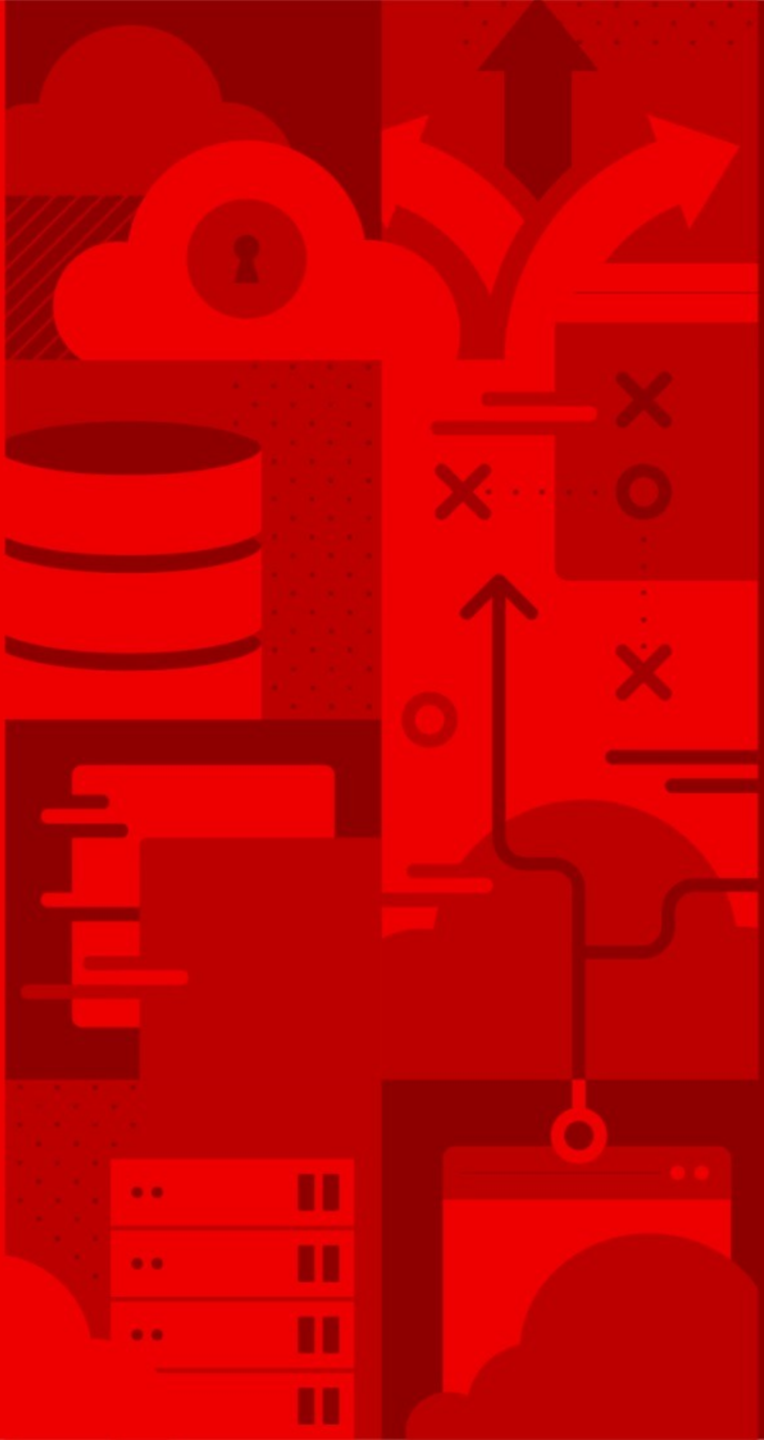
Red Hat Enterprise Linux

Formal verification made easy

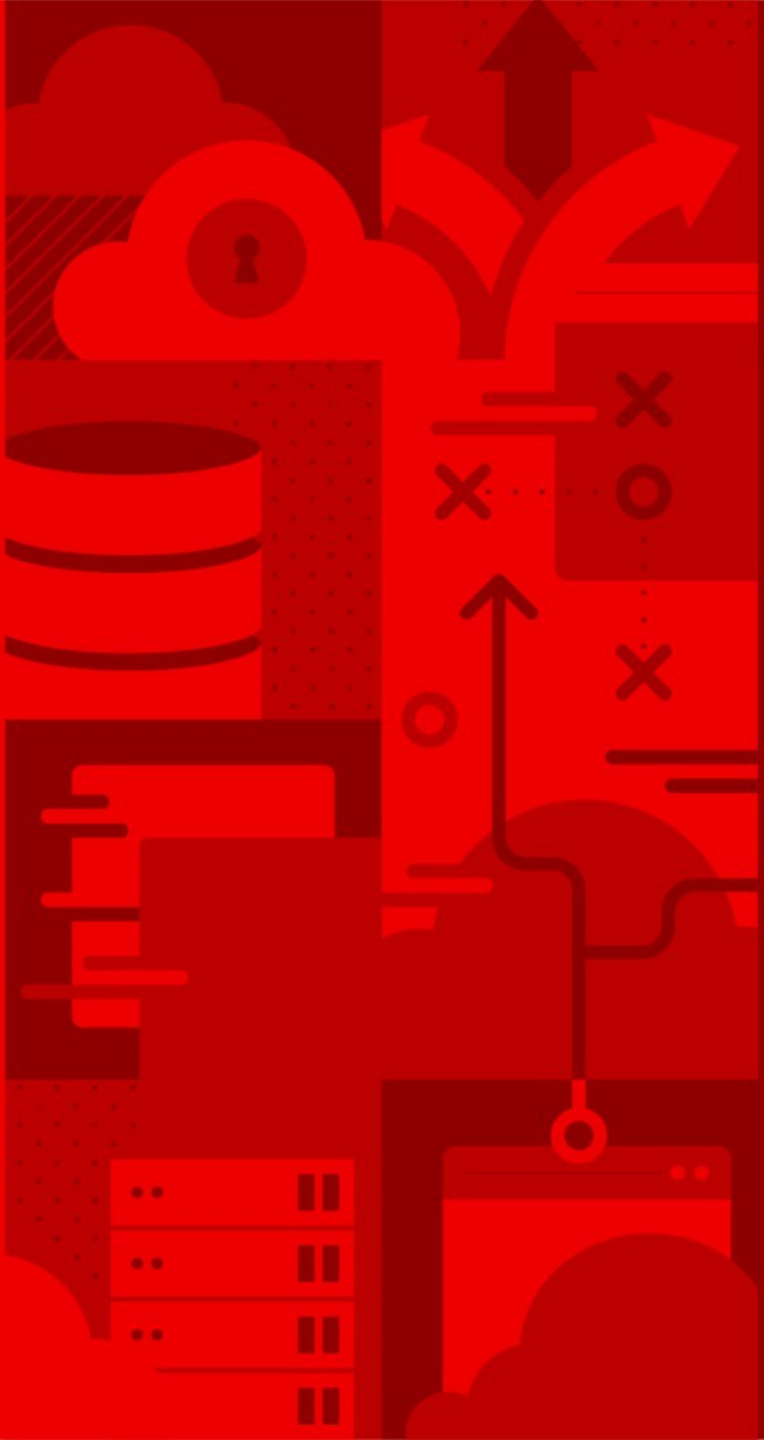
And fast!

Daniel Bristot de Oliveira
Principal Software Engineer

Red Hat



Linux is complex.



Linux is critical.

We need to be sure that Linux _***behaves***_ as _***expected***_ .

What do we _***expect***_?

What do we _*expect*_?

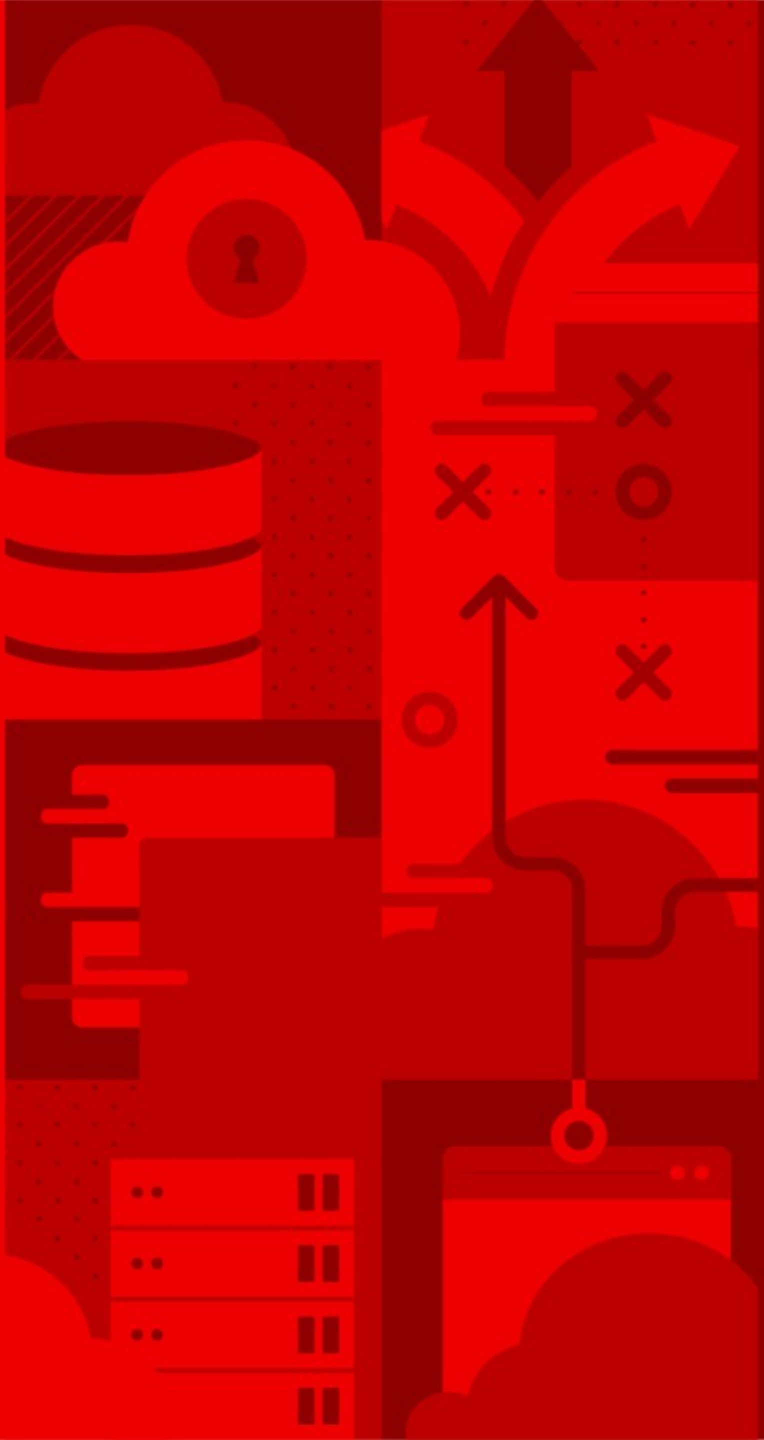
- We have a lot of documentation explaining what is expected!
  - In many different languages!
- We have a lot of “ifs” that asserts what is expected!
- We have lots of tests that check if part of the system behaves as expected!

These things are good. But…

- How do we check that our reasoning is right?
- How do we check that our asserts are not contradictory?
- How do we check that we are covering all cases?

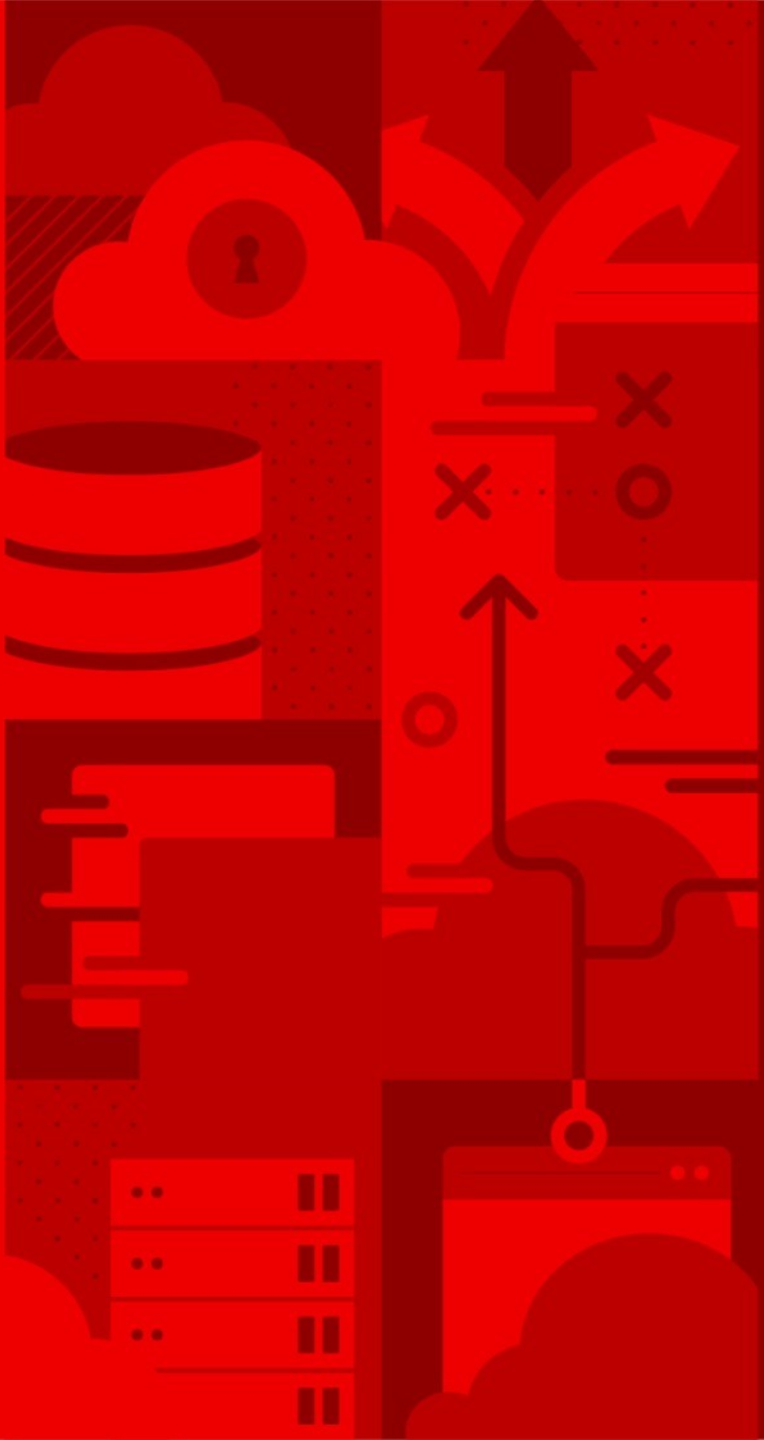
What do we need?

- An intuitive way to describe what we expect
- Using a method that enables the verification of the description
- And a methodology that allows us to cover all “cases”
  - While scaling well...

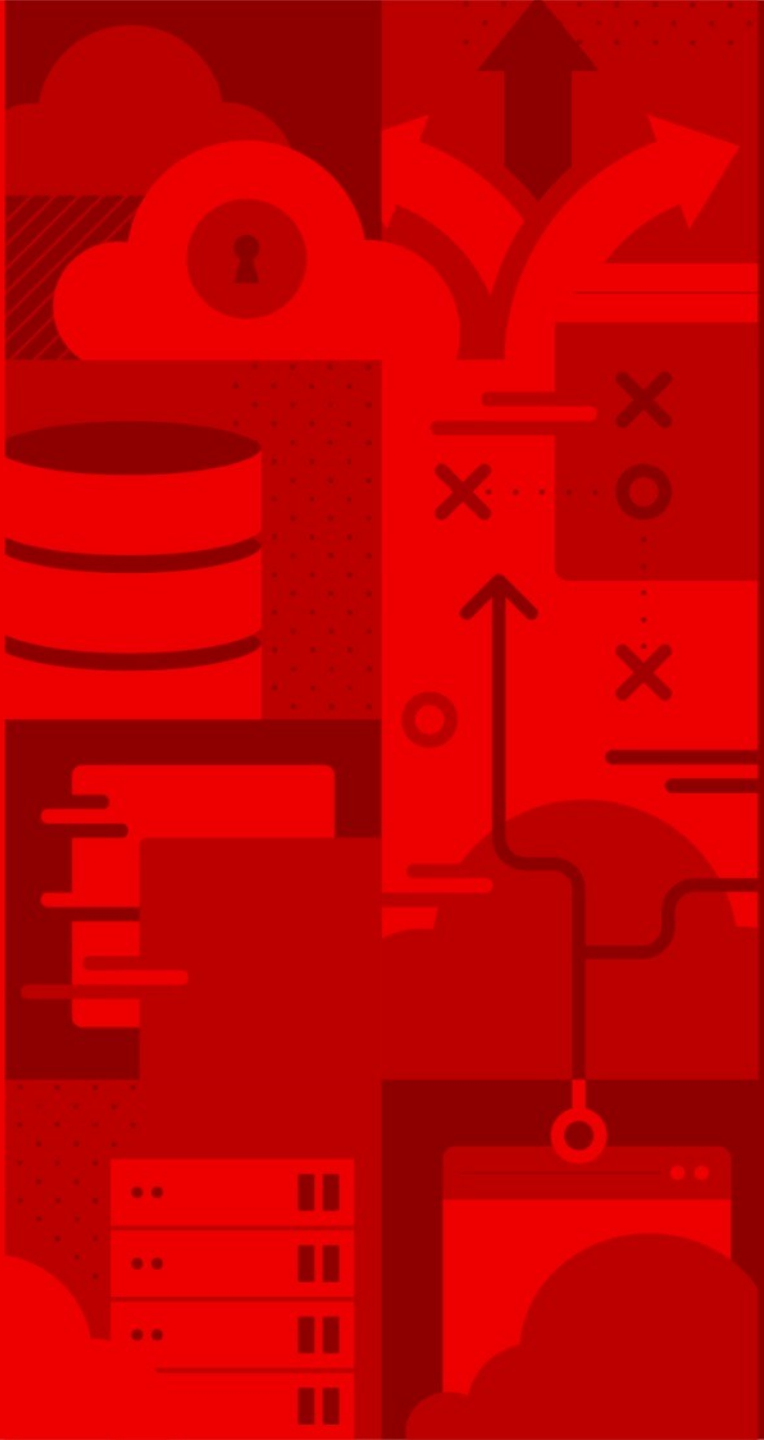


We need formal models.

Red Hat



We already have some examples!



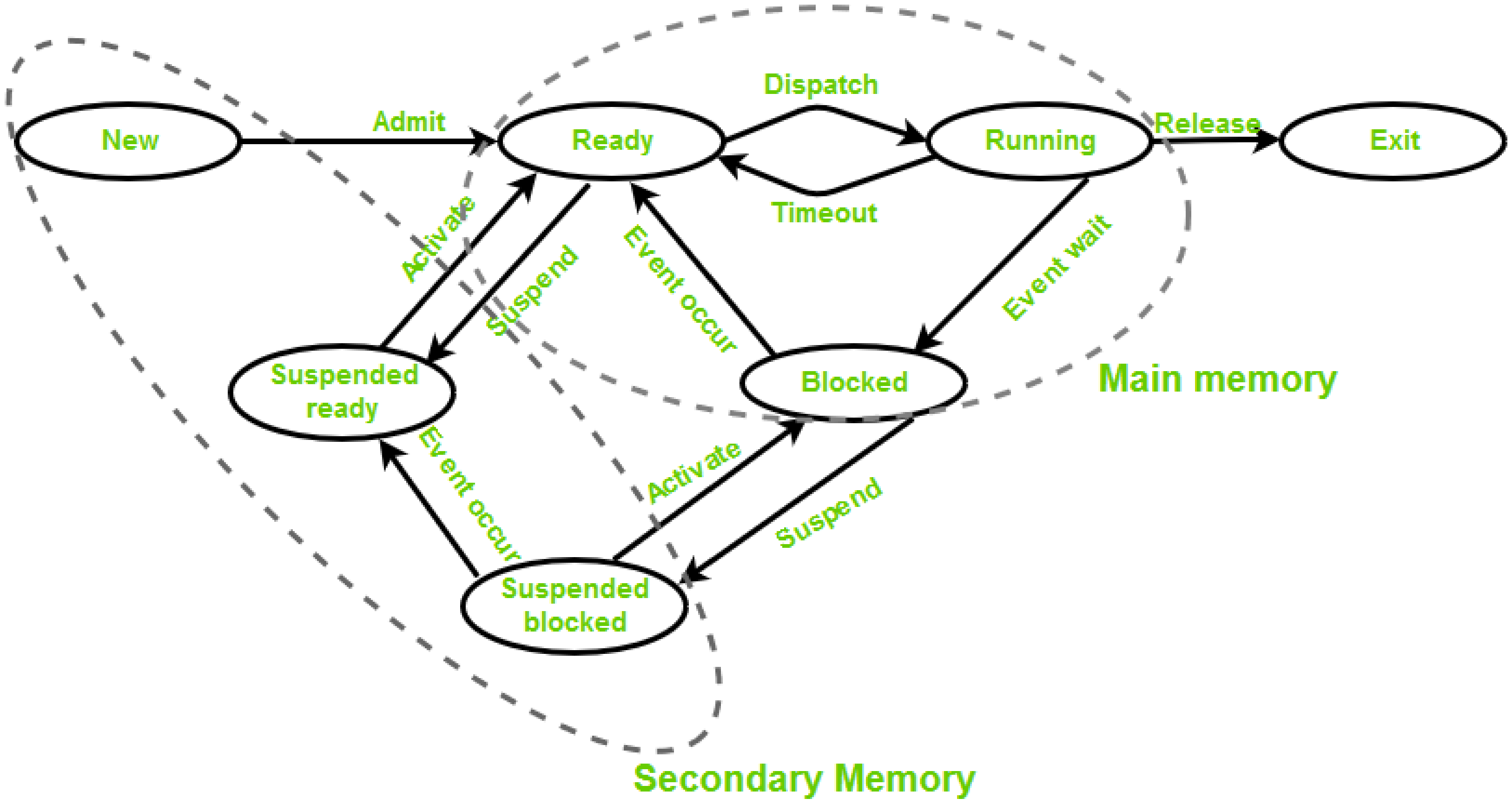
But we need a more “generic” and “intuitive way” for modeling.

How can we turn modeling easier?

- Using a formal method that looks natural for us!
- How do we naturally "observe" the dynamics of Linux?

We trace events!

While tracing we…

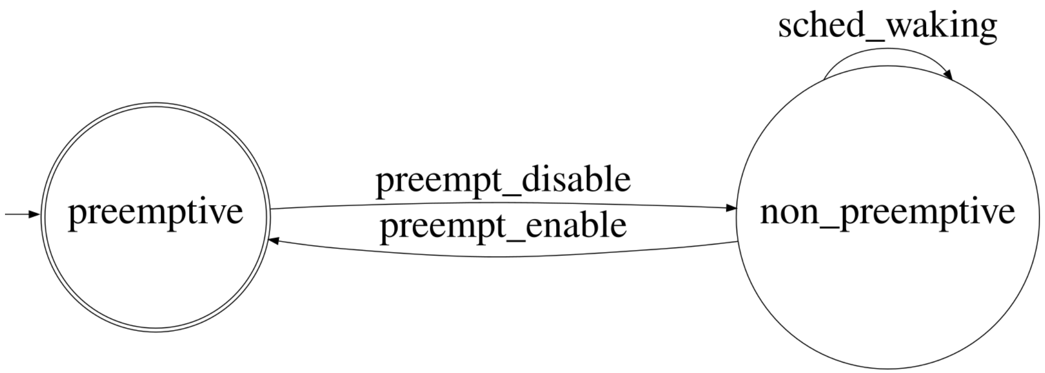


^C^V from
https://www.geeksforgeeks.org/states-of-a-process-in-operating-systems/

State-Machines

- State machines are Event-driven systems
- Event-driven systems describe the system evolution as trace of events
- As we do for run-time analysis.

```
tail-5572  [001] ....1..  2888.401184: preempt_enable: caller=_raw_spin_unlock_irqrestore+0x2a/0x70 parent=         (null)
tail-5572  [001] ....1..  2888.401184: preempt_disable: caller=migrate_disable+0x8b/0x1e0 parent=migrate_disable+0x8b/0x1e0
tail-5572  [001] ....111  2888.401184: preempt_enable: caller=migrate_disable+0x12f/0x1e0 parent=migrate_disable+0x12f/0x1e0
tail-5572  [001] d..h212  2888.401189: local_timer_entry: vector=236
```

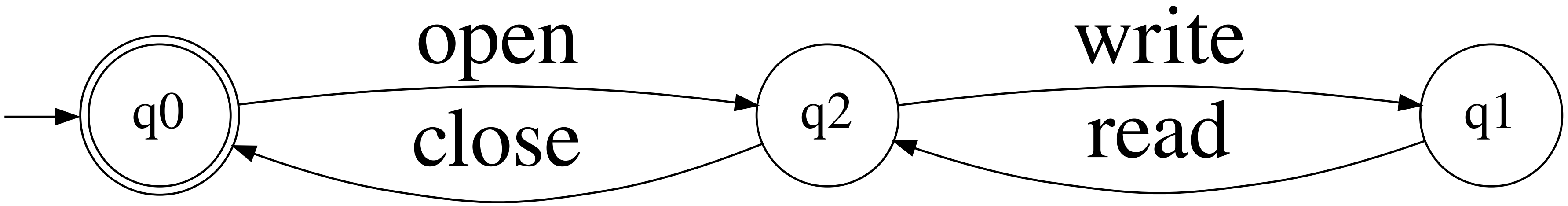


Red Hat

I’ve heard this story before…

This is the continuation of last year’s talk here at LPC:

Using automata as formal language



Red Hat

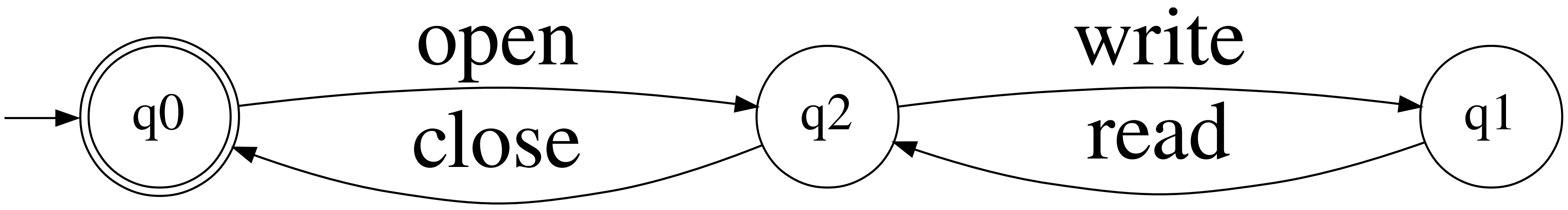
Is formally defined.

- Automata is a method to model Discrete Event Systems (DES)
- Formally, an automaton G is defined as:
  - $G = \{X, E, f, x_0, X_m\}$, where:
    - X = finite set of states;
    - E = finite set of events;
    - *F* is the transition function = (X x E) → X;
    - x_0 = Initial state;
    - X_m = set of final states.
- The language - or traces - generated/recognized by G is the L(G).

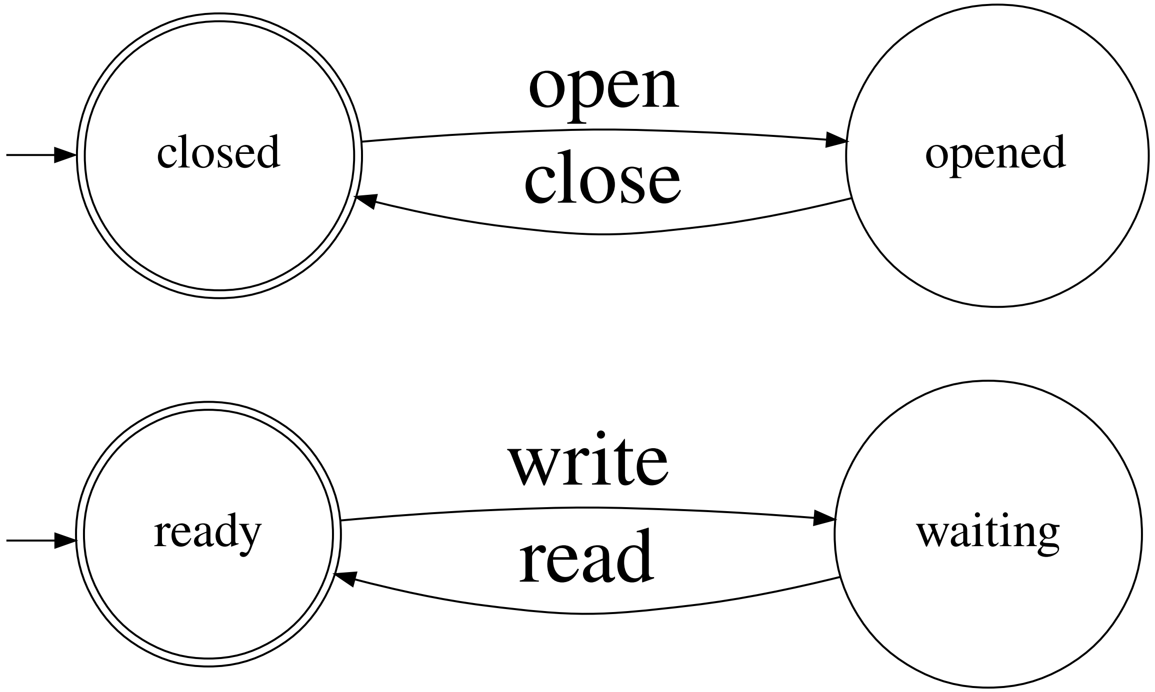
Automata allows

- The verification of the model
  - Deadlock free? Live-lock free?
- Operations
  - Modular development

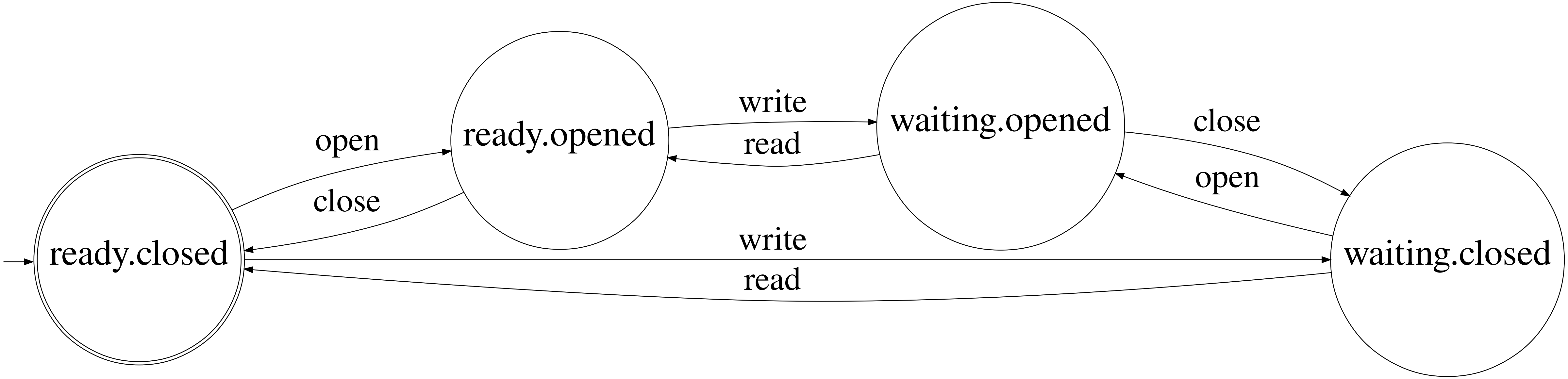
The previous example



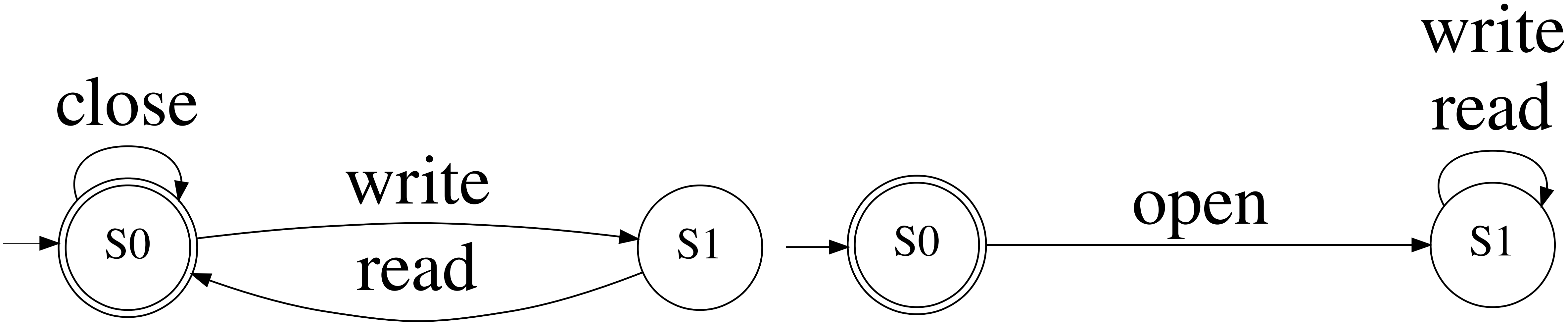
Generators



Sync of generators

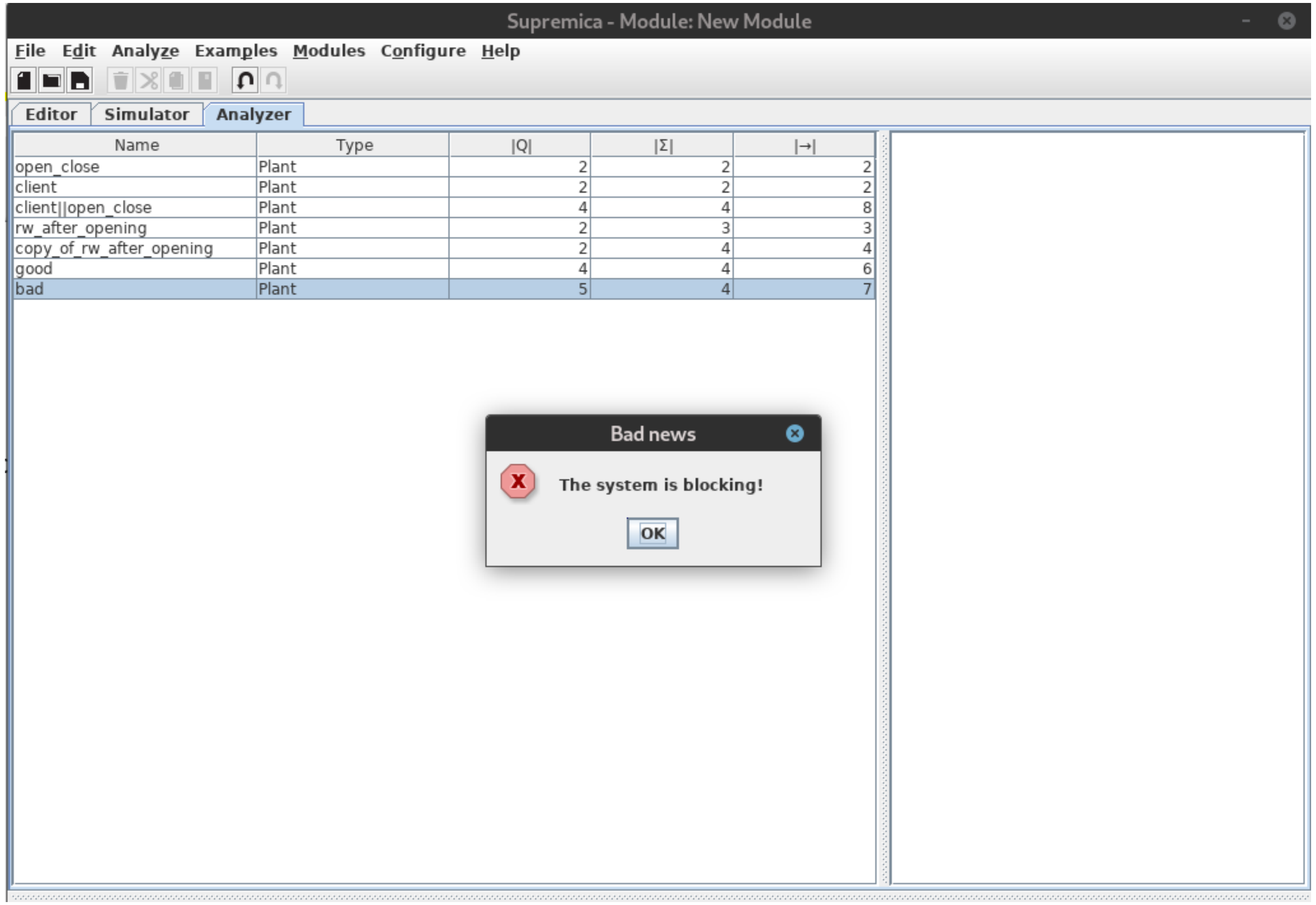


Specification

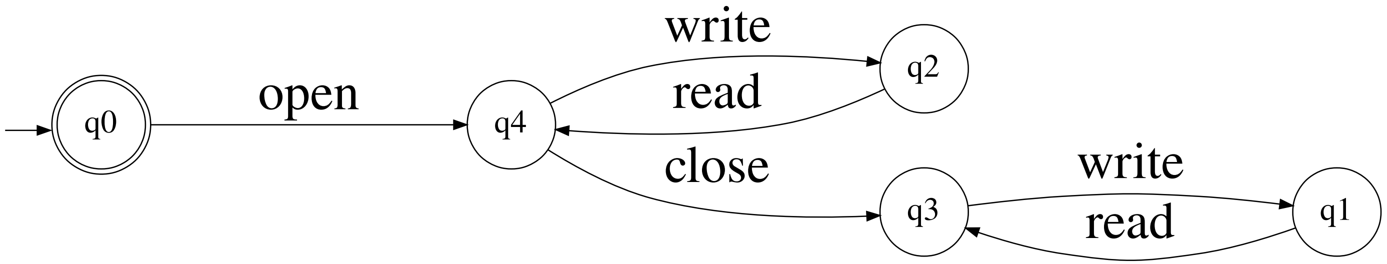


Red Hat

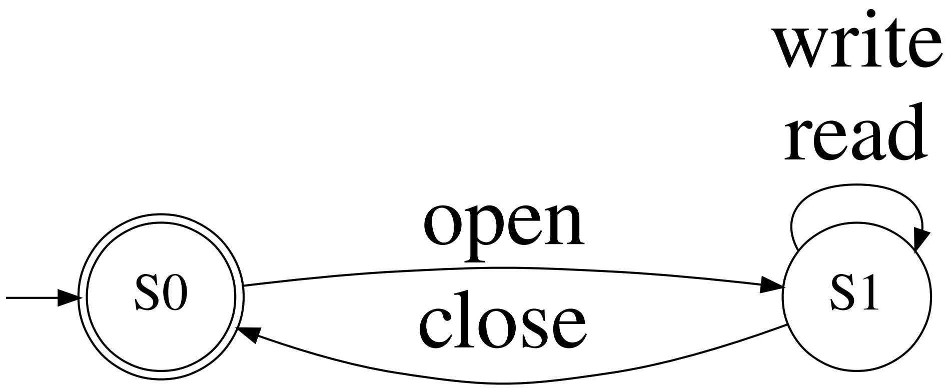
Verification

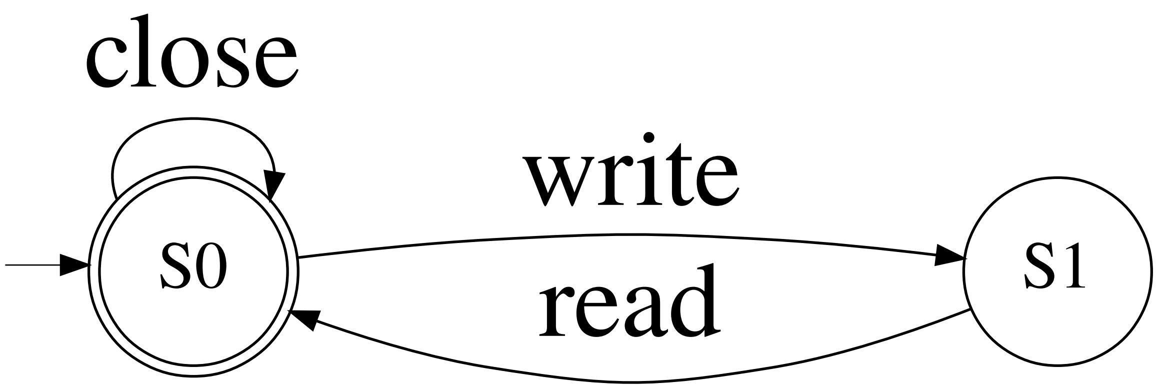


Synch of Generators and Specifications

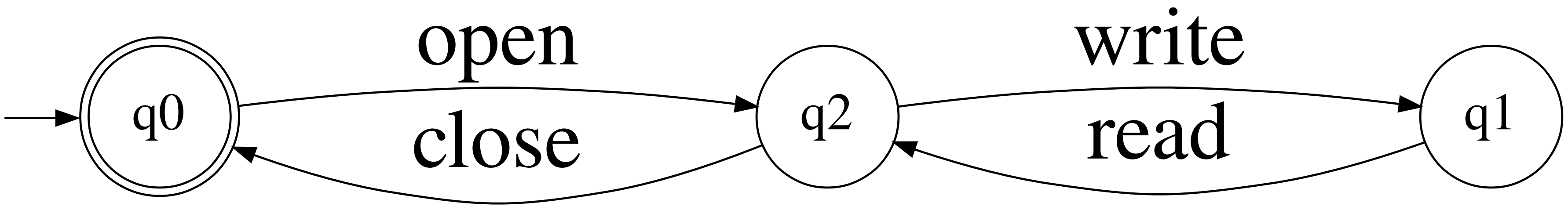


Specifications





Sync of Generators and Specifications



Why not just draw it?

PREEMPT_RT model

- The PREEMPT RT task model has:
  - 12 generators
  - 33 specifications
  - 9017 states!
  - 23103 transitions!
- During development found 3 bugs that would not be detected by other tools...

Academically accepted

Untangling the Intricacies of Thread Synchronization in the PREEMPT_RT Linux Kernel.

Daniel Bristot de Oliveira, Rômulo Silva de Oliveira & Tommaso Cucinotta

2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)

Modeling the Behavior of Threads in the PREEMPT_RT Linux Kernel Using Automata

Daniel Bristot de Oliveira, Tommaso Cucinotta & Romulo Silva De Oliveira

8th Embedded Operating Systems Workshop (EWiLi 2018)

Automata-Based Modeling of Interrupts in the Linux PREEMPT RT Kernel

Daniel Bristot de Oliveira, Rômulo Silva de Oliveira, Tommaso Cucinotta and Luca Abeni

Proceedings of the 22nd IEEE International Conference on Emerging Technologies And Factory Automation (ETFA 2017)

How to verify that the system _***behaves***_?

Comparing system execution against the model!

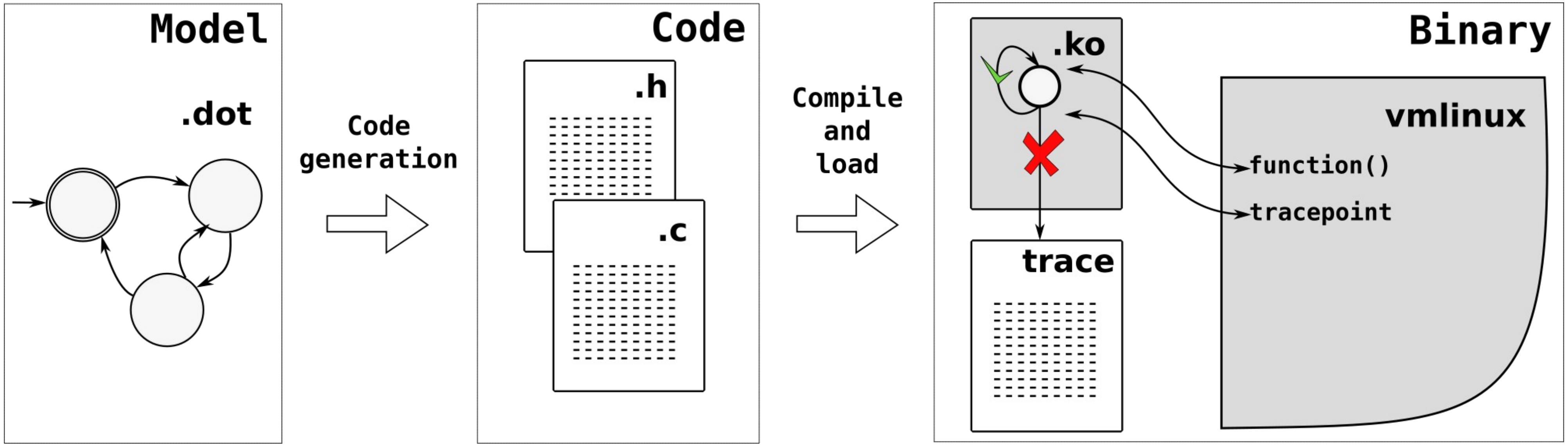
Previous version

Logical correctness for task model

- Example of patch catch'ed with the model
  - [PATCH RT] sched/core: Avoid__schedule() being called twice, the second in vain
- I am doing the model verification in user-space now:
  - Using perf + (sorry, peterz) tracepoints
  - It works, but requires a lot of memory/data transfer:
    - Single core, 30 seconds = 2.5 GB of data
    - We don't need all the data, only from a safe state to the problem.
  - It performs well, because the automata verification is O(1).
  - But still, the amount of data is massive.

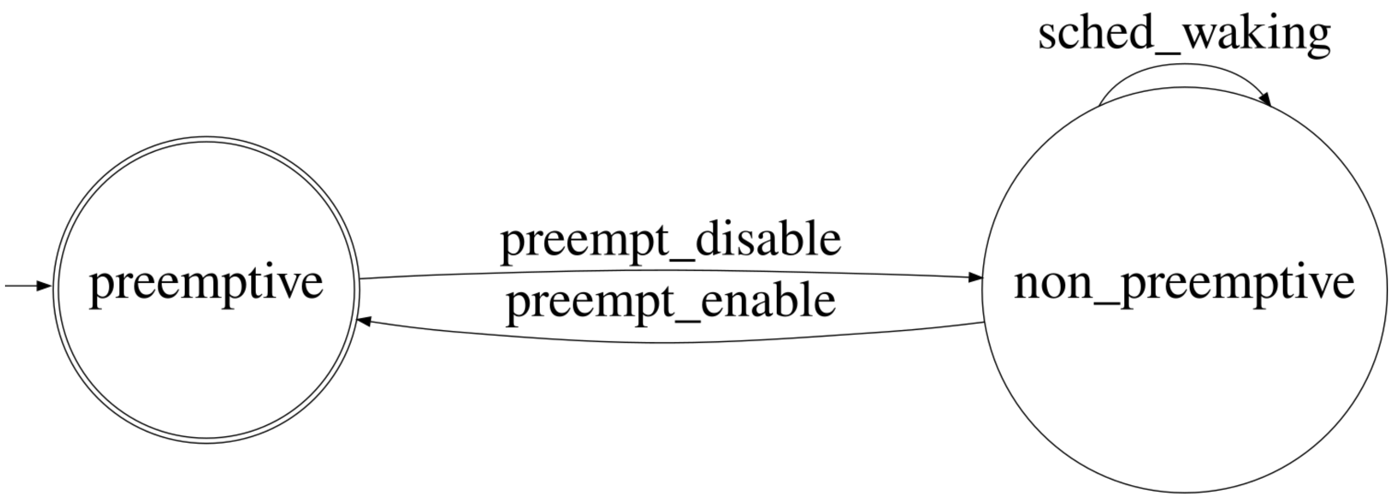
34 redhat.

New approach



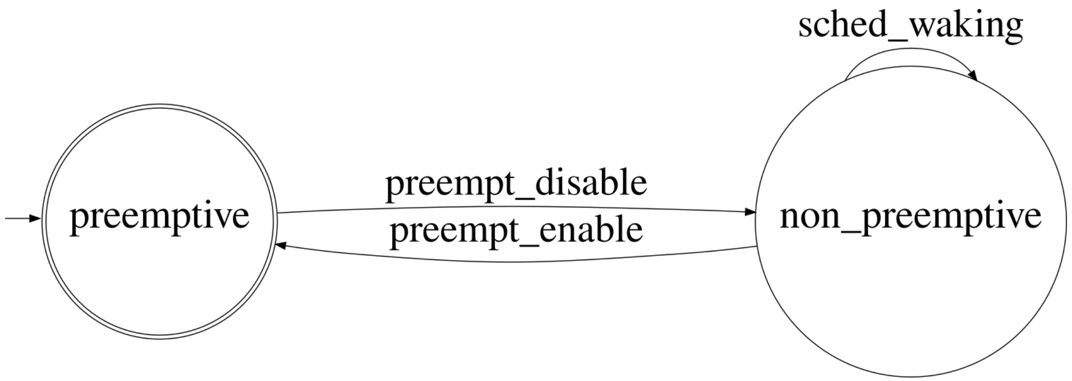
Red Hat

Automata example...



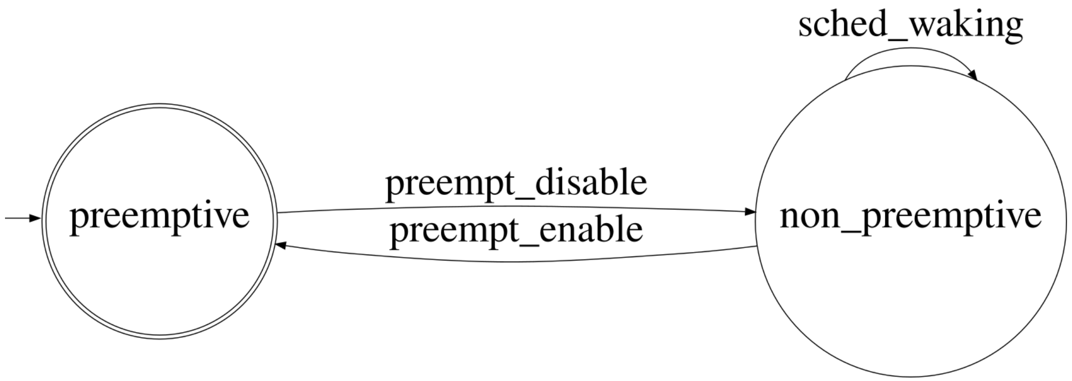
Automaton in C

```
enum states {
            preemptive = 0,
            non_preemptive,
            state_max
};

enum events {
            preempt_disable = 0,
            preempt_enable,
            sched_waking,
            event_max
};

struct automaton {
            char *state_names[state_max];
            char *event_names[event_max];
            char function[state_max][event_max];
            char initial_state;
            char final_states[state_max];
};
```



Automaton in C

```
enum states {
            preemptive = 0,
            non_preemptive,
            state_max
};

enum events {
            preempt_disable = 0,
            preempt_enable,
            sched_waking,
            event_max
};
....
struct automaton aut = {
            .event_names = { "preempt_disable", "preempt_enable", "sched_waking" },
            .state_names = { "preemptive", "non_preemptive" },
            .function = {
                        { non_preemptive,         -1,             -1 },
                        {             -1, preemptive, non_preemptive },
            },
            .initial_state = preemptive,
            .final_states = { 1, 0 }
};
```



Red Hat

Processing one event

```
char process_event(struct verification *ver, enum events event)
{
	int curr_state = get_curr_state(ver);
	int next_state = get_next_state(ver, curr_state, event);

	if (next_state >= 0) {
		set_curr_state(ver, next_state);

		debug("%s -> %s = %s %s\n",
			get_state_name(ver, curr_state),
			get_event_name(ver, event),
			get_state_name(ver, next_state),
			next_state ? "" : "safe!");

		return true;
	}

	error("event %s not expected in the state %s\n",
		get_event_name(ver, event),
		get_state_name(ver, curr_state));

	stack(0);

	return false;
}
```

Processing one event

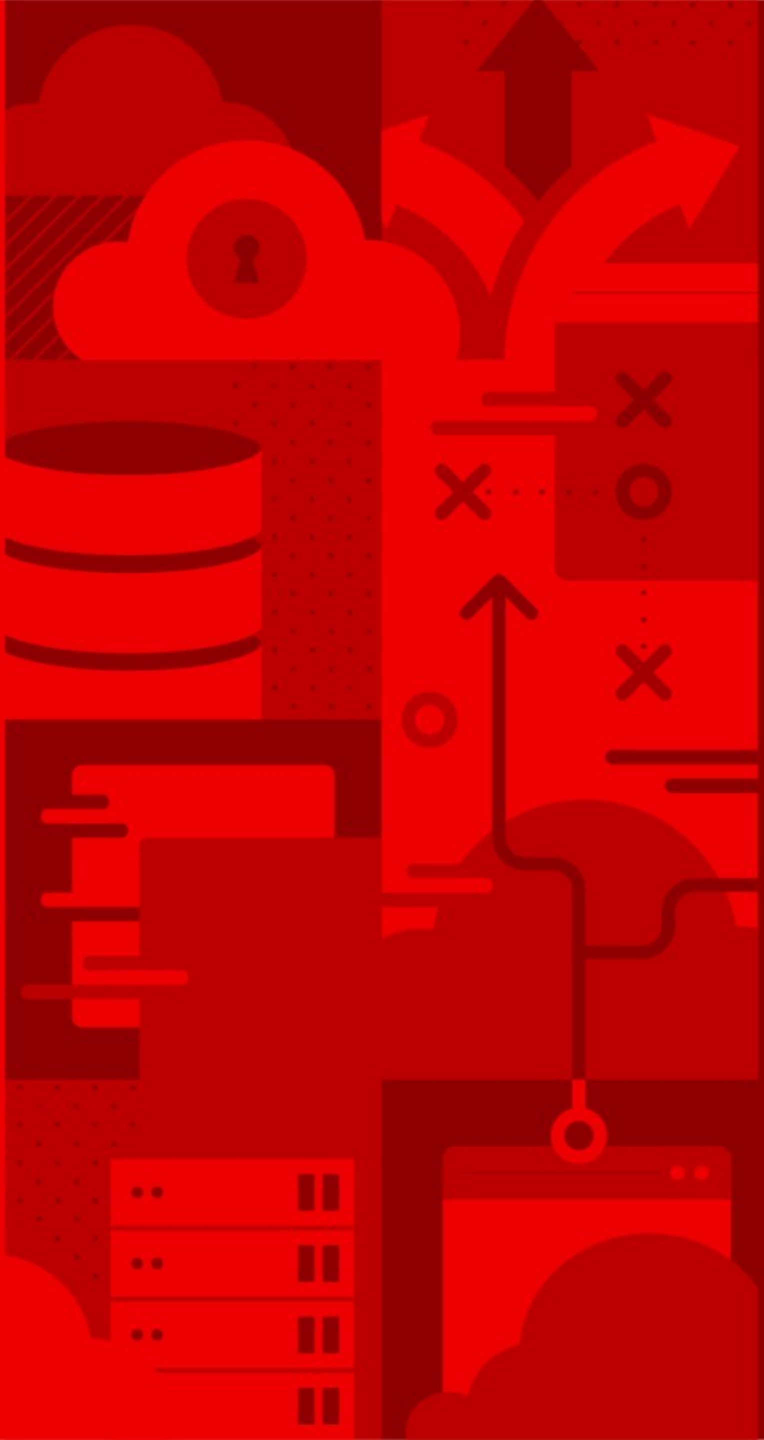
```
char *get_state_name(struct verification *ver, enum states state)
{
            return ver->aut->state_names[state];
}

char *get_event_name(struct verification *ver, enum events event)
{
            return ver->aut->event_names[event];
}

char get_next_state(struct verification *ver, enum states curr_state, enum events event)
{
            return ver->aut->function[curr_state][event];
}

char get_curr_state(struct verification *ver)
{
            return ver->curr_state;

}

void set_curr_state(struct verification *ver, enum states state)
{
            ver->curr_state = state;
}
```

Processing one event

```
char *get_state_name(struct verification *ver, enum states state)
{
            return ver->aut->state_names[state];
}

char *get_event_name(struct verification *ver, enum events event)
{
            return ver->aut->event_names[event];
}

char get_next_state(struct verification *ver, enum states curr_state, enum events event)
{
            return ver->aut->function[curr_state][event];
}

char get_curr_state(struct verification *ver)
{
            return ver->curr_state;

}

void set_curr_state(struct verification *ver, enum states state)
{
            ver->curr_state = state;
}
```

All operations are O(1)!

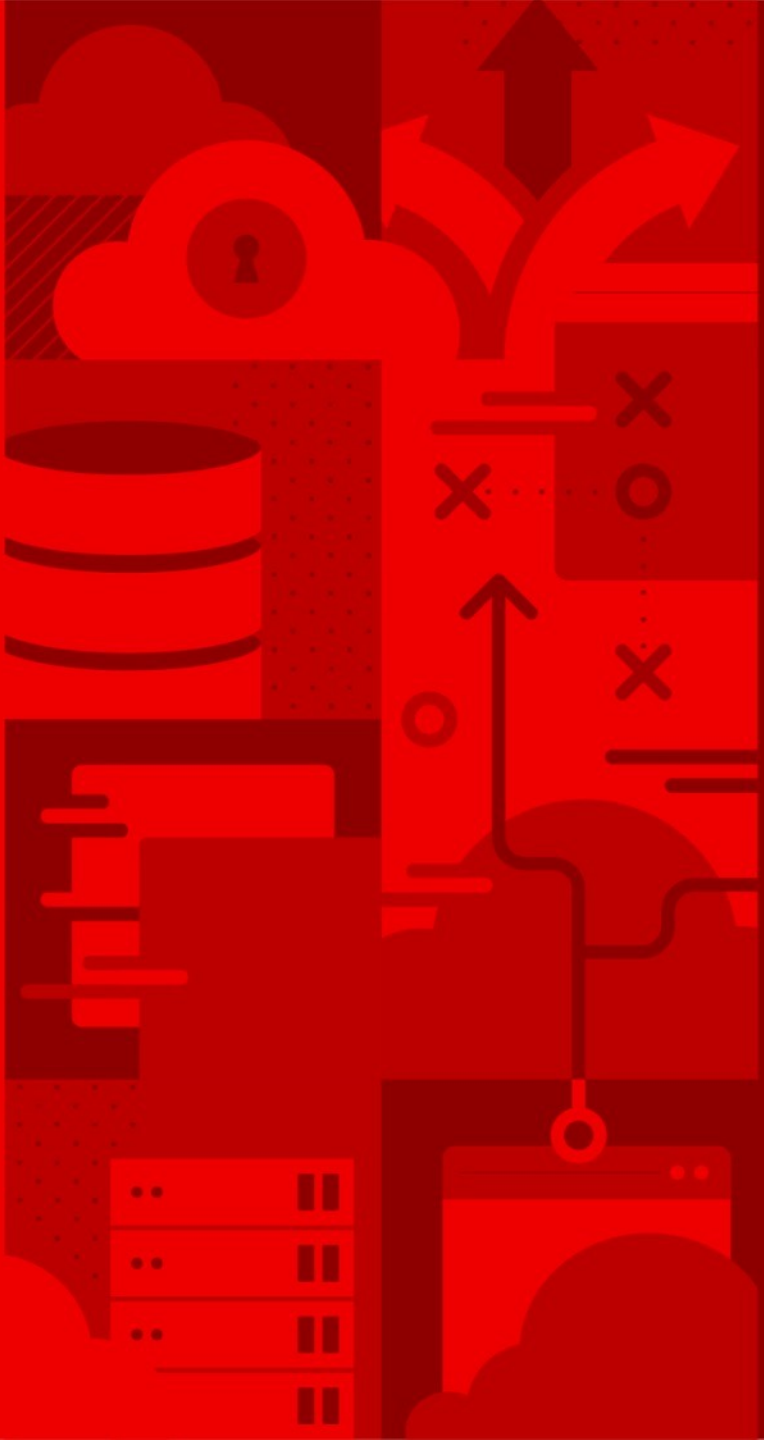
Only one variable to keep the state!



There is not free meal!

The price is in the data structure

- The vectors and matrix are not “compact” data structure
- BUT!
- The PREEEMPT_RT model, with:
  - 9017 states!
  - 23103 transitions!
  - Compiles in a module with < 800KB
  - **Acceptable, no?**

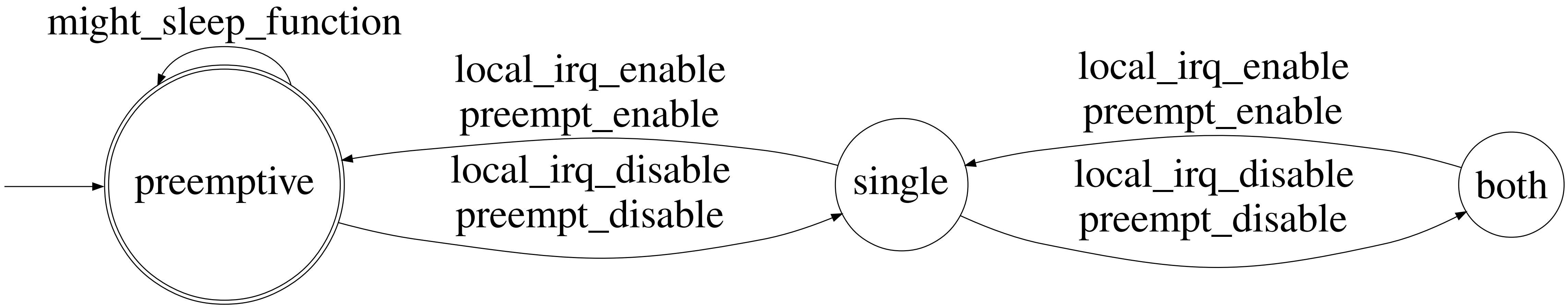


How _***efficient***_ is this ideia?

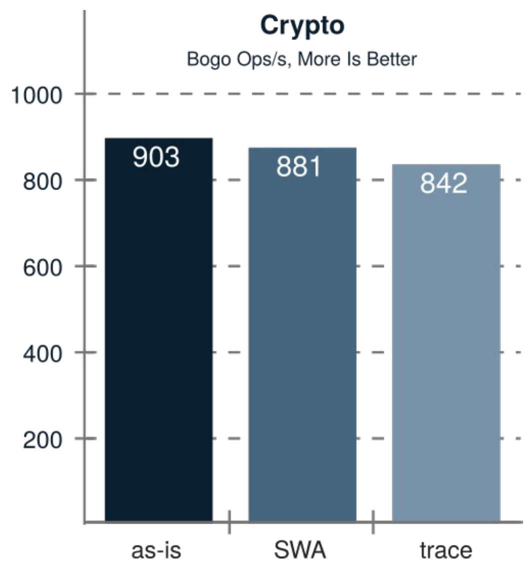
Efficiency in practice: a benchmark

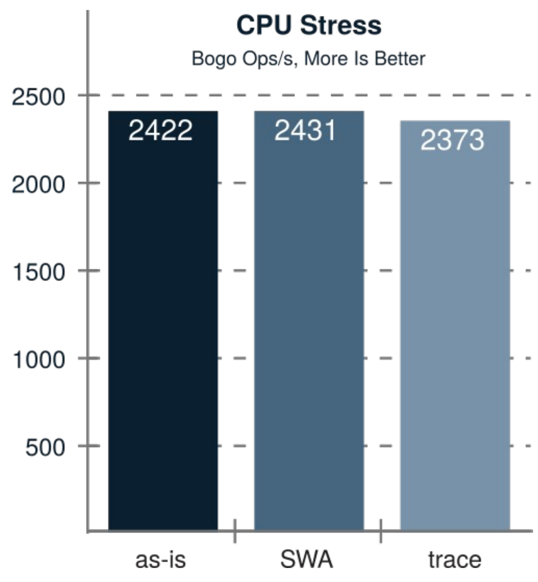
- Two benchmarks
  - Throughput: Using the Phoronix Test Suite
  - Latency: Using cyclictest
- Base of comparison:
  - **as-is**: The system without any verification or trace.
  - **trace**: Tracing (ftrace) the same events used in the verification
    - Only trace! No collection or interpretation.

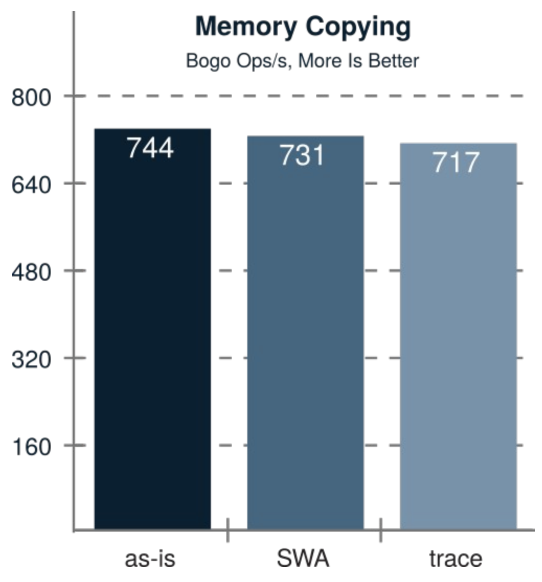
Throughput: SWA model



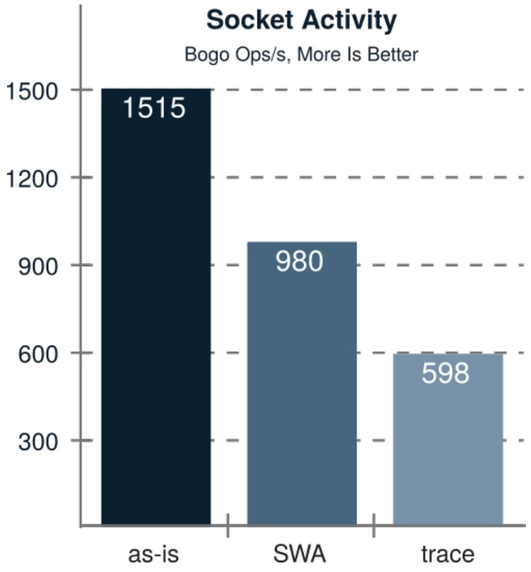
Benchmark: Thoughput – Low kernel activation

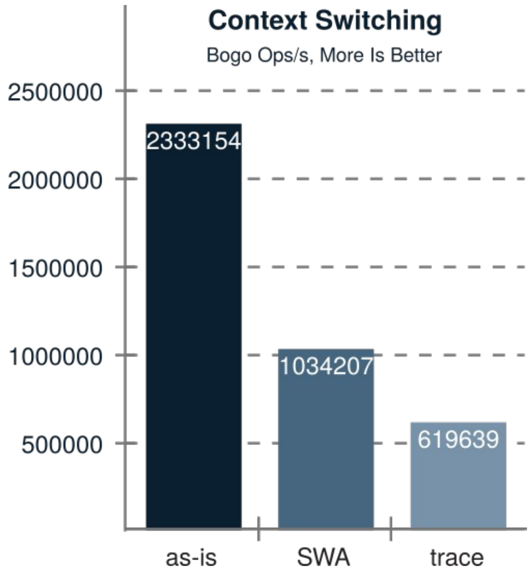


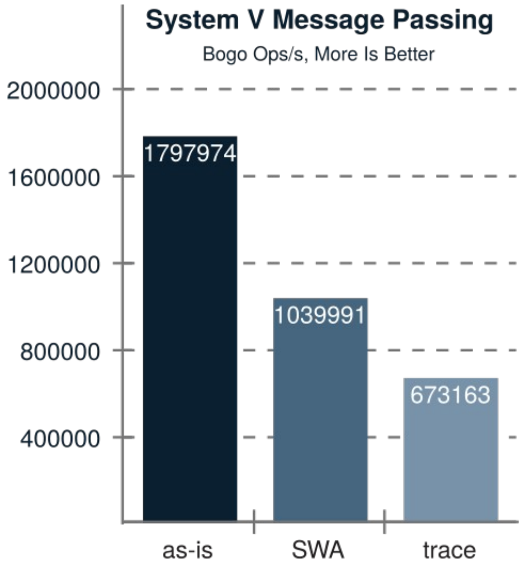




Benchmark: Thoughput – High kernel activation

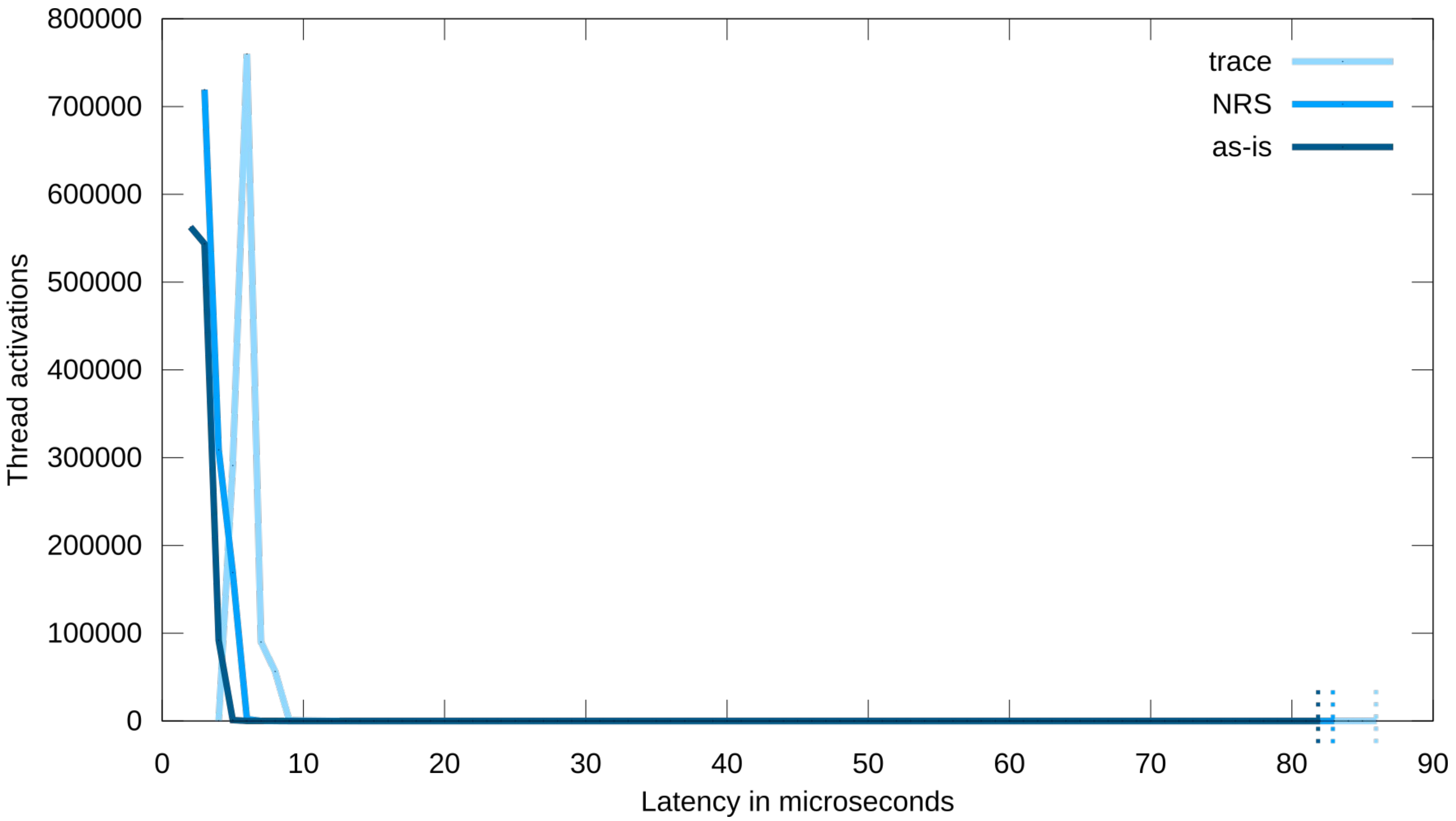






Red Hat

Benchmark: Cyclictest latency



Academically accepted

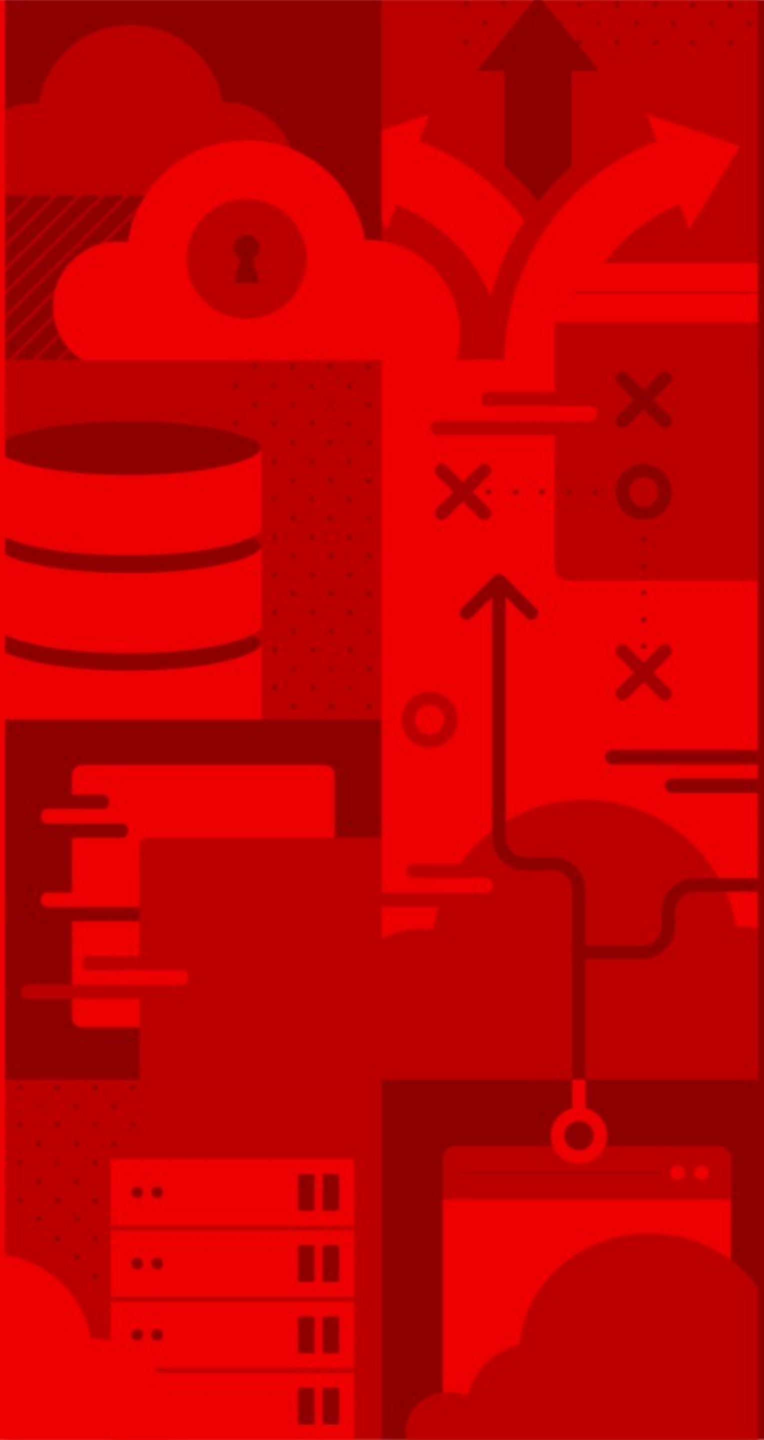
Efficient Formal Verification for the Linux Kernel

Daniel Bristot de Oliveira, Rômulo Silva de Oliveira & Tommaso Cucinotta

17th International Conference on Software Engineering and Formal Methods (SEFM)

More info here: http://bristot.me/efficient-formal-verification-for-the-linux-kernel/

Red Hat



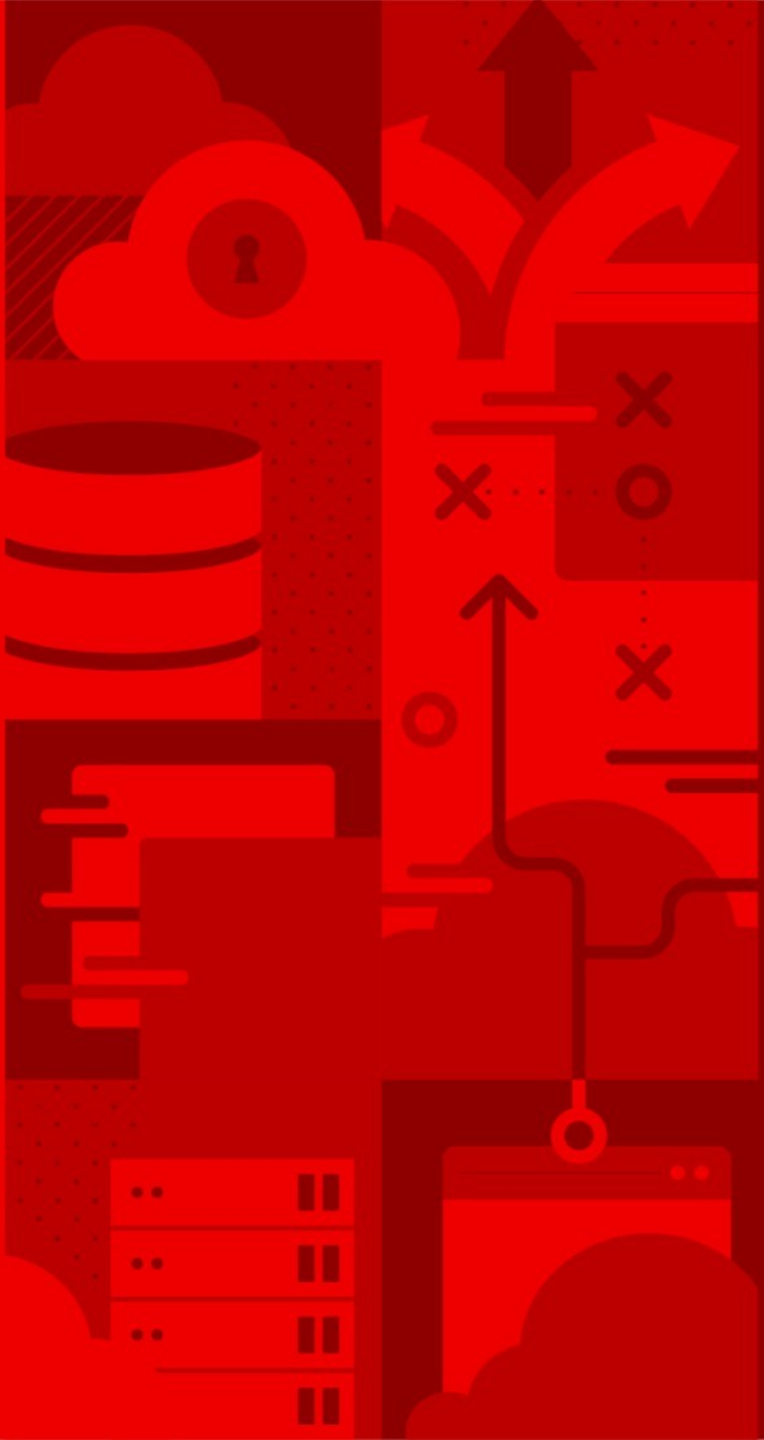
So, what is next?

A better interface

- Loading the module is not that practical
- How about an interface like ftrace?
  - /sys/kernel/debug/verification/
  - Would enable many verification models to be loaded
  - Enable/disable verification
  - Enable/disable options
- Or should I use eBPF + perf?
  - perf verify "model.dot" translation_trace_to_events.txt

What should we model?

- I am currently working to make the RT task model to work
  - Different viewpoint: from per-task to per-cpu
- But there are other possible things to model
  - Locking (part of lockdep)
    - Why?
    - Run-time without recompile/reboot.
  - RCU?
  - Schedulers?



Something else?

Thank you!

This work is made in collaboration with:

the Retis Lab @ Scuola Superiore Sant'Anna (Pisa – Italy)

Universidade Federal de Santa Catarina (Florianópolis - Brazil)