

From requirements to HIL testing

Building a life-saving medical device in Rust

HISC 2026, Alex Senier (Ferrous Systems)

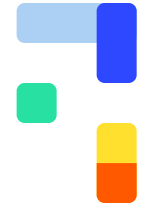
Who am I?

Alex Senier

- Product manager at Ferrous Systems
- More than 20 years in critical systems
- **Today:** What is still missing for Rust for highest safety levels?

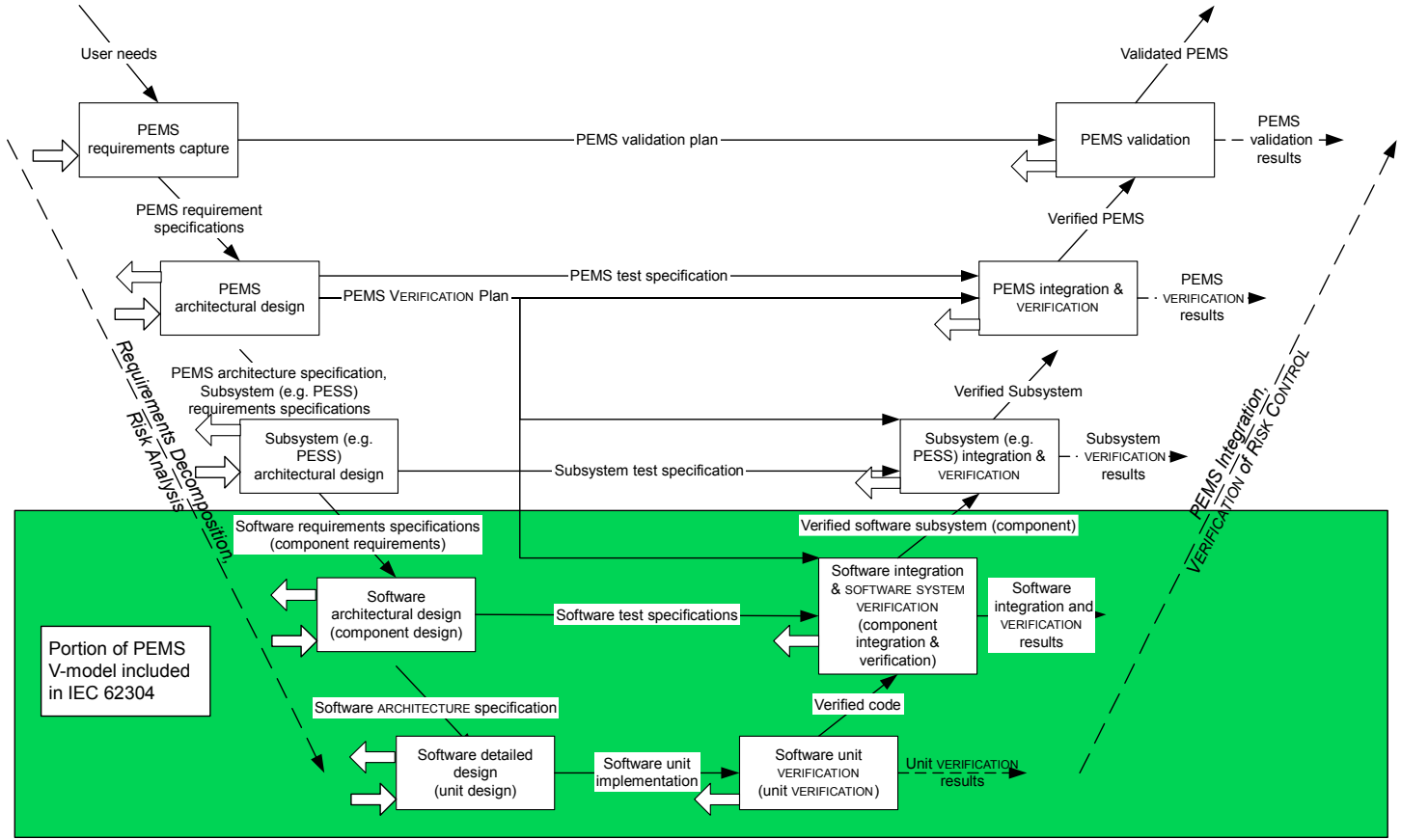


What did we build?

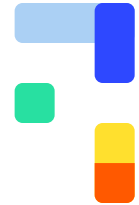


- **OxyPrem NOAH**
- Neonatal brain oxygen monitoring
- Sensor
  - Embedded firmware in Rust
  - ARM Cortex-M
  - `no_std` (and no allocator)
  - RTIC RTOS
- Tablet (not built by us)
  - Used by medical personal
  - Monitoring, alarms, etc.
- Requires IEC 62304 certification

IEC 62304: Software as part of the V-Model

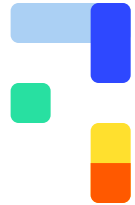






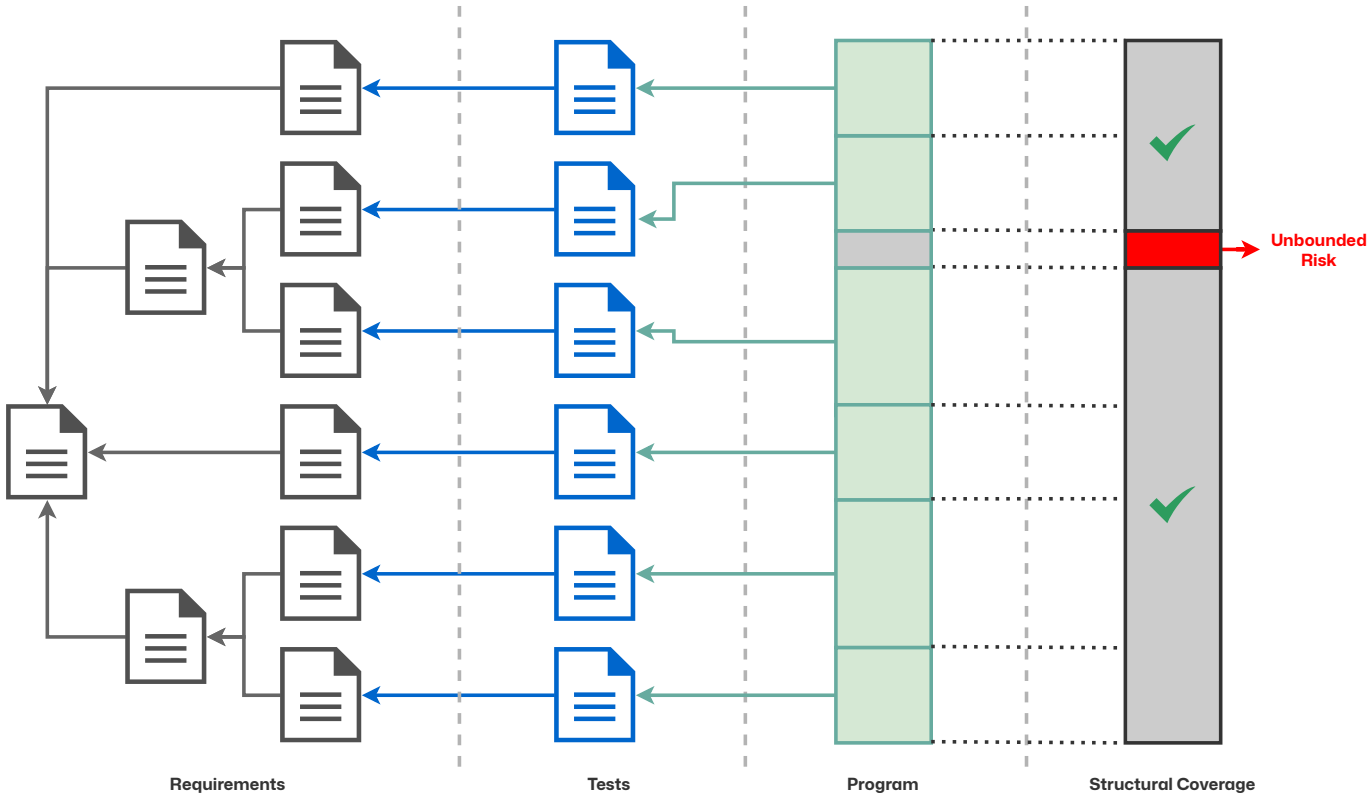
Deadline: Can we make yesterday work? 👉👈

Perfect Storm

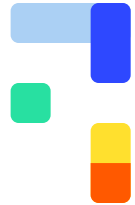


- Challenging embedded system (custom drivers, limited resources)
- Rigid safety process (traceability, structural coverage, documentation, change management)
- Crazy deadlines (experiments with real blood, fixed slots for certification labs)
- Little babies at risk (tons of safety requirements)
- And then some surprises!

How do you show safety?



Challenges

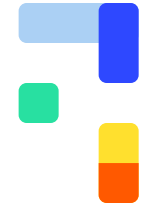


Make sure that requirements are

- consistent / current (Word doc)
- implemented (code vs. requirements)
- completely verified (tests vs. requirements)

Make sure that the firmware

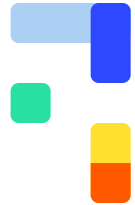
- is verified in realistic environment (real target)
- is covered by tests (requirements coverage)
- only has known functionality (structural coverage)



We know how to do this.

But how do you do that in 4 months?

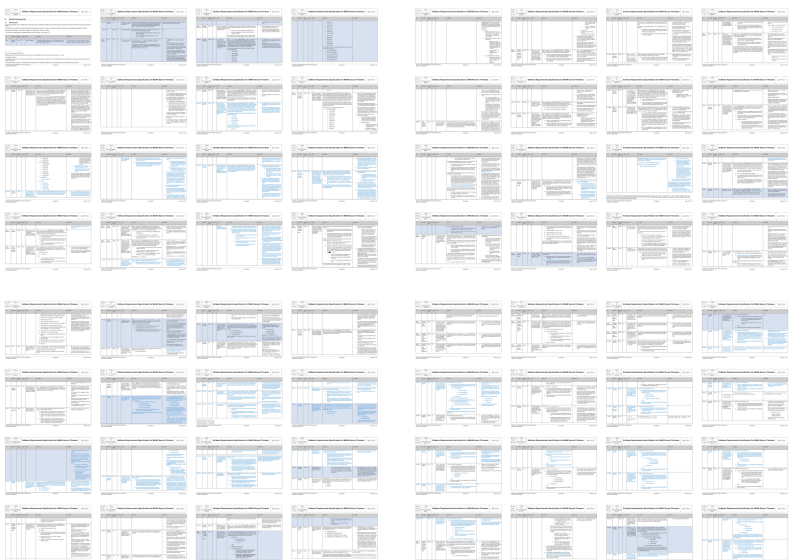
Mantra



- Ferrocene's lightweight requirements tracing for Rust projects
- **Requirement states**
  - **Failed** … At least one test mapping to the requirement has failed
  - **Verified** … Requirements that fulfill *verified conditions*
  - **Skipped** … At least one test mapping to the requirement was skipped
  - **Unverified** … Requirements that don't fulfill *verified conditions*
  - **Deprecated** … Requirements that have been marked as deprecated
  - **Excluded** … Requirements that have been marked as excluded

Challenge #1: Consistent / current requirements

Defining requirements



```
# `req-1`: Requirement Title

- **Parents:** ["req-2"]
- **Some-Key:** "Some JSON5-like value"

Requirement description...

# `req-2`: Other Requirement

Description for the other requirement...
```

(markdown is just one way to keep requirements)

Challenge #2: Requirements vs. Code

Tracing requirements to code

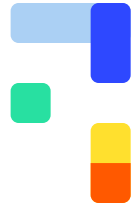
```rust
#[mantra_macros::req_satisfied("gs-req-1")]
fn foo() -> bool {
    // ...
    true
}
```

Challenge #3: Tests vs. Requirements

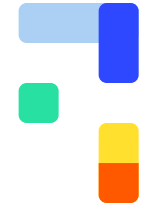
Tracing tests to code

```rust
#[test]
fn test_foo_1() {
    mantra_macros::assert_req!("gs-req-1" => foo(), "Verification using assert macro");
}

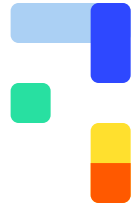
#[mantra_macros::req_test("gs-req-1")]
#[test]
fn test_foo_2() {
    core::assert!(foo(), "Verification using attribute macro");
}
```



Mantra Reports



Challenges



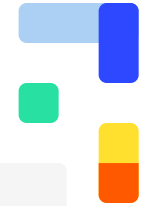
Make sure that requirements are

- consistent / current (Word doc)
- implemented (code vs. requirements)
- completely verified (tests vs. requirements)

Make sure that the firmware

- is verified in realistic environment (real target)
- is covered by tests (requirements coverage)
- only has known functionality (structural coverage)

Challenge #4: On-target testing



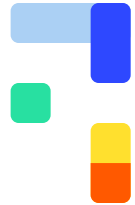
- **Ferrocene Embsinth:** Embedded system and integration test harness for Rust projects
  - abstract setup, connection, flashing etc.
  - tests feel like normal host integration tests
  - additional simulation hardware can be used (ours was a mock device)
- We had 2 pre-production devices of the NOAH sensor as Github self-hosted runners
- A mock devices would simulate the actual oxygen monitoring endpoint
- Every PR would run block on failed tests

```rust
#[embsinth::test]
fn system_test() {
    let app_probe = embsinth::probe::ProbeId::
      with_serial_nr(0x1366, 0x1051, "001050272949")
        .attach_under_reset("nRF52840_xxAA")
        .expect("Failed to attach to rad target")
    let sim_probe = embsinth::probe::ProbeId::new(0x1366, 0x1052)
        .attach_under_reset("nRF52840_xxAA")
        .expect("Failed to attach to rad target")

    let app_connection = app_probe
        .flash_once_and_connect("<APP binary filepath>")
        .expect("Failed to flash application binary");
    let sim_connection = sim_probe
        .flash_and_connect("<simulation sequence binary filepath>")
        .expect("Failed to flash simulation binary");

    let timeout = std::time::Duration::from_secs(10);
    assert!(app_connection
            .search_msg_for(timeout, |msg| {
                msg.message
                .starts_with("<app has started>")
            })
            .is_some(),
```

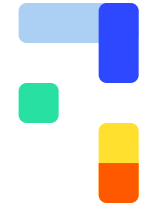
Challenges with HIL testing



- Slow
- Sometimes flaky
- We only had 2 devices
- Realistic, but some conditions cannot be tested on device (e.g. over temperature)

What would be a good compromise between realism, completeness and speed?

Ferrocene testing facades



- Rust test framework needs standard library
- But bare metal targets don't provide it
- **Testing facades**
  - Just enough `std` functionality for testing and structural coverage
  - Cross-compile bare metal code with qualified compiler and run emulated on the host
  - Feels like normal unit tests
  - Test can also run on as unit tests on the host

```rust
#[cfg(target_os = "linux")]
fn only_on_facade() { /* ... */ }

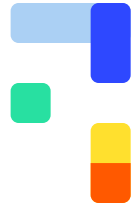
#[cfg_attr(target_os = "linux", derive(Debug))]
struct OnlyHasDebugOnFacade { /* ... */ }

fn only_prints_on_facade(value: u32) {
   let x = value;
   // ...

   #[cfg(target_os = "linux")]
   {
      println!("Received {x}");
   }

   // ...
   x
}
```

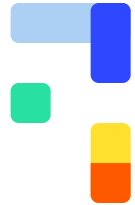
Challenge #5: Requirements coverage



- Strict separation: business logic / device drivers
- **Business logic**
  - Requirements traced using Ferrocene facade targets + Mantra
  - Compiled for the actual target using qualified compiler
  - Fast, reliable, available for every developer
- **Device drivers**
  - integration tests
- **Whole system / integration**
  - Requirements traced in final firmware in CI using (predecessor of) embsinth
  - Test driven through a simulated device

Majority of requirements verified automatically, a few through manual review (e.g. some error modes)

Challenge #6: Structural coverage

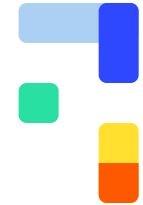


- **Business logic**:
  - 100% statement coverage
  - 100% branch coverage
  - Measured in CI on every commit (Ferrocene testing facades)
- **Drivers:**
  - Independent review / implementation by different embedded system experts

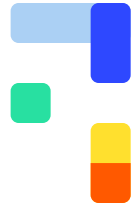
Current limits to our approach

Drivers

- On-target requirements tracing was limited (timing!)
- Structural coverage could not be done easily (error paths, tracing)
- **Future:** Facade targets with full device emulation and fault injection (hard!)

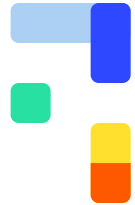


Other useful tools

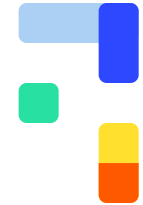


- Soak tests over the weekends
- Automatic deployment of all artifacts in the CI

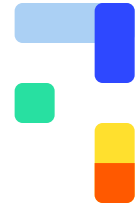
Other challenges



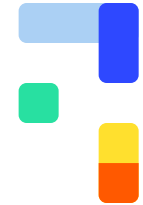
- Tons of crazy hardware issues
- Too few production devices
- CI took long towards the end of the project
- Change management made pull requests pile up (~20 PRs for a team of 5 devs at some point)
- `cargo test` has not been qualified, yet (welcome to shell script hell)
- We accidentally used a compiler flag that was not in our own safety manual 😬



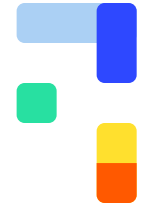
Lessons learned



#1: Don’t do medical projects!

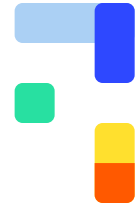


#2: Don’t use hardware!



#3: Don’t accept unrealistic timelines from your customers!

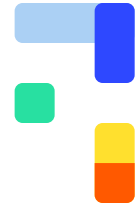
In case you cannot avoid any of those:



- Make requirements testing a first class citizen (not an afterthought)
- If possible have (your customer put) requirements in a structured, machine-readable format
- Measure coverage (requirements + structural) in your CI
- Implement a ratchet on those measures and fail the CI if they drop
- Integrate actual hardware into your CI workflow
- Make sure your developers have direct access to (the real) hardware
- Be prepared that you **will have** hardware issues

Thank you!

Questions?



Alex Senier, alex.senier@ferrous-systems.com