ELISA
Enabling Linux in Safety Applications

WORKSHOP

NASA Goddard

Documenting Design: The Linux Kernel

Low Level Requirement Template Proposal

Chuck Wolber, Gabrielle Paolini, Kate Stewart

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.

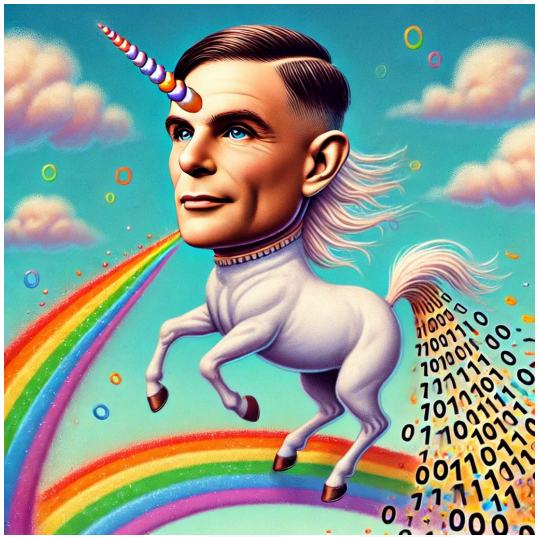
Decomposition: Big ideas broken down into smaller ideas. (idea == testable expectation)

Low Level Requirement:

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

Bug Free Code in Three Simple Steps!

1. Document all testable expectations.
2. Write a pass/fail test for every testable expectation.
3. Validate the test with code coverage.
    a. If code not covered, goto 1.
4. Profit!



“I need a satirical image that shows a happy unicorn with Alan Turing's head spewing 0's and 1's from its posterior end, while happily flying over a rainbow.”

ELISA Enabling Linux in Safety Applications

WORKSHOP

Kernel Design Expression

- (Testable) Design expression does not exist.
- Current testing shows no evidence of alignment to developer intent.
- We cannot reason about design while guessing at intent.

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 Tags

Tag Name	Entity	String
SPDX-Req-ID	(1,1)	Hash generation method.
SPDX-Req-HKey	(1,1)	Hash generation method.
SPDX-Req-Child	(0,*)	SPDX-Req-ID
SPDX-Req-Sys	(1,1)	Human readable string.
SPDX-Req-Lines	(0,1)	Comma delimited, relative ranges.
SPDX-Req-Text	(1,*)	Human readable string.
SPDX-Req-Note	(0,*)	Human readable string.

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

- Intent: Enable stable tracing.
- Entity Relationship: Mandatory, one and only one.
- Tag String:
  - … a hash generated with the Hash Generation Method (HGM).
  - … stable and does not change over the life of the requirement.

SPDX-Req-HKEY

- Intent:
  - Prevent semantic drift.
  - Document review activity.
- Entity Relationship: Mandatory, one and only one.
- Tag String:
  - … a hash generated with the Hash Generation Method (HGM).
  - … updated whenever a change is made to an HGM criteria.

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)

HGM (Hash Generation Method)

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 requirement template applies to.

The hash string is then generated as a basic string concatenation:

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

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-Lines~~

- Intent: Tie code to requirement when proximity is insufficient.
- Entity Relationship: Optional, zero or more.
- Tag String:
  - Comma delimited relative line ranges.
  - Line 1 is first line of compilable code below template instance.
  - Example: `// SPDX-Req-Lines: 8-13,15,22-54`

SPDX-Req-Text

- Intent: The requirement text.
- Entity Relationship: Required, one or more.
- Tag String:
  - Human readable.
  - Complies with requirement definition best practices.
  - 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

Multi-Line Tags

SPDX-Req-Text and SPDX-Req-Note are examples of multi-line tags.

- Simple concatenation makes the common case easy.
- Add a leading space on subsequent lines as needed.
- Use an underscore “_” (`0x5F`) to signal newline intent (e.g. paragraph, ASCII art, etc).

```
// SDPX-Req-Text: This is some arbitrary text that is intended to wrap to a
// SPDX-Req-Text:  second line to demonstrate the common case.
```

This is some arbitrary text that is intended to wrap to a second line to demonstrate the common case.

```
// SPDX-Req-Text:_| Col 1 | Col 2 | Col 3 |
// SPDX-Req-Text:_| ----- | ----- | ----- |
// SPDX-Req-Text:_| r1c1  | r1c2  | r1c3  |
// SPDX-Req-Text:_| r2c1  | r2c2  | r3c3  |
```

Terse LLR Example

```
/**
 * SPDX-Req-Id:
 * SPDX-Req-HKey:
 * SPDX-Req-Sys: INTEGER MATH
 * SPDX-Req-Text: The function shall return a 32 bit signed integer that is the
 * SPDX-Req-Text:  sum of the even 32 bit integers in the inclusive range of 1
 * SPDX-Req-Text:  to 10.
 */
int sum_evens() {
        int sum = 0;
        for (int i = 1; i <= 10; ++i)
                sum += i % 2 == 0 ? i : 0;
        return sum;
}
```

The function shall return a 32 bit signed integer that is the sum of the even 32 bit integers in the inclusive range of 1 to 10.

ELISA Enabling Linux in Safety Applications

WORKSHOP

Verbose LLR Example

```
/**
 * SPDX-Req-Id:
 * SPDX-Req-HKey:
 * SPDX-Req-Sys: INTEGER MATH
 * SPDX-Req-Text: The function shall initialize a signed 32-bit integer
 * SPDX-Req-Text:_ variable sum to 0.
 * SPDX-Req-Text: The function shall iterate over all signed 32-bit integers in
 * SPDX-Req-Text:_ the inclusive range of 1 to 10 using a loop variable i.
 * SPDX-Req-Text: During each iteration, if i is even (i.e., i % 2 == 0), the
 * SPDX-Req-Text:  function shall add i to sum, otherwise, the function shall
 * SPDX-Req-Text:_ add 0 to sum.
 * SPDX-Req-Text: After completing the iteration, the function shall return the
 * SPDX-Req-Text:_ value of sum as a signed 32-bit integer.
 */
int sum_evens() {
        int sum = 0;
        for (int i = 1; i <= 10; ++i)
                sum += i % 2 == 0 ? i : 0;
        return sum;
}
```

The function shall initialize a signed 32-bit integer variable sum to 0.
The function shall iterate over all signed 32-bit integers in the inclusive range of 1 to 10 using a loop variable i.
During each iteration, if i is even (i.e., i % 2 == 0), the function shall add i to sum, otherwise, the function shall add 0 to sum.
After completing the iteration, the function shall return the value of sum as a signed 32-bit integer.

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.

Next Steps

Technically we envision to:

- Demonstrate Feasibility - Use template on a target subsystem (TBD)
- Propose changes to the Linux Kernel Documentation to introduce the requirement template and to improve the guidelines for documenting different design elements
- Scale to other subsystems/drivers (long term goal)

How to achieve the technical goals?

- Should we first do it in ELISA and create an RFC to send upstream?
- Should we spawn a new Linux Kernel Requirements project right away to work on the RFC above as a first step?