TECHNISCHE UNIVERSITÄT WIEN

Execution and monitoring of HOA automata with HOAX

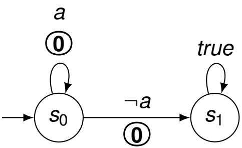
Luca Di Stefano

TU Wien, Austria

RV, September 16, 2025

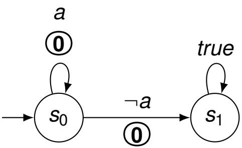
HOA automata

```
HOA: v1
States: 2
Start: 0
AP: 1 "a"
Acceptance: 1 Inf(0)
Alias: @a 0
--BODY--
State: 0 "s0"
[@a] 0 {0}
[!@a] 1 {0}
State: 1 "s1"
[t] 1
--END--
```



HOA automata

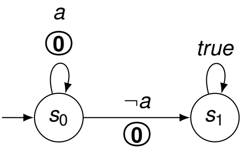
```
HOA: v1
States: 2
Start: 0
AP: 1 "a"
Acceptance: 1 Inf(0)
Alias: @a 0
--BODY--
State: 0 "s0"
[@a] 0 {0}
[!@a] 1 {0}
State: 1 "s1"
[t] 1
--END--
```



$G\ a$

HOA automata

```
HOA: v1
States: 2
Start: 0
AP: 1 "a"
Acceptance: 1 Inf(0)
Alias: @a 0
--BODY--
State: 0 "s0"
[@a] 0 {0}
[!@a] 1 {0}
State: 1 "s1"
[t] 1
--END--
```



$G\ a$

Tools: Owl, Spin, Spot, Prism, SemML, Strix, AutoHyper . . .

Competitions: SYNTCOMP

HOAX - A HOA eXecutor

- Execute automaton based on the semantics of HOA

HOAX - A HOA eXecutor

- Execute automaton based on the semantics of HOA
- Configurable input sources for automata inputs (drivers)
  - User input (prompt)
  - (Biased) random variables
  - File-based

HOAX - A HOA eXecutor

- Execute automaton based on the semantics of HOA
- Configurable input sources for automata inputs (drivers)
  - User input (prompt)
  - (Biased) random variables
  - File-based
- Provide scriptable condition-reaction mechanisms (hooks)
  - If (state reached / trace longer than x / etc.) ...
  - ... Then (reset automaton / print message / etc.)

Basic usage + Sample configuration

Command line

```
$ hoax aut.hoa --config conf.toml
```

conf.toml

```
# Main section (mandatory)
[hoax]
name = "My HOAX config"   # Name for the configuration
version = 1               # Config file version (mandatory)

# Driver for propositions that have none defined
default-driver = "user"   # User prompt (the default)

[[driver.flip]]  # Notice the double brackets
aps = ["a"]      # APs driven by this driver
bias = 0.7       # Bias of "true" (optional, default=0.5)

[[driver.flip]]
# (multiple "flip" drivers may be defined)
aps = ["b", "c"]
```

Hook API example (Python)

- Runner objects track execution of an automaton
- We create Hook objects and append them to Runners

hook_example.py

```python
# Instantiate runner
aut = ...                    # HOA automaton (eg., parsed from file)
conf = DefaultConfig()       # or: conf = TomlConfigV1("config.toml")
run = SingleRunner(aut, conf.driver)

# Instantiate and add hook
condition = Reach(2)         # If state with index=2 is reached...
action = Reset()             # Then reset automaton to initial state
run.add_transition_hook(Hook(condition, action))

# Basic run loop
run.init()
try:
    while True:
        run.step()
except StopRunner:
    pass
```

Monitoring with HOAX

Just a natural application of the tool features

We built Condition classes that turn true iff the run so far is enough to establish that the run will be accepted

Then we merely add them to the runner via hooks

Monitoring with HOAX

Just a natural application of the tool features

We built Condition classes that turn true iff the run so far is enough to establish that the run will be accepted

Then we merely add them to the runner via hooks

Command line

```
$ hoax aut.hoa --config conf.toml --monitor
```

Monitoring with HOAX

Just a natural application of the tool features

We built Condition classes that turn true iff the run so far is enough to establish that the run will be accepted

Then we merely add them to the runner via hooks

Command line

```
$ hoax aut.hoa --config conf.toml --monitor
```

Clearly only some acceptance conditions are actually monitorable. Best effort in general case

Acceptance Conditions in HOA

S = non-empty subset of states (or transitions)

$$\texttt{acc} ::= \mathrm{Inf}(S) \mid \mathrm{Fin}(S) \mid \texttt{acc} \wedge \texttt{acc} \mid \texttt{acc} \vee \texttt{acc}$$

Acceptance Conditions in HOA

S = non-empty subset of states (or transitions)

$$\mathtt{acc} ::= \mathrm{Inf}(S) \mid \mathrm{Fin}(S) \mid \mathtt{acc} \wedge \mathtt{acc} \mid \mathtt{acc} \vee \mathtt{acc}$$

$\mathrm{Inf}(S)$ Visit at least one element of S infinitely often

Acceptance Conditions in HOA

S = non-empty subset of states (or transitions)

$$\texttt{acc} ::= \mathrm{Inf}(S) \mid \mathrm{Fin}(S) \mid \texttt{acc} \wedge \texttt{acc} \mid \texttt{acc} \vee \texttt{acc}$$

$\mathrm{Inf}(S)$ Visit at least one element of S infinitely often

$\mathrm{Fin}(S)$ Do not visit any element of S infinitely often

Trap sets

A trap set T of automaton $\mathcal{A}$ is a non-empty subset of its states st. once a run enters T, it may never leave

- Trap sets may be nested
- Minimal trap sets have no nested trap sets
- They correspond to bottom SCCs of $\mathcal{A}$

Trap sets

A trap set T of automaton $\mathcal{A}$ is a non-empty subset of its states st. once a run enters T, it may never leave

- Trap sets may be nested
- Minimal trap sets have no nested trap sets
- They correspond to bottom SCCs of $\mathcal{A}$

We can build a mapping MINTRAPSETOF from each state of $\mathcal{A}$ to the smallest trap set that contains it

- Reduction from SCC/condensation graph of $\mathcal{A}$
- Indeed, it will return a set of SCCs
- If only 1 SCC, it is a bottom SCC, thus a minimal trap set

Trap sets

A trap set T of automaton $\mathcal{A}$ is a non-empty subset of its states st. once a run enters T, it may never leave

- Trap sets may be nested
- Minimal trap sets have no nested trap sets
- They correspond to bottom SCCs of $\mathcal{A}$

We can build a mapping MINTRAPSETOF from each state of $\mathcal{A}$ to the smallest trap set that contains it

- Reduction from SCC/condensation graph of $\mathcal{A}$
- Indeed, it will return a set of SCCs
- If only 1 SCC, it is a bottom SCC, thus a minimal trap set

Monitoring by comparing current trap to acceptance set

Example: checking for $\mathrm{Inf}(S)$

Execute this algorithm after every step:

Input : Det. complete automaton $\mathcal{A}$; its current state $q \in Q$
Output: good, bad, ugly, or $\perp$.

Comps $\leftarrow$ MinTrapSetOf(q) # $O(|Q|)$
$T \leftarrow \bigcup Comps$ # $O(|Q|)$
if $T \subseteq S$ **then return** good # $O(|S|)$
if $T \cap S = \emptyset$ **then return** bad # $O(|S|)$
if T is minimal **then**
    if $T \setminus S$ is transient **then return** good **else return** ugly
    # $O(|S| + |E_S|)$ `(sub-graph limited to` S`)`
return $\perp$ `# No verdict in this step`

Example: checking for $\mathrm{Inf}(S)$

Execute this algorithm after every step:

Input : Det. complete automaton $\mathcal{A}$; its current state $q \in Q$
Output: good, bad, ugly, or $\bot$.

$Comps \leftarrow$ MinTrapSetOf(q) `#` $O(|Q|)$
$T \leftarrow \bigcup Comps$ `#` $O(|Q|)$
if $T \subseteq S$ **then return** good `#` $O(|S|)$
if $T \cap S = \emptyset$ **then return** bad `#` $O(|S|)$
if T is minimal **then**
  if $T \setminus S$ is transient **then return** good **else return** ugly
  `#` $O(|S| + |E_S|)$ `(sub-graph limited to` S`)`
return $\bot$ `# No verdict in this step`

- Good prefix $\Leftrightarrow$ the run will be accepted

Example: checking for $\mathrm{Inf}(S)$

Execute this algorithm after every step:

Input : Det. complete automaton $\mathcal{A}$; its current state $q \in Q$
Output: good, bad, ugly, or $\bot$.

$Comps \leftarrow$ MinTrapSetOf(q) # $O(|Q|)$
$T \leftarrow \bigcup Comps$ # $O(|Q|)$
if $T \subseteq S$ **then return** good # $O(|S|)$
if $T \cap S = \emptyset$ **then return** bad # $O(|S|)$
if T is minimal **then**
  if $T \setminus S$ is transient **then return** good **else return** ugly
  # $O(|S| + |E_S|)$ (sub-graph limited to S)
return $\bot$ # No verdict in this step

- Good prefix $\Leftrightarrow$ the run will be accepted
- Bad prefix $\Leftrightarrow$ the run will be rejected

Example: checking for $\mathrm{Inf}(S)$

Execute this algorithm after every step:

Input : Det. complete automaton $\mathcal{A}$; its current state $q \in Q$
Output: good, bad, ugly, or $\bot$.

$Comps \leftarrow$ MinTrapSetOf(q) # $O(|Q|)$
$T \leftarrow \bigcup Comps$ # $O(|Q|)$
if $T \subseteq S$ **then return** good # $O(|S|)$
if $T \cap S = \emptyset$ **then return** bad # $O(|S|)$
if T is minimal **then**
  if $T \setminus S$ is transient **then return** good **else return** ugly
  # $O(|S| + |E_S|)$ (sub-graph limited to S)
return $\bot$ # No verdict in this step

- Good prefix $\Leftrightarrow$ the run will be accepted
- Bad prefix $\Leftrightarrow$ the run will be rejected
- X transient $\Leftrightarrow$ every run of $\mathcal{A}$ leaves X infinitely often

Example: checking for $\mathrm{Inf}(S)$

Execute this algorithm after every step:

Input : Det. complete automaton $\mathcal{A}$; its current state $q \in Q$
Output: good, bad, ugly, or $\perp$.

$Comps \leftarrow$ MinTrapSetOf(q) `# ` $O(|Q|)$
$T \leftarrow \bigcup Comps$ `# ` $O(|Q|)$
if $T \subseteq S$ **then return** good `# ` $O(|S|)$
if $T \cap S = \emptyset$ **then return** bad `# ` $O(|S|)$
if T is minimal **then**
  if $T \setminus S$ is transient **then return** good **else return** ugly
  `# ` $O(|S| + |E_S|)$ `(sub-graph limited to` S`)`
return $\perp$ `# No verdict in this step`

- Good prefix $\Leftrightarrow$ the run will be accepted
- Bad prefix $\Leftrightarrow$ the run will be rejected
- X transient $\Leftrightarrow$ every run of $\mathcal{A}$ leaves X infinitely often
- Ugly prefix $\Leftrightarrow$ further monitoring is hopeless

Conclusion

- Configurable tool to execute HOA automata
- Scriptable behaviour through Hooks API
- (Best-effort) monitoring of any acceptance condition

Future work:

- Performance improvements
  - Possibly enabled by Python/cPython evolution
- DSL to script hooks within configuration
- Go beyond Booleans?

Conclusion

- Configurable tool to execute HOA automata
- Scriptable behaviour through Hooks API
- (Best-effort) monitoring of any acceptance condition

Future work:

- Performance improvements
  - Possibly enabled by Python/cPython evolution
- DSL to script hooks within configuration
- Go beyond Booleans?

Try HOAX today!

```
$ pipx install hoax
$ uv tool install hoax
```

https://github.com/lou1306/hoax