



Formal modeling (and verification) made easy And fast!

Daniel Bristot de Oliveira
Principal Software Engineer



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 we need something
more robust.

Like...

- 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?
- How do we verify the runtime behavior of Linux?



How do we convince
other communities
about our properties?




What computer
scientists say about it?



Formal methods!



We already have some examples!



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

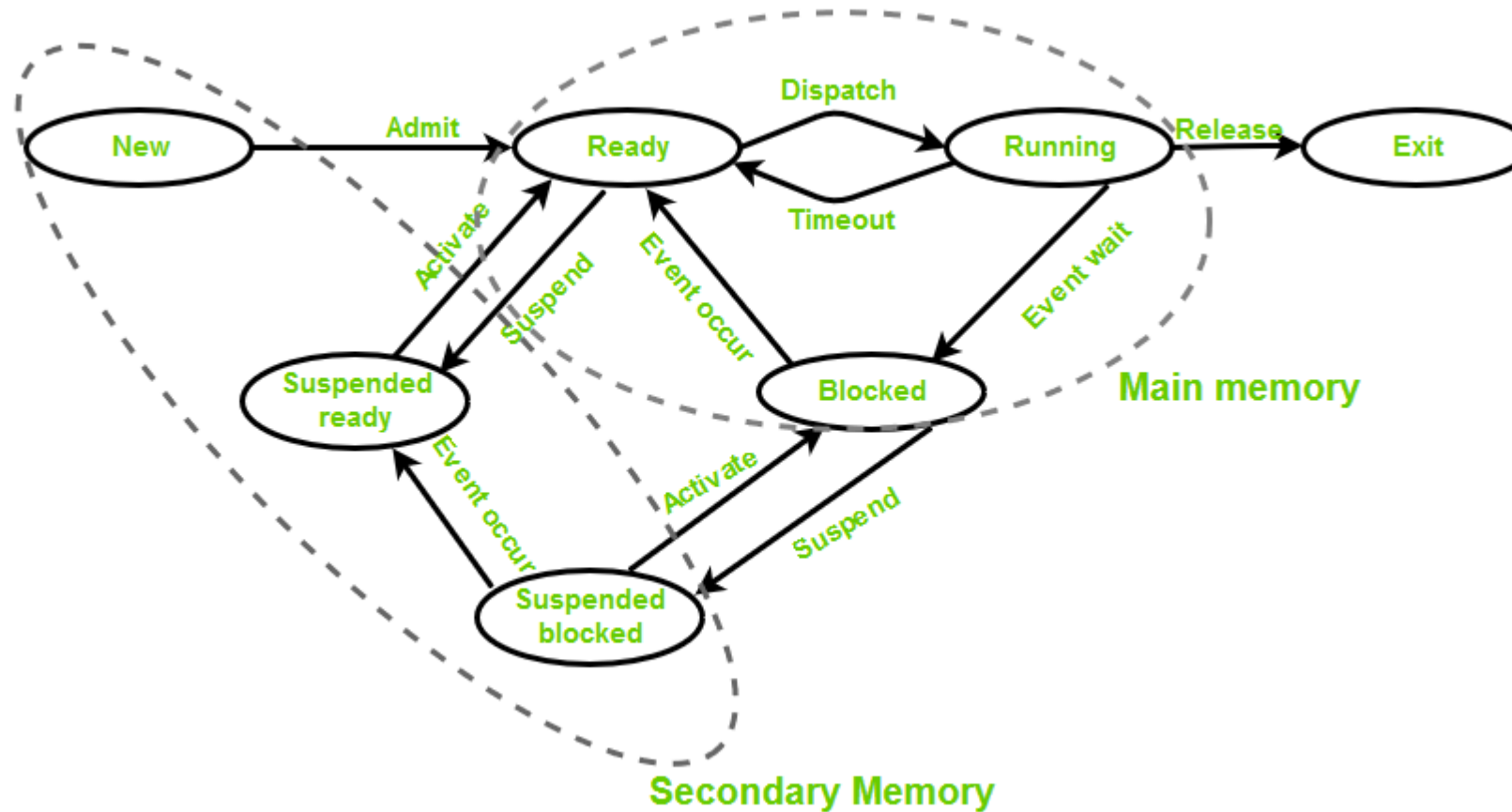
How can we turn modeling easier?

- Using a *formal language* that looks *natural* for us!
- How do we *naturally* “observe” the dynamics of Linux?



We trace events!

While tracing we...



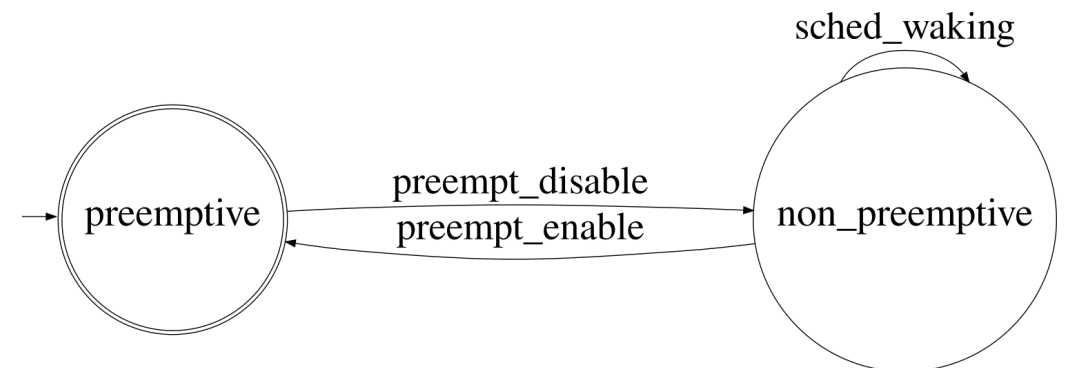
^C^V from

<https://www.geeksforgeeks.org/states-of-a-process-in-operating-systems/>

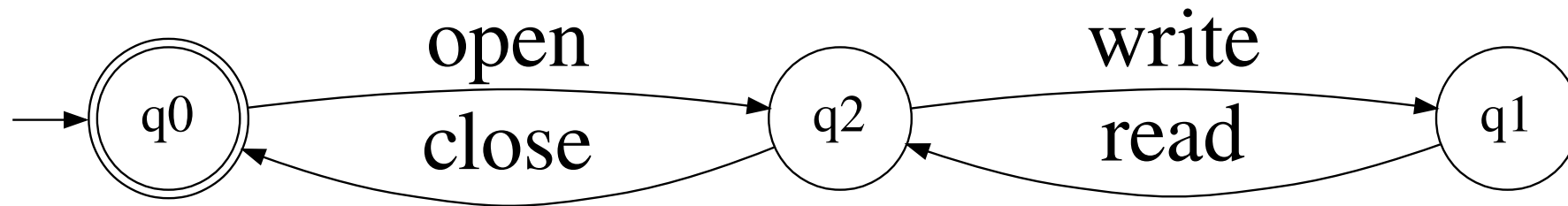
State-machines + FM = Automata!

- 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
```



Using automata as formal language



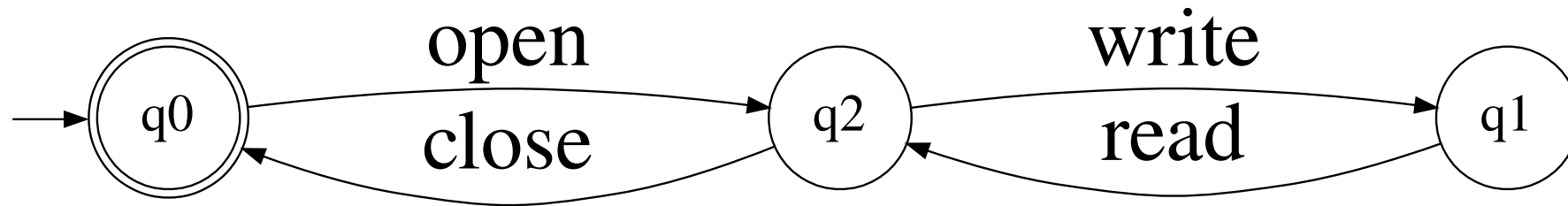
Is formally defined

- 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 = $(X \times E) \rightarrow X$;
 - x_0 = Initial state;
 - X_m = set of final states.
- The language - or traces - generated/recognized by G is the $L(G)$.

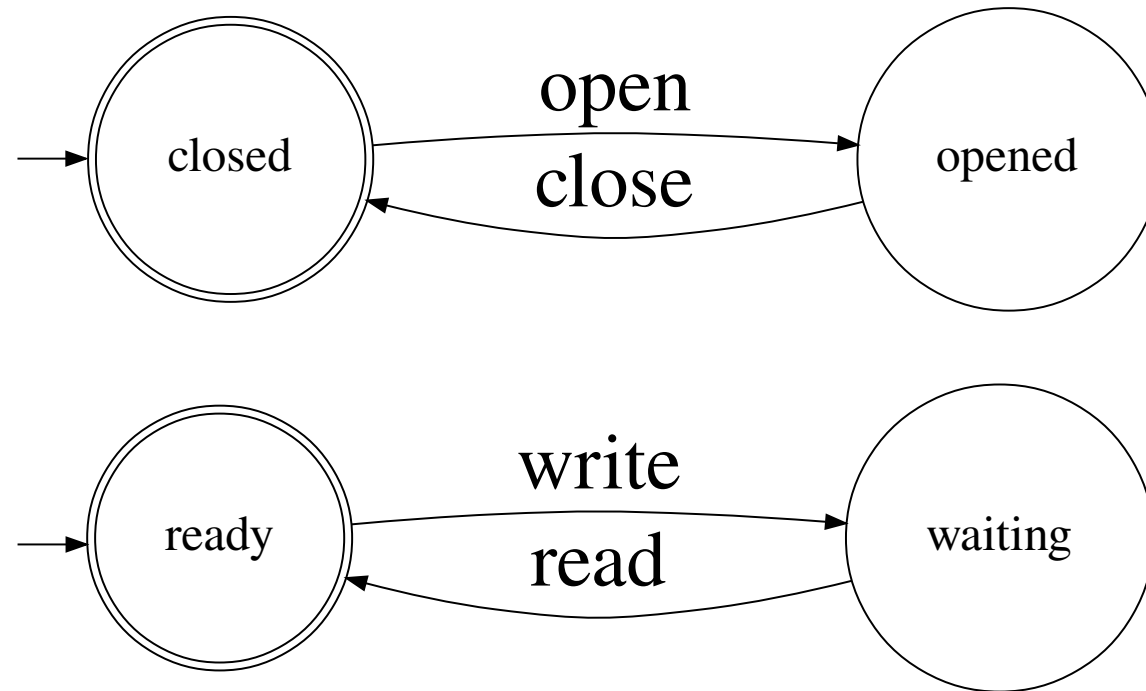
Automata allows

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

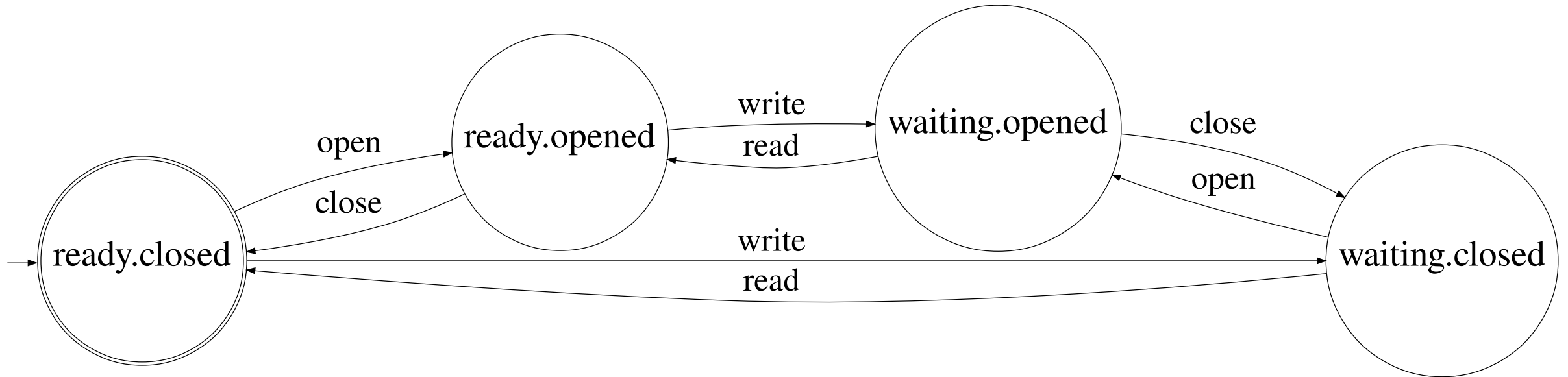
The previous example



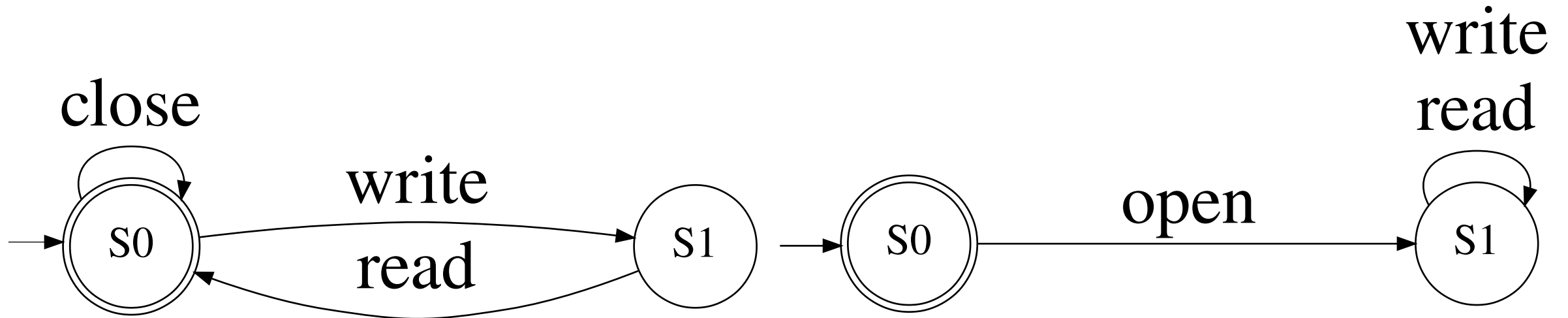
Generators



Sync of generators



Specification



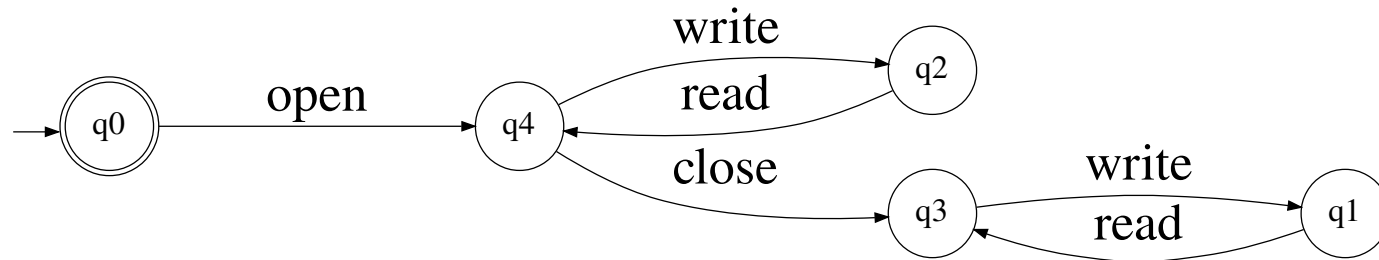
Verification

The screenshot shows the Supremica software interface. The title bar reads "Supremica - Module: New Module". The menu bar includes "File", "Edit", "Analyze", "Examples", "Modules", "Configure", and "Help". Below the menu bar is a toolbar with icons for file operations and simulation. The main window has three tabs: "Editor", "Simulator", and "Analyzer". The "Analyzer" tab is active, displaying a table with the following data:

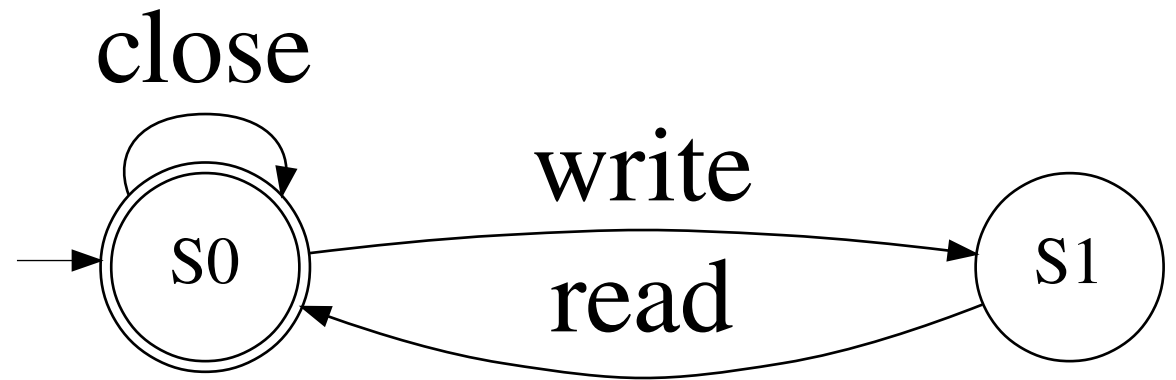
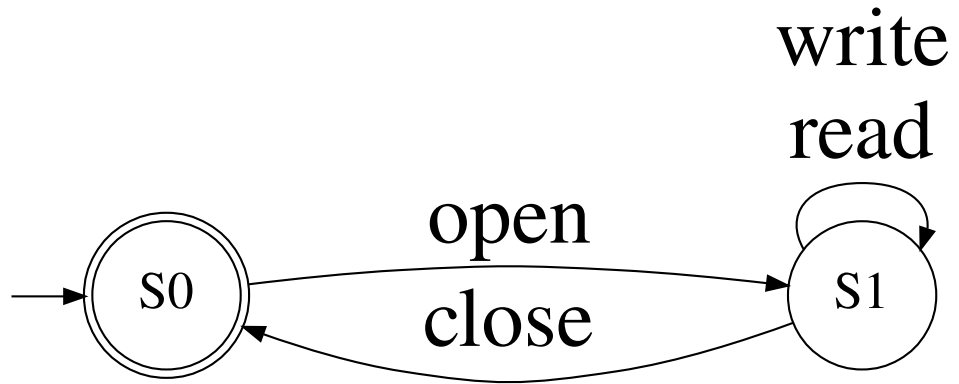
Name	Type	Q	Σ	→
open_close	Plant	2	2	2
client	Plant	2	2	2
client open_close	Plant	4	4	8
rw_after_opening	Plant	2	3	3
copy_of_rw_after_opening	Plant	2	4	4
good	Plant	4	4	6
bad	Plant	5	4	7

A dialog box titled "Bad news" is overlaid on the table. It contains a red "X" icon and the text "The system is blocking!". An "OK" button is located at the bottom of the dialog box.

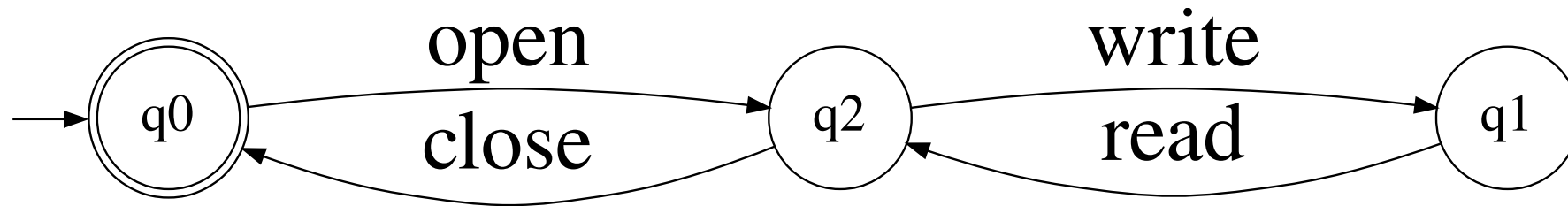
Synch of Generators and Specifications



Specifications



Sync of Generators and Specifications





Why not just draw it?

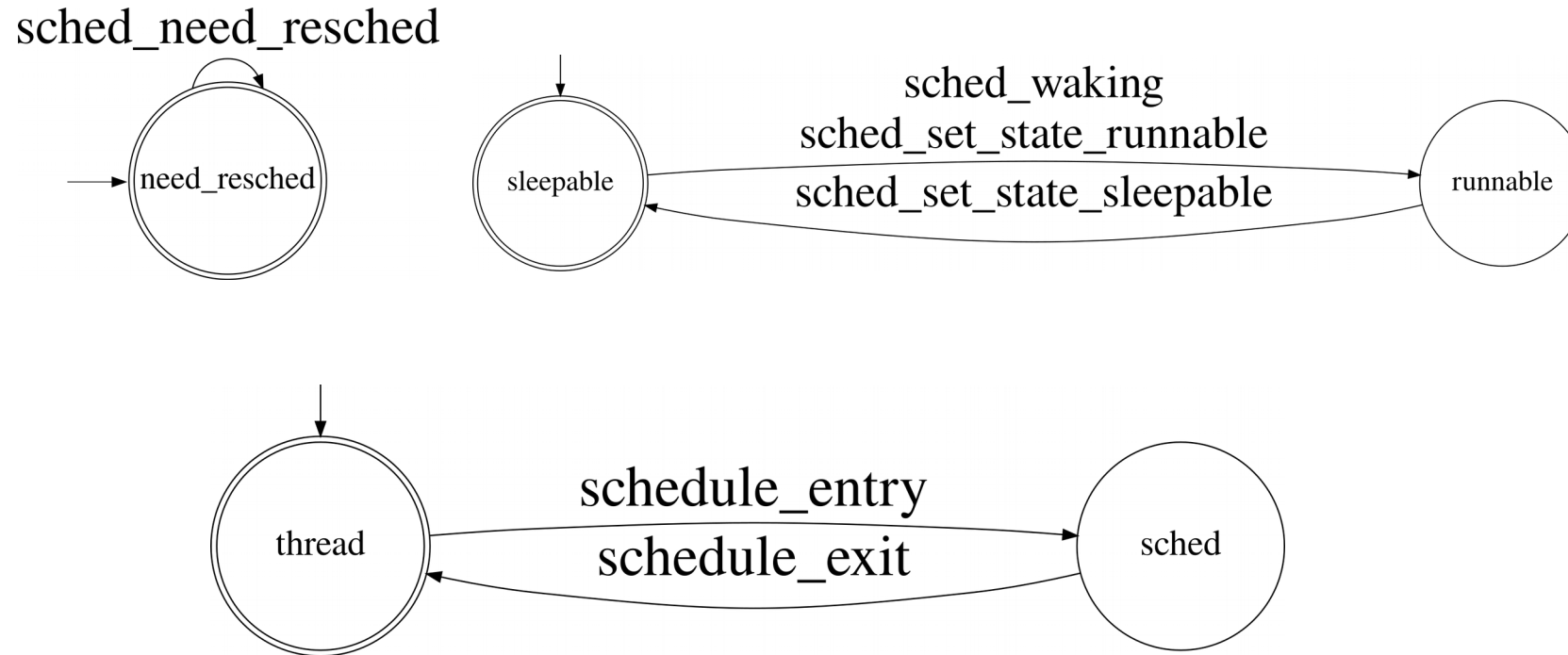


Linux is Complex!

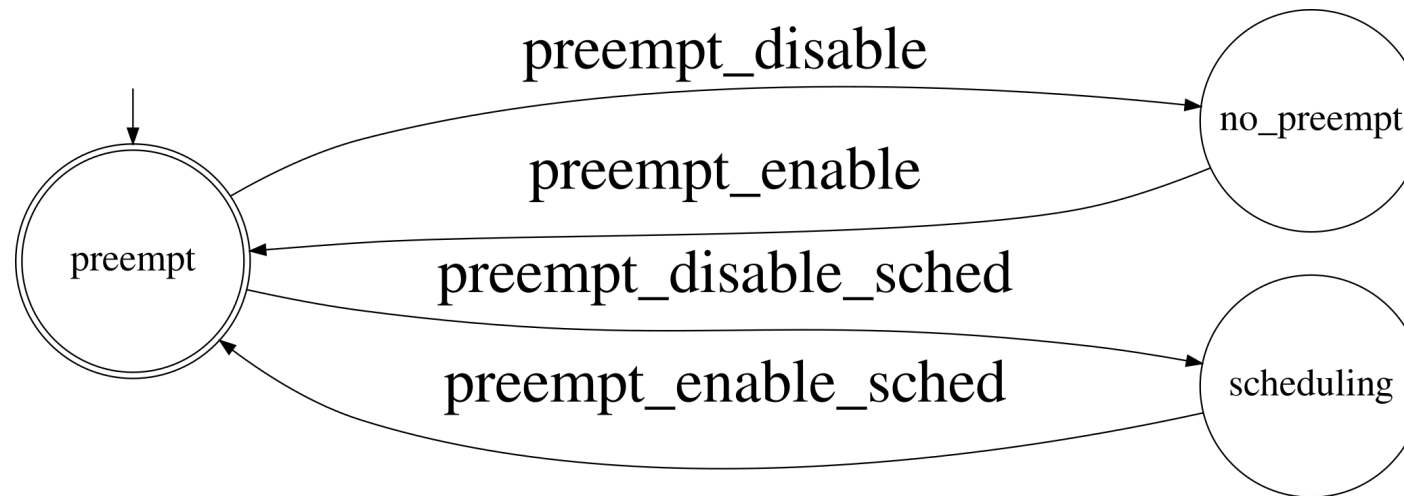
PREEMPT_RT model

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

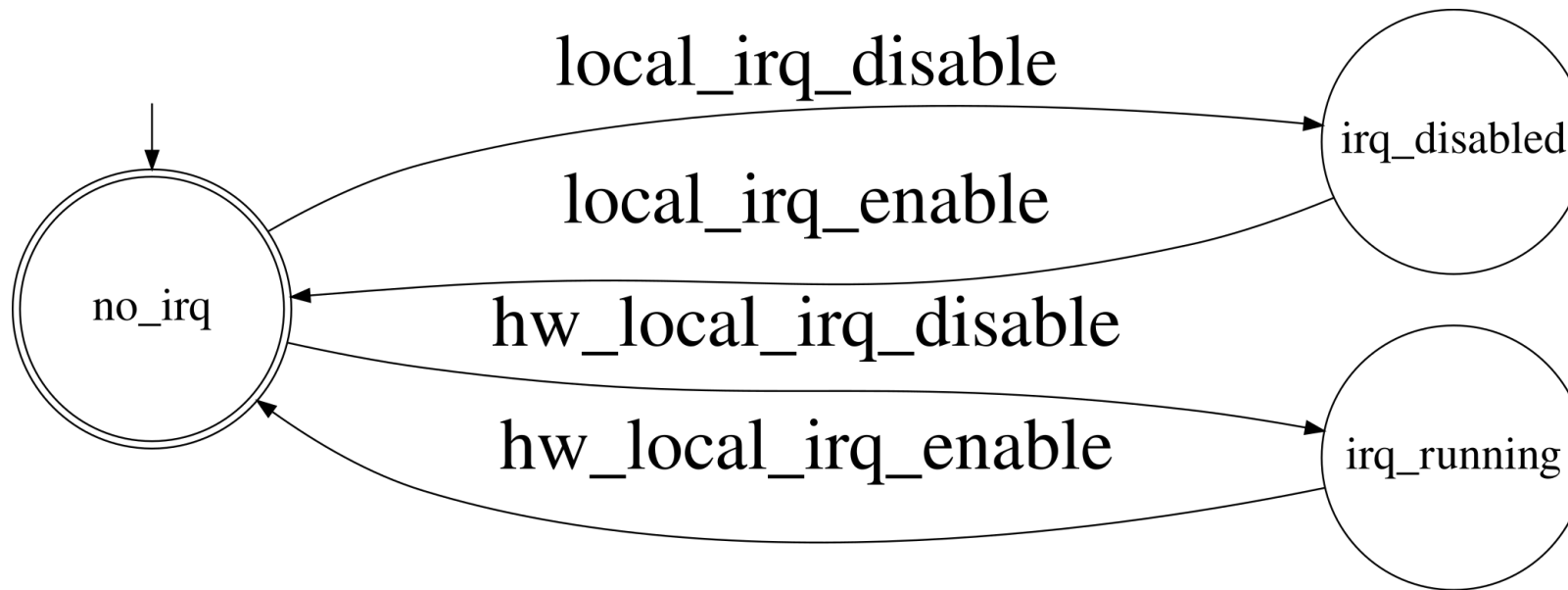
A more complex case



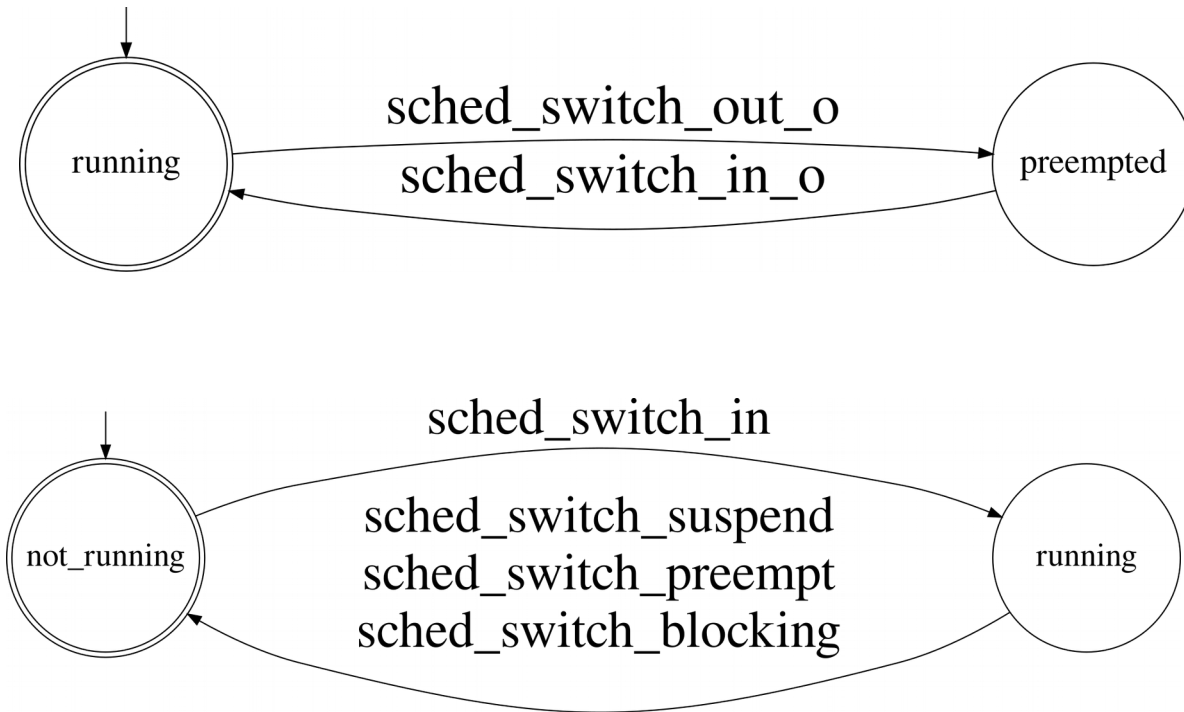
Independend “generators”



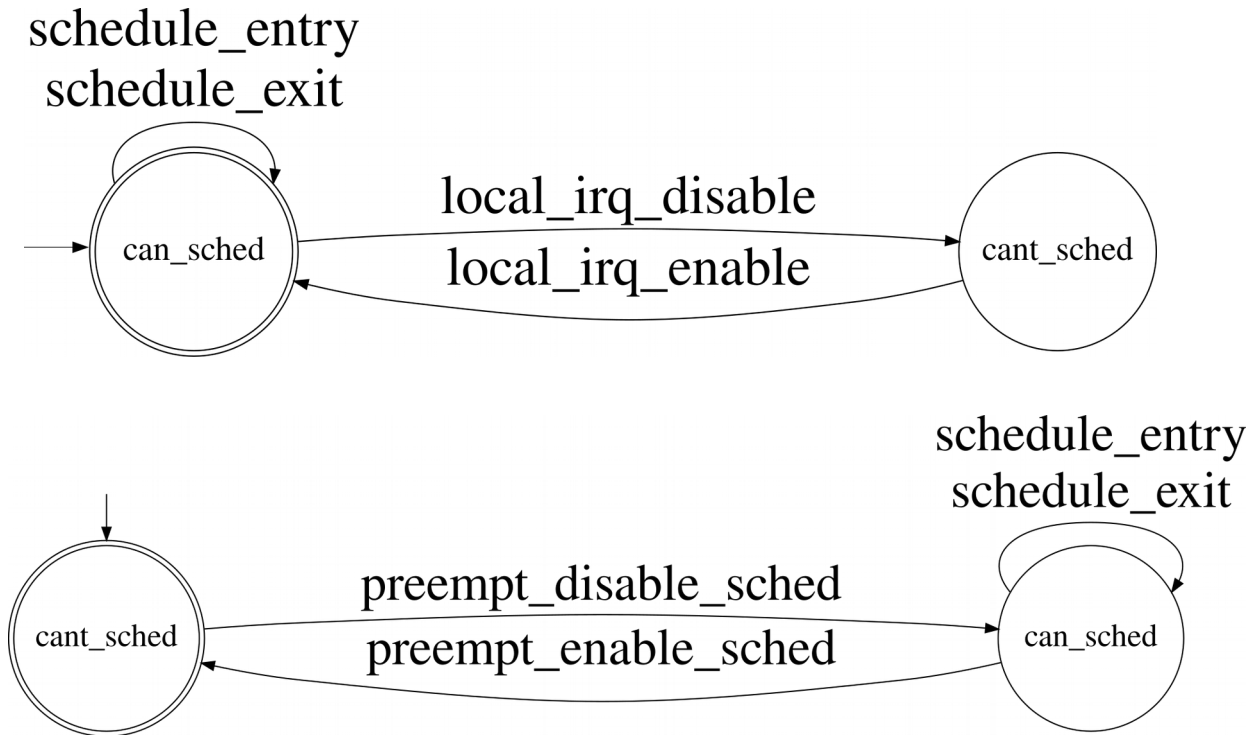
Independend “generators”



Independend “generators”

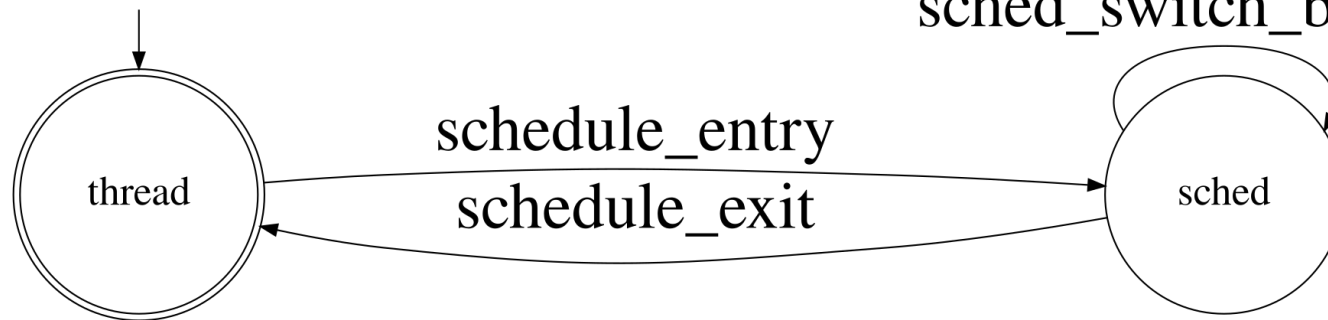


Necessary conditions

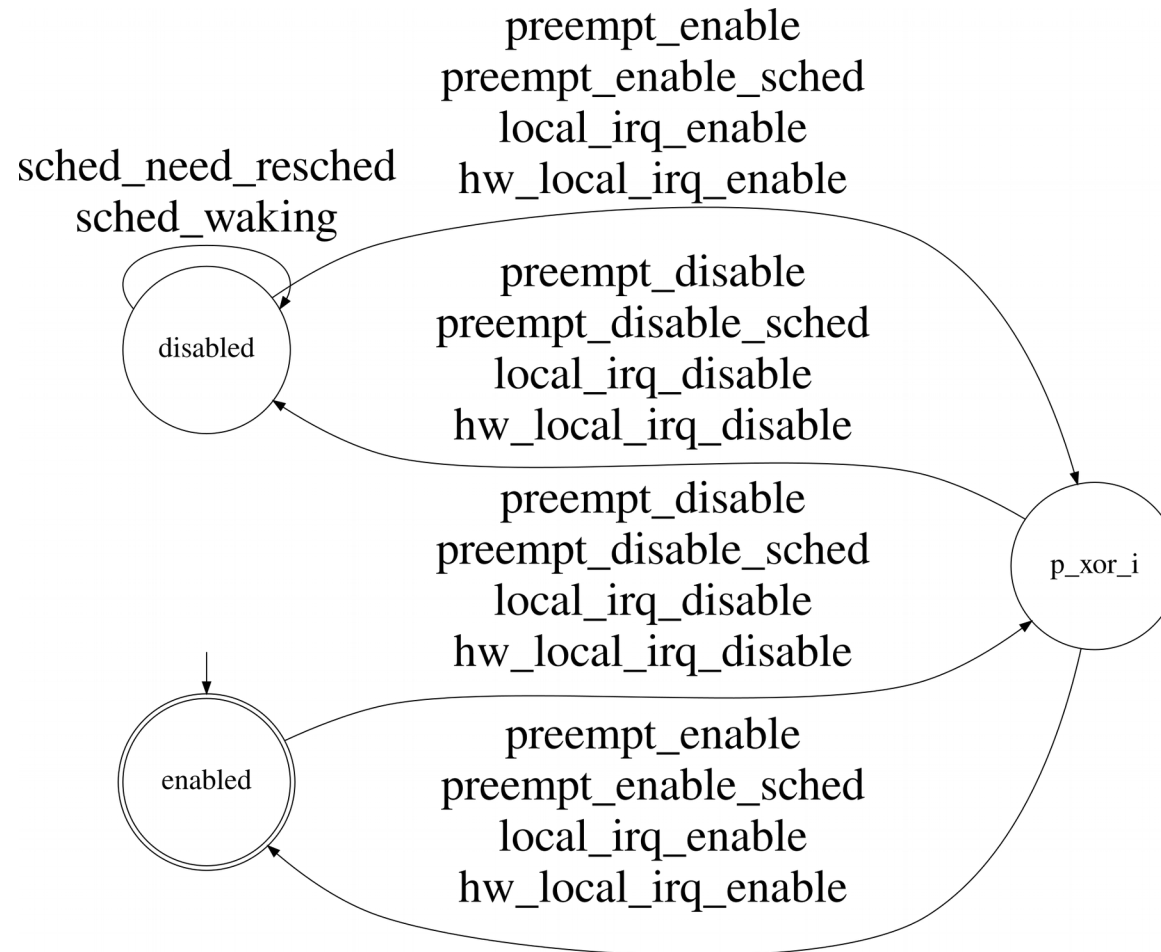


Necessary conditions

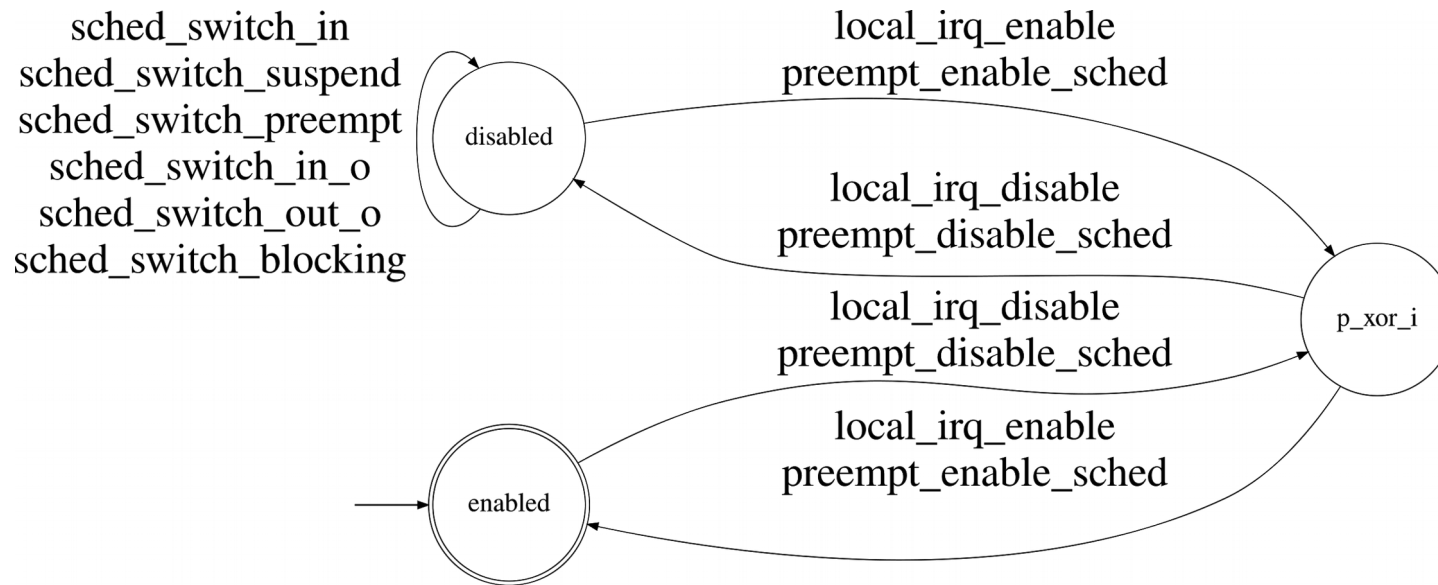
sched_switch_in
sched_switch_in_o
sched_switch_suspend
sched_switch_preempt
sched_switch_out_o
sched_switch_blocking



Necessary conditions

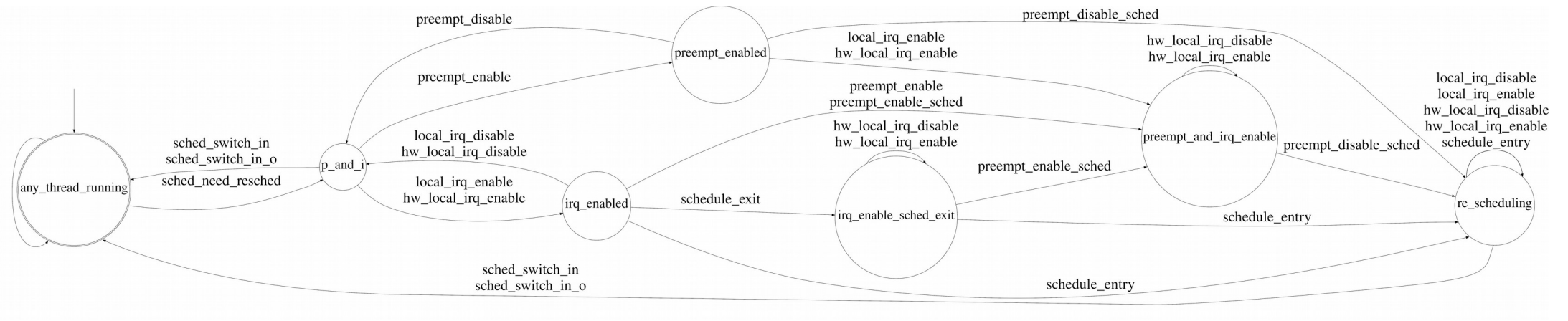


Necessary conditions



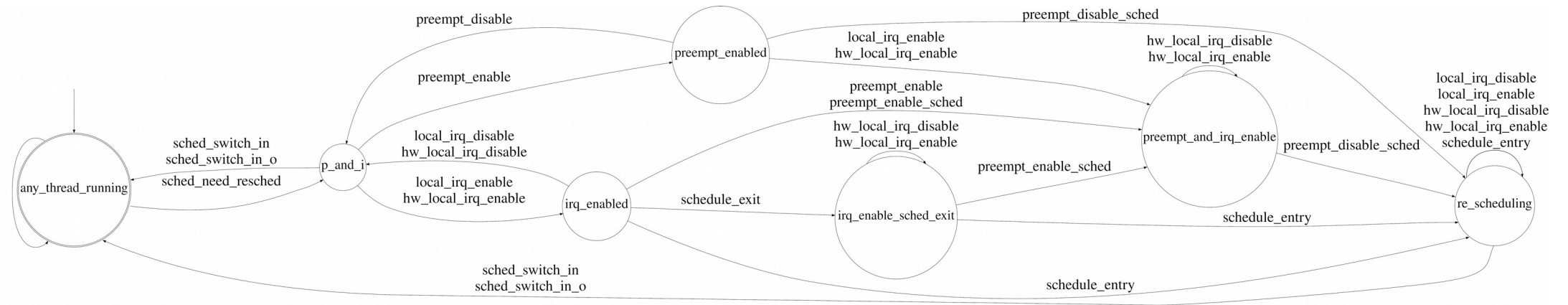
Sufficient conditions

write_abandon
write_acquired
write_blocked
write_lock
mutex_abandon
mutex_acquired
mutex_blocked
mutex_lock
read_abandon
read_acquired
read_blocked
read_lock
preempt_disable_sched
preempt_enable_sched
hw_local_irq_disable
hw_local_irq_enable
local_irq_disable
local_irq_enable
preempt_disable
preempt_enable
schedule_entry
schedule_exit
sched_switch_in
sched_switch_in_o



“PREEMPT”_RT is deterministic

write_abandon
write_acquired
write_blocked
write_lock
mutex_abandon
mutex_acquired
mutex_blocked
mutex_lock
read_abandon
read_acquired
read_blocked
read_lock
preempt_disable_sched
preempt_enable_sched
hw_local_irq_disable
hw_local_irq_enable
local_irq_disable
local_irq_enable
preempt_disable
preempt_enable
schedule_entry
schedule_exit
sched_switch_in
sched_switch_in_o



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)

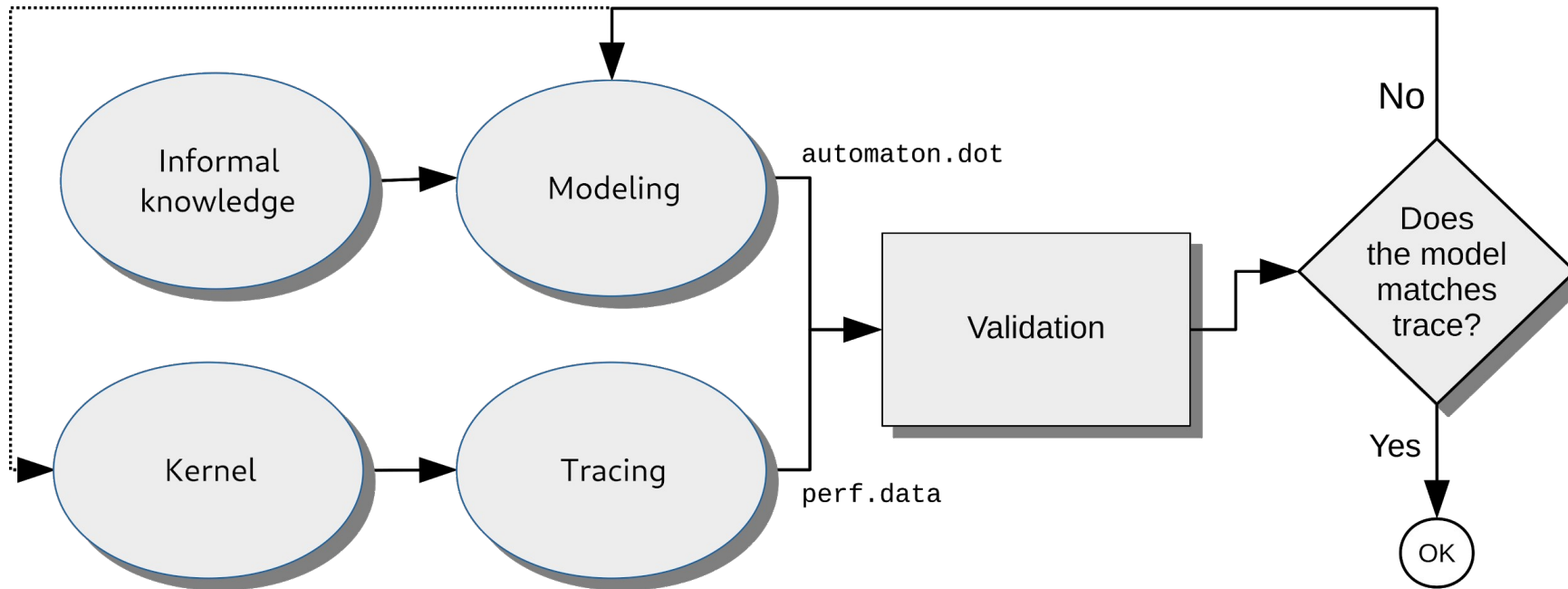


How to verify that the
system *behaves*?



Comparing system
execution against the
model!

Offline & Asynchronous



But...

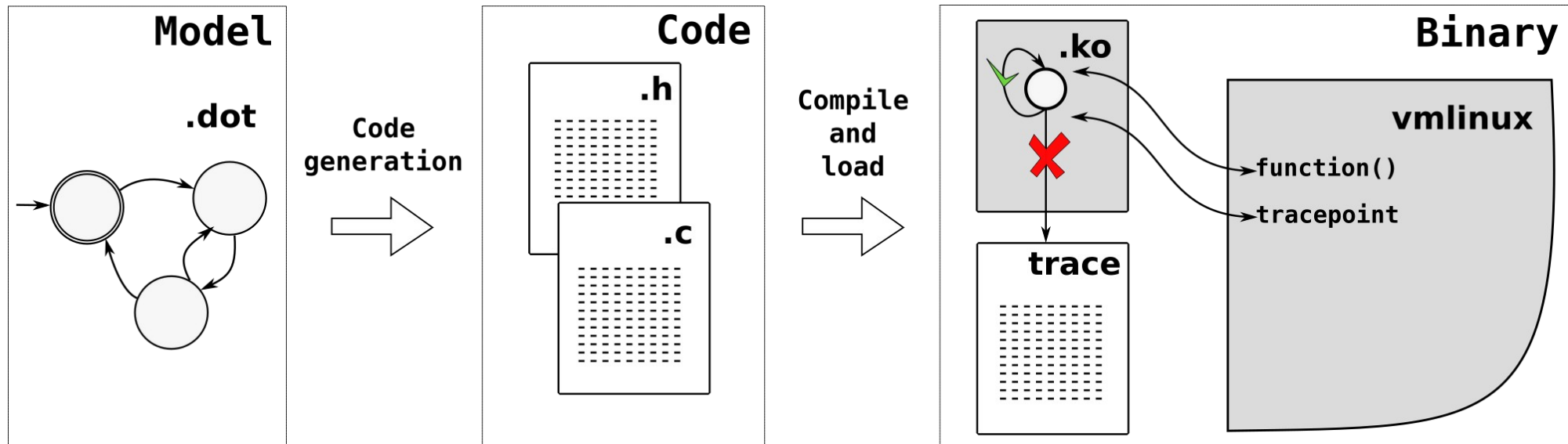
Logical correctness for task model

- 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.

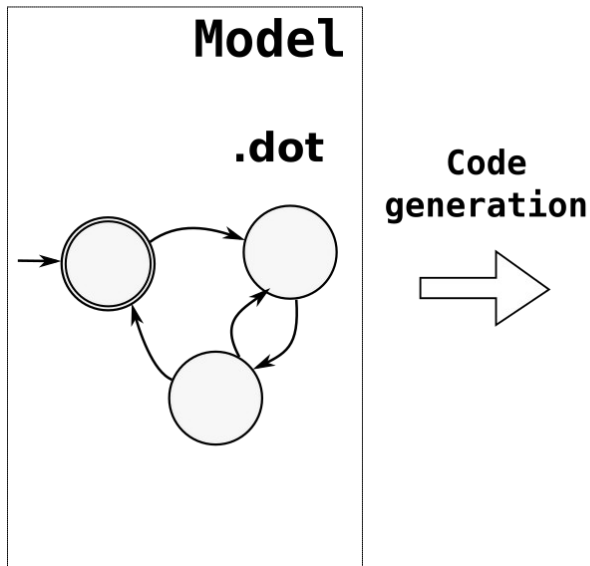


What can we do?

Online & Synchronous RV



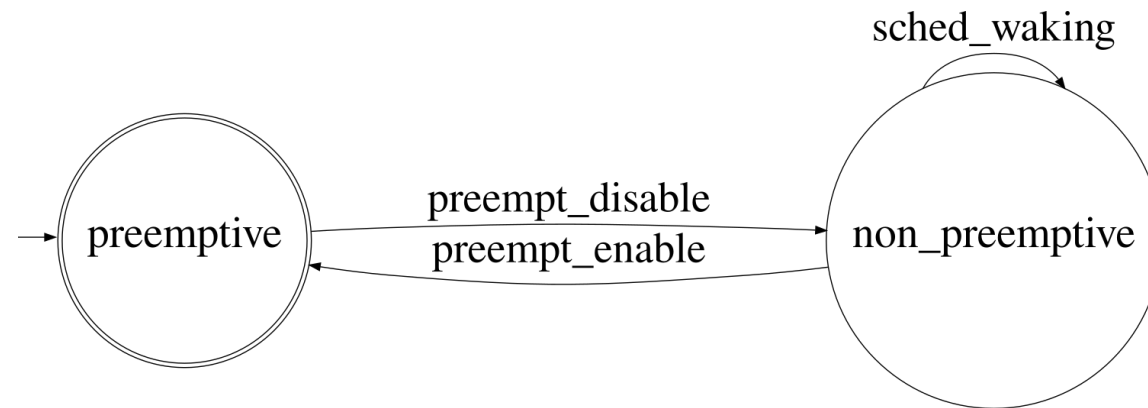
1) Code generation



- We develop the **dot2c** tool to translate the model into code
- It is a python program that has one input:
 - An automaton model in the **.dot** format
 - It is an open format (graphviz)
 - Supremica tool exports models with this format

Code generation

Wakeup in preemptive model:



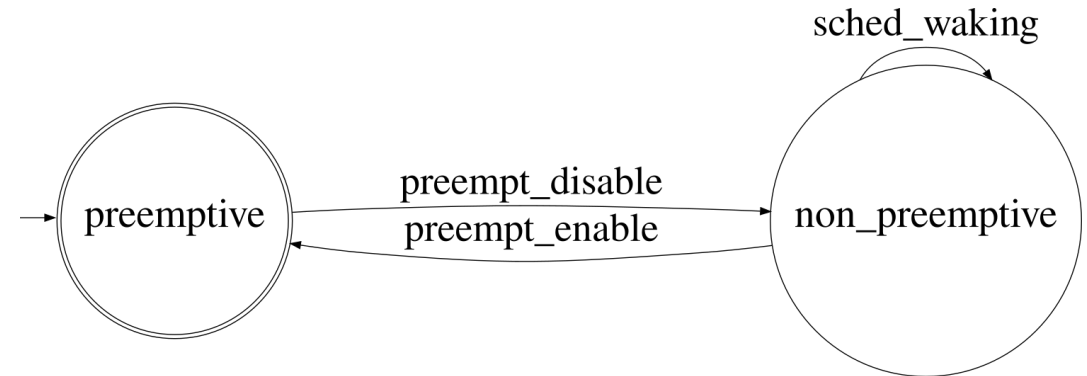
Code generation:

```
[bristot@t460s dot2c]$ ./dot2c wakeup_in_preemptive.dot
```

.....

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];  
};
```

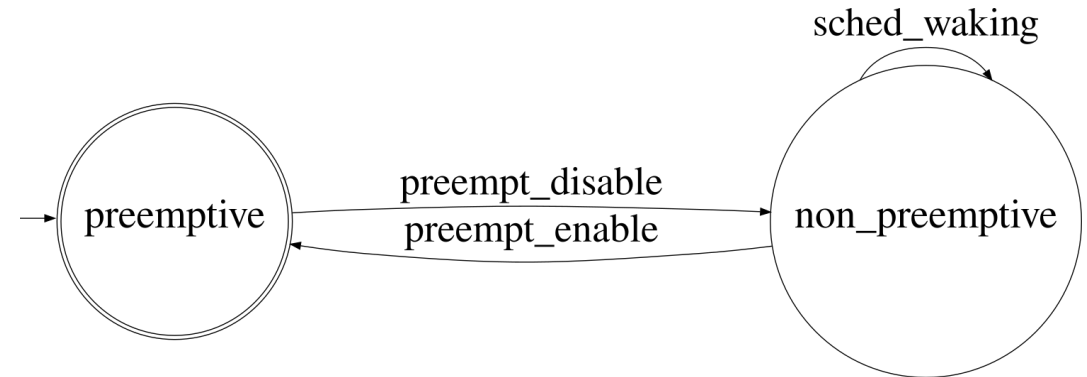


Automaton in C

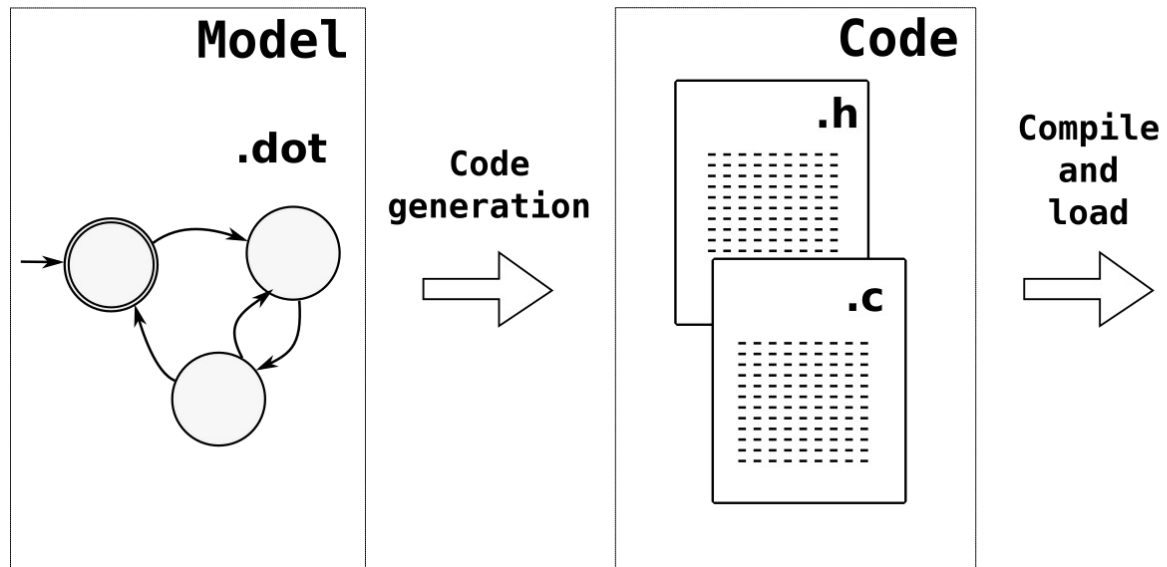
```
enum states {
    preemptive = 0,
    non_preemptive,
    state_max
};

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 }
};
```



Processing functions



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];
}
```

All operations are O(1)!

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

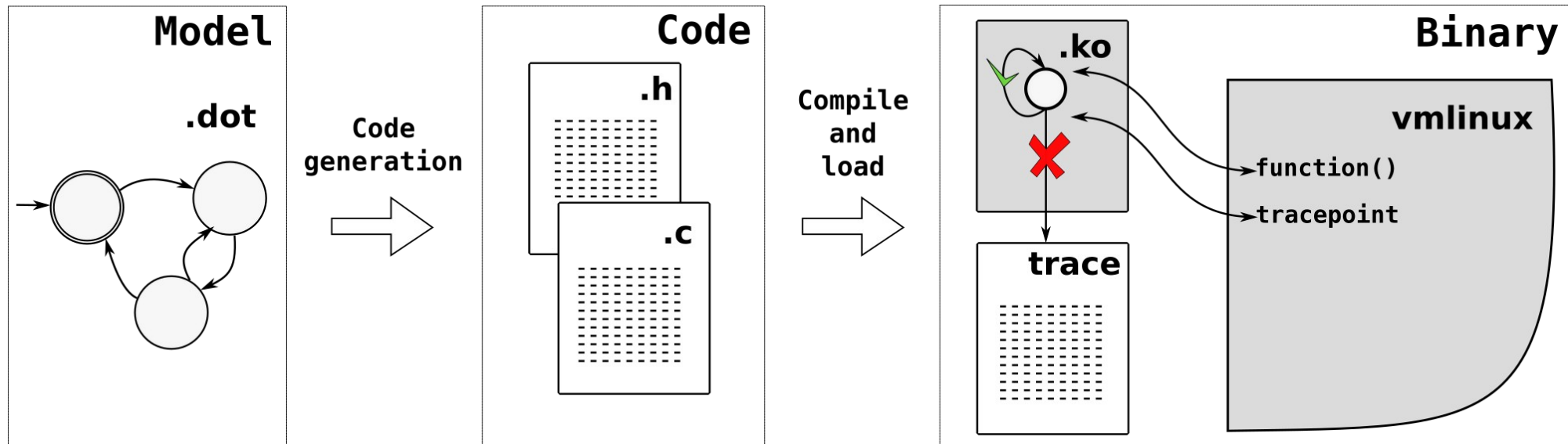
```
char get_next_state(struct verification *ver, enum states curr_state, enum events event)
{
    return ver->aut->function[curr_state][event];
}
```

```
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;
}
```

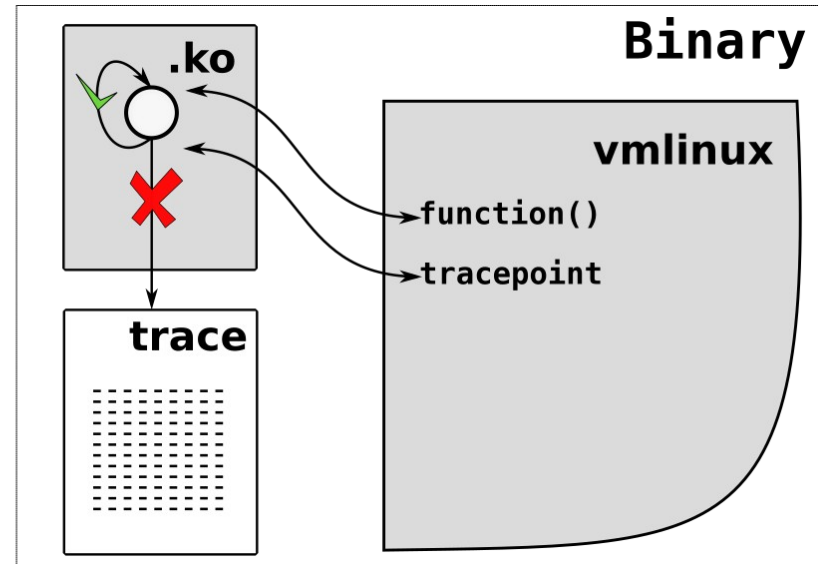
Only one variable to keep the state!

3) Verification



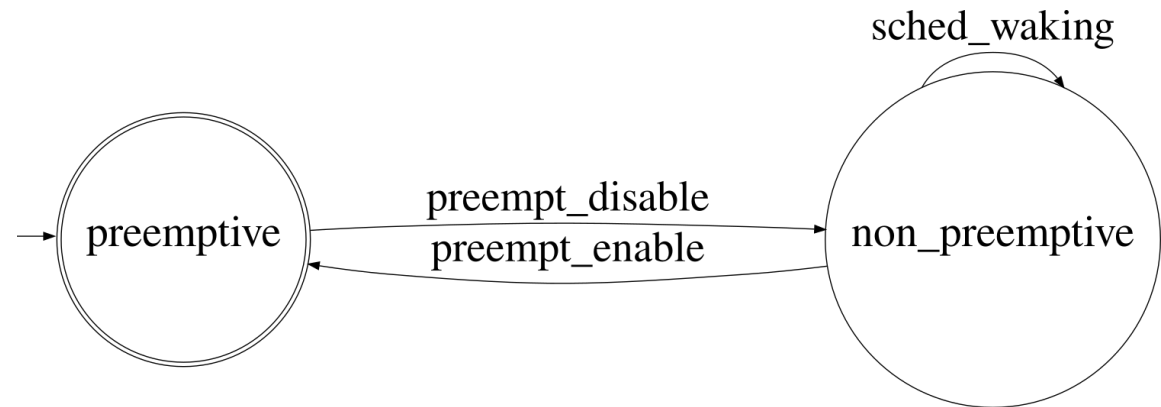
Verification

- Verification code is compiled as a kernel module
- Kernel module is loaded to a running kernel
 - While no problem is found:
 - Either print all event's execution
 - Or run silently
 - If an unexpected transitions is found:
 - Print the error on trace buffer



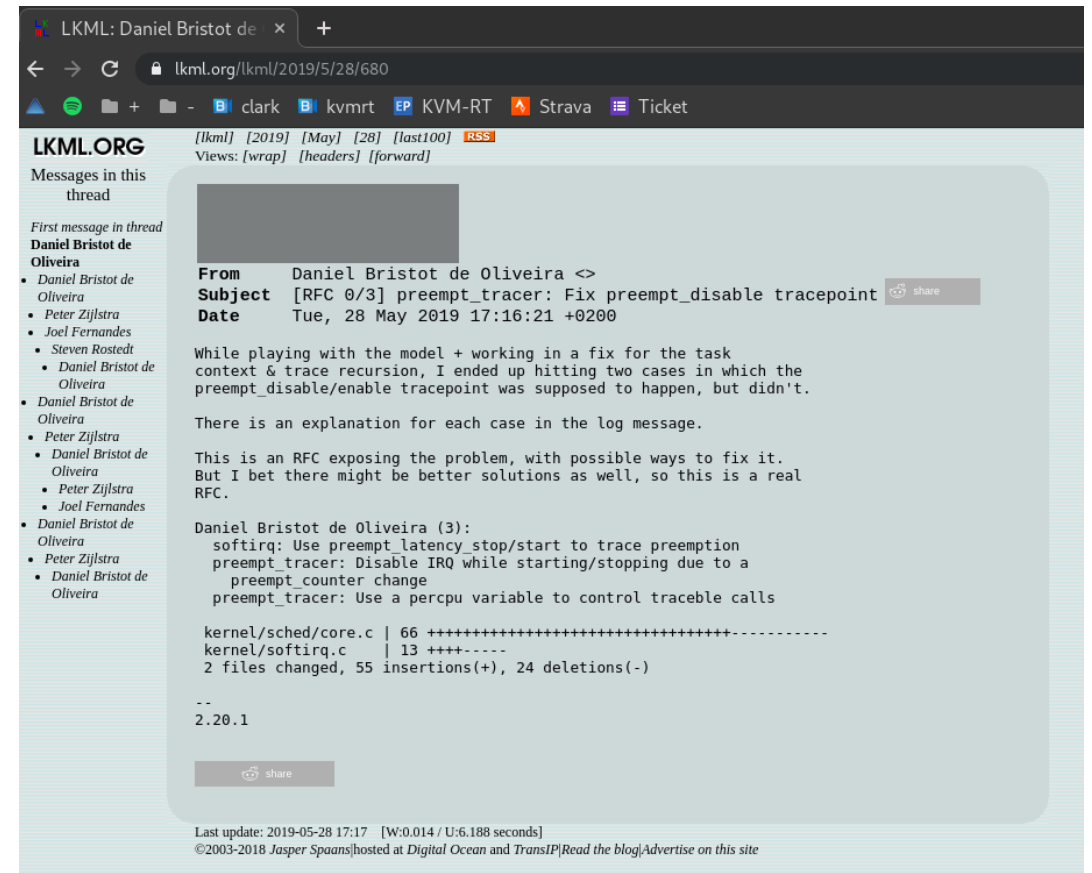
Error output

```
bash-1157 [003] ....2.. 191.199172: process_event: non_preemptive -> preempt_enable = preemptive safe!  
bash-1157 [003] dN..5.. 191.199182: process_event: event sched_waking not expected in the state preemptive  
bash-1157 [003] dN..5.. 191.199186: <stack trace>  
=> process_event  
=> __handle_event  
=> ttwu_do_wakeup  
=> try_to_wake_up  
=> irq_exit  
=> smp_apic_timer_interrupt  
=> apic_timer_interrupt  
=> rcu_irq_exit_irqson  
=> trace_preempt_on  
=> preempt_count_sub  
=> __raw_spin_unlock_irqrestore  
=> __down_write_common  
=> anon_vma_clone  
=> anon_vma_fork  
=> copy_process.part.42  
=> _do_fork  
=> do_syscall_64  
=> entry_SYSCALL_64_after_hwframe
```



Practical example

- A problem with tracing subsystem was reported using this model's module
- <https://lkml.org/lkml/2019/5/28/680>
<recall to open the link>



The screenshot shows a web browser displaying an email thread on the LKML.ORG website. The browser's address bar shows the URL `lkml.org/lkml/2019/5/28/680`. The page header includes navigation links like `[[lkm]]`, `[2019]`, `[May]`, `[28]`, `[last100]`, and an `RSS` link. Below the header, there's a list of participants in the thread, including Daniel Bristot de Oliveira, Peter Zijlstra, and Joel Fernandes. The main content of the email is as follows:

From: Daniel Bristot de Oliveira <>
Subject: [RFC 0/3] preempt_tracer: Fix preempt_disable tracepoint
Date: Tue, 28 May 2019 17:16:21 +0200

While playing with the model + working in a fix for the task context & trace recursion, I ended up hitting two cases in which the preempt_disable/enable tracepoint was supposed to happen, but didn't.

There is an explanation for each case in the log message.

This is an RFC exposing the problem, with possible ways to fix it. But I bet there might be better solutions as well, so this is a real RFC.

Daniel Bristot de Oliveira (3):
softirq: Use preempt_latency_stop/start to trace preemption
preempt tracer: Disable IRQ while starting/stopping due to a preempt_counter change
preempt_tracer: Use a percpu variable to control traceable calls

```
kernel/sched/core.c | 66 ++++++-----  
kernel/softirq.c   | 13 +++-----  
2 files changed, 55 insertions(+), 24 deletions(-)
```

--
2.20.1

At the bottom of the email, there is a 'share' button and a footer with the text: "Last update: 2019-05-28 17:17 [W:0.014 / U:6.188 seconds] ©2003-2018 Jasper Spaans|hosted at Digital Ocean and TransIP|Read the blog|Advertise on this site".



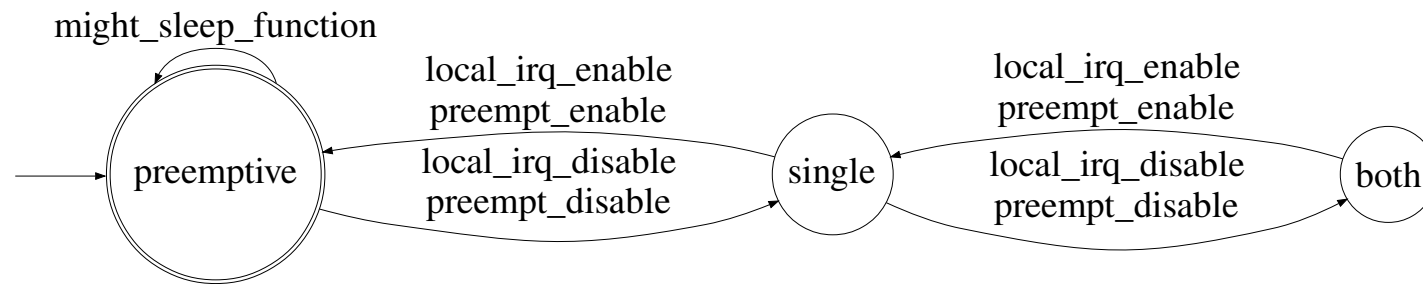
There is not free meal!

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?**

In practice... also..

- Complete models like the PREEMPT_RT are not necessarily need.
- Small models can be created as “test cases”
- For example:



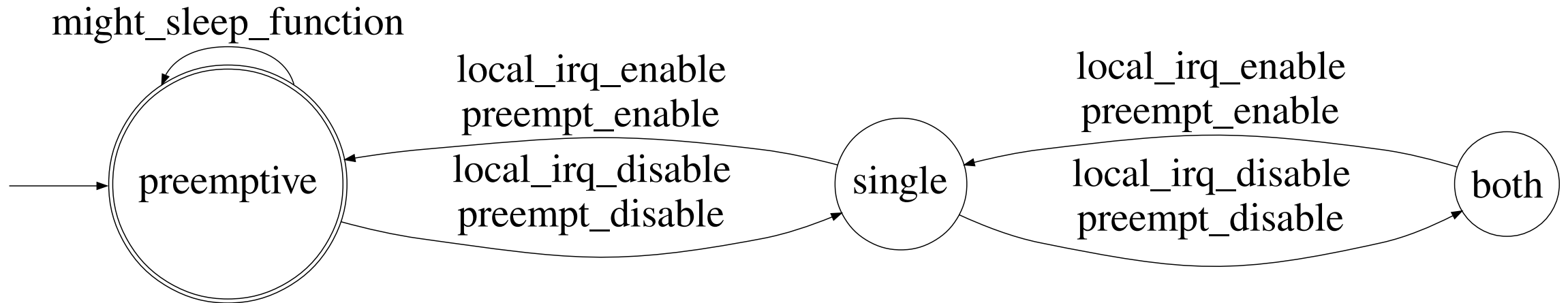


How *efficient* is this idea?

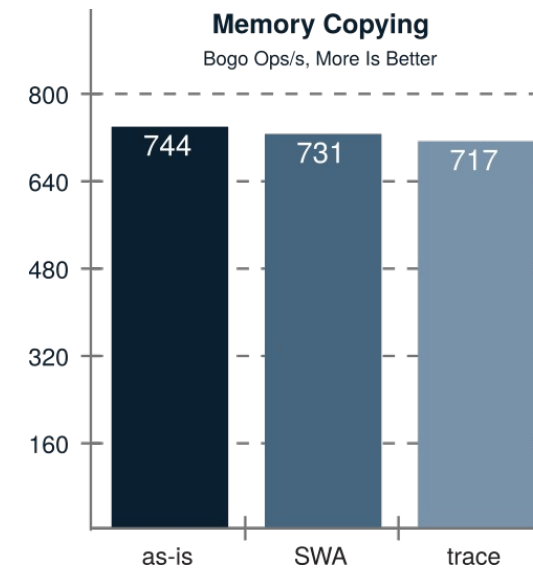
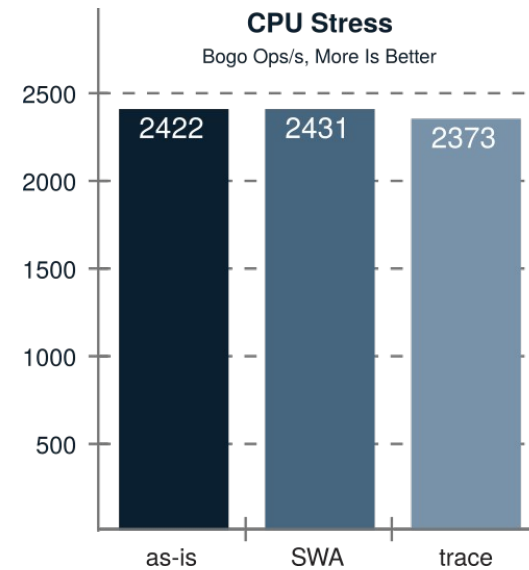
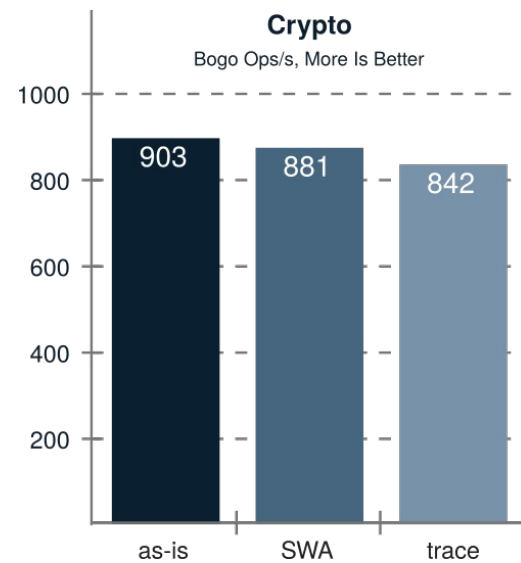
Efficiency in practice: a benchmark

- Two benchmarks
 - Throughput: Using the Phoronix Test Suite
 - Latency: Using `cyclictst`
- 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.

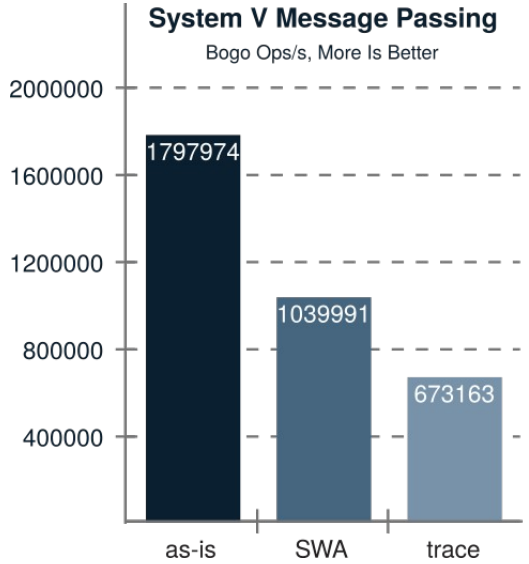
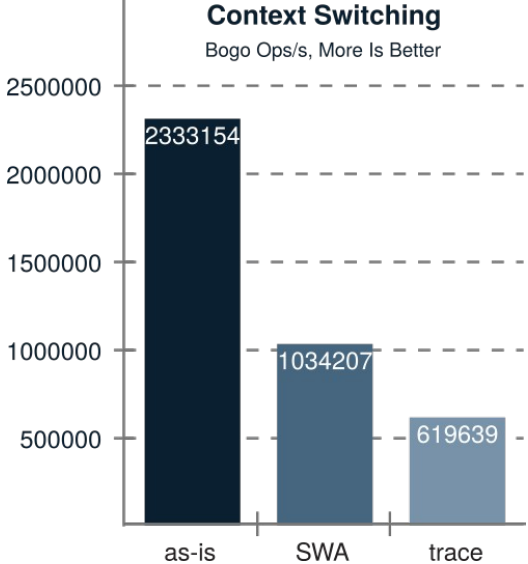
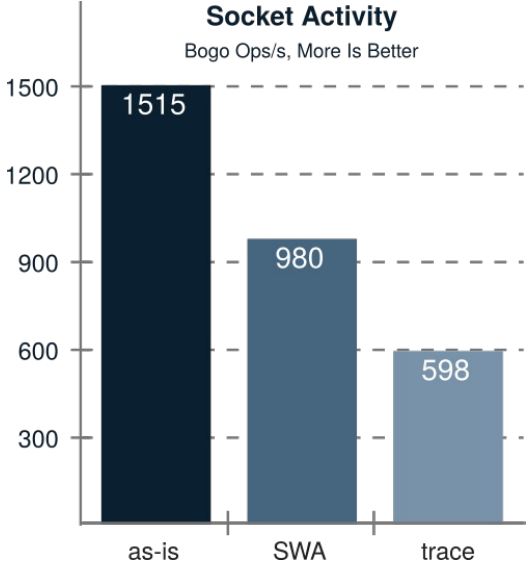
Throughput: SWA model



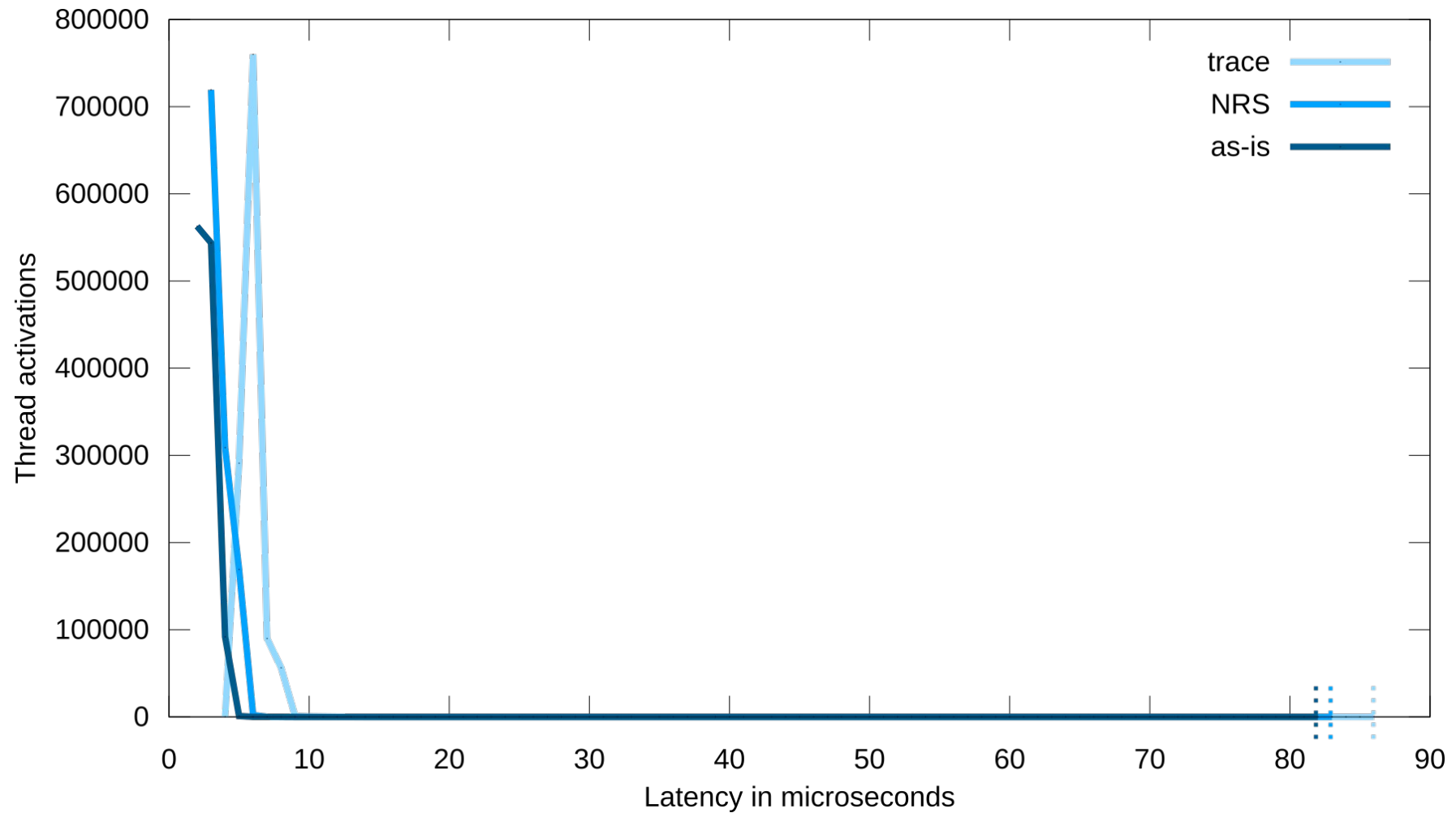
Benchmark: Throughput – Low kernel activation



Benchmark: Throughput – High kernel activation



Benchmark: Cyclicttest latency



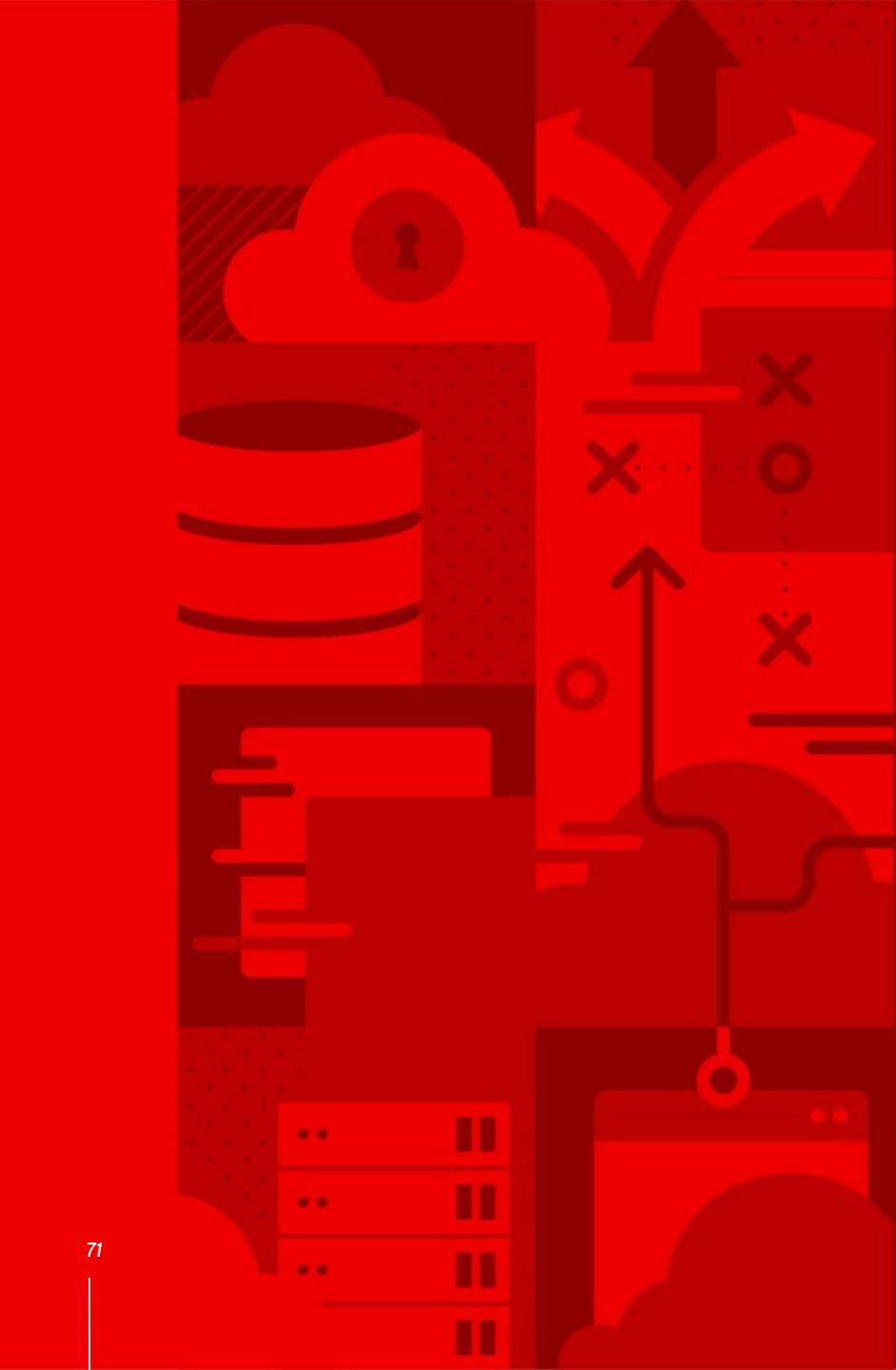
Academically accepted

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/>



So...

So...

- It is possible to model complex behavior of Linux
 - Using a formal language
 - Creating big models from small ones
- It is possible to verify properties of models
 - And so properties of the system
 - Bonus: It is possible to use other more complex methods by using the automata
 - LTL and so on
- It is possible to verify the runtime behavior of Linux

What's next?

- Better interface
 - Working in a perf/ebpf version of the runtime verification part
 - And also working with a “ftrace” like interface
 - Then I will compare both
- Documenting the process in a “linux developer way”
 - IOW: translating the papers into LWN articles

What should we model?

- There are other possible things to model
 - Locking (part of lockdep)
 - Why?
 - Run-time without recompile/reboot.
 - RCU?
 - Schedulers?



Worth Mentioning



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)