ELISA
Enabling Linux in Safety Applications

WORKSHOP

ELISA Workshop
Lund, Sweden

Linux Kernel Low Level Requirements

Kate Stewart, Gabriele Paoloni, Chuck Wolber

Terminology

Bug: A violation of expectations.

Requirement: “Developer Statement of Testable Intent” or “Testable Expectation”

Testable: Provable (“Shall”).

Untestable: Unprovable or not reasonably proven (“Shall Not”).

Design: Decomposition of intent into (pass/fail) testable form.

Decomposition: Big ideas broken down into smaller ideas. (idea == testable expectation)

Low Level Requirement:

> “Software requirements from which Source Code can be **directly** implemented **without** further information.” - DO-178C

Validation vs. Verification

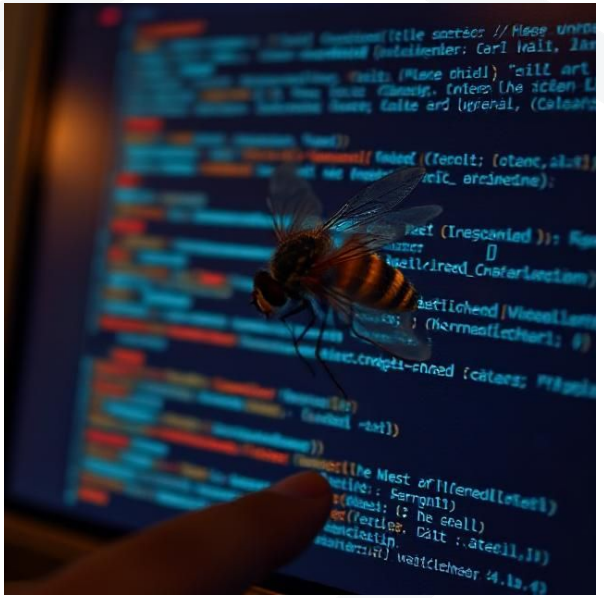
> “Validation is making sure you do the right thing.
> Verification is making sure you do the thing right.”
> - Author Unknown

- Verification Bug - “This violates my expectations”
  - The code is incorrect.
  - Traces to: pass/fail test.
- Validation Bug - “My expectation is wrong”
  - Code correctness is irrelevant.
  - Traces to: Maintainer `Signed-off-by` on requirement patch.

Bug Free Code Principles

1. Document testable expectations (low level requirements).
2. Trace testable expectation to developer intent (e.g. Maintainer Signed-off-by on requirement patch.)
3. Develop pass/fail test and validate testing with code coverage.

The Linux Kernel Requirements initiative is focused on points 1) and 2) while point 3) can be ground for future initiatives.



ELISA Enabling Linux in Safety Applications

WORKSHOP

Past Events

2025 Requirement Pilots in the Safety Architecture WG

- Initial requirement pilots drafted for the TRACING subsystems
- Initial feedbacks received from Steven Rostedt to define short term directions

2024 ELISA Workshop at NASA:

- Presented the Requirements Template Draft
- Agreed with Steven Rostedt to use the TRACING subsystem as experimental ground for requirements' definition

2024 Linux Plumbers Conference*:

- Maintain low level developer intent in-line with code.
- Machine readable template (consistency and automation)
- Build it and they will come.

* https://www.youtube.com/watch?v=stqGiy85s_Y

Template Requirements

- Should avoid substantive changes to current Linux Kernel development processes.
- Each requirement shall have a stable ID.
- Each requirement shall have a Hash Key.
- Stable IDs and Hash Keys:
  - Shall be globally unique (e.g. SHA-256)
  - Shall be generated without central coordination.
  - Shall be reproducible (i.e. generated in a standard way).
- Requirement template shall be embedded with the relevant code.
- The template format:
  - Shall be machine readable.
  - Should be compact and succinct to avoid unnecessary clutter.
  - Should be easy to read and biased against complex formatting.
  - Shall be standardized (e.g. through SPDX) to support use in other OSS projects.
  - Shall enable detection when relevant criteria changes.
  - Shall support being referenced from other systems.

Proposed Template Tags

Tag Name	Cardinality	Argument Mutability	Locations	Quick Explanation
SPDX-Req-ID	(1,1)	Immutable	Inline, Sidecar	Unique requirement identifier.
SPDX-Req-End	(1,1)	N/A	Inline	End of a requirement.
SPDX-Req-Ref	(0,*)	Immutable	Inline	Cross-reference additional code.
SPDX-Req-HKey	(1,1)	Mutable	Inline	Modification sentinel.
SPDX-Req-Child	(0,*)	Mutable	Sidecar	Pointer to decomposed requirement.
SPDX-Req-Sys	(1,1)	Mutable	Sidecar	Subsystem identifier.
SPDX-Req-Text	(1,1)	Mutable	Sidecar	Testable expectation (requirement text).
SPDX-Req-Note	(0,1)	Mutable	Sidecar	Optional clarifying notes.

Line Format

- Tag + String
  - `^SPDX-Req-[A-Za-z0-9-_]*:[ _]{1}.*\n$`
- All characters UTF-8.
- No text processing, escapes, or variable expansion.
- Line length 80 character common sense rule.
- Leading white space preserved, trailing white space trimmed.

TBD: Grouping, enumeration, and offloading.

SPDX-Req-ID

Unique Requirement Identifier

It is generated using the following string concatenation formula:

```
echo -nE "${PROJECT}${FILE_PATH}${INSTANCE}${CODE}" | sha256sum
```

Where:

- 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 - Immutable, generated when requirement is created.

SPDX-Req-HKEY

Generated using the same formula as SPDX-Req-ID, this field is used to highlight any change in:

- 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.

The very first time a requirement is created SPDX-Req-ID and SPDX-Req-HKEY are identical; as any field above change SPDX-Req-ID stays unchanged while SPDX-Req-HKEY evolves. It is expected to be used to for change management purposes

SPDX-Req-Child

- Intent: Trace to a decomposed requirement.
- Entity Relationship: Optional, zero or more.
- Tag String:
  - A valid SPDX-Req-ID
  - Acyclic (no loops)

SPDX-Req-Sys

- Intent: Provide context when requirement is exported or offloaded.
- Entity Relationship: Required, one and only one.
- Tag String:
  - For Linux Kernel, first relevant subsystem in `MAINTAINERS` file.
  - Other projects decide their own policy.

SPDX-Req-Text

- Intent: The requirement text.
- Entity Relationship: Required, one or more.
- Tag String:
  - Human readable.
  - Complies with requirement definition best practices (including semantic criteria defined later in the deck).
  - LLR: Detailed enough to support blind reimplementation without additional information, while avoiding a pseudocode-like repetition of the existing implementation.

SPDX-Req-Note

- Intent:
  - Enhance understanding.
  - Close down invalid avenues of interpretation.
- Entity Relationship: Optional, zero or more.
- Tag String: Human Readable

Semantic aspects of Kernel Requirements

What do we write under “`SPDX-Req-Text`”?

- From a semantic point of view it is important to document the intended or expected behavior (from a developer or integrator point of view respectively) in consideration of the different design aspects impacting it.
- Such behavior shall be described in a way that makes it possible to define test cases unambiguously.
- To this extent it is important to document design elements impacting the expected behavior and the design elements impacted by the expected behavior

The idea is to extend and refine the current design documentation guidelines (e.g. “Writing kernel-doc comments” for low level Kernel functions)

Possible elements impacting the expected behavior

- **Input parameters**: parameters passed to the API being documented;
- **State variables**: global and static data (variables or pointers);
- **Software dependencies**: external SW APIs invoked by the code under analysis;
- **Hardware dependencies**: HW design elements directly impacting the behavior of the code in scope;
- **Firmware dependencies**: FW design elements that have an impact on the behavior of the API being documented (e.g. DTB or ACPI tables, or runtime services like SCMI and ACPI AML);
- **Compile time configuration parameters**: configuration parameters parsed when compiling the Kernel Image;
- **Runtime configuration parameters** (AKA calibration parameters): parameters that can be modified at runtime.

Design elements impacted by the expected behavior

- **Return values**, including pointer addresses;
- **Input pointers**: pointers passed as input parameter to the API being documented;
- **State variables**: global and static data (variable or pointers);
- **Hardware design elements** (e.g. HW registers);

Some Considerations

Testability considerations: the impact of each of the documented “design elements impacting the expected behavior” shall be described in terms of effect on the “design element impacted by the expected behavior” and, in doing so, it is important to document allowed or not allowed ranges of values, corner cases and error conditions; so that it will be possible to define a meaningful test plan according to different equivalence classes.

Scalability considerations: the described expected behavior shall be limited to the scope of the code under analysis so for example the Software, Firmware and Hardware dependencies shall be described in terms of possible impact on the invoking code deferring further details to the respective documentation of these. The goal is to build a hierarchical documentation

Feasibility considerations: Only the “meaningful” and “useful” expected behavior, and the design elements impacting it, shall be considered (e.g. a printk logging some info may be omitted).
To this extent the Linux experts shall play a significant role in deciding the right level of detail, what to document and what to omit.

What have we done?

1) Created a development branch to write requirements for the Linux Kernel: https://github.com/elisa-tech/linux/tree/linux_requirements_wip
2) Defined requirements for a sample of functions from the TRACING subsystem:
   a) `trace_array_set_clr_event`
   b) `trace_set_clr_event`
   c) `__ftrace_set_clr_event_nolock`
   d) `__ftrace_event_enable_disable`
3) Created an initial automation to populate SPDX-Req-ID identifiers within the source code
4) Found a bug in __ftrace_event_enable_disable and pushed the fix upstream

Lessons Learned

1) When writing requirements from pre-existing code it is easy to fall into the mistake of writing pseudo code: requirements should be agnostic of the implementation
2) The Linux Kernel is complex: multiple requirements (as testable expectations) often map to a single function
3) In order to get the requirements framework accepted upstream it is better to show the value first. Hence we are now prioritizing the documentation of testable expectations over completing the requirements framework and automation

Next Steps

1) Use the SPDX-Req-Text section of the current requirements pilots to create documentation patches and push them upstream (this is needed to later get consensus on the proposed requirements framework)
2) Present the requirements work at OSS NA
3) Bring more maintainers and respective subsystems on board for expanding the requirements pilots
4) Recruit more people and keep working on requirements pilots and automation
5) Discuss at Linux Plumbers the final steps to get the requirements' framework upstream