Safety Architecture Working Group - Annual Update

Gabriele Paoloni & the WG participants

ELISA
Enabling Linux in Safety Applications

Aerospace · Automotive · Linux Features
Medical Devices · OS Engineering Process
Safety Architecture · Space Grade Linux · Systems · Tools

Agenda

- Working Group Intro
- Major milestone and achievements in 2024
- Current focus and activities
- What’s coming up in 2025 and areas and opportunities for collaboration
- Onboarding resources and how to get involved

The Safety Architecture Working Group

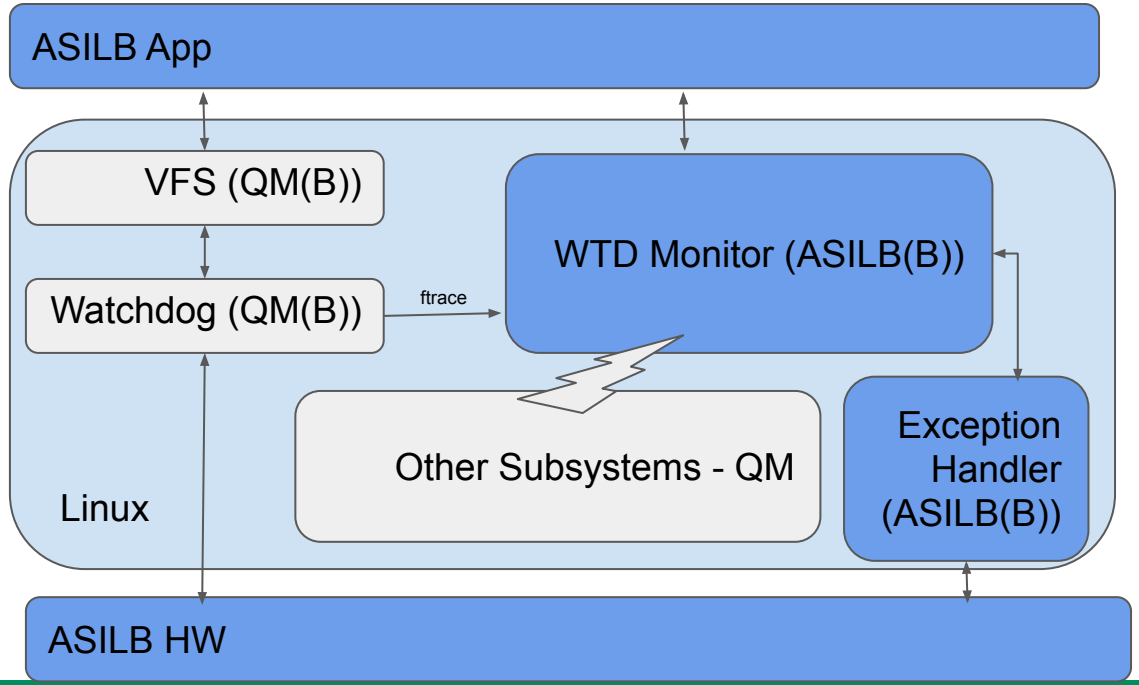
Mission:

According to technical safety requirements produced by domain specific WGs the focus of the Safety Architecture WG is to determine critical Linux subsystems and components in supporting safety functions, define associated safety requirements and scalable architectural assumptions, deliver corresponding safety analyses for their individual qualification and their integration into the safety critical system.

Past year activities

1) Kernel Safety Claims by Runtime Verification Monitors
2) Revisited STPA vs Expert Driven FMEA comparison
3) Evaluating and improving the Linux Kernel documentation
4) Linux Kernel Requirements

Kernel Safety Claims by Runtime Verification Monitors



We analysed the challenges in monitoring the Kernel through a monitor that also run within the Kernel. Mainly:

a) Understanding the Kernel code to be monitored (to design an effective monitor)

b) Claiming Freedom From Interference between the monitor and the rest of the Kernel

Revisited STPA vs Expert Driven FMEA comparison

STPA better suits a system level analysis; e.g. to define and break down safety requirements for a safety concept

Expert Driven FMEA better suits a SW level analysis; e.g. to define and break down safety requirements for SW Components.

In both cases in order to claim the completeness and correctness of the analyses it is crucial to have a comprehensive description of the element under analysis (The Linux Kernel in our case)

Evaluating and improving the Linux Kernel documentation

Main goals of the document:

- Analyze the current templates and guidelines that are available in the Linux Kernel documentation,
- Evaluate if and how they fulfill architecture and design aspects required by functional safety standards,
- Define improvements also in consideration of maintenance challenges deriving from a continuously evolving code baseline

This document triggered a session presented at Linux Plumbers 2024, following such a discussion the audience realized that we need to formalize and define testable requirements in Linux

Linux Kernel Requirements

Tag Name	Cardinality	Argument Mutability	Locations
SPDX-Req-ID	(1,1)	Immutable	Inline, Sidecar
SPDX-Req-End	(1,1)	N/A	Inline
SPDX-Req-Ref	(0,*)	Immutable	Inline
SPDX-Req-HKey	(1,1)	Mutable	Sidecar
SPDX-Req-Child	(0,*)	Mutable	Sidecar
SPDX-Req-Sys	(1,1)	Mutable	Sidecar
SPDX-Req-Text	(1,1)	Mutable	Sidecar
SPDX-Req-Note	(0,1)	Mutable	Sidecar

During the last ELISA Workshop at NASA a requirements template proposal has been presented and we decided, as next step, to prototype some examples and the automation associated with their generation and maintenance

Current Focus

- Prototyping Linux Kernel Requirements
- Prototyping the automation to check patchsets against requirements and vice-versa
- Finalizing the initial requirements framework, automation and examples
- Finalizing the Linux Kernel Requirements white paper and get it published by the Linux Foundation

Prototyping Linux Kernel Requirements

```
1498    /**
1499     * SPDX-Req-ID: [TODO automatically generate it]
1500     * SPDX-Req-Text:
1501     * trace_set_clr_event - enable or disable an event within a system
1502     * @system: system name (NULL for any system)
1503     * @event: event name (NULL for all events, within system)
1504     * @set: 1 to enable, 0 to disable (any other value is invalid)
1505     *
1506     * This is a way for other parts of the kernel to enable or disable
1507     * event recording.
1508     *
1509     * sequence of events:
1510     * 1) retrieve the global tracer
1511     * 2) locks the global event_mutex
1512     * 3) invokes __ftrace_set_clr_event_nolock
1513     * 4) unlocks the global event_mutex
1514     *
1515     * Returns 0 on success, -ENODEV if the global tracer cannot be retrieved,
1516     * -EINVAL if the parameters do not match any registered events, any other
1517     * error condition returned by __ftrace_set_clr_event_nolock
1518     */
1519    int trace_set_clr_event(const char *system, const char *event, int set)
1520    {
```

We started prototyping requirements for some functions of the tracing subsystem.
Requirements shall be:

- Testable
- Maintainable inline within the source code
- Compatible with pre-existing Kernel Documentation
- Hierarchically traceable

The main challenge is identifying the main design elements to be documented starting from the pre-existing code

While it is important to refer to design elements in the requirements (in order to write testable requirements), on the other hand it is not possible/feasible to mention them all (otherwise requirements would be as complex as the code itself)

Prototyping the automation to check patchsets against requirements

✓ **scripts/reqs/idgen.py: Add script for SPDX-Req-ID management"**

```
This script scans and processes all .c and .h files within a directory
tree.
It performs two main tasks:
* Preprocessing: Detects existing SPDX-Req-ID entries, updating a
  global map to track the highest progressive ID per file hash.
* ID Assignment: Updates or assigns new SPDX-Req-ID identifiers where
  missing, based on the file's hash and the next available progressive
  ID.

The script ensures efficient directory traversal by maintaining a single
file system scan and processes files in place, emitting warnings for
invalid or mismatched IDs.

Signed-off-by: Alessandro Carminati <acarmina@redhat.com>
```

alessandrocarminati committed 5 days ago

A script to automate the generation of Requirements' IDs (`SPDX-Req-ID`) is in-progress.

The goal is to generate a unique one that cannot change along the life of the requirements

"`SPDX-Req-HKey`" will instead be used to flag if, following code changes or requirement's text changes, the requirement shall be reviewed against the code (and vice versa).
"`SPDX-Req-HKey`" hashes are produced based on the following criteria:

- PROJECT: The name of the project (e.g. linux)
- FILE_PATH: The file the code resides in, relative to the root of the project repository.
- INSTANCE: The requirement template instance, minus tags with hash strings.
- CODE: The code that the SPDX-Req applies to.

"`SPDX-Req-ID`" is the very first "`SPDX-Req-HKey`" generated

Finalizing the initial requirements framework, automation and examples

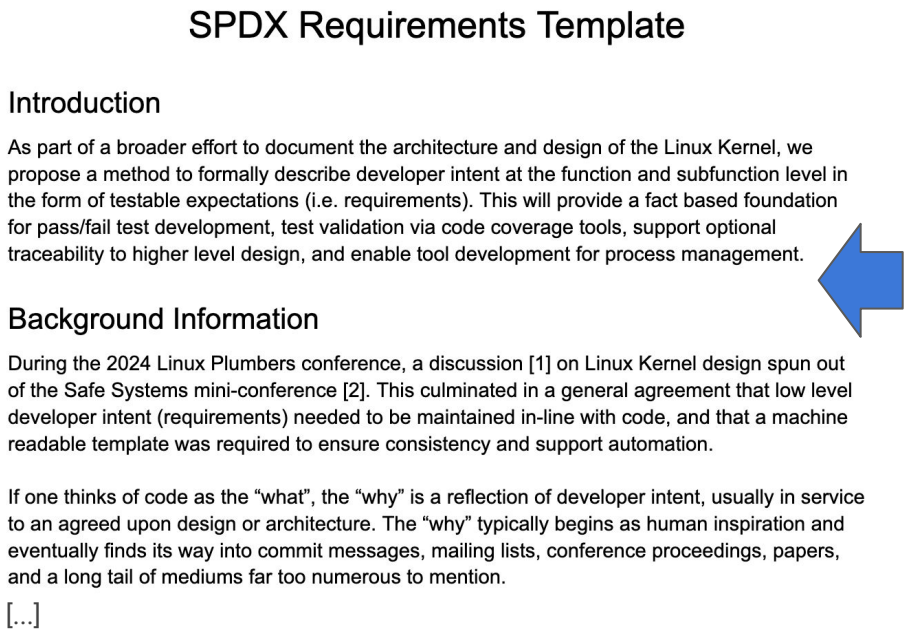
The working group is collaborating on a development branch:
https://github.com/elisa-tech/linux/tree/linux_requirements_wip

The very next steps are:

- Complete the requirements definition for `trace_array_set_clr_event()` and `trace_set_clr_event()`

- Complete the automation for maintaining all the SPDX requirements' fields

Finalizing the Linux Kernel Requirements white paper and get it published by the Linux Foundation



SPDX Requirements Template

Introduction

As part of a broader effort to document the architecture and design of the Linux Kernel, we propose a method to formally describe developer intent at the function and subfunction level in the form of testable expectations (i.e. requirements). This will provide a fact based foundation for pass/fail test development, test validation via code coverage tools, support optional traceability to higher level design, and enable tool development for process management.

Background Information

During the 2024 Linux Plumbers conference, a discussion [1] on Linux Kernel design spun out of the Safe Systems mini-conference [2]. This culminated in a general agreement that low level developer intent (requirements) needed to be maintained in-line with code, and that a machine readable template was required to ensure consistency and support automation.

If one thinks of code as the “what”, the “why” is a reflection of developer intent, usually in service to an agreed upon design or architecture. The “why” typically begins as human inspiration and eventually finds its way into commit messages, mailing lists, conference proceedings, papers, and a long tail of mediums far too numerous to mention.

[...]

The working group is collaborating on a work in progress draft here.

Following the finalization of the initial requirements’ framework and examples, the draft will be refined and a whitepaper should be published to engage with the community of Linux developers

What’s coming up in 2025

- Upstreaming initial requirements framework, automation and examples
- Re-focusing requirements in alignment with the key subsystems identified by the LFSCS working groups (or subsystems that are key to domain specific working groups)
- Using Linux Kernel requirements to support safety analyses in the Kernel

Onboarding resources and how to get involved

WG Webpage: https://lists.elisa.tech/g/safety-architecture

WG Regular Public Meeting
All ELISA public meetings can be accessed here https://zoom-lfx.platform.linuxfoundation.org/meetings/elisa
You can register for a specific WG meeting to receive the direct meeting calendar invitation.
You can also subscribe to calendar feed here https://lists.elisa.tech/g/safety-architecture/calendar

GitHub Repo
Please go to https://github.com/elisa-tech/Safety_Architecture_WG for additional details including current work led by this group and how to collaborate.

ELISA
Enabling Linux in
Safety Applications

Thank You!