Red Hat
Enterprise Linux

RV: where are we?

Daniel Bristot de Oliveira
Senior Principal Software Engineer

Red Hat

Runtime Verification

- What is RV?
- Where are we from?
- RV and safety critical systems
- Where are we?
    - Current RV structure in kernel
- What is next

The initial patch set was under submission when I submitted this topic, but it was merged on 6.0!

So we will have a look at the current structure and talk about what is next!

Runtime Verification

- Runtime Verification (RV) is a lightweight (yet rigorous) **formal method** that with a more practical approach for complex systems.
- Instead of relying on a fine-grained model of a system (e.g., a re-implementation a instruction level), **RV works by analyzing the trace of the system's actual execution**, comparing it against a formal specification of the system behavior.

This topic was explored during Daniel's PhD, where he used it as the basis for the creation of an preemption model for the PREEMPT_RT, then used to proof the scheduling latency bound for the kernel-rt

Runtime Verification: IOW

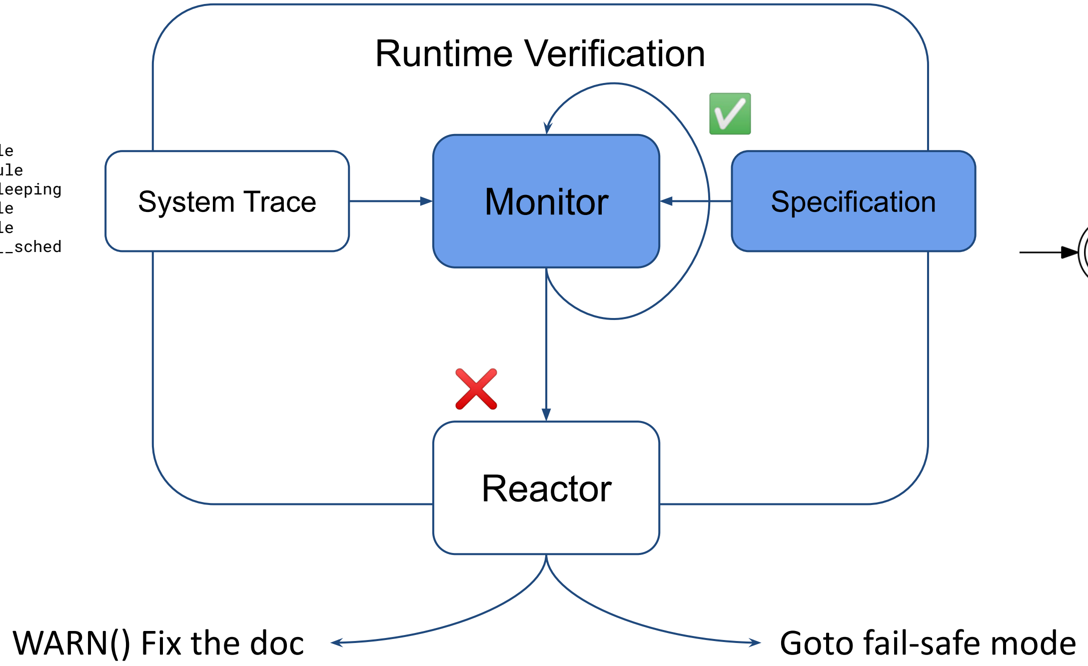
- As the system runs, it generates events to be analyzed
- These events are analyzed against a well-defined description of the system
  - Online Synchronous if it blocks the system
  - Online Asynchronous if it does not block
  - Offline if it verifies the system based on a trace file
- The analysis produces a verdict
- The system can react to an unexpected event
  - Only makes sense for online RV

The current supported method is online synchronous.

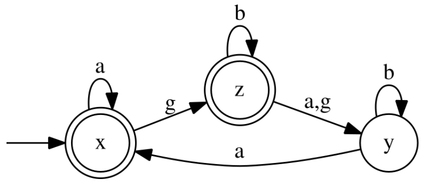
Runtime Verification on Linux

Linux Realm

```
247309: schedule <-worker_thread
247309: preempt_count_add <-schedule
247309: wq_worker_sleeping <-schedule
247309: kthread_data <-wq_worker_sleeping
247310: preempt_count_sub <-schedule
247310: preempt_count_add <-schedule
247310: rcu_note_context_switch <-__sched
```



Formal Realm



Why formal?

- Because it is precise and unambiguous
  - It is about reasoning, not about code
  - Math is math
  - See this "The Man Who Revolutionized Computer Science With Math*": https://www.youtube.com/watch?v=rkZzg7Vowao
- It is possible to analyze formal properties of your reasoning
  - Does the logic have contradictions?
  - Is the reasoning deadlock free?
- It is closer to other formal types of demonstration
  - Like the demonstration of the scheduling latency (my talk at LPC 2020).
- It helps to document the code
  - And adds value to the documentation for safety critical systems

We currently support only one formalism, but more methods are are under development.

* The man is Laslie Lamport, not me :-)

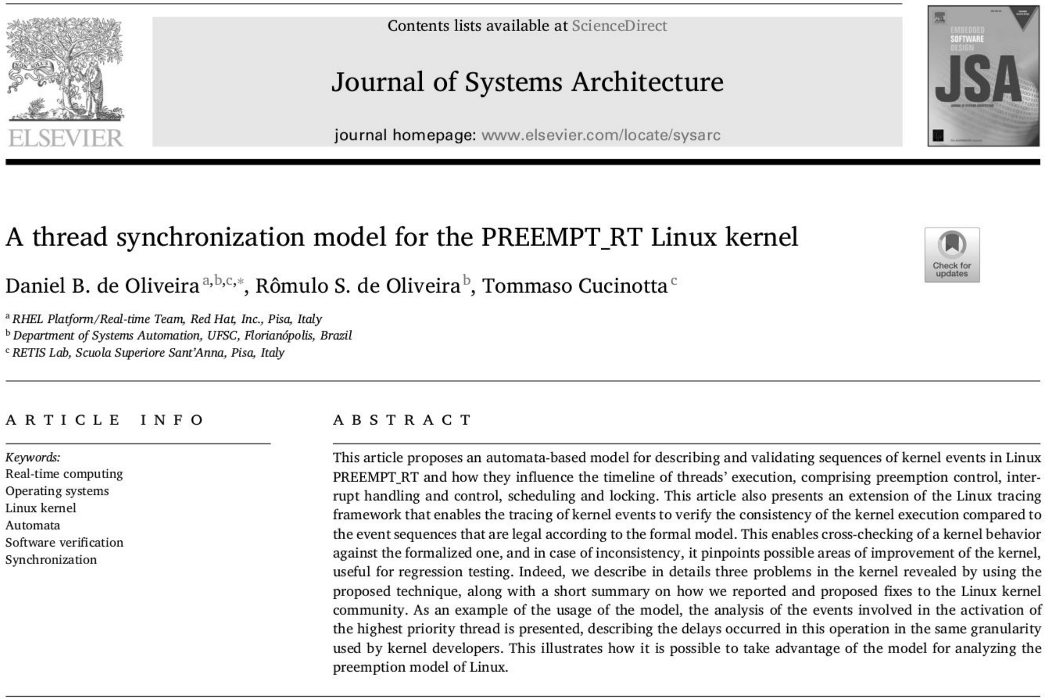
Where are we from?

How it started

- How can I demonstrate the bound for the scheduling latency?
  - In a way that I could convince theoretical researchers
  - But that could also be meaning full for Linux people
- I solved this problem using an automaton model explaining the PREEMPT_RT synchronization.
  - LPC 2018 talk Mind the gap - between real-time Linux and real-time theory
  - Used the model specifications to derive a bound for the scheduling latency
  - LPC 2020 talk: "A theorem for the RT scheduling latency (and a measuring tool too!)

Automata for complex models

- Automata is a formal method with a large set of operations for composition and validation
- A large automata can be built form a set of small automaton.
- The PREEMPT_RT thread model has:
    - +9k states
    - +21k transitions
    - Build on a set of automata
        - The vast majority of the automaton modules have 2/3 states
        - The largest has 10 states
- (Scientific) journal paper on the subject ->



Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

ELSEVIER

A thread synchronization model for the PREEMPT_RT Linux kernel

Daniel B. de Oliveira [a,b,c,*], Rômulo S. de Oliveira [b], Tommaso Cucinotta [c]

[a] RHEL Platform/Real-time Team, Red Hat, Inc., Pisa, Italy
[b] Department of Systems Automation, UFSC, Florianópolis, Brazil
[c] RETIS Lab, Scuola Superiore Sant'Anna, Pisa, Italy

A R T I C L E I N F O

Keywords:
Real-time computing
Operating systems
Linux kernel
Automata
Software verification
Synchronization

A B S T R A C T

This article proposes an automata-based model for describing and validating sequences of kernel events in Linux PREEMPT_RT and how they influence the timeline of threads' execution, comprising preemption control, interrupt handling and control, scheduling and locking. This article also presents an extension of the Linux tracing framework that enables the tracing of kernel events to verify the consistency of the kernel execution compared to the event sequences that are legal according to the formal model. This enables cross-checking of a kernel behavior against the formalized one, and in case of inconsistency, it pinpoints possible areas of improvement of the kernel, useful for regression testing. Indeed, we describe in details three problems in the kernel revealed by using the proposed technique, along with a short summary on how we reported and proposed fixes to the Linux kernel community. As an example of the usage of the model, the analysis of the events involved in the activation of the highest priority thread is presented, describing the delays occurred in this operation in the same granularity used by kernel developers. This illustrates how it is possible to take advantage of the model for analyzing the preemption model of Linux.

Red Hat

Automata for RV

- Automata is a well define formalism
- 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 = 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).

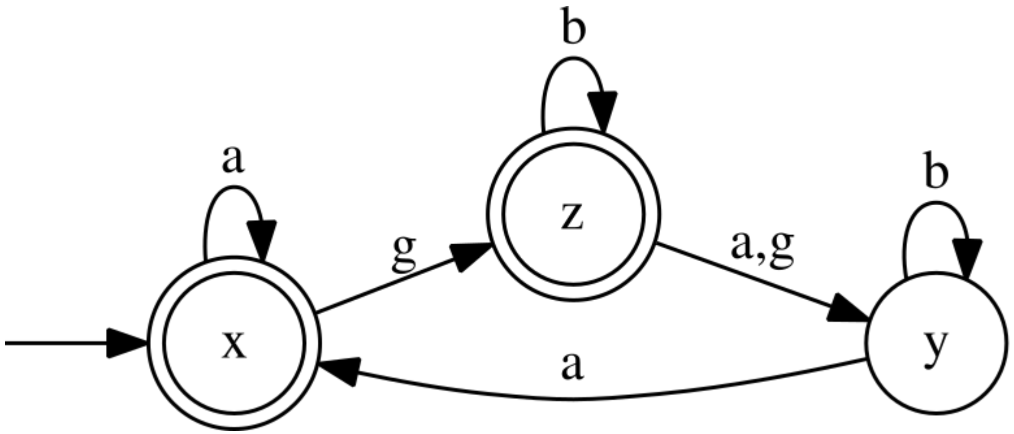
There are multiple different automata definitions, in this case we are talking about Deterministic Automata (DA)

Kernel doc explains it!

Red Hat

Automata for RV

- The good thing about automata is that it also has an graphical representation:



A state-machine is a sort of automata! It is one of the basis of CS.

Using automata to prove things

- It worked! By explaining the logical behavior, we derived a timing bound!
- While developing and testing the model ended up finding kernel bugs
    - So people asked my, why not make it generic?

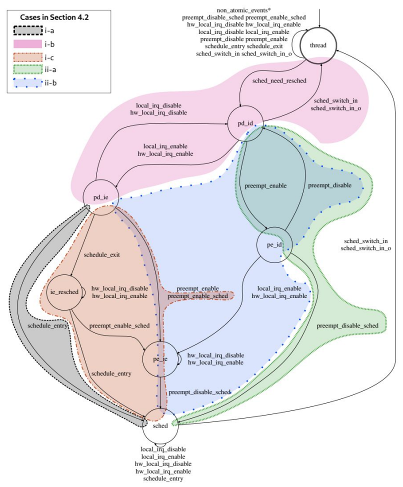
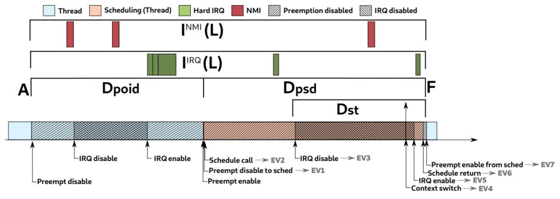


Figure 21 Setting need resched always causes a context switch (R14).



Demystifying the Real-Time Linux Scheduling Latency

Daniel Bristot de Oliveira
Red Hat, Inc, Italy
bristot@redhat.com

Daniel Casini
Scuola Superiore Sant'Anna, Italy
daniel.casini@santannapisa.it

Rômulo Silva de Oliveira
Universidade Federal de Santa Catarina, Brazil
romulo.deoliveira@ufsc.br

Tommaso Cucinotta
Scuola Superiore Sant'Anna, Italy
tommaso.cucinotta@santannapisa.it

▶ **Lemma 7.**

$$L^{IF} \leq max(D_{ST}, D_{POID}) + D_{PAIE} + D_{PSD}, \tag{6}$$

Proof. The lemma follows by noting that cases (i-a), (i-b), (i-c), (ii-a), (ii-b) are mutually-exclusive and cover all the possible sequences of events from the occurrence of `RHP`$_i$ and `set_need_resched`, to the time instant in which τ_i^{THD} is allowed to execute (as required by Definition 1), and the right-hand side of Equation 6 simultaneously upper bounds the right-hand sides of Equations 2, 3, 4, and 5. ◀

Theorem 8 summarizes the results derived in this section.

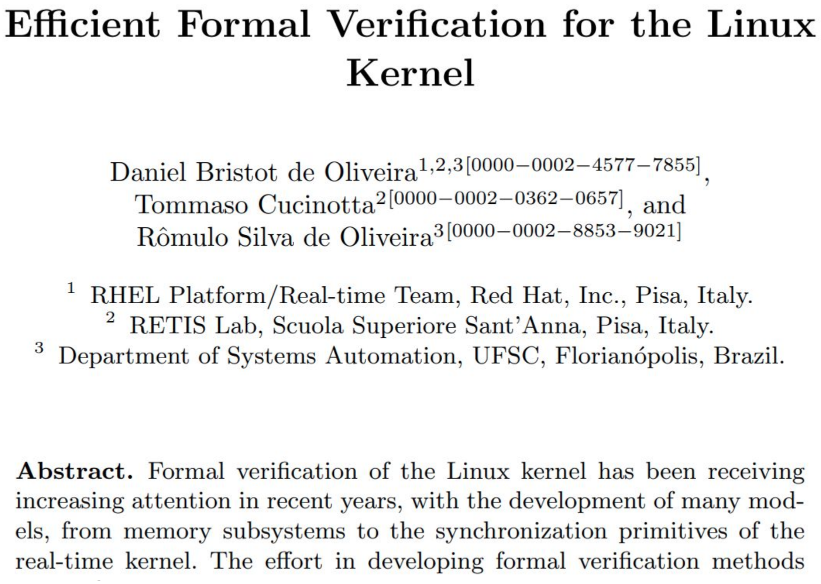
▶ **Theorem 8.** *The scheduling latency experienced by an arbitrary thread τ_i^{THD} is bounded by the least positive value that fulfills the following recursive equation:*

$$L = max(D_{ST}, D_{POID}) + D_{PAIE} + D_{PSD} + I^{NMI}(L) + I^{IRQ}(L) \tag{7}$$

Proof. The theorem follows directly from Lemmas 7 and Equation 1. ◀

Efficient automata verification

- The PREEMPT_RT model was a super-high-frequency one
  - Multiple events per microsecond
- Doing it in user-space was not efficient due to trace buffer overhead
- So we developed a way to transform the automata into code and run it in the kernel
  - O(1) operating
  - It was faster than just tracing
    - Because it is a simpler operation
- And we ended up finding more bugs

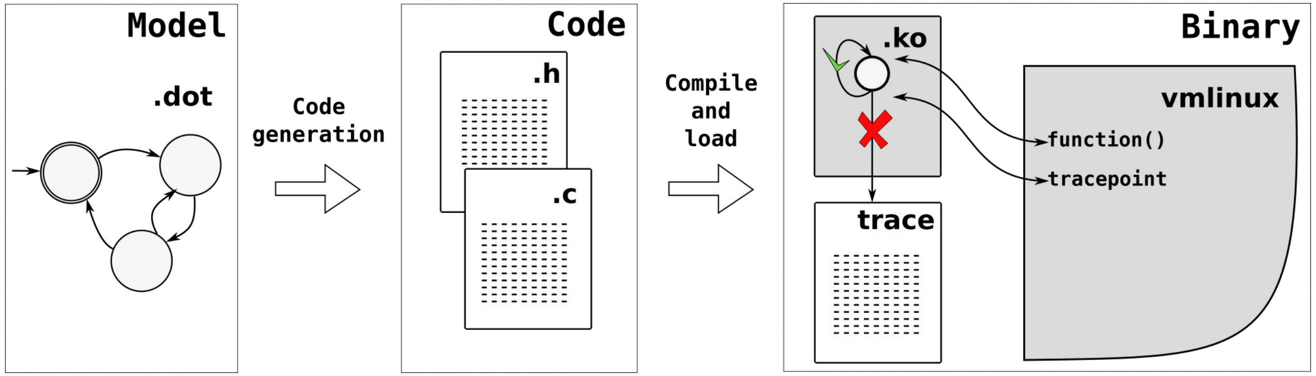


Efficient Formal Verification for the Linux Kernel

Daniel Bristot de Oliveira[1,2,3][0000−0002−4577−7855],
Tommaso Cucinotta[2][0000−0002−0362−0657], and
Rômulo Silva de Oliveira[3][0000−0002−8853−9021]

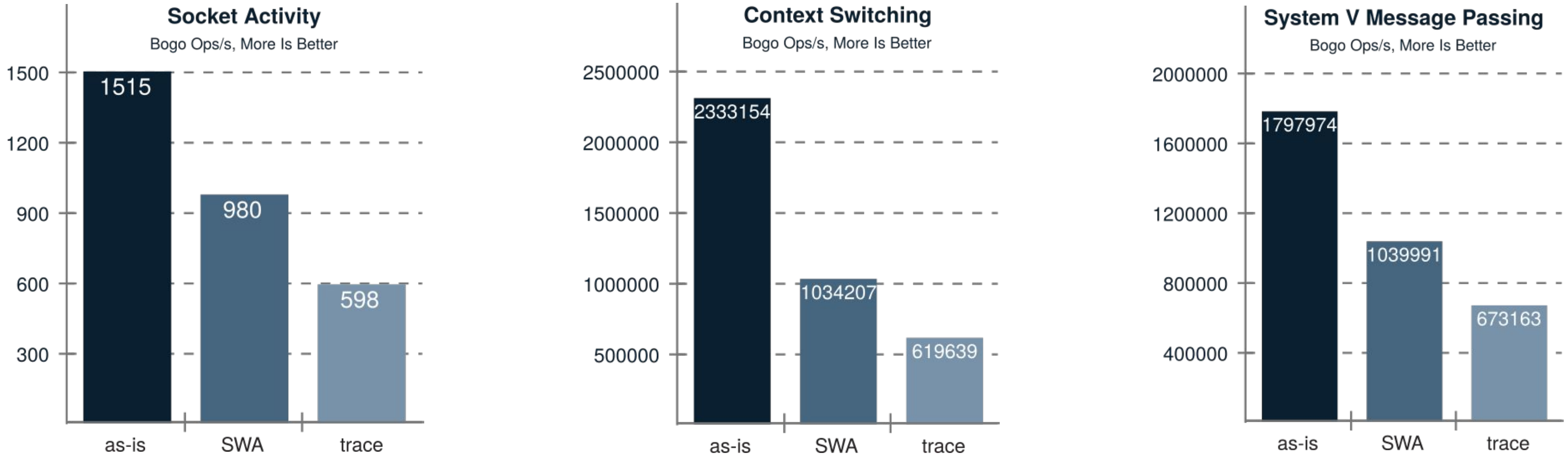
[1] RHEL Platform/Real-time Team, Red Hat, Inc., Pisa, Italy.
[2] RETIS Lab, Scuola Superiore Sant'Anna, Pisa, Italy.
[3] Department of Systems Automation, UFSC, Florianópolis, Brazil.

Abstract. Formal verification of the Linux kernel has been receiving increasing attention in recent years, with the development of many models, from memory subsystems to the synchronization primitives of the real-time kernel. The effort in developing formal verification methods



End efficient

- Running RV code in kernel is faster than tracing it!



RV meets safety-critical systems

RV and safety-critical system

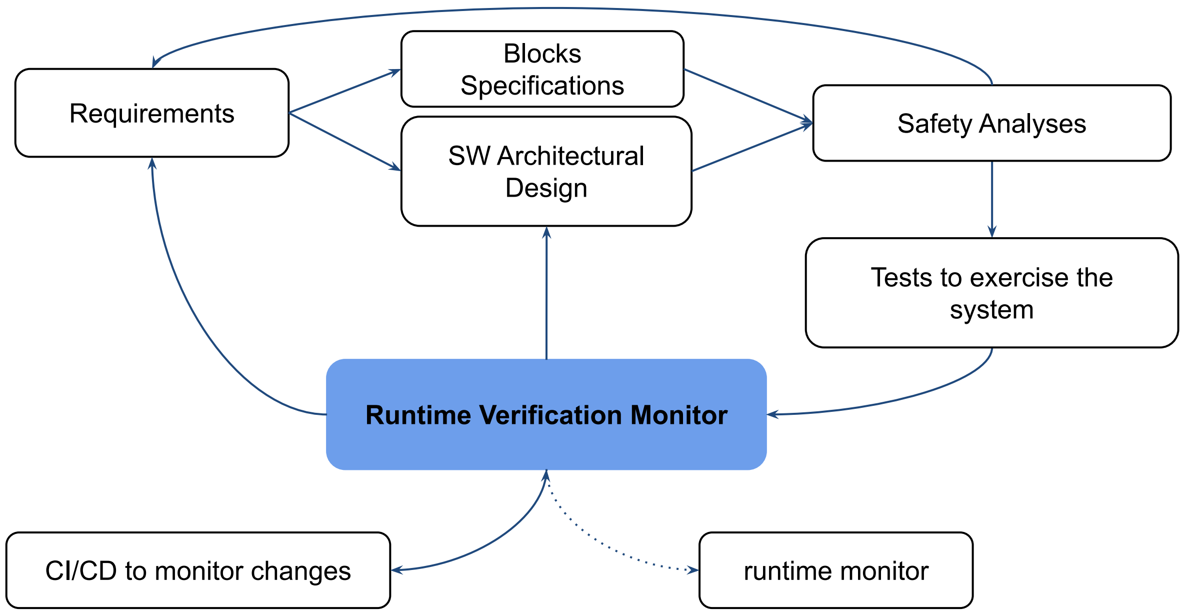
- RV uses formal-method
  - Which is at the top of the list of approaches to use for certification
- Three birds with a single stone!
  - Documents the system using automata
  - It verifies the system at the development/testing phases
  - It monitors the system at runtime
    - Reacting to unexpected events
- It only uses well-established technologies in the safety-critical fields:
  - Pure and straightforward C code
  - Statically allocated memory
  - No loops and so on

Talk at ELCE 2019:
Formal Verification Made Easy
(and fast!)

https://www.youtube.com/watch?v=BfTuEHafNgg

RV and safety-critical system

- I started working with people in the Elisa Project
- Gabriele Paoloni and I are working on the approach presented at LPC 2021



Red Hat

RV and safety-critical system

- RV can be used to document the kernel
- But also to document how the system should behave:
  - How the kernel is suppose to behave
  - How the user is supposed to behave
- It can be used to split subsystems into small parts
  - Each small part can be qualified as a black box (ISO 26262 part 12)
  - RV monitors the interface (ISO 26262 part 6)
- Further information:
  - See talk ->
  - Together with Gabriele Paoloni

A Maintainable, Scalable, and Verifiable SW Qualification Approach for Automotive in Linux

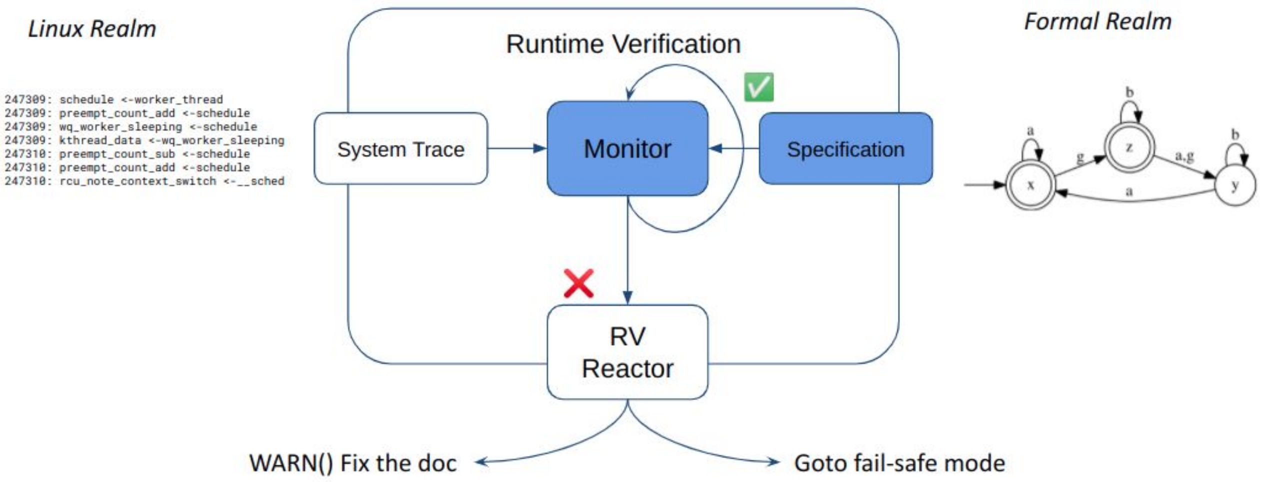
https://www.youtube.com/watch?v=6_gvarChkAg

Where are we?

Merged!

Interface and concepts

- RV subsystem is an interface and a set of tools to make RV accessible
- A RV **monitor** composed of:
    - A model/specification
    - The instrumentation
- A **reactor** is an action available to the monitor
    - Can be invoked by the monitor if an unexpected even happens
- **Monitors** can be enabled at runtime
    - Each monitor can have one enabled **reactor**



Interface demo

Demo video

https://youtu.be/V42BFNKbh0g

Monitor synthesis

- The RV enables conversion of an automata.dot -> kernel RV monitor
  - This is done with a dot2k utility
- **dot2k** creates a skeleton of a RV monitor module
- The monitor also uses a set of **C macros** to generate the **monitor code** for the specific type of monitor
- The only thing left for the humans to do is the **instrumentation**
    - The connection between a kernel event and the model event

Using code generation automatizes the process, making it less prone to error and easier to qualify.

Monitor synthesis demo

Demo video

https://youtu.be/3yDxz1SI4k8

RV Structure

RV Interface

- kernel/tracing/rv/
  - rv.c: rv interface
    - Startup
    - Register & control monitors
  - rv_reactor.c:
    - Startup
    - Register & control reactors
  - Reactors

```
[bristot@x1 linux]$ ls -l kernel/trace/rv/
total 56
-rw-r--r-- 1 bristot bristot  2233 Aug 24 08:38 Kconfig
-rw-r--r-- 1 bristot bristot   299 Aug 24 08:38 Makefile
drwxr-xr-x 4 bristot bristot  4096 Aug 24 08:38 monitors
-rw-r--r-- 1 bristot bristot   972 Aug 24 08:38 reactor_panic.c
-rw-r--r-- 1 bristot bristot   989 Aug 24 08:38 reactor_printk.c
-rw-r--r-- 1 bristot bristot 19406 Aug 24 08:38 rv.c
-rw-r--r-- 1 bristot bristot  1524 Aug 24 08:38 rv.h
-rw-r--r-- 1 bristot bristot 11129 Aug 24 08:38 rv_reactors.c
[bristot@x1 linux]$
```


Red Hat

RV Monitors

- kernel/tracing/rv/monitors
  - Each monitor has its own directory
  - They are independent from the interface
    - The interface only controls the monitoring session
  - For deterministic automata monitors, there are two files:
    - .h - the automata
    - .c - instrumentation and control
  - Other types of monitors might have different files
    - But it is important to keep the specification separated from the code

```
[bristot@x1 linux]$ ls -l kernel/trace/rv/monitors/
total 8
drwxr-xr-x 2 bristot bristot 4096 Aug 24 08:38 wip
drwxr-xr-x 2 bristot bristot 4096 Aug 24 08:38 wwnr
[bristot@x1 linux]$ ls -l kernel/trace/rv/monitors/wip/
total 8
-rw-r--r-- 1 bristot bristot 2066 Aug 24 08:38 wip.c
-rw-r--r-- 1 bristot bristot 1023 Aug 24 08:38 wip.h
[bristot@x1 linux]$
```


Red Hat

RV Headers

- include/linux
  - rv.h: monitor interface
- include/rv
  - Instrumentation.h: helper functions for instrumentation
  - automata.h: helpers for automata operation
  - da_monitor.h: helpers for deterministic automata monitor
  - Other types of monitor (timed automata) will add files here

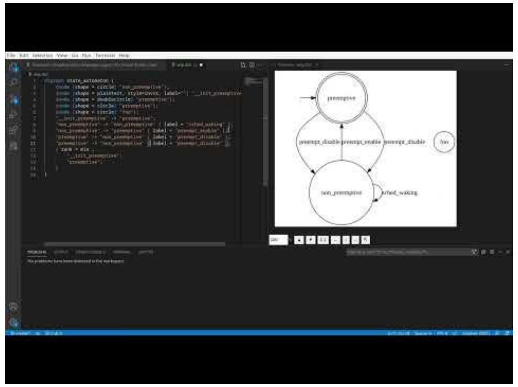
```
[bristot@x1 linux]$ ls -l include/rv
total 28
-rw-r--r-- 1 bristot bristot  2559 Aug 24 08:38 automata.h
-rw-r--r-- 1 bristot bristot 17416 Aug 24 08:38 da_monitor.h
-rw-r--r-- 1 bristot bristot   885 Aug 24 08:38 instrumentation.h
[bristot@x1 linux]$ ls -l include/linux/rv.h
-rw-r--r-- 1 bristot bristot 1595 Aug 24 08:38 include/linux/rv.h
[bristot@x1 linux]$ 
```


Red Hat

RV Tools

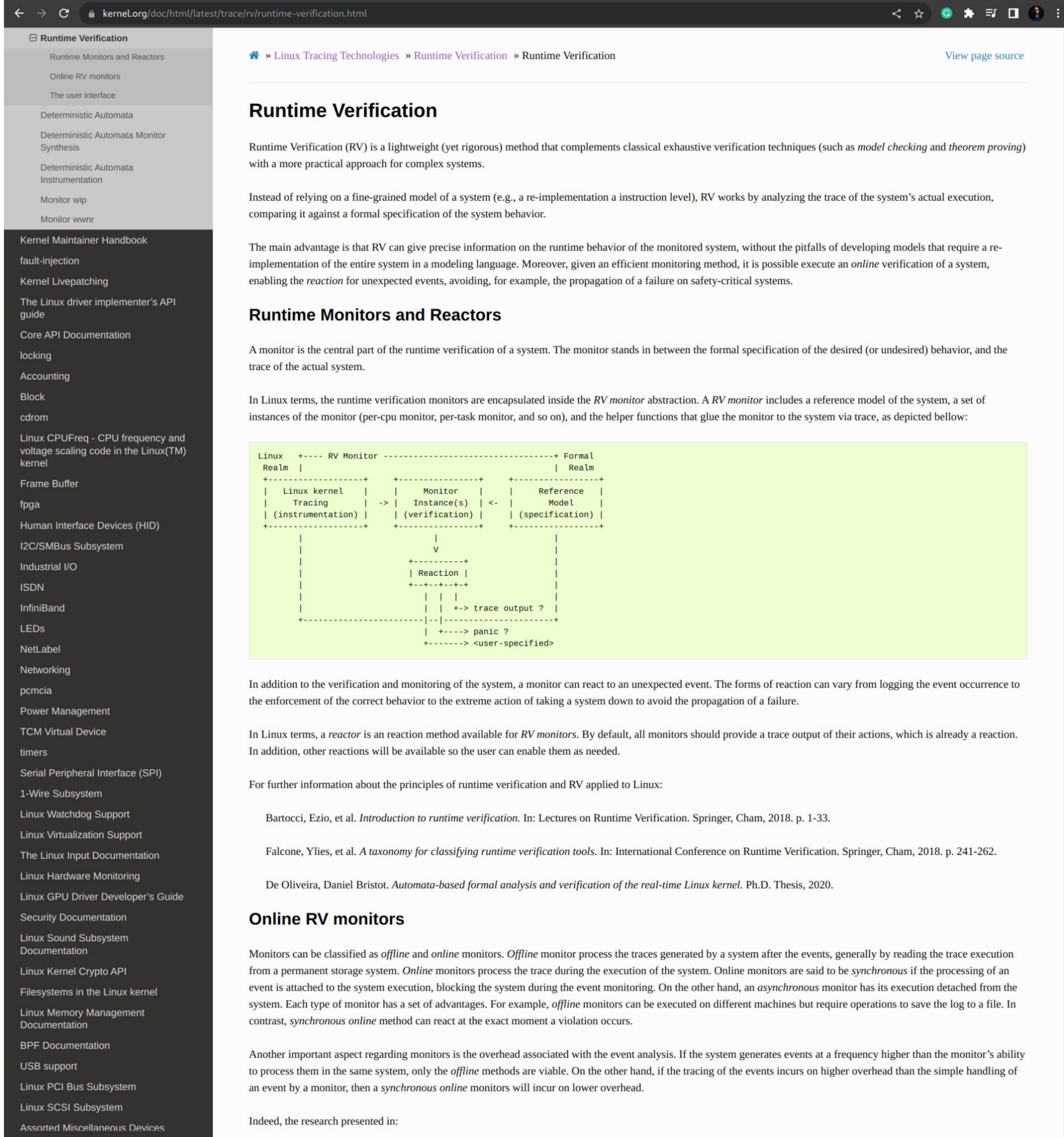
- tools/verification:
  - dot2/:
    - dot2 tools: tools to create the skeleton of the monitor
    - dot2k_templates/: templates for the monitor
  - models/
    - A place to store models/specification
    - Sample monitors that can be used as starting point

```
[bristot@x1 linux]$ ls -lR tools/verification/ | grep -v total
tools/verification/:
drwxr-xr-x 3 bristot bristot 4096 Aug 24 08:38 dot2
drwxr-xr-x 2 bristot bristot 4096 Aug 24 08:38 models

tools/verification/dot2:
-rw-r--r-- 1 bristot bristot 5879 Aug 24 08:38 automata.py
-rw-r--r-- 1 bristot bristot  942 Aug 24 08:38 dot2c
-rw-r--r-- 1 bristot bristot 7959 Aug 24 08:38 dot2c.py
-rw-r--r-- 1 bristot bristot 1746 Aug 24 08:38 dot2k
-rw-r--r-- 1 bristot bristot 5942 Aug 24 08:38 dot2k.py
drwxr-xr-x 2 bristot bristot 4096 Aug 24 08:38 dot2k_templates
-rw-r--r-- 1 bristot bristot  667 Aug 24 08:38 Makefile

tools/verification/dot2/dot2k_templates:
-rw-r--r-- 1 bristot bristot 1851 Aug 24 08:38 main_global.c
-rw-r--r-- 1 bristot bristot 1858 Aug 24 08:38 main_per_cpu.c
-rw-r--r-- 1 bristot bristot 1859 Aug 24 08:38 main_per_task.c

tools/verification/models:
-rw-r--r-- 1 bristot bristot 602 Aug 24 08:38 wip.dot
-rw-r--r-- 1 bristot bristot 573 Aug 24 08:38 wwnr.dot
[bristot@x1 linux]$
```



RV Documentation

- Documentation/trace/rv/:
  - runtime-verification.rst
  - da_monitor_instrumentation.rst
  - da_monitor_synthesis.rst
  - deterministic_automata.rst
  - For each monitor:
    - monitor_wip.rst
    - monitor_wwnr.rst



kernel.org/doc/html/latest/trace/rv/runtime-verification.html

» Linux Tracing Technologies » Runtime Verification » Runtime Verification   View page source

Runtime Verification

Runtime Verification (RV) is a lightweight (yet rigorous) method that complements classical exhaustive verification techniques (such as *model checking* and *theorem proving*) with a more practical approach for complex systems.

Instead of relying on a fine-grained model of a system (e.g., a re-implementation a instruction level), RV works by analyzing the trace of the system's actual execution, comparing it against a formal specification of the system behavior.

The main advantage is that RV can give precise information on the runtime behavior of the monitored system, without the pitfalls of developing models that require a re-implementation of the entire system in a modeling language. Moreover, given an efficient monitoring method, it is possible execute an *online* verification of a system, enabling the *reaction* for unexpected events, avoiding, for example, the propagation of a failure on safety-critical systems.

Runtime Monitors and Reactors

A monitor is the central part of the runtime verification of a system. The monitor stands in between the formal specification of the desired (or undesired) behavior, and the trace of the actual system.

In Linux terms, the runtime verification monitors are encapsulated inside the *RV monitor* abstraction. A *RV monitor* includes a reference model of the system, a set of instances of the monitor (per-cpu monitor, per-task monitor, and so on), and the helper functions that glue the monitor to the system via trace, as depicted bellow:

```
Linux   +---- RV Monitor ----------------------------------+ Formal
 Realm  |                                                  |  Realm
 +-------------------+     +----------------+     +-----------------+
 |   Linux kernel    |     |    Monitor     |     |    Reference    |
 |     Tracing       |  -> |  Instance(s)   | <-  |      Model      |
 | (instrumentation) |     | (verification) |     | (specification) |
 +-------------------+     +----------------+     +-----------------+
        |                          |                       |
        |                          V                       |
        |                     +----------+                 |
        |                     | Reaction |                 |
        |                     +--+--+--+-+                 |
        |                        |  |  |                   |
        |                        |  |  +-> trace output ?  |
        +------------------------|--|----------------------+
                                 |  +----> panic ?
                                 +-------> <user-specified>
```

In addition to the verification and monitoring of the system, a monitor can react to an unexpected event. The forms of reaction can vary from logging the event occurrence to the enforcement of the correct behavior to the extreme action of taking a system down to avoid the propagation of a failure.

In Linux terms, a *reactor* is an reaction method available for *RV monitors*. By default, all monitors should provide a trace output of their actions, which is already a reaction. In addition, other reactions will be available so the user can enable them as needed.

For further information about the principles of runtime verification and RV applied to Linux:

Bartocci, Ezio, et al. *Introduction to runtime verification.* In: Lectures on Runtime Verification. Springer, Cham, 2018. p. 1-33.

Falcone, Ylies, et al. *A taxonomy for classifying runtime verification tools.* In: International Conference on Runtime Verification. Springer, Cham, 2018. p. 241-262.

De Oliveira, Daniel Bristot. *Automata-based formal analysis and verification of the real-time Linux kernel.* Ph.D. Thesis, 2020.

Online RV monitors

Monitors can be classified as *offline* and *online* monitors. *Offline* monitor process the traces generated by a system after the events, generally by reading the trace execution from a permanent storage system. *Online* monitors process the trace during the execution of the system. Online monitors are said to be *synchronous* if the processing of an event is attached to the system execution, blocking the system during the event monitoring. On the other hand, an *asynchronous* monitor has its execution detached from the system. Each type of monitor has a set of advantages. For example, *offline* monitors can be executed on different machines but require operations to save the log to a file. In contrast, *synchronous online* method can react at the exact moment a violation occurs.

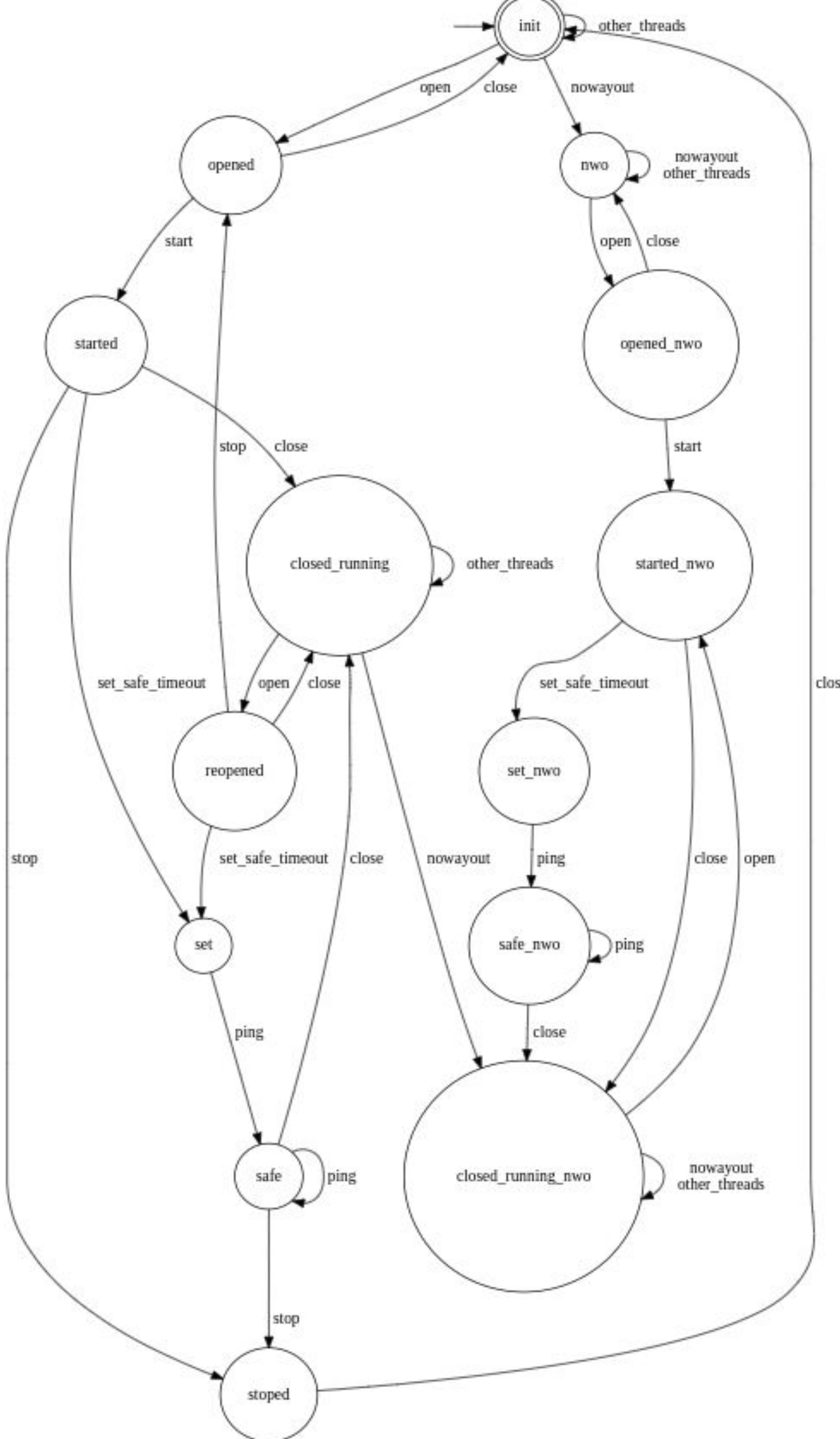
Another important aspect regarding monitors is the overhead associated with the event analysis. If the system generates events at a frequency higher than the monitor's ability to process them in the same system, only the *offline* methods are viable. On the other hand, if the tracing of the events incurs on higher overhead than the simple handling of an event by a monitor, then a *synchronous online* monitors will incur on lower overhead.

Indeed, the research presented in:

RV: what is next?

Watchdog monitor

- It was part of the first patchset (working with Elisa people)
- The idea is to monitor the watchdog usage until reaching a safe state
  - Open -> start -> set a timeout -> ping at least once
  - Avoid some features to reduce the amount of code to be inspected without changing the code
    - Like the hrtimers usage on watchdog dev
- The monitor automaton is generic
- In the patchset I added some options requested by the safety analysis made in the Elisa group
  - But because I did it all in a single patch, the explanation was not clear
- The patchset also included a user-space tool to exercise the monitor and to serve as starting point for the monitor



Monitor options

- We can enrich monitors with additional options/parameters
- For example:
  - int value to be the max safe watchdog timeout
- Each monitor has its folder also to store these specific options
- I am not sure if I will:
  - Add a file per option
    - Easy to use
    - More memory
  - A single option file
    - It will need a syntax like the event's format file
    - Less memory

Modular monitors

- Now models are built-in only
- But they can be loaded as module - I started them as module
- I removed the export symbols in the initial patch set to avoid problems
- I just need to export symbols in include/linux/rv.h
- It has a drawback:
    - Each monitor will have its own DECLARE_TRACE
    - That is why each monitor has a src dir: to store these monitor specific things

```
#ifdef CONFIG_RV_REACTORS
struct rv_reactor {
	const char		*name;
	const char		*description;
	void			(*react)(char *msg);
};
#endif

struct rv_monitor {
	const char		*name;
	const char		*description;
	bool			enabled;
	int			(*enable)(void);
	void			(*disable)(void);
	void			(*reset)(void);
#ifdef CONFIG_RV_REACTORS
	void			(*react)(char *msg);
#endif
};

bool rv_monitoring_on(void);
int rv_unregister_monitor(struct rv_monitor *monitor);
int rv_register_monitor(struct rv_monitor *monitor);
int rv_get_task_monitor_slot(void);
void rv_put_task_monitor_slot(int slot);

#ifdef CONFIG_RV_REACTORS
bool rv_reacting_on(void);
int rv_unregister_reactor(struct rv_reactor *reactor);
int rv_register_reactor(struct rv_reactor *reactor);
#endif /* CONFIG_RV_REACTORS */
```


Red Hat

Monitors with ebpf

- It is also possible to have monitors in eBPF
- I have a dot2bpf implementation
  - C & libbpf
  - Process the automata in the kernel
  - Feedback to user-space
- But it works and will share most of the kernel headers
  - To deduplicate the code of the eBPF and in-kernel option
- There is also a new set of monitors that I am doing with ETH Zurich that will use eBPF for user-space processing

Qualifying the monitors code

- We need to qualify the monitors' code according to the safety regulations
- This is part of the work we do with Elisa
- Red hat is committed to that as well
- The monitor was designed with this in mind
  - Pure C
  - No memory allocation
  - Self-generated
- But it will certainly require some changes that might add overhead
  - So we might have a da_safe_monitor.h

Other monitors

- Integrating the preemption model
  - It will require some extra work in the instrumentation, adjusting existing tracepoints
  - It will happen along with rtsl integration on rtla
    - So we will have the logical and the timing verification of the proven scheduling latency.
- Other monitors for RT
  - Function calls that are not guaranteed to be real-time
  - [Potential] priority inversion scenarios
- I am working with ETH Zurich and the University of Copenhagen on other types of formalism
  - They are more complex than automata, and allow timing in the equations

Questions?