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

Untangling the Intricacies of Thread Synchronization in the PREEMPT_RT Linux Kernel

Daniel B. de Oliveira[1,2,3], Rômulo S. de Oliveira[3], Tommaso Cucinotta[2]
bristot@redhat.com, tommaso.cucinotta@santannapisa.it, romulo.deoliveira@ufsc.br,

Agenda

- Introduction
- Related Work
- Background on automata theory
- Proposed approach
- Application of the model
- Conclusions and future work

Linux as a RTOS

- Linux has been used as RTOS on many academic and industrial projects.
  - It has become a fundamental block of real-time distributed systems, e.g.:
    - Sensor Networks
    - Robotics
    - Factory automation
    - Military Drones
    - Distributed and service oriented multimedia systems
    - Distributed high frequency trading systems

Determinism on Linux

- The PREEMPT RT changes a set of in-kernel operations that enhance the deterministic operation of Linux.
- Operations, however, are not atomic.
  - Incurring in non-negligible delays;
  - Even for tasks that are not related.
- The understanding of these rules and how they affect the timing behavior of Linux are fundamental for the development of real-time applications and algorithms.

Complexity of Linux

- The in-kernel synchronization mechanisms are complex
  - They involve various task contexts (Threads, IRQs, NMI)
  - Low-level hardware details
  - Kernel hacks
  - Not a single place in the code to understand
- It may take years to understand them all
  - That is why many projects ignore them
    - But they end up not landing on Linux
- How can we explain Linux synchronization?
  - And what are the benefits of it?

Tracing and DES

- Linux developers use tracing features to analyse the system:
  - They see tracing *events* that cause *states* change of the system.
- Discrete Event Systems (DES) methods also use these concepts:
  - *events*, *trace* and *states*...
- DES can be used in the formalization of system.
- So, why not try to describe Linux using a DES method?

Paper contributions

- Proposes an automata-based model for describing and verifying the behavior of thread management code in the Linux kernel:
  - Considers he FULLY_PREEMPTIVE mode
  - Includes
    - IRQ/NMI (and its management)
    - Locking: Mutex, rw locks and semaphores
    - Scheduling

- Presents the extension of the Linux trace features that enables the trace of the events used by the automata in a real scenario.
- Presents how the model can be used to understand Linux
- Presents how the model helps catching bugs in Linux

Agenda

- Introduction
- Related Work
- Background on automata theory
- Proposed approach
- Application of the model
- Conclusions and future work

Related work: Automata

Automata and discrete-event systems have been extensively used to verify timing properties of real-time systems:

- Usage of timed automata for schedulability tests
  - Daws and Yovine - 1995
  - Cimatti, Palopoli, Ramadian - 2008
  - Wang, Li, Wonham - 2016
- To reduce the complexity of the system by using compositions of automata;
  - Lampka, Perathoner, and Thiele - 2013
- Schedulability analysis and code generation
  - Amnell, Fersman, Mokrushin, Pettersson, and Yi - 2004
- None of them explores the details of in-kernel (or complex os) mechanisms.

Related work: Formal verification

- Usage of BLAST tool with control flow automata, along with techniques for state-space reduction, applied to the verification of safety properties of OS drivers for the Linux and Microsoft Windows NT kernels.
  - Henzinger, Jhala, Majumdar, and Sutre (2002)
- MAGIC, a tool for automatic verification of sequential C programs against finite state machine specifications.
  - Chaki, Clarke, Groce, Jha, and Veith (2004)
  - MAGIC has been used to verify locking correctness (deadlock-freedom) in the Linux kernel.

Related work: Linux kernel

- lockdep mechanism built into the Linux kernel, capable of identifying errors in using locking primitives that might eventually lead to deadlocks.
- Linux Memory Model
  - Alglave, Maranget, McKenney, Parri, and Stern (2018)

A model for thread synchronization

To the best of our knowledge, none of these techniques ventured into the challenging goal of building a formal model for the understanding and validation of the Linux PREEMPT RT kernel code sections responsible for such low-level operations such as task scheduling, IRQ and NMI management, and their delicate interplay, as done in this paper.

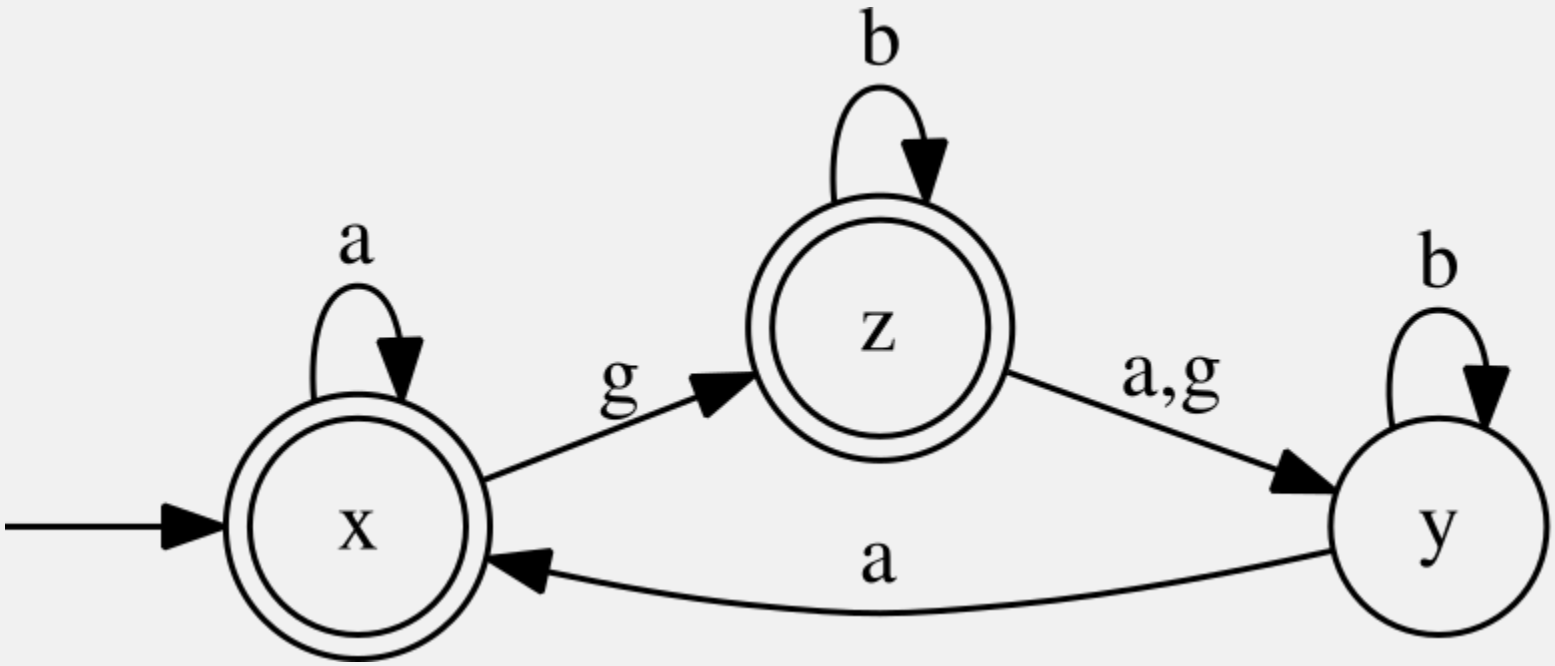
Agenda

- Introduction
- Related Work
- Background on automata theory
- Proposed approach
- Application of the model
- Conclusions and future work

Background

- Automata is a method to model Discrete Event Systems (DES)
- Formally, an automaton 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 = $(X \text{ x } E) \rightarrow X$;
    - x_0 = Initial state;
    - X_m = set of final states.
- The language - or traces - generated/recognized by G is the L(G).

Graphical format



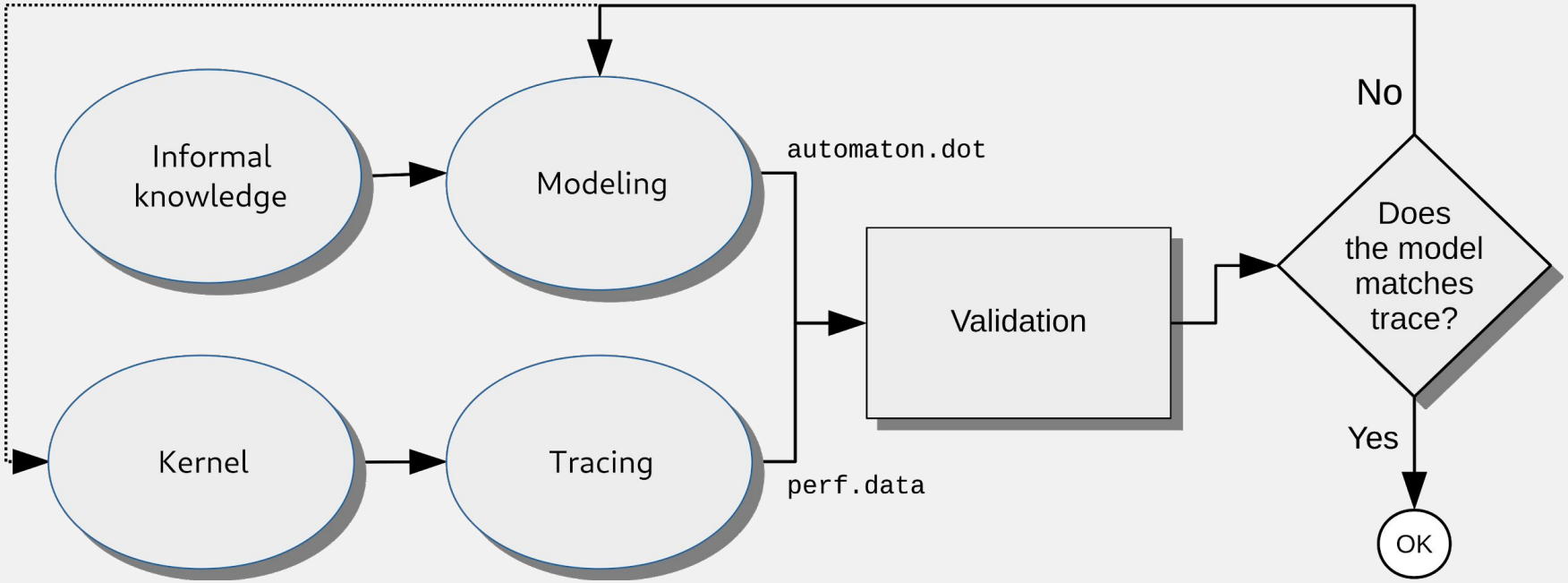
Modeling of complex systems

- Rather than modeling the system as a single automaton, the modular approach uses **generators** and **specifications.**
  - Generators:
    - Independent subsystems models
    - Generates all chain of events (without control)
  - Specification:
    - Control/synchronization rules of two or more subsystems
    - Blocks some events
- The parallel composition operation synchronizes the generators and specifications.
  - The result is an automaton with all chain of events possible in a controlled system.

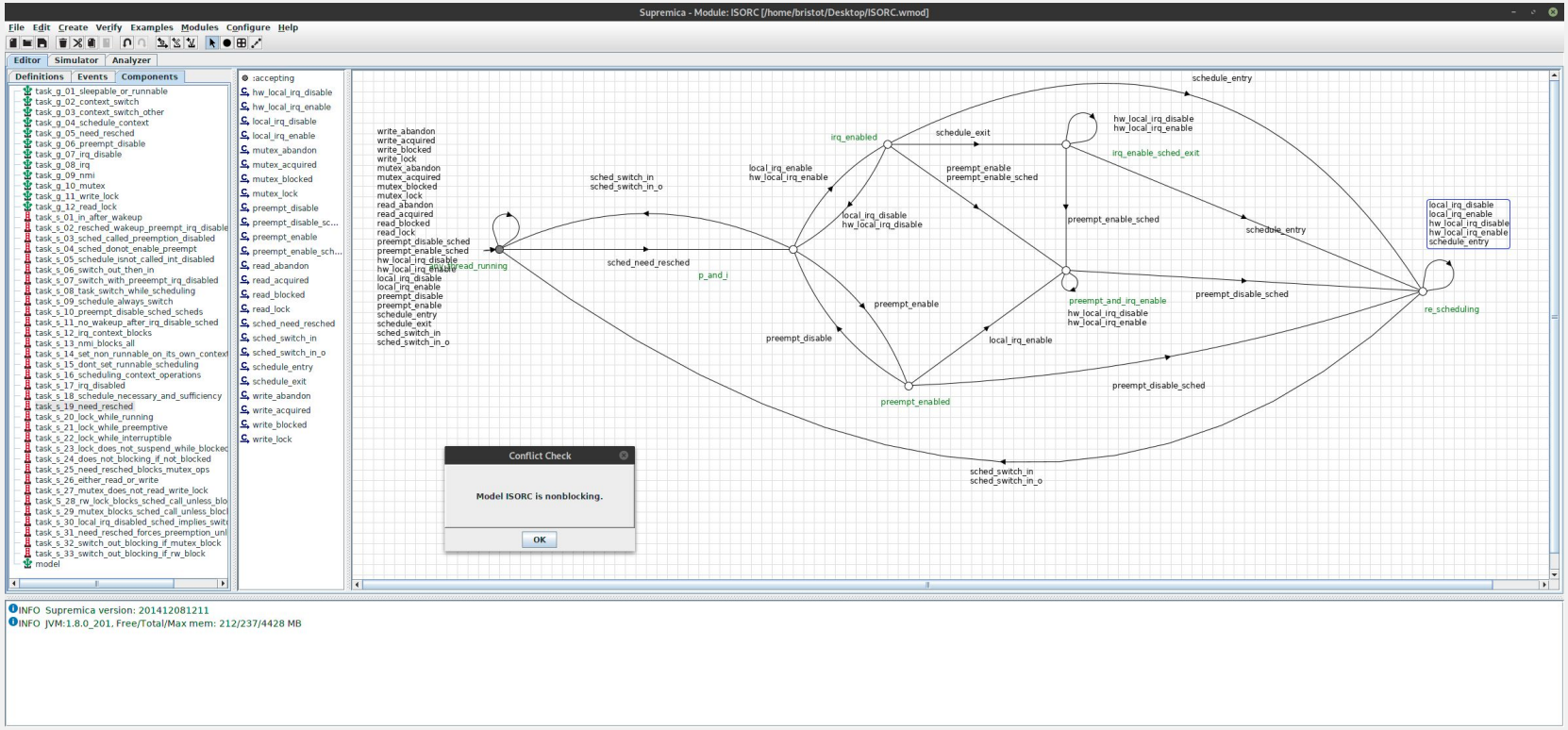
Agenda

- Introduction
- Related Work
- Background on automata theory
- Proposed approach
- Application of the model
- Conclusions and future work

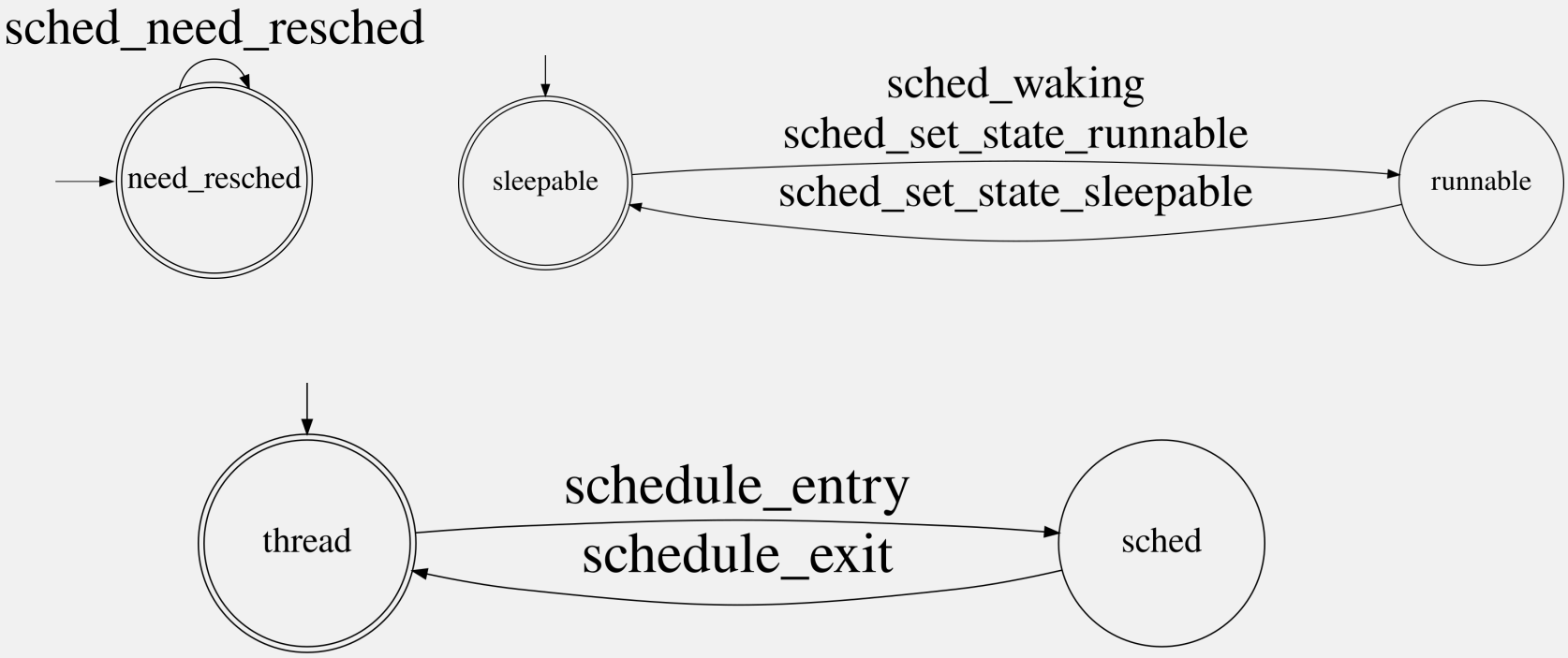
Proposed approach



Modeling


Supremica - Module: ISORC [/home/bristot/Desktop/ISORC.wmod]
File Edit Create Verify Examples Modules Configure Help
Editor Simulator Analyzer
Definitions Events Components
task_g_01_sleepable_or_runnable
task_g_02_context_switch
task_g_03_context_switch_other
task_g_04_schedule_context
task_g_05_need_resched
task_g_06_preempt_disable
task_g_07_irq_disable
task_g_08_irq
task_g_09_nmi
task_g_10_mutex
task_g_11_write_lock
task_g_12_read_lock
task_s_01_in_after_wakeup
task_s_02_resched_wakeup_preempt_irq_disable
task_s_03_sched_called_preemption_disabled
task_s_04_sched_donot_enable_preempt
task_s_05_schedule_!snpt_called_int_disabled
task_s_06_switch_out_then_in
task_s_07_switch_with_preempt_irq_disabled
task_s_08_task_switch_while_scheduling
task_s_09_schedule_always_switch
task_s_10_preempt_disable_sched_scheds
task_s_11_no_wakeup_after_irq_disable_sched
task_s_12_irq_context_blocks
task_s_13_nmi_blocks_all
task_s_14_set_non_runnable_on_its_own_context
task_s_15_dont_set_runnable_scheduling
task_s_16_scheduling_context_operations
task_s_17_irq_disabled
task_s_18_schedule_necessary_and_sufficiency
task_s_19_need_resched
task_s_20_lock_while_running
task_s_21_lock_while_preemptive
task_s_22_lock_while_interruptible
task_s_23_lock_does_not_suspend_while_blocked
task_s_24_does_not_blocking_if_not_blocked
task_s_25_need_resched_blocks_mutex_ops
task_s_26_either_read_or_write
task_s_27_mutex_does_not_read_write_lock
task_s_28_rw_lock_blocks_sched_call_unless_blo
task_s_29_mutex_blocks_sched_call_unless_bloc
task_s_30_local_irq_disabled_sched_implies_swit
task_s_31_need_resched_forces_preemption_unl
task_s_32_switch_out_blocking_if_mutex_block
task_s_33_switch_out_blocking_if_rw_block
model
:accepting
hw_local_irq_disable
hw_local_irq_enable
local_irq_disable
local_irq_enable
mutex_abandon
mutex_acquired
mutex_blocked
mutex_lock
preempt_disable
preempt_disable_sc...
preempt_enable
preempt_enable_sch...
read_abandon
read_acquired
read_blocked
read_lock
sched_need_resched
sched_switch_in
sched_switch_in_o
schedule_entry
schedule_exit
write_abandon
write_acquired
write_blocked
write_lock
schedule_entry
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
irq_enabled
schedule_exit
hw_local_irq_disable
hw_local_irq_enable
irq_enable_sched_exit
sched_switch_in
sched_switch_in_o
local_irq_enable
hw_local_irq_enable
preempt_enable
preempt_enable_sched
local_irq_disable
hw_local_irq_disable
preempt_enable_sched
schedule_entry
local_irq_disable
local_irq_enable
hw_local_irq_disable
hw_local_irq_enable
schedule_entry
sched_need_resched
p_and_i
preempt_enable
preempt_and_irq_enable
hw_local_irq_disable
hw_local_irq_enable
preempt_disable_sched
re_scheduling
preempt_disable
local_irq_enable
preempt_enabled
preempt_disable_sched
sched_switch_in
sched_switch_in_o
Conflict Check
Model ISORC is nonblocking.
OK
INFO Supremica version: 201412081211
INFO JVM:1.8.0_201, Free/Total/Max mem: 212/237/4428 MB

Example of generators: *G05, G01 and G04*



Automata & Kernel events

Model: IRQ events

Automaton event	Kernel event	Description
hw_local_irq_disable	**irq:local_irq_disable**	Begin IRQ handler
hw_local_irq_enable	**irq:local_irq_enable**	Return IRQ handler
local_irq_disable	**irq:local_irq_disable**	Mask IRQs
local_irq_enable	**irq:local_irq_enable**	Unmask IRQs
nmi_entry	**irq_vectors:nmi**	Begin NMI handler
nmi_exit	**irq_vectors:nmi**	Return NMI Handler

Model: Preemption/Scheduler related events

Automaton event	Kernel event	Description
preempt_disable	**sched:sched_preempt_disable**	Disable preemption
preempt_enable	**sched:sched_preempt_enable**	Enable preemption
preempt_disable_sched	**sched:sched_preempt_disable**	Disable preemption to call the scheduler
preempt_enable_sched	**sched:sched_preempt_enable**	Enables preemption returning from the scheduler
schedule_entry	**sched:sched_entry**	Begin of the scheduler
schedule_exit	**sched:sched_exit**	Return of the scheduler
sched_need_resched	**sched:set_need_resched**	Set need resched

Model: State of threads events

Automaton event	Kernel event	Description
sched_waking	sched:sched_waking	Activation of a thread
sched_set_state_runnable	**sched:sched_set_state**	Thread is runnable
sched_set_state_sleepable	**sched:sched_set_state**	Thread can go to sleepable

Model: Context switch events

Automaton event	Kernel event	Description
sched_switch_in	sched:sched_switch	Switch in of the thread under analysis
sched_switch_suspend	sched:sched_switch	Switch out due to a suspension of the thread under analysis
sched_switch_preempt	sched:sched_switch	Switch out due to a preemption of the thread under analysis
sched_switch_blocking	sched:sched_switch	Switch out due to a blocking of the thread under analysis
sched_switch_in_o	sched:sched_switch	Switch in of another thread
sched_switch_out_o	sched:sched_switch	Switch out of another thread

Model: Mutex events

Automaton event	Kernel event	Description
mutex_lock	**lock:rt_mutex_lock**	Requested a RT Mutex
mutex_blocked	**lock:rt_mutex_block**	Blocked in a RT Mutex
mutex_acquired	**lock:rt_mutex_acquired**	Acquired a RT Mutex
mutex_abandon	**lock:rt_mutex_abandon**	Abandoned the request of a RT Mutex

Model: Write lock events

Automaton event	Kernel event	Description
write_lock	**lock:rwlock_lock**	Requested a R/W Lock or Sem as writer
write_blocked	**lock:rwlock_block**	Blocked in a R/W Lock or Sem as writer
write_acquired	**lock:rwlock_acquired**	Acquired a R/W Lock or Sem as writer
write_abandon	**lock:rwlock_abandon**	Abandoned a R/W Lock or Sem as writer

Model: Read lock events

Automaton event	Kernel event	Description
read_lock	**lock:rwlock_lock**	Requested a R/W Lock or Sem as reader
read_blocked	**lock:rwlock_block**	Blocked in a R/W Lock or Sem as reader
read_acquired	**lock:rwlock_acquired**	Acquired a R/W Lock or Sem as reader
read_abandon	**lock:rwlock_abandon**	Abandoned a R/W Lock or Sem as reader

Generators and Specifications

Components: Generators

Name	*States*	*Events*	*Transitions*
G01 Sleepable or runnable	**2**	**3**	**3**
G02 Context switch	2	4	4
G03 Context switch other thread	**2**	**2**	**2**
G04 Scheduling context	2	2	2
G05 Need resched	**1**	**1**	**1**
G06 Preempt disable	3	4	4
G07 IRQ Masking	**2**	**2**	**2**
G08 IRQ handling	2	2	2
G09 NMI	**2**	**2**	**2**
G10 Mutex	3	4	6
G11 Write lock	**3**	**4**	**6**
G12 Read lock	3	4	6

Components: Specifications (part 1)

Name	*States*	*Events*	*Transitions*
S01 Sched in after wakeup	**2**	**3**	**5**
S02 Resched and wakeup sufficiency	3	10	18
S03 Scheduler with preempt disable	**2**	**4**	**4**
S04 Scheduler doesn't enable preemption	2	6	6
S05 Scheduler with interrupt enabled	**2**	**4**	**4**
S06 Switch out then in	2	20	20
S07 Switch with preempt/irq disabled	**3**	**10**	**14**
S08 Switch while scheduling	2	8	8
S09 Schedule always switch	**3**	**6**	**6**
S10 Preempt disable to sched	2	3	4
S11 No wakeup right before switch	**3**	**5**	**8**
S12 IRQ context disable events	2	27	27
S13 NMI blocks all events	**2**	**34**	**34**
S14 Set sleepable while running	2	6	6
S15 Don't set runnable when scheduling	**2**	**4**	**4**
S16 Scheduling context operations	2	3	3

Components: Specifications (part 2)

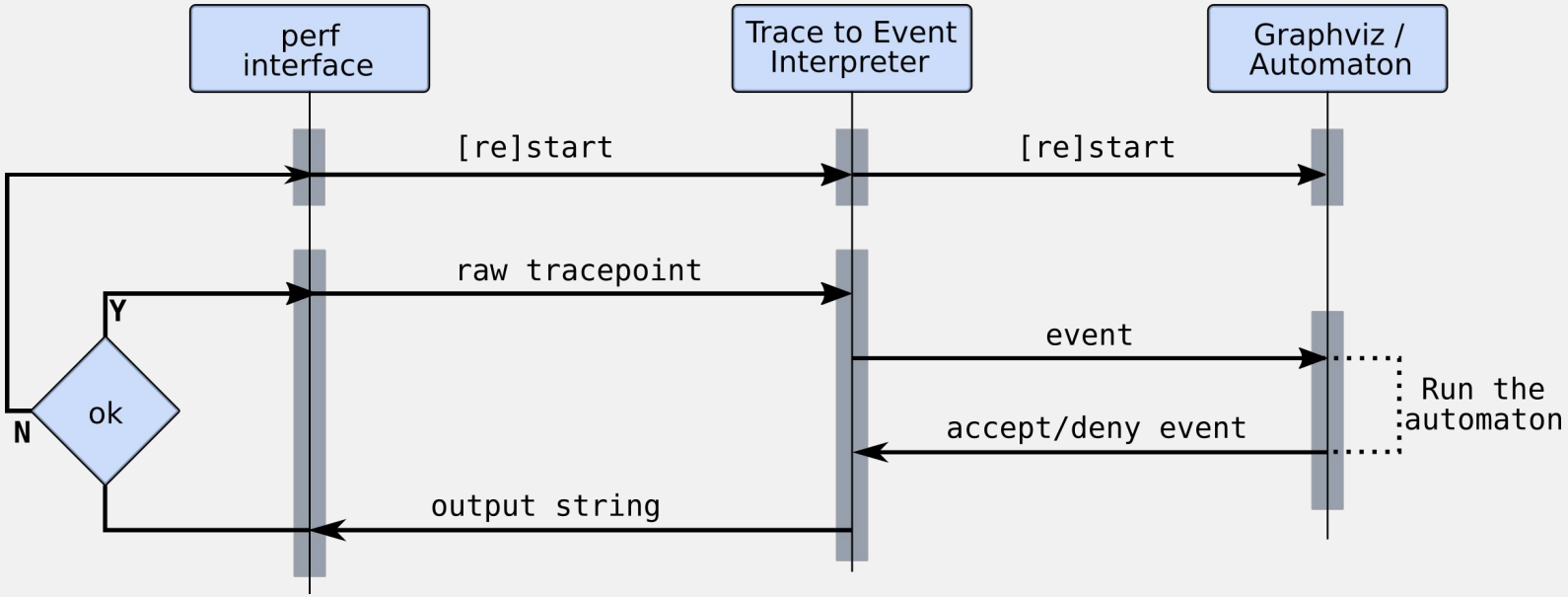
Name	*States*	*Events*	*Transitions*
S17 IRQ disabled	**3**	**4**	**4**
S18 Schedule necessary and sufficient	7	9	22
S19 Need resched forces scheduling	**7**	**27**	**59**
S20 Lock while running	2	16	16
S21 Lock while preemptive	**2**	**16**	**16**
S22 Lock while interruptible	2	16	16
S23 No suspension in lock algorithms	**3**	**10**	**19**
S24 Sched blocking if blocks	3	10	20
S25 Need resched blocks lock ops	**2**	**15**	**17**
S26 Lock either read or write	3	6	6
S27 Mutex doesn't use rw lock	**2**	**11**	**11**
S28 RW lock does not sched unless block	4	11	22
S29 Mutex does not sched unless block	**4**	**7**	**16**
S30 Disable IRQ in sched implies switch	5	6	10
S31 Need resched preempts unless sched	**3**	**5**	**11**
S32 Does not suspend in mutex	3	5	11
S33 Does not suspend in rw lock	**3**	**8**	**16**

Model

- The final model has:
  - 13906 states;
  - 31708 transitions;
  - It would be impossible to model it directly.
- Using the modular approach, the final model is composed of:
  - 34 events;
  - 12 generators;
  - 33 specifications.
    - The most complex module (a specification) has 7 states!

Model Verification

Model Validation: *perf task model*



perf task_model output

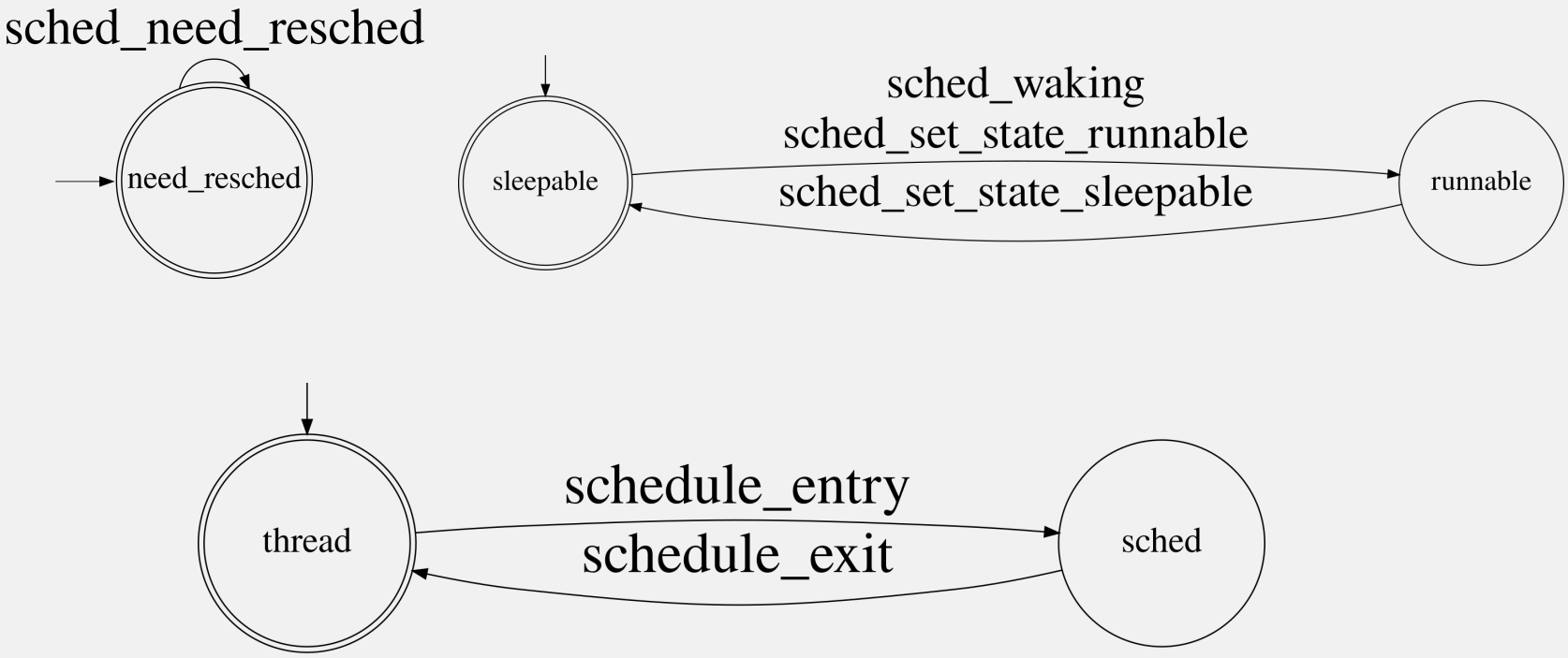
```
 1: Reference model: isorc.dot
 2: +----->  +=thread of interest - .=other threads
 3: | +-> T=Thread - I=IRQ - N=NMI
 4: | |
 5: | |     TID |   timestamp   | cpu |           event               | state   | safe?
 6: . T       8    436.912532    [000]              preempt_enable ->       q0   safe
 7: . T       8    436.912534    [000]            local_irq_disable ->    q8102
 8: . T       8    436.912535    [000]              preempt_disable ->   q19421
 9: . T       8    436.912535    [000]                 sched_waking ->      q99
10: . T       8    436.912535    [000]           sched_need_resched ->   q14076
11: . T       8    436.912535    [000]             local_irq_enable ->    q1965
12: . T       8    436.912536    [000]               preempt_enable ->   q12256
13: . T       8    436.912536    [000]        preempt_disable_sched -> q18615,q23376
14: . T       8    436.912536    [000]               schedule_entry -> q16926,q17108,q2649
15: . T       8    436.912537    [000]            local_irq_disable -> q11700,q14046,q21391
16: . T       8    436.912537    [000]           sched_switch_out_o -> q10337,q20018,q21933
17: . T       8    436.912537    [000]              sched_switch_in -> q10268,q20126
18: + T    1840    436.912537    [000]             local_irq_enable ->   q20036
19: + T    1840    436.912538    [000]                schedule_exit ->   q21033
```

Agenda

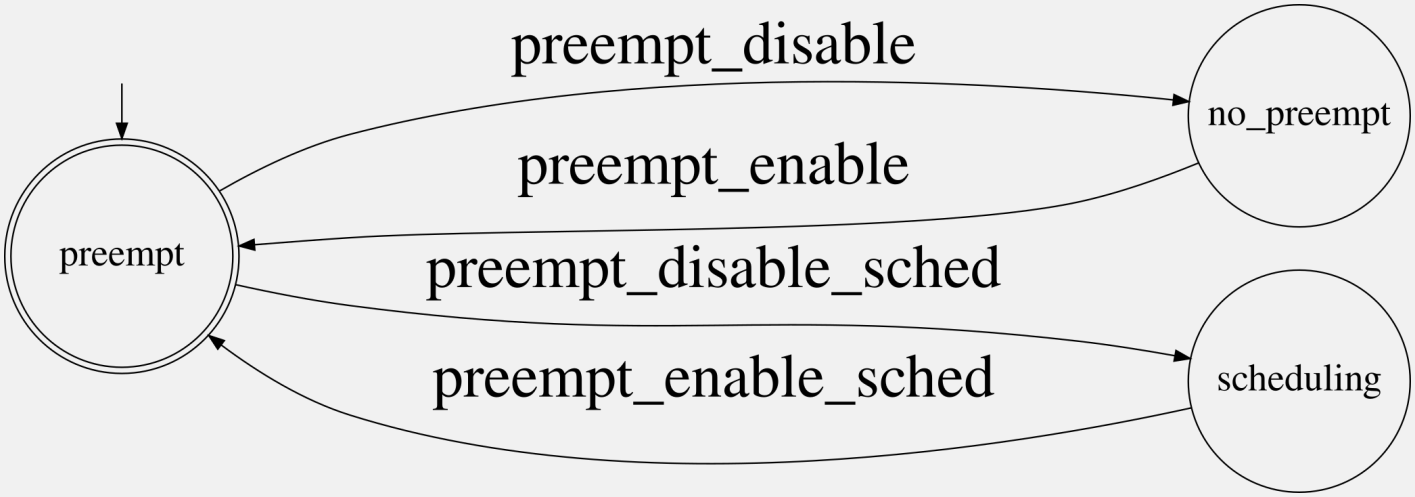
- Introduction
- Related Work
- Background on automata theory
- Proposed approach
- Application of the model
- Conclusions and future work

Example of application of the Model: Analysis of activation of the highest priority thread.

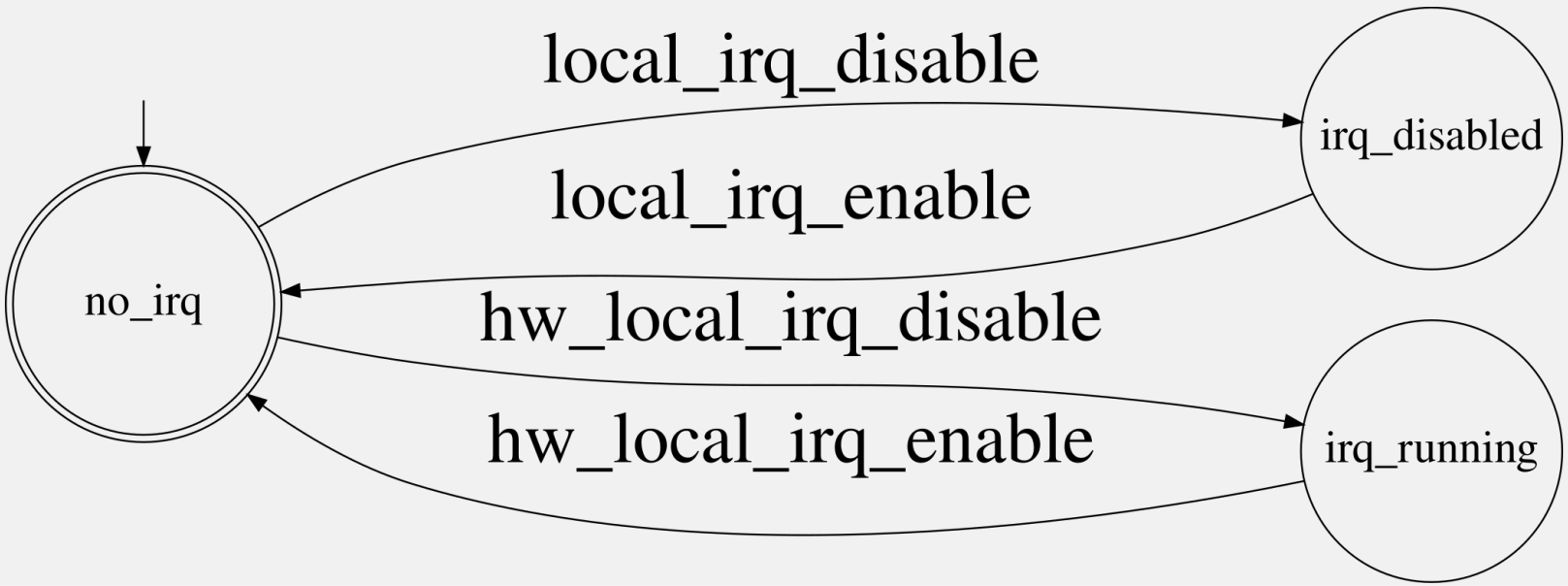
Highest priority task activation: Generators



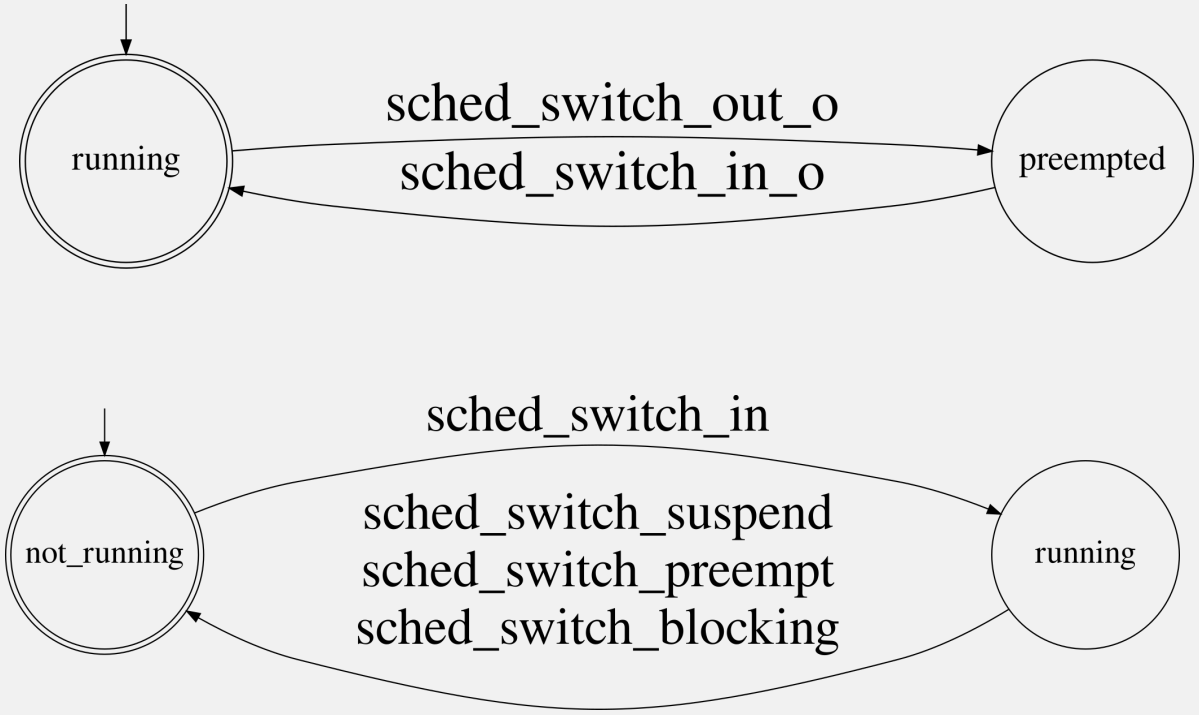
Highest priority task activation: Generators



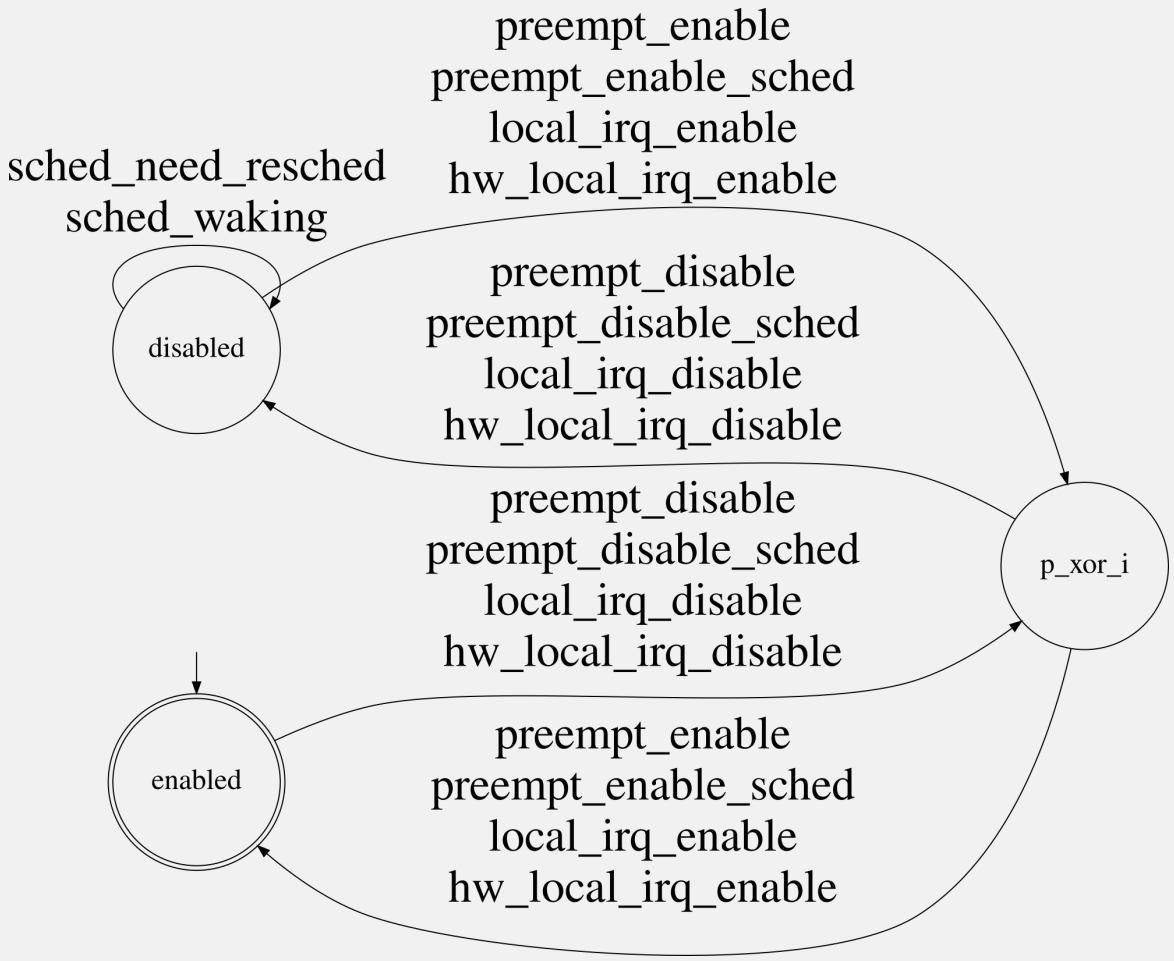
Highest priority task activation: Generators



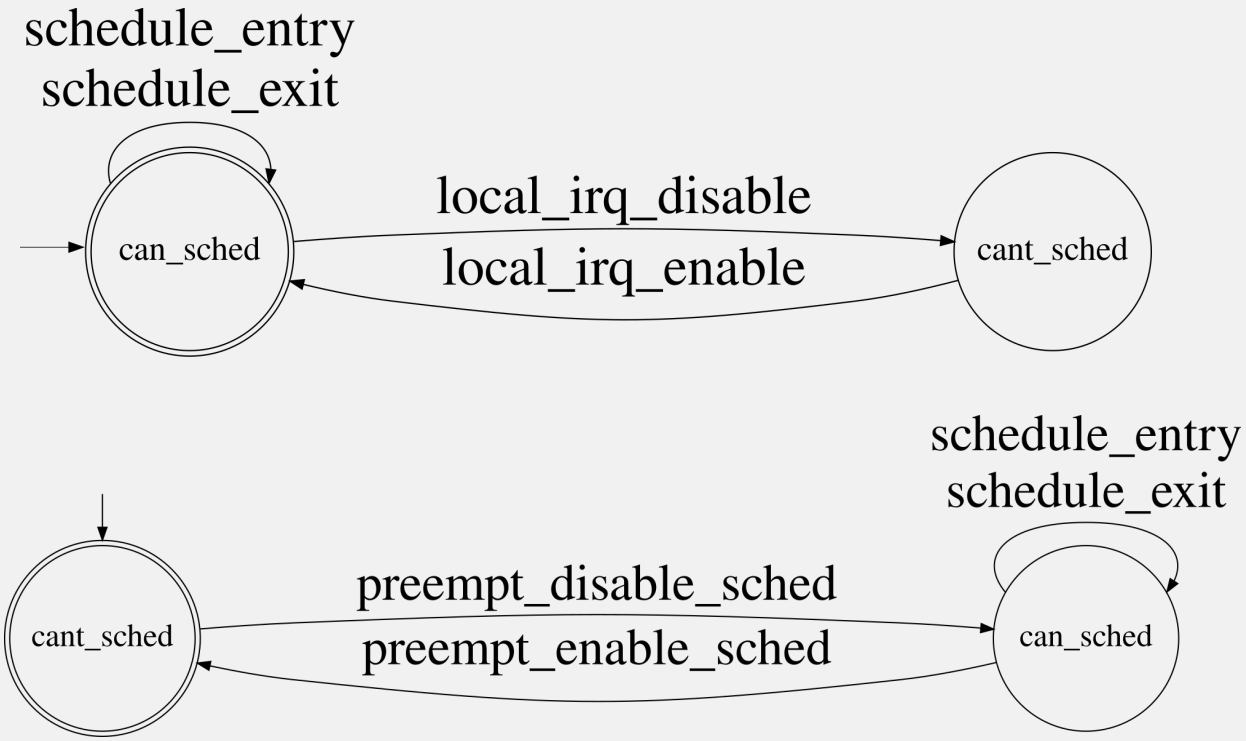
Highest priority task activation: Generators



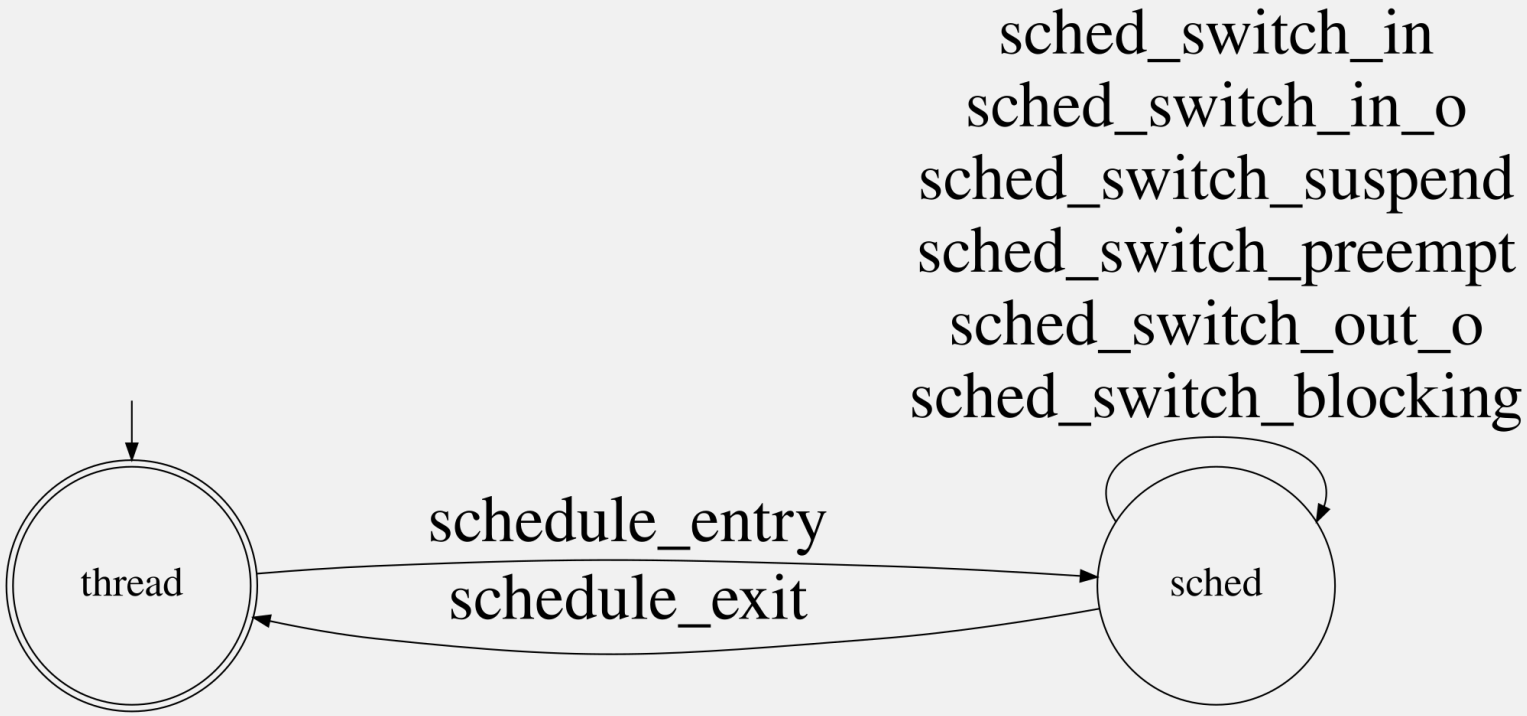
Highest priority task activation: Specifications



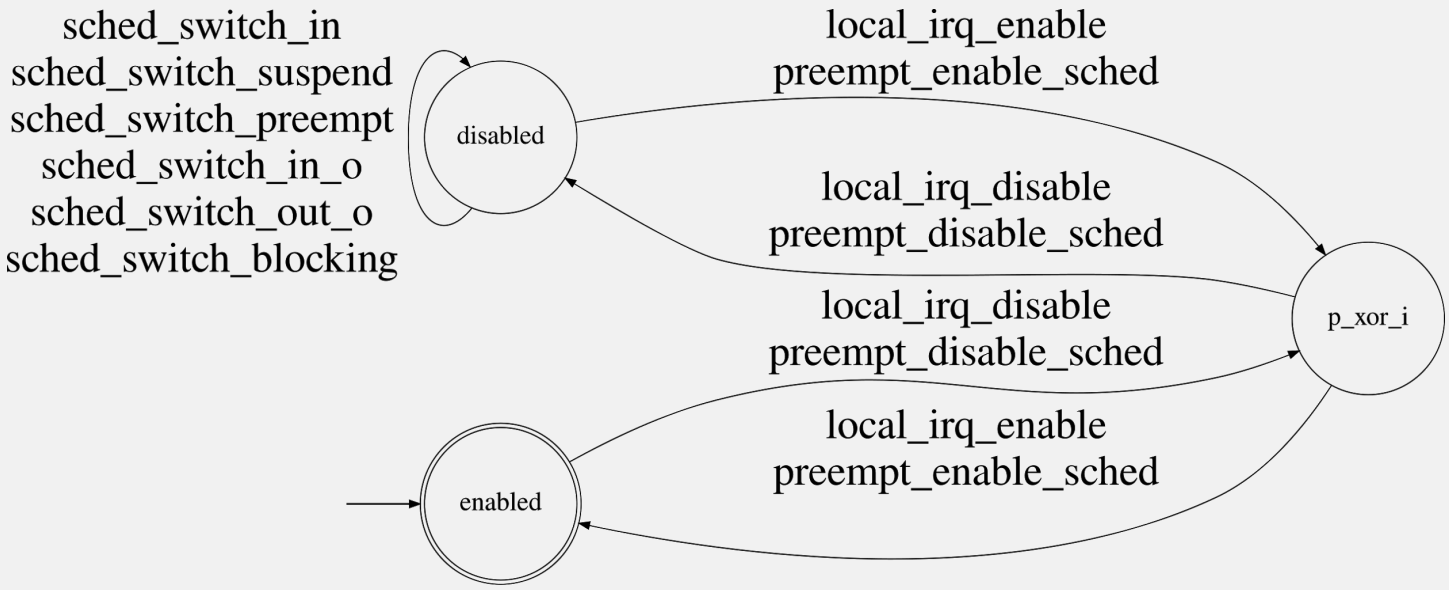
Highest priority task activation: Specifications



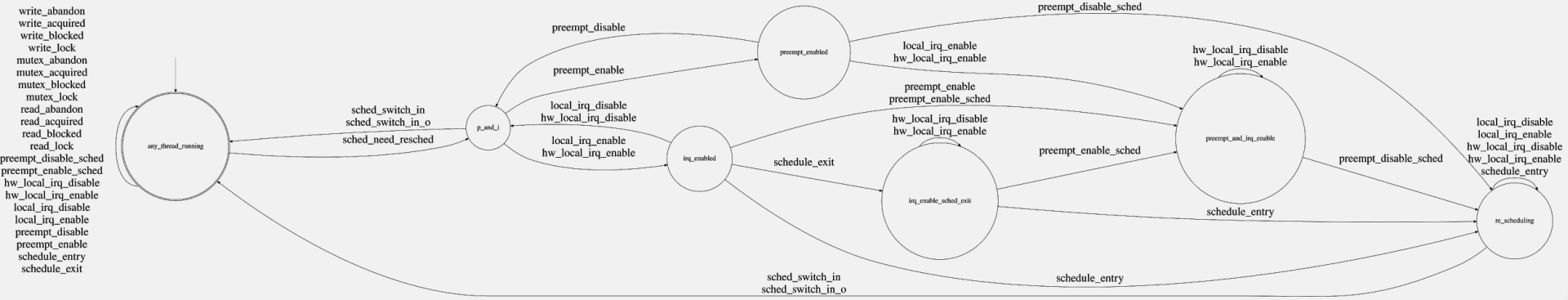
Highest priority task activation: Specifications



Highest priority task activation: Specifications



Highest priority task activation: Specifications



Other results

- By modeling the *expected* behavior, we can catch cases in which the kernel does not behave as expected.
  - We found two problems on kernel
    - One unexpected call to schedule():
      - Schedule called in vain.
        - Resulted in a kernel patch.
    - Perf & Ftrace missing events:
      - It is a problem in the trace recursion control
        - Patch suggested and under discussion

Agenda

- Introduction
- Related Work
- Background on automata theory
- Proposed approach
- Early results
- Conclusions and future work

Conclusion

- The definition of the operations that affect the timing behavior of threads are fundamental for the improvement of real-time Linux.
- Linux is complex! But the complexity was successfully “broken” by using the modular approach.
- The model is useful to understand the kernel dynamics.
- But also to find problems in the kernel!

Future work

- Model the multicore behavior.
- Try to fit the approach in existing schedulability analysis methods.
- Create more efficient model checker for Linux.
  - To find regressions: both timing and logical regressions.
  - This was discussed with developers and it is a wish.

All is OPEN

- All the source code (Kernel, model,...) are available at the Linux Task Model page:
  - http://bristot.me/linux-task-model/

Thanks!

Questions?