Automatisiertes Testen von eingebetteten Systemen mittels SiL und HiL

Ein Einstieg

Matthias Miehl

2019-11-13 18:00

Testautomatisierung Meetup

TOP-Tagungszentren AG
Emil-Figge-Str. 43, Dortmund

Vorstellung

- Matthias Miehl
- Elektrotechnikstudium
- Softwareentwickler seit 4 Jahren

Bis Oktober	bei SOREL GmbH	in Wetter.
Ab Januar	bei cbb Software GmbH	in Lübeck.

makomi.net

- Projekte (Python, TDD)
- Blog (Git, Z-Wave)
- Telegram-Kanal: embedded_sw_quality

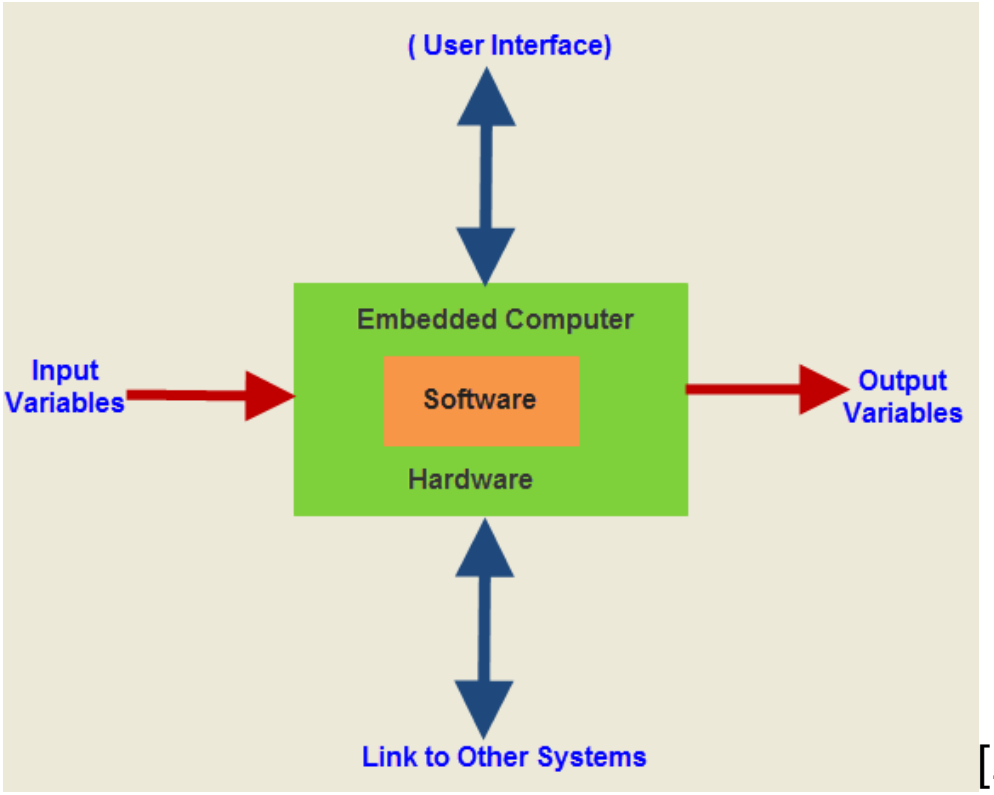
  “Relevant articles, news, and insights regarding all aspects of software quality with a focus on embedded systems.“

Vortragsziele

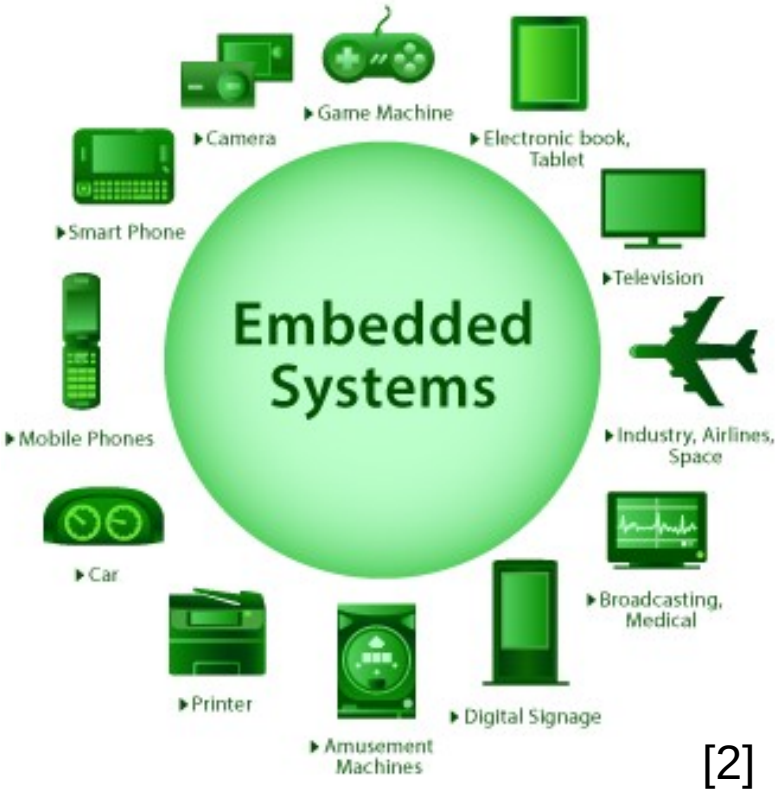
- Einblick in automatisiertes Testen von eingebetteten Systemen geben.
- Besonderheiten beim Testen von eingebetteten Systemen aufzeigen.
- Aufgekommende Herausforderungen/ Fragen teilen.
- Interessanten Austausch.

Was ist ein eingebettetes System?

Unsere Arbeitsdefinition
Ein physisches Gerät, das einen kleinen Computer beinhaltet, der auf die jeweilige Aufgabe zugeschnitten ist. Dieser Computer übernimmt typischer Weise Mess- und Steueraufgaben.

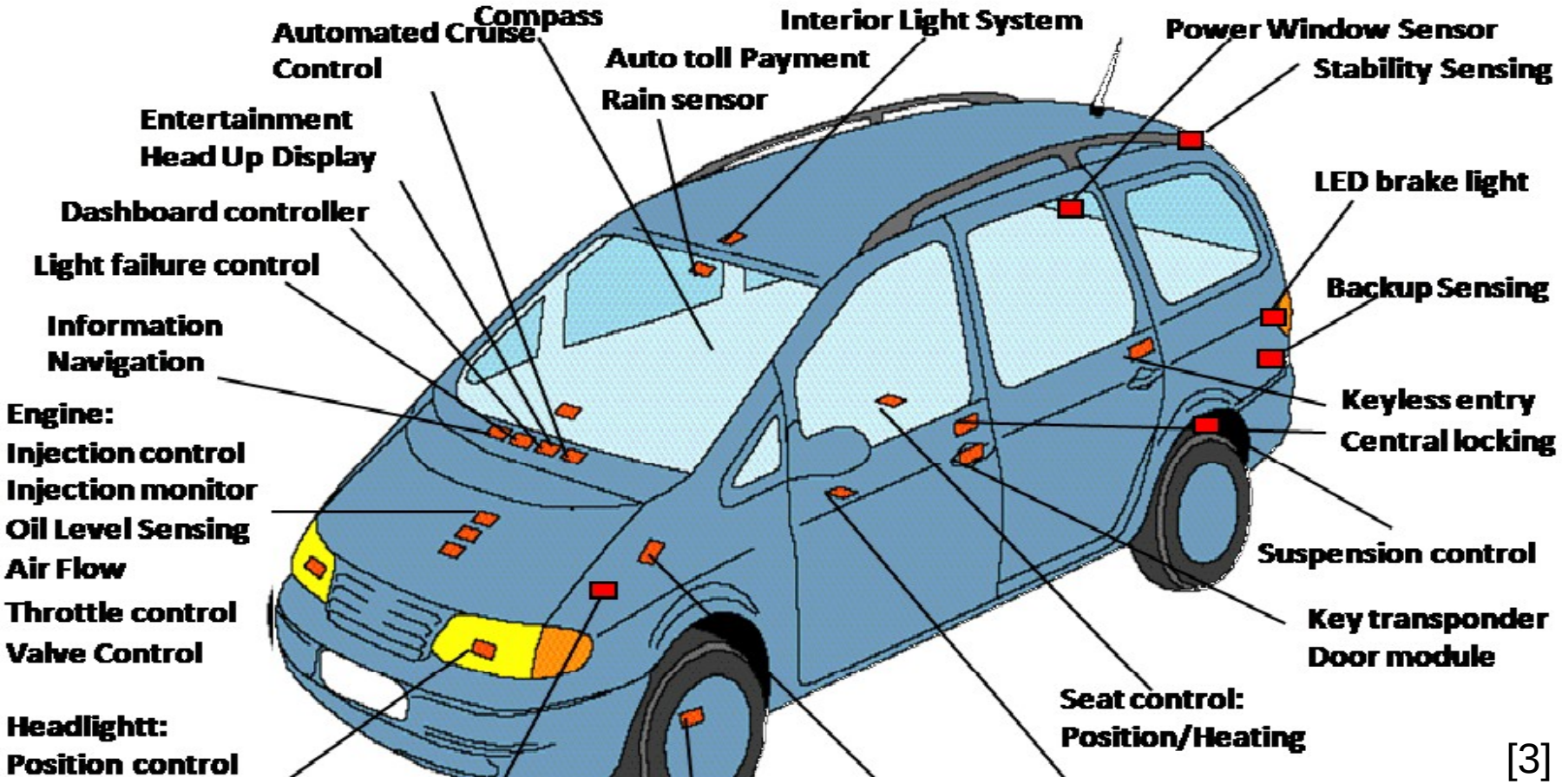


[1]



[2]

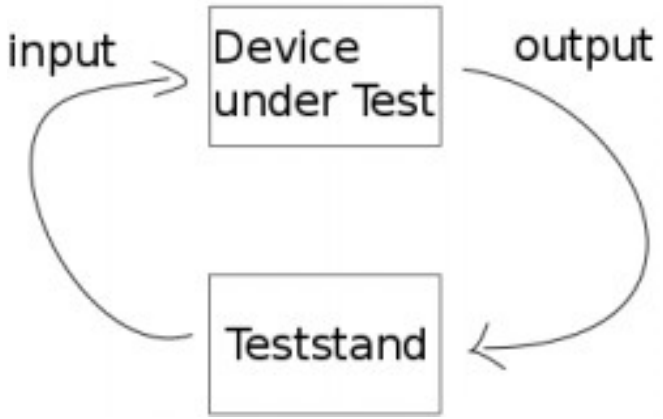
Was ist ein eingebettetes System? (cont‘d)



[3]

Was sind SiL- und HiL-Tests?

Software in the Loop, Hardware in the Loop



Es wird der SW bzw. HW vorgegaukelt, in einem echten System verbaut zu sein.

Hierzu adaptiert ein Teststand alle IOs.

Testart: funktionaler Blackbox-Test.
Fungieren bei uns als acceptance tests.

Beispiel: HiL-Teststand

Sendet elektkrische Signale an das DUT, um externe HW zu emulieren.
Interpretiert die Ausgänge des DUTs, um seine Reaktionen zu erfassen.

Parametriert das DUT (z.B. per UART). Typw. durch Anwender/ andere ES.

- Setup: Werkseinstellungen laden, Programm wählen, Params einstellen, ...
- Teardown: ...

Für die verschiedenen IO-Arten des DUTs werden Adapterplatinen benötigt.

Über SOREL

- 1991 gegründet
- Ursprünglich: Analoge Heizungsregler
- Aktuell: Digitale Heizungsregler und Raumthermostate

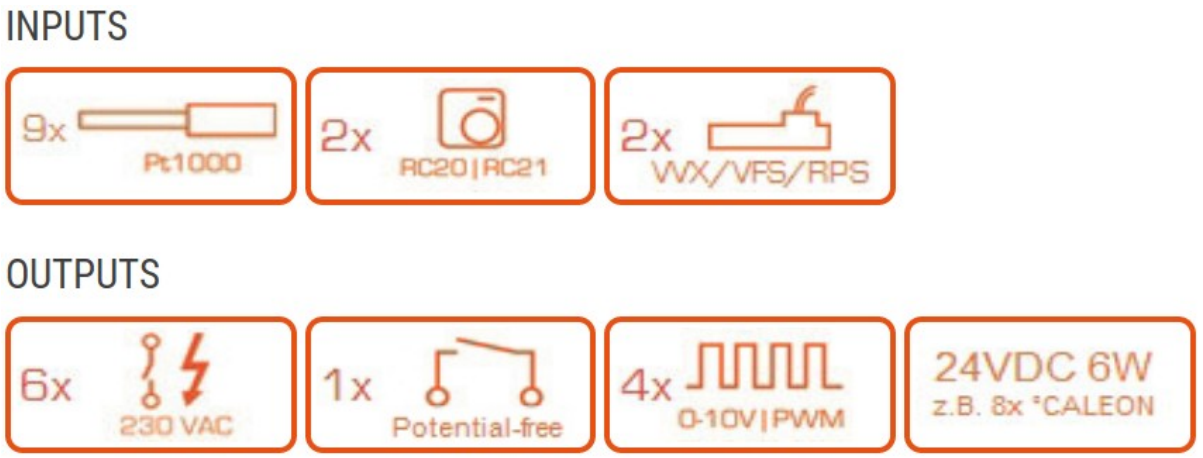
SOREL® Mikroelektronik GMBH

Ausgangssituation

Entwicklungsressourcen

- SW devs: 3 intern, 4 extern + 1 bis 2 Studenten
- Tester: 1 Hauptverantwortlicher + 2 bis 3 Studenten

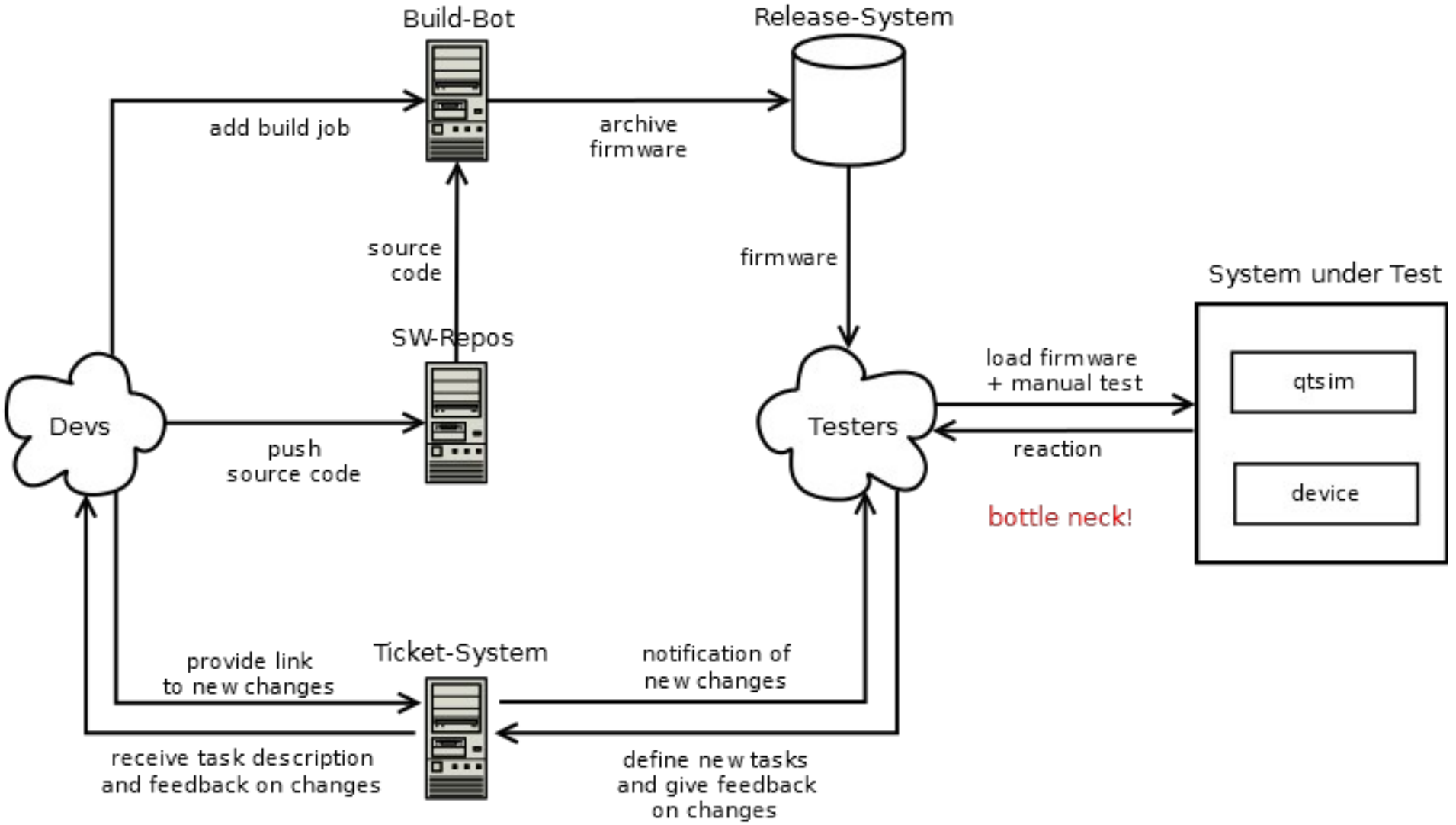
Zu testendes Produkt



Embeddedspezifische Herausforderung beim automatisierten Testen:
Verbindungen zur HW herstellen. Jedes eingebettete System ist anders
und stellt eigene Anforderungen an die Testinfrastruktur. Daher oft DIY-Lösungen.

Aktueller Entwicklungsprozess

Interaktionen.



Typische Schwierigkeiten

Was ist die Ursache und was muss sich ändern?

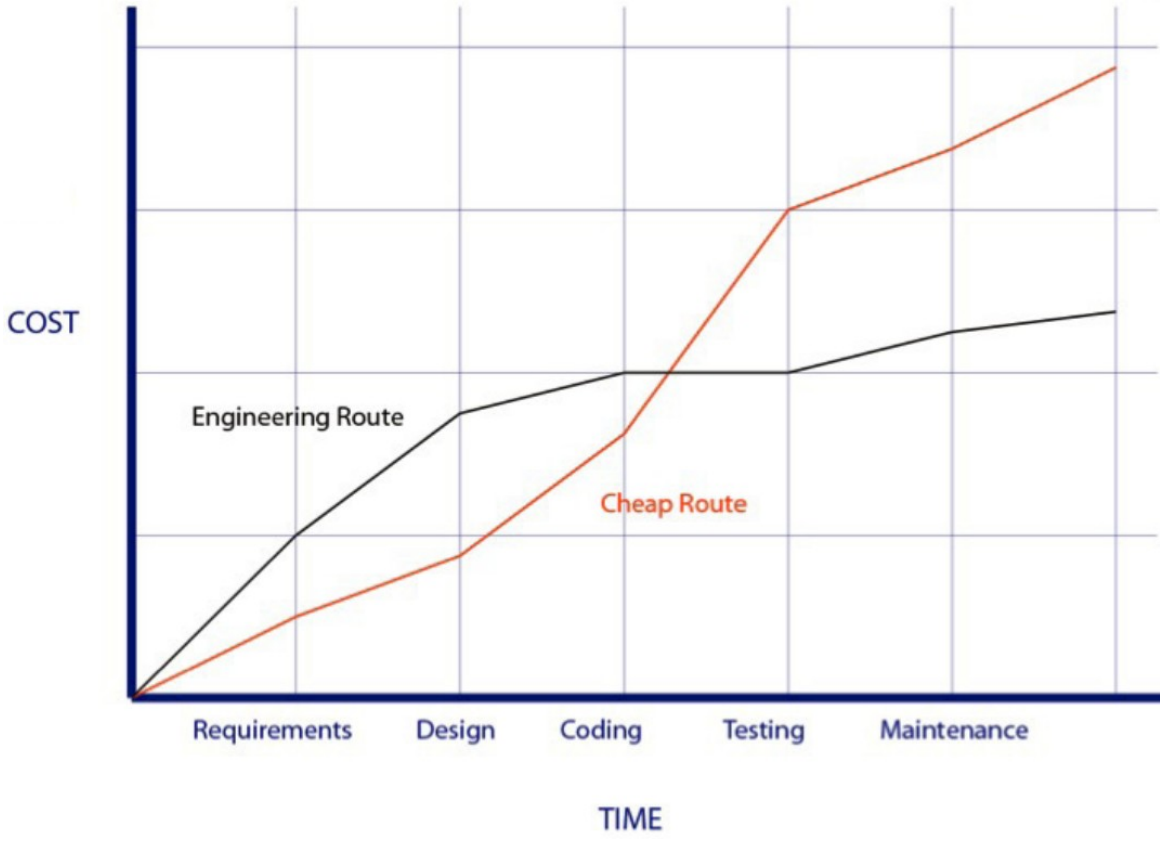
- Aufgaben werden nicht innerhalb der geplanten Zeit fertig.
  Je komplexer die Aufgabe, um so mehr wird sich verschätzt.
- Die Software startet über einen längeren Zeitraum nicht oder stürzt immer wieder ab und der Grund wird nicht gefunden.
- Es werden immer mehr Fehler und es dauert immer länger, bis selbst einfache Fehler gefunden und behoben werden.
- < 50% der Zeit bleibt für Refactoring, Planung und neue Features. Tendenz fallend.
- SW devs erhalten zu oft unvollständige und/ oder fehlerhafte Spezifikation. Führt dazu, dass sie sich, statt zu programmieren, eine möglichst plausible Kunden- und technische Anforderung überlegen – was natürlich besonders bei komplexen Aufgaben nicht funktioniert.
- Dinge, die mal funktionierten, funktionieren plötzlich nicht mehr.
- ...

Folge: Der int+ext Stresslevel für alle Beteiligten steigt kontinuierlich.

Erkenntnisse hinsichtlich SW-Entw.prozess

Nicht beschränkt auf eingebettete Systeme oder automatisiertes Testen.

„The Economics of Software Quality“ [4]



- Je komplexer die Aufgabe, umso wichtiger ist sauberes Engineering (spec, design, TDD, Clean Code, ...).
- Jede Testart findet unterschiedliche Fehler. Mehrere Testarten kombinieren, um möglichst viele Defekte zu finden.
- Man kann nicht alles gleichermaßen optimieren (Arbeitsgeschw., Codegröße, Ausführungsgeschwindigkeit, Sauberkeit). Man muss priorisieren und was zu letzt kommt, leidet am meisten (typw. Test).
- Wenn man in kleinen Inkrementen arbeiten möchte, d.h. mit schnellen feedback loops (Agile), ist Testautomatisierung ein Muss.
- Bei der Softwareentwicklung bestimmt der Entwicklungsprozess die Softwarequalität.
- Nur Bugs fixen, für die man einen fehlschlagenden Test hat.
- Best practices von Projektstart an einsetzen. -> Anhang B.

Lösungsansatz

a.k.a. der interessante Teil des Vortrags.

1. Neue Schäden verhindern (technical debt)
2. Bestehende Schäden reparieren (technical debt)
3. High ROI tasks automatisieren
4. Metriken
5. Acceptance und system tests nutzen (SiL, HiL)

-> Neuer Entwicklungsprozess

Lösungsansatz

a.k.a. der interessante Teil des Vortrags.

1. Neue Schäden verhindern (technical debt)

(a) Only make changes to code, if the code is in a good condition, i.e. it is easy to add a feature and obvious how to fix a bug. Otherwise, refactor the code prior to making any change.

(b) Fix defects as early as possible in the development process by using quality gates after major steps. This will allow us to expect a certain level of quality in the following step.

Lösungsansatz

a.k.a. der interessante Teil des Vortrags.

2. Bestehende Schäden reparieren (technical debt)

Problematic parts of the software that are large and central to the system, are refactored proactively to reduce the amount of new technical debt being created in their vicinity, by being the basis for new code.

Furthermore, since they are large, refactoring them takes some time and must be done before a feature needs to be added or a defect needs to be fixed in them. (cp. 1a)

Lösungsansatz

a.k.a. der interessante Teil des Vortrags.

3. High ROI tasks automatisieren

Create faster feedback loops by automating what currently makes the most sense to be automated:
- Has good ratio of being useful (task takes a lot of time + performed often)
- Easy/ quick to automate
- Makes sense in the bigger picture of where we want to be in the end

Example:

1. Automate integration
2. Pre-commit tests (Git best practices, coding guidelines, ...)
3. Unit tests

We don't need to fully implement each step, prior to taking the next. Once the ground work is done for one step, it is relatively easy to add more steps and improve existing ones. In the end, previously separate steps can be combined to one highly automated big step.

Lösungsansatz

a.k.a. der interessante Teil des Vortrags.

4. Metriken

We define metrics, collect data, and interpret it to understand what is happening/ whether we are improving, and what can be optimized.
Measure your current situation to determine whether you improve.

Interesting metrics:

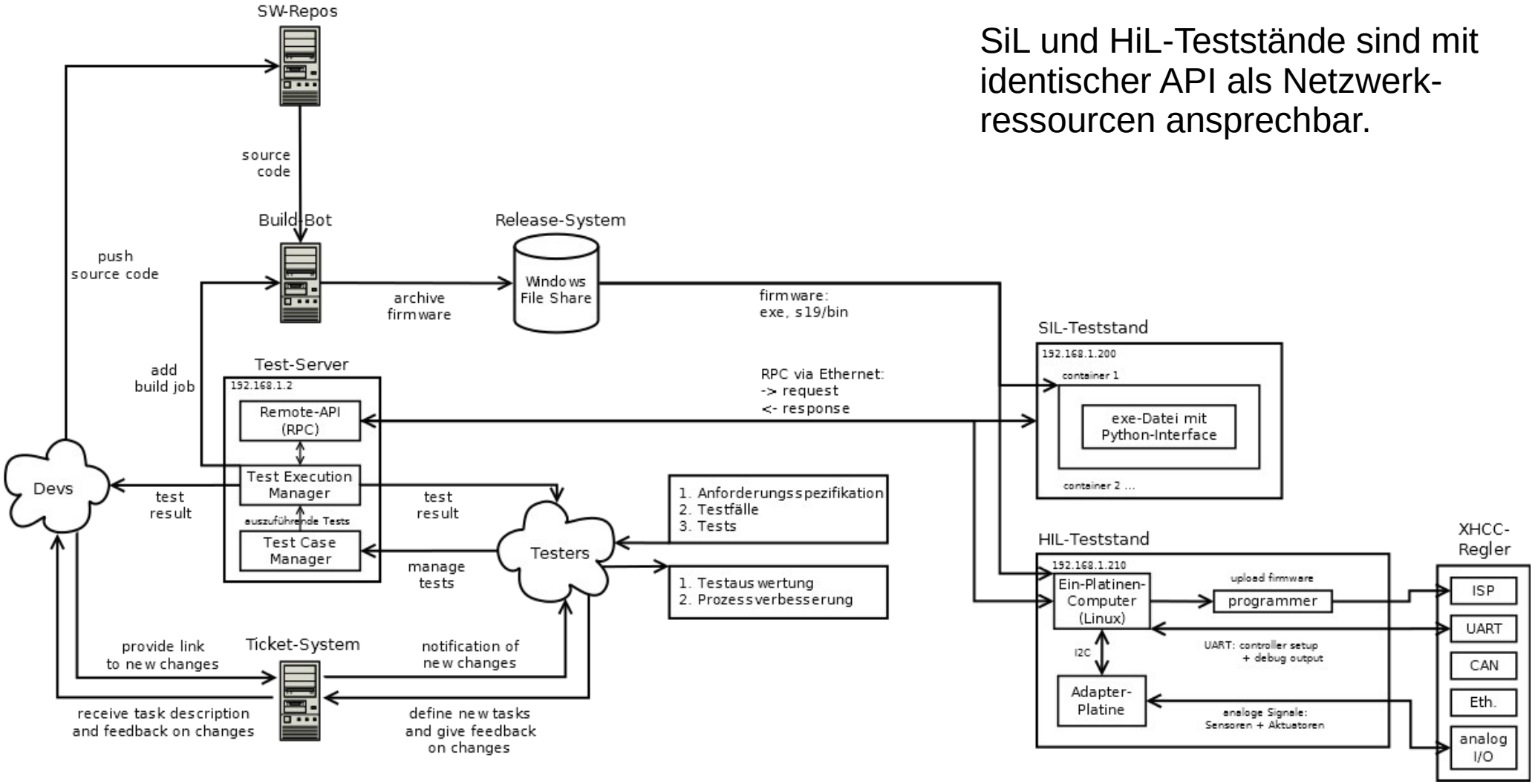
- Static analysis: compiler, linters, code complexity, ...
- Ratio between time spent implementing features vs. finding&fixing defects.
- Defect removal efficiency.
  Adopt the technologies that will get us above 95% in total defect removal efficiency before delivering the software.
- Cost of delivering a feature of certain complexity.
- Cost of fixing a defect at a certain stage.

Also: Keep an eye out for the path towards automated SiL and HiL testing.
One important prerequisite: written requirements/ functional specification.

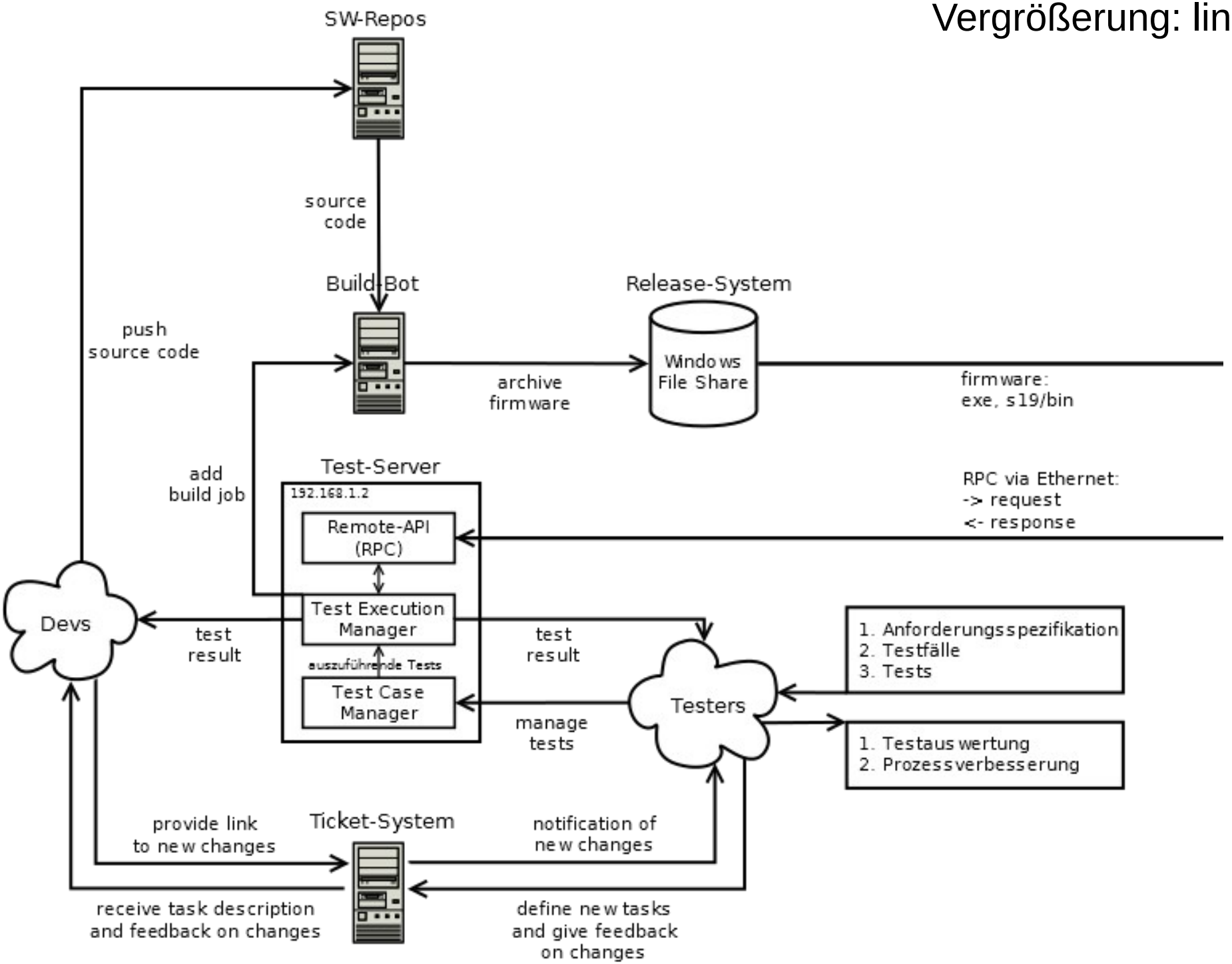
Neuer Entwicklungsprozess

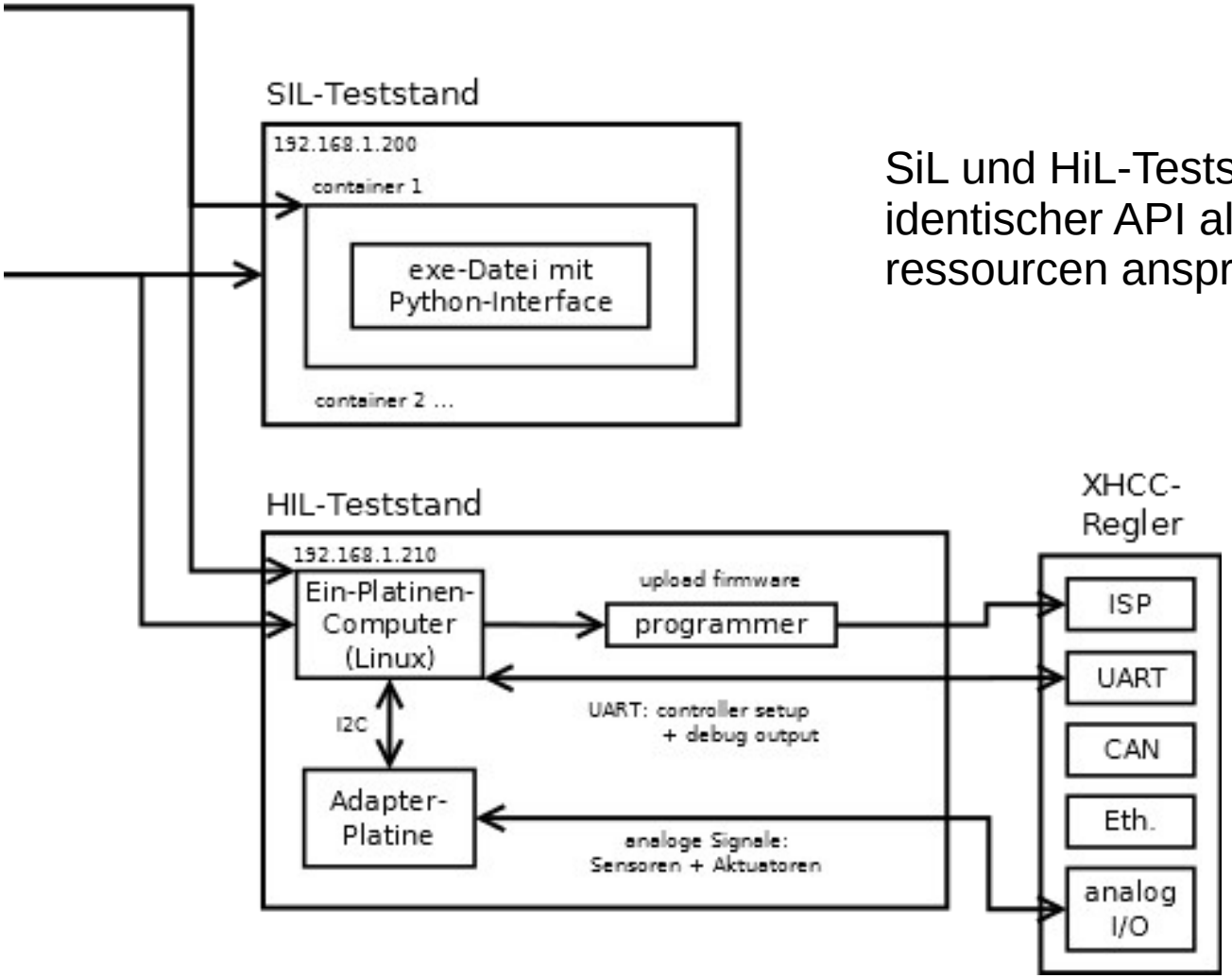
a.k.a. der interessante Teil des Vortrags.

(1) Interaktionen



SiL und HiL-Teststände sind mit identischer API als Netzwerk-ressourcen ansprechbar.


SW-Repos
source code
Build-Bot
Release-System
push source code
archive firmware
Windows File Share
firmware: exe, s19/bin
add build job
Test-Server
192.168.1.2
Remote-API (RPC)
RPC via Ethernet:
-> request
<- response
Devs
Test Execution Manager
test result
test result
auszuführende Tests
Test Case Manager
1. Anforderungsspezifikation
2. Testfälle
3. Tests
Testers
manage tests
1. Testauswertung
2. Prozessverbesserung
provide link to new changes
Ticket-System
notification of new changes
receive task description and feedback on changes
define new tasks and give feedback on changes

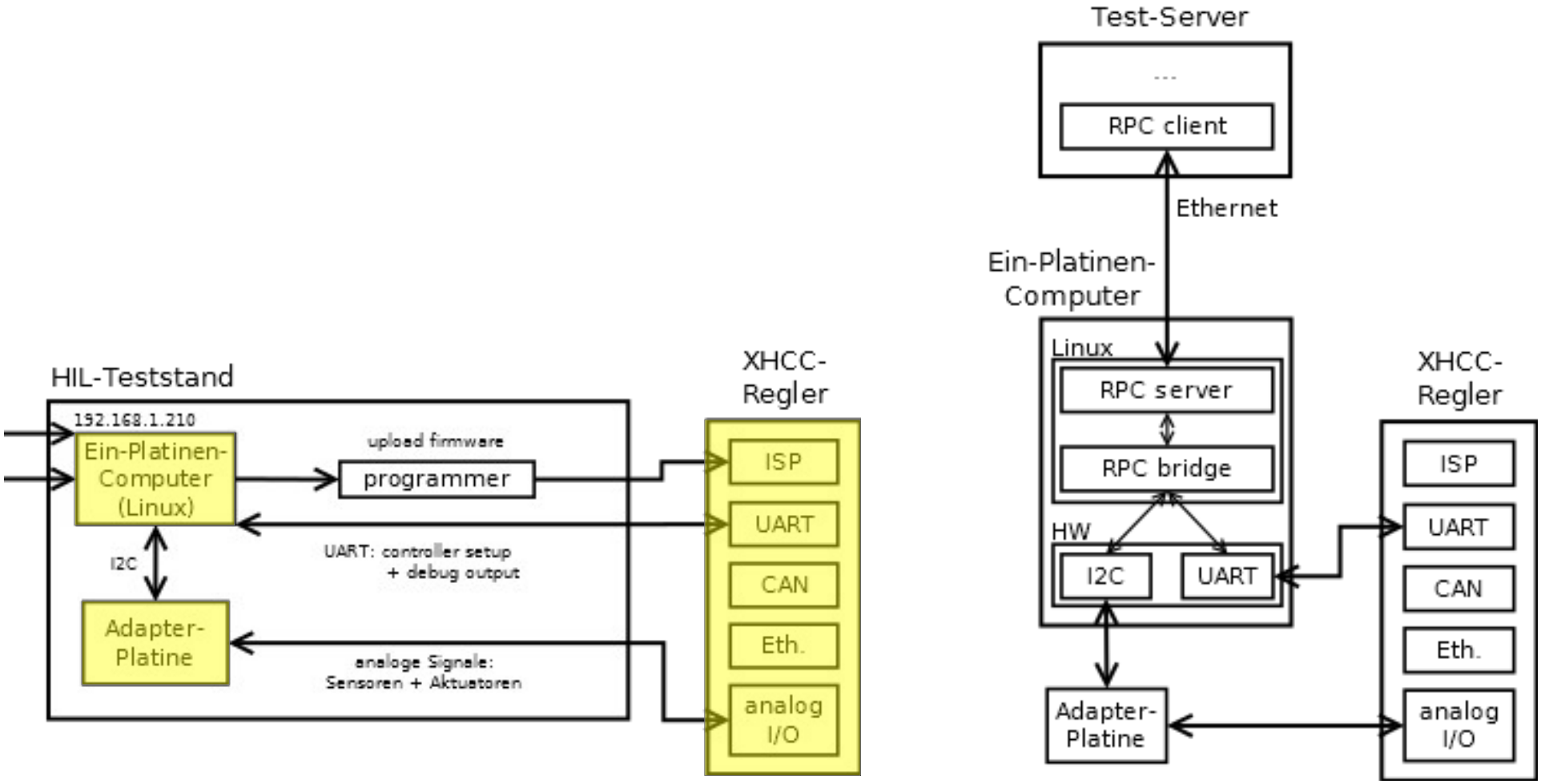


SiL und HiL-Teststände sind mit identischer API als Netzwerk-ressourcen ansprechbar.

Neuer Entwicklungsprozess

a.k.a. der interessante Teil des Vortrags.

(1) Interaktionen: HiL-Teststand



Neuer Entwicklungsprozess

a.k.a. der interessante Teil des Vortrags.

(2) Prozess

1. Idea

2. Specification

2.1 User requirements aka requirements spec (Lastenheft)
2.2 QG1: Validation with customer
2.3 Technical requirements aka functional spec (Pflichtenheft)
2.4 QG2: Make sure the tech reqs are complete, clear, consistent, and correct.

3. Implementation (by oneself) (-> details p23)

3.1 SW design
3.2 Code implementation
Trace code back to tech reqs by linking tests and commits to lines/ items of tech reqs.
3.3 QG3
(a) Pre-commit tests (engineering/ tech reqs for developing the SW) (-> details p24)
(b) Acceptance tests (SiL)
3.4 Request pre-commit peer code review

4. Implementation (group)

4.1 QG4: Manual pre-commit peer code review
4.2 Automated integration into dev branch (where all other devs base their work on)

Neuer Entwicklungsprozess

a.k.a. der interessante Teil des Vortrags.

(2) Prozess (cont‘d)

5. In depth tests

5.1 QG5

- (a) Production and debug build
  All following stages use this binary -> *Deploy what you test!*
  Generate trend data: warnings, (more in-depth/ better) static analysis, unit tests, ...
- (b) Acceptance tests (HiL)
  Generate trend data: memory usage, timing data, … [a]
  Feeds it to the CI server for visualization.
- (c) (in parallel/ continuously done) manual/ exploratory tests
- (d) (in parallel/ continuously done) stress tests (time intensive)
  Ex.: capacity/ throughput, long running, fuzzing, property testing

5.2 Queue Release Candidate (RC) for release
+ tag respective commit in VCS with RC ID to connect it to all test results

6. Deployment

6.1 Automatically upload RC to update server once manually authorized
6.2 Receive device telemetry for field tests
Ex.: warnings, errors, kernel panics, sensor/ actuator anomalies
6.3 QG6: Customer ;-)

Neuer Entwicklungsprozess

a.k.a. der interessante Teil des Vortrags.

(2) Prozessdetail: devs submitting a new increment of work

1. Rebase on development branch

2. Compile code (personal build)

3. Run pre-commit tests

4. Run SiL acceptance tests

5. Make changes and return to #2 until done

6. Once AOK: Request peer Code Review

Neuer Entwicklungsprozess

a.k.a. der interessante Teil des Vortrags.

(2) Prozessdetail: pre-commit tests

Keep them fast and reliable!

Example: Let devs to run parts of the pre-commit tests during implementation.

- Git Best Practices
- No translation file issues?
- No new static analyzer issues (in the lines/ files touched)?
  (linters, code complexity)
- Does it compile?
- No new warnings (in the lines/ files touched)?
  `-Wall -Wextra -pedantic …`
- All unit tests green?
- Smoke test 1: Does the application run?
- Smoke test 2: Does the application perform its most fundamental function?
- Dynamic code analysis

Neuer Entwicklungsprozess

a.k.a. der interessante Teil des Vortrags.

(3) Ausbaustufen/ Designüberlegungen für die Infrastruktur

a. Integration teilautomatisieren (bis sie letztlich vollautomatisiert ist)

b. TDD oder zumindest unit tests

c1. CI/CD-Infra-Design entwerfen
c2. CI/CD-Infra neu aufsetzen und dokumentieren
c3. Unit tests in neue Infra integrieren
c4. Weiteres in neue Infra integrieren: static analysis, code size monitoring, ...

d1. HiL-Teststände bauen
d2. Acceptance tests schreiben und in Infra integrieren

e. SiL-Teststände bauen und in Infra integrieren

f1. SiL + Prozesssimulation
f2. HiL + Prozesssimulation

g1. Gleichzeitiges Testen von mehreren per CAN verbundenen Reglern
g2. Virtuelle per CAN ansprechbare Regler

Aktuelle Herausforderung

Softwarequalität (Testen, CI/CD, embedded) ist ein großer Bereich.
- Als Neueinsteiger schwer zu erfassen.
- Es ist schwer zu erkennen, was die eigene Situation auszeichnet.
  - Was ist das ursächliche Problem? (root cause)
  - Welche Anforderungen haben wir und welche sind leicht umzusetzen?

  Daher schwer, von Null an eine zur Situation passende Schritt-für-Schritt-Anleitung zum Bau einer CI/CD-Infrastruktur zu entwerfen.

Schlechte Voraussetzungen
Viel Neues: Testen, CI/CD, embedded. Wenig Zeit und Leute.

Resultat
Langsamer Fortschritt und große Unklarheit darüber was am meisten hilft und in welcher Reihenfolge Testarten implementiert werden sollten.

Iterativer und explorativer Ansatz nötig.
- Nachteil: Kosten- und zeitintensiv.
- Vorteil: Internes Know How und eigene (optimal passende?) Lösung.

Konkrete Unklarheiten und Fragen

Einigermaßen klar; Hat jemand Erfahrungen aus der Praxis?

1) SiL/ HiL: Eigene Lösung bauen vs. einkaufen?

2) Design
   Wie detailiert machen?
   Welche Abstraktionen wählen?
   Wieviel Modularität ist genug?

Weniger klar

3) Welche Tests zuerst? (Unit, Integration, System)

4) Den Kern Open Source machen und öffentlich entwickeln?

5) Requirements engineering für ein so kleines Unternehmen
   5.1) Welcher Grad von Traceability ist sinnvoll?
   5.2) Welche Software zur Verwaltung der Traceabilitymatrix und Tests verwenden?

6) Wie die Prozesssimulation am besten angehen?
   Hat jemand praktische Erfahrung?

Konkrete Unklarheiten und Fragen (cont‘d)

7) Wie praktisch die "defect removal efficiency" bestimmen?
8) Wie gut ist unser aktueller Ansatz auf unsere Situation abgestimmt?
   Konzentrieren wir uns auf das Wichtigste?
   Wie detailiert sollte die Analyse und der Plan sein?
9) Wie am besten mit dem fehlenden Wissen und
   der fehlenden Erfahrung umgehen, um dennoch
   guten und schnellen Fortschritt zu erzielen?
10) Welche Abstraktionen/ Architekturdiagramme sind
    am besten geeignet, um einen bei der Automatisierung
    eines Prozesses zu unterstützen?
    (Infrastruktur, Interaktionen, decision flow graphs, ...)

Anhang A: Nützliche Tools und Themen

Toolchain in a docker container
- dockcross: https://github.com/dockcross/dockcross
  Explanation: CppCon 2018: Michael Caisse
  “Modern C++ in Embedded Systems - The Saga Continues”
  https://www.youtube.com/watch?v=LfRLQ7IChtg&t=14m39s

Automated code formatting
- clang-format: clangformat.com

Compile time reduction
- Combining SCons and Ninja builds: https://el-tramo.be/blog/scons2ninja/
- Activate LTO (Link Time Optimization): `-Flto` (youtube.com/watch?v=dOfucXtyEsU&t=56m)

Git Best Practices
- git-cop: https://github.com/bkuhlmann/git-cop
- pre-commit: https://pre-commit.com/
- git-style-guide: https://github.com/agis/git-style-guide

Complexity metrics
- lizard: https://github.com/terryyin/lizard
- ravioli: https://github.com/ElectronVector/ravioli

Anhang A: Nützliche Tools und Themen (cont‘d)

Static code analysis
- Frama-C: http://frama-c.com/download.html
- clang static analyzer: https://clang-analyzer.llvm.org/
- clang-tidy: https://clang.llvm.org/extra/clang-tidy/
- infer: https://github.com/facebook/infer
- cppcheck: https://github.com/danmar/cppcheck
- PVS-Studio: https://www.viva64.com/en/pvs-studio/
- gcc-poison: https://github.com/leafsr/gcc-poison (w/o `strlen` and `strcmp`)

Dynamic code analysis
- Google AddressSanatizer: https://github.com/google/sanitizers/wiki/AddressSanitizer
- LLVM Clang: https://clang.llvm.org/
- gcc: `-fsanitize=address -fsanitize=undefined -std=c11 -fdiagnostics-color`
  Instrumentation options: https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html

Symbolic execution
- History of symbolic execution: https://github.com/enzet/symbolic-execution

Property testing
- Fuzzing vs. Property Testing: https://news.ycombinator.com/item?id=20279500
  HackerNews discussion and explanation
- rapidcheck: https://github.com/emil-e/rapidcheck

Anhang B: Best practices für neue Projekte

Lightweight processes for improving code quality and identifying problems early:

1) Fix all of your warnings
2) Set up a static analysis tool for your project (linter, code complexity, ...)
3) Measure and tackle complexity in your software
4) Create automated code formatting rules
5) Have your code reviewed
6) „s/“ directory for common actions (https://chadaustin.me/2017/01/s/)
7) Setup Doxygen
8) Setup a code formatter
9) Be reasonable
   - Use a VCS like Git
   - Use a sensible directory structure (bin, build, doc, tools, src, …)
   - Automate the use of static analysis tools
   - ...

Basiert auf diesem Artikel von Embedded Artistry:
https://embeddedartistry.com/newsletter-archive/2018/3/5/march-2018-lightweight-processes-to-improve-quality

Tipp: „Embedded Artistry“-Newsletter abonnieren.

Bildreferenzen

[1] https://www.aldec.com/images/content/TySOM_Embedded-Systems.png
[2] https://www.elprocus.com/wp-content/uploads/2016/10/Embedded-System.png
[3] http://www.theengineeringprojects.com/wp-content/uploads/2016/11/automotive.png
[4] http://www.informit.com/store/economics-of-software-quality-9780132582209

Textreferenzen

[a] Jacob Beningo, „5 Embedded System Characteristics Every Engineer Should Monitor“, https://www.beningo.com/5-embedded-system-characteristics-every-engineer-should-monitor/, 2019-02-14, Zugriff: 2019-11-12

Fragen?

Vielen Dank!

E-Mail: say-hi@makomi.net

Folien: makomi.net/posts/automated-testing-of-embedded-systems/

Telegram-Kanal: embedded_sw_quality

“Relevant articles, news, and insights regarding all aspects of software quality with a focus on embedded systems.“