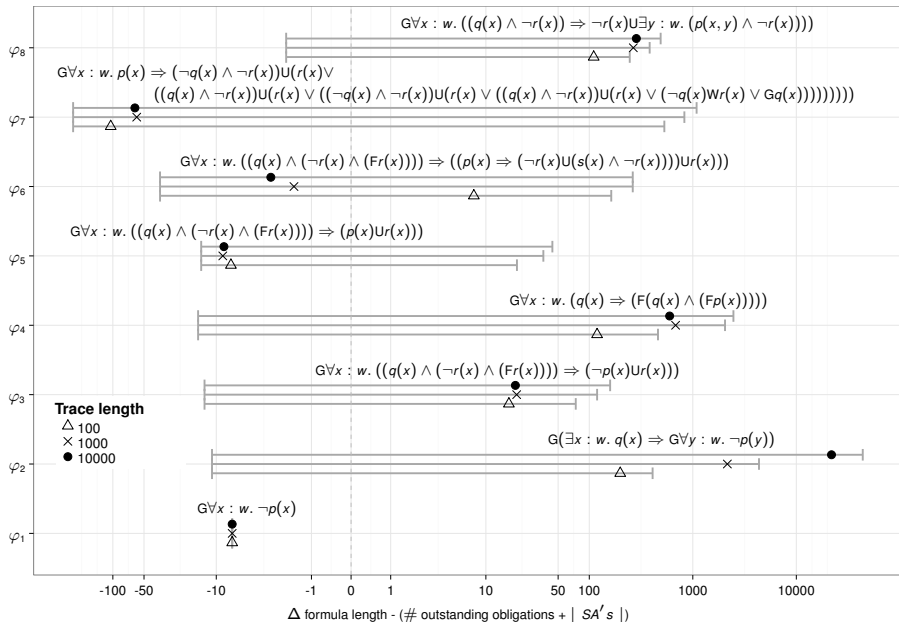
It can be used as a command line tool. If you run `java -jar Ltfo2mon.jar` it will show you the following options:

```
Ltfo2mon v1.0 beta
Usage: Ltfo2mon [options] <ltlfo-formula> <trace-on-stdin>

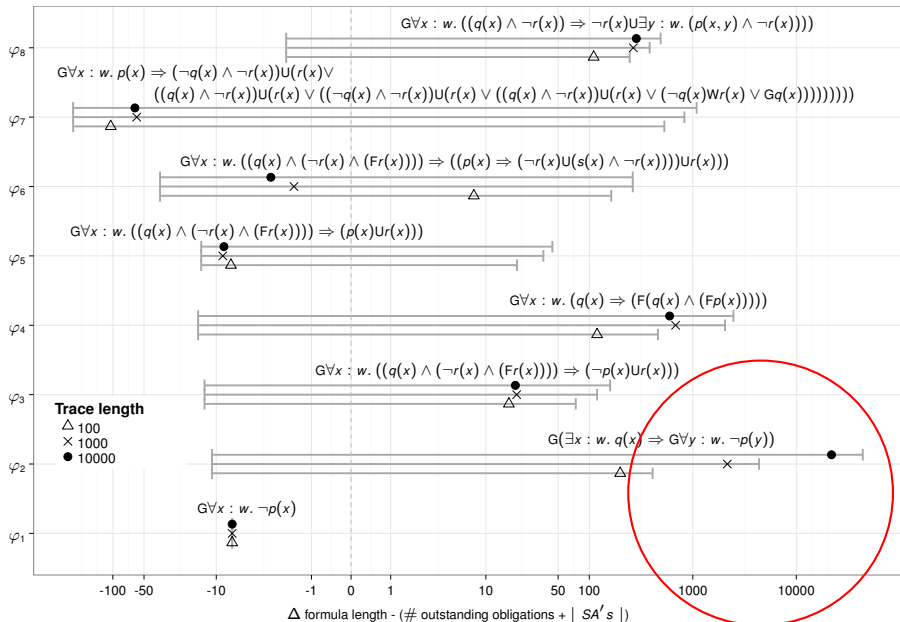
-o <file> | --output <file>
    Write monitor's statistics (size, number of submonitors, etc.) to file.
-p | --progression
    Use progression/formula-rewriting as monitor; default is SA-based monitor.
-v | --verbose
    Show monitor's statistics (size, number of submonitors, etc.) after each step.
-xx | --verbose level 2
    Show monitor's inner-state after each step.
<ltlfo-formula>
    LtLFO formula.
<trace-on-stdin>
    Monitor reads a single trace from stdin.
```

It comes with predefined interpreted predicates `seq(x,y)`, `last(x,y)`, `max(x)`, `min(x)`, and the uninterpreted predicates `x`, `w` to specify actions in the trace. To define your own predicates, functions or constants, please

# SA-based monitor vs. formula rewriting

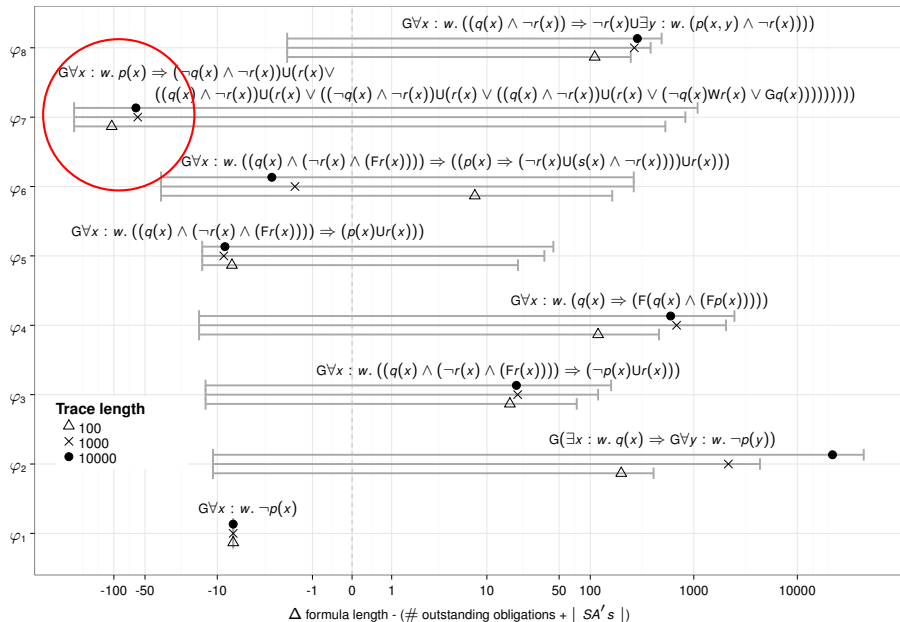


# SA-based monitor vs. formula rewriting





# SA-based monitor vs. formula rewriting



## Complexity results: LTL vs. LTL<sup>FO</sup>

	<b>Satisfiability</b>	<b>Word problem</b>	<b>Model checking</b>	<b>Prefix problem</b>
LTL	PSpace-complete	< Bilinear-time	PSpace-complete	PSpace-complete

## Complexity results: LTL vs. LTL<sup>FO</sup>

	<b>Satisfiability</b>	<b>Word problem</b>	<b>Model checking</b>	<b>Prefix problem</b>
LTL	PSpace-complete	< Bilinear-time	PSpace-complete	PSpace-complete
LTL <sup>FO</sup>	Undecidable	PSpace-complete	ExpSpace-membership, PSpace-hard	Undecidable

# Summary

- ▶ A first-order temporal logic, called  $LTL^{FO}$ , suitable for reasoning about systems with data at runtime.
- ▶ New automaton model called spawning automaton.
- ▶ Monitor algorithm for  $LTL^{FO}$  based on spawning automaton.
  - ▶ <https://github.com/jckuester/ltlfo2mon>
- ▶ Complexity results  $LTL$  vs.  $LTL^{FO}$