Traum und Wirklichkeit beim automatisch erzeugten Unit-Test und bei Concurrency-Tests

Beobachtungen und Visionen

Embedded Testing 2017

Stephan Grünfelder

Stephan Grünfelder

1994 – 1998 TU Wien, Siemens

1998 – 2000 Austrian Aerospace

2001 – 2002 Medcare Flaga, IS

2003 – 2006 Robert Bosch GmbH

2007 – 2011 RUAG Space

2011 – Riedel Communications

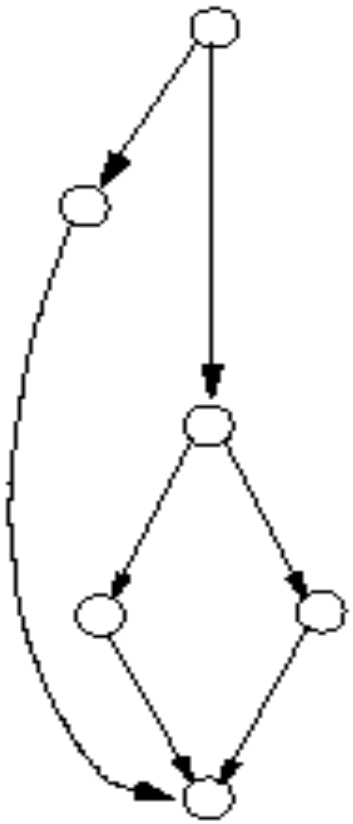
2007 – selbständiger Trainer

Inhalt

- **Strukturelle Coverage von Tests**
- **Automatische Testfallgenerierung bei Unit-Tests**
  - **Funktion von Werkzeugen der akademischen Forschung**
  - **Werkzeuge im industriellen Einsatz**
- **Traum/Wirklichkeit #1: Praxistauglichkeit von automatisch erzeugten Unit Tests**
- **Coverage-Analyse ohne Instrumentierung**
  - **Funktionsprinzip**
  - **Randbedingungen**
- **Data-Race-Analyse, Deadlock-Analyse**
- **Traum/Wirklichkeit #2: Dynamische Data-Race-Analyse und Deadlock-Analyse ohne Instrumentierung**

Strukturelle Testüberdeckung - Structural Coverage

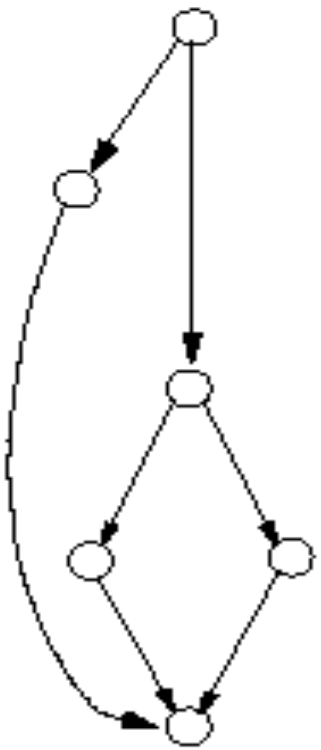
- **Statement Coverage**
- **Decision Coverage**
- **MC/DC**
- **McCabe‘s Basis Path Coverage (structural unit test coverage)**



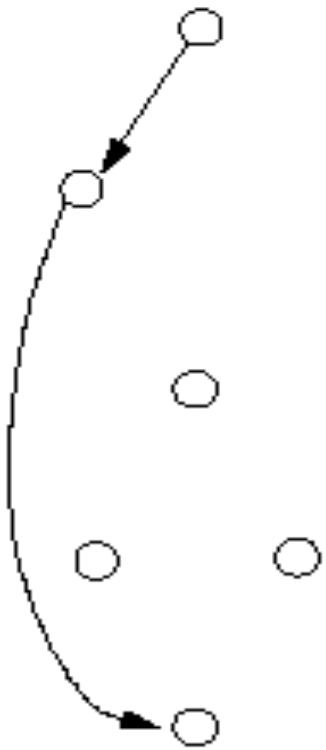
```
If (a && b && c) { ... }
```

Strukturelle Testüberdeckung - McCabe

Complexity = 7 - 6 + 2 = 3

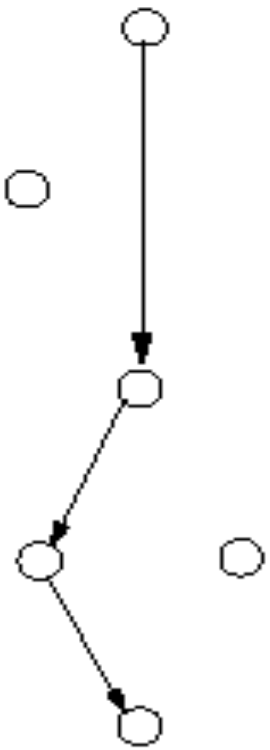


Baseline path



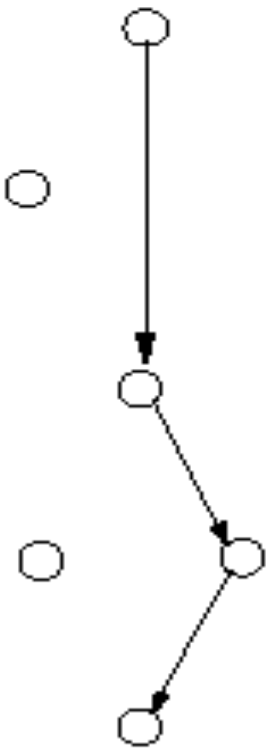
Test Case 1

Switch decision 1



Test Case 2

Switch decision 2



Test Case 3

Werkzeugunterstütztes Unit-Testing m. Coverage-Analyse

- **Schon lange:**
  Testwerkzeuge „lesen“ den Code und erzeugen Skelette von Stubs und Testtreibern oder erzeugen Default-Stubs und erleichtern somit die Arbeit
- **Schon lange:**
  Testwerkzeuge ermitteln die erreichte strukturelle Testabdeckung (z.B. Decision Coverage)
- **Relativ neu:**
  „automatische Erzeugung von Unit-Tests“

Definition der automatischen Erzeugung von Unit Tests

- **Das Testwerkzeug analysiert den Quellcode. Tests werden daher immer *nach* dem Codieren erzeugt.**
- **Tests orientieren**
  Quellcode, könne
  einer (Design-)Sp
- **Vom Werkzeug e**
  Erreichen von 100% struktureller Testabdeckung ab.

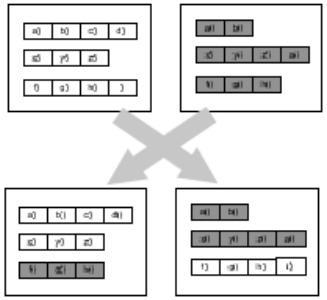
Eigentlich keine „Tests“ sondern nur Inputs der Tests, denn das erwartete Ergebnis kann nur erraten werden.

Definition der automatischen Erzeugung von Unit Tests

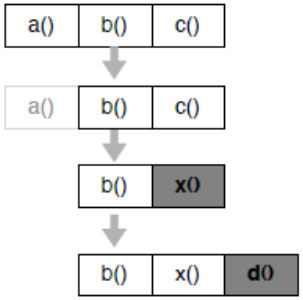
- **Das Testwerkzeug analysiert den Quellcode. Tests werden daher immer *nach* dem Codieren erzeugt.**
- **Tests orientieren sich ausschließlich am Quellcode, können daher nicht das Einhalten einer (Design-)Spezifikation überprüfen.**
- **Vom Werkzeug erzeugte Testfälle zielen auf Erreichen von 100% struktureller Testabdeckung ab.**

Akademische Werkzeuge

- **Genetische Algorithmen (z.B. EvoSuite)**



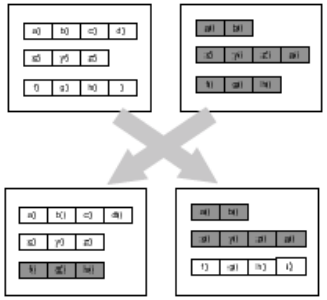
(a) Crossover



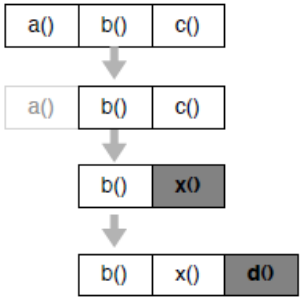
(b) Mutation

Akademische Werkzeuge

- **Genetische Algorithmen (z.B. EvoSuite)**
  - **Auswahl einer Startpopulation von Individuen (Individuum = Testfall)**
  - **„Erbmaterial“ eines Individuums = Testparameter**
  - **Individuen paaren sich und sterben gemäß survival of the fittest**
  - **Fitness function = Test Coverage**
  - **Ende: 100% Coverage oder Zeitüberschreitung**



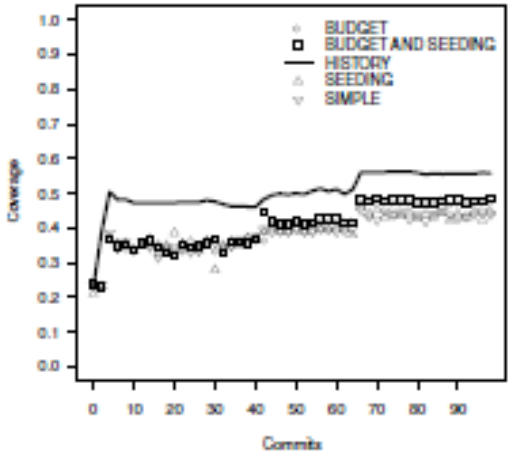
(a) Crossover



(b) Mutation

Akademische Werkzeuge

- **Dynamic Symbolic Execution (z.B. AUSTIN)**
  - **Es wird mitprotokolliert welche Parameter Verzweigungsbedingungen beeinflussen**
  - **und diese für neue Testfälle gezielt geändert**



Kommerzielle Werkzeuge

- **Backtracking (z.B. Cantata)**
  - **Algorithmus bewertet regelmäßig den Status der Lösungssuche**
  - **Kehrt nötigenfalls wieder an eine frühere Position zurück um von dort aus neue Wege zu erforschen**

5	3	1	2	7	6	8	9	4
6	2	4	1	9	5	2		
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Kommerzielle Werkzeuge

- **100% Decision Coverage in < 2s**

Cantata Test Results Summary

Project: *StackTest* **Overall Result: Pass**

Summary Information

Hostname	VIE-GRU-E6420	Cantata version	v6.2
Summary generated	09.09.2015 21:16	Time elapsed during test run	4 seconds

Build Summary

Total tests	1
Compile attempted (files)	2
Link attempted (tests)	1
Execute attempted (tests)	1

Results Summary

PASSED	1
FAILED	0
Total checks	40
Total checks failed	0
Total script errors/call failures	0

Coverage Summary

Entry point (E)	100%
Statement (S)	100%
Decision (D)	100%
Call-return (C)	100%
MC/DC - masking (M)	-
MC/DC - unique cause (U)	-

Problems | Tasks | Console | Properties | Coverage Viewer

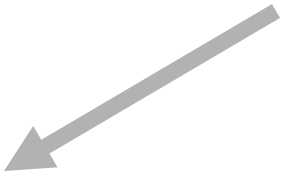
Cantata messages for: StackTest

```
=-----------------------------------------------------------------------=
= 1: init_stack          0      0      6      0      0      PASS =
= 2: resize              0      0      7      0      0      PASS =
= 3: push                0      0      5      0      0      PASS =
= 4: push                0      0      5      0      0      PASS =
= 5: pop                 0      0      7      0      0      PASS =
= 6: pop                 0      0      6      0      0      PASS =
= Other                  0      0      4      0      0      PASS =
=-----------------------------------------------------------------------=
= TOTALS                 0      0     40      0      0      PASS =
=======================================================================
Test complete
```

Zusätzliche Features mancher akademischer Tools

- **Indexüberläufe, Nullpointerdereferenzierung, Division durch Null**

```
void test(int x)
{
    foo[x] = 0;
}
```



```
void test(int x)
{
    if (x < 0)
        throw new NegativeArraySizeException();
    if (x >= foo.length)
        throw new ArrayIndexOutOfBoundsException();
    foo[x] = 0;
}
```

Vision der Forscher

- **Bessere Testfälle durch Werkzeugunterstützung**
- **Schneller 100% Testabdeckung erzielen**

Fehlerfindung mit und ohne autom. Testfallerzeugung

- **JUnit vs. EvoSuite**
- **Testabdeckung bei manuellen Tests gleich hoch wie bei automatisch erzeugten Tests**
- **Fehlerfindungsquote gleich**

Welcher Nutzen bleibt dann?

Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study

Gordon Fraser, Department of Computer Science, University of Sheffield,
Regent Court, 211 Portobello
S1 4DP, Sheffield, UK
Gordon.Fraser@sheffield.ac.uk
Matt Staats, SnT Centre for Security, Reliability and Trust, University of Luxembourg,
4 rue Alphonse Weicker
L-2721 Luxembourg, Luxembourg,
staatsm@gmail.com
Phil McMinn, Department of Computer Science, University of Sheffield,
Regent Court, 211 Portobello
S1 4DP, Sheffield, UK
p.mcminn@sheffield.ac.uk
Andrea Arcuri, Certus Software V&V Center at Simula Research Laboratory,
P.O. Box 134, Lysaker, Norway
arcuri@simula.no
Frank Padberg, Karlsruhe Institute of Technology,
Karlsruhe, Germany

Work on automated test generation has produced several tools capable of generating test data which achieves high structural coverage over a program. In the absence of a specification, developers are expected to manually construct or verify the test oracle for each test input. Nevertheless, it is assumed that these generated tests ease the task of testing for the developer, as testing is reduced to checking the results of tests. While this assumption has persisted for decades, there has been no conclusive evidence to date confirming it. However, the limited adoption in industry indicates this assumption may not be correct, and calls into question the practical value of test generation tools. To investigate this issue, we performed two controlled experiments comparing a total of 97 subjects split between writing tests manually and writing tests with the aid of an

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging

General Terms: Algorithms, Experimentation, Reliability, Theory

Additional Key Words and Phrases: Unit testing, automated test generation, branch coverage, empirical software engineering

This work is supported by a Google Focused Research Award on "Test Amplification"; the National Research Fund, Luxembourg (with grant FNR/P10/03); EPSRC grants EP/K038383/1 ("EXOGEN") and EP/I010396/1 ("RE-COST: REducing the Cost of Oracles in Software Testing"); and the Norwegian Research Council.

DOI : http://dx.doi.org/10.1145/0000000.0000000

ACM Transactions on Software Engineering Methodology, vol. 24, no. 4, 2015

Traum/Wirklichkeit #1: Praxistauglichkeit von automatisch erzeugten Unit Tests

TEST FAST

- **(Design for testability).**
- **Testwerkzeug Testfälle generieren lassen, Tauglichkeit *ohne Blick auf Code* bewerten, Testfälle ohne Relevanz streichen.**
- **Test-Set um methodisch hergeleitete Black-Box-Testfälle erweitern.**
- **Testabdeckung messen.**
  Wenn < 100%: scharf nachdenken, mehr Testfälle.
  Wenn das nichts nützt:
  Quellcode inspizieren, debuggen.

Diskussion des Stands der Technik

```
unsigned max(unsigned a, unsigned b, unsigned c)
{
   unsigned max = 0;
   if (a > c) max = a;
   if (b > a) max = b;
   if (c > b) max = c;
   return max;
}
```

- **max(3, 5, 7) == 7**
- **max(5, 7, 3) == 7**
- **max(7, 3, 5) == 7**
- **max(1, 0, 0) == 1**
- **max(0, 1, 0) == 1**
- **max(0, 0, 1) == 1**

Diskussion des Stands der Technik

```
unsigned max(unsigned a, unsigned b, unsigned c)
{
   unsigned max = 0;
   if (a > c) max = a;
   if (b > a) max = b;
   if (c > b) max = c;
   return max;
}
```

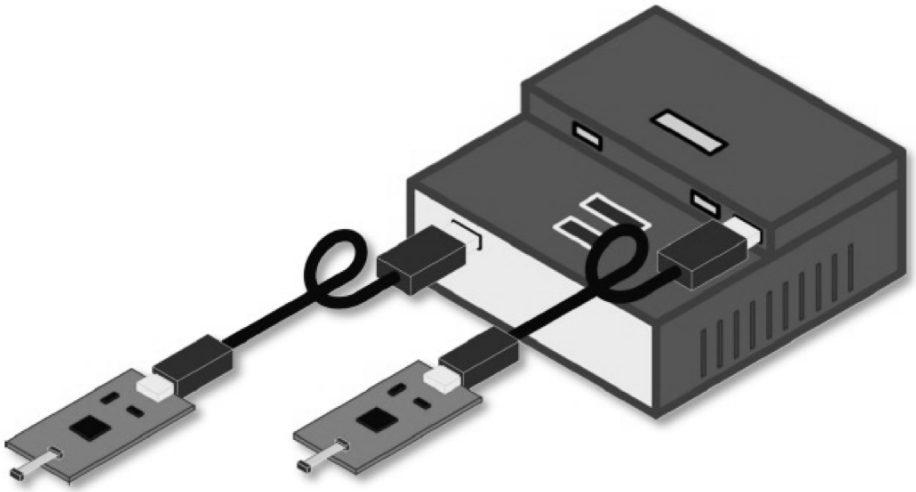
- **max(MAX_INT, 0, 0) == MAX_INT**
- **max(0, MAX_INT, 0) == MAX_INT**
- **max(0, 0, MAX_INT) == MAX_INT**
- **max(MAX_INT, MAX_INT, MAX_INT) == 0**
- **max(0, 0, 0) == 0**

Schlussfolgerungen

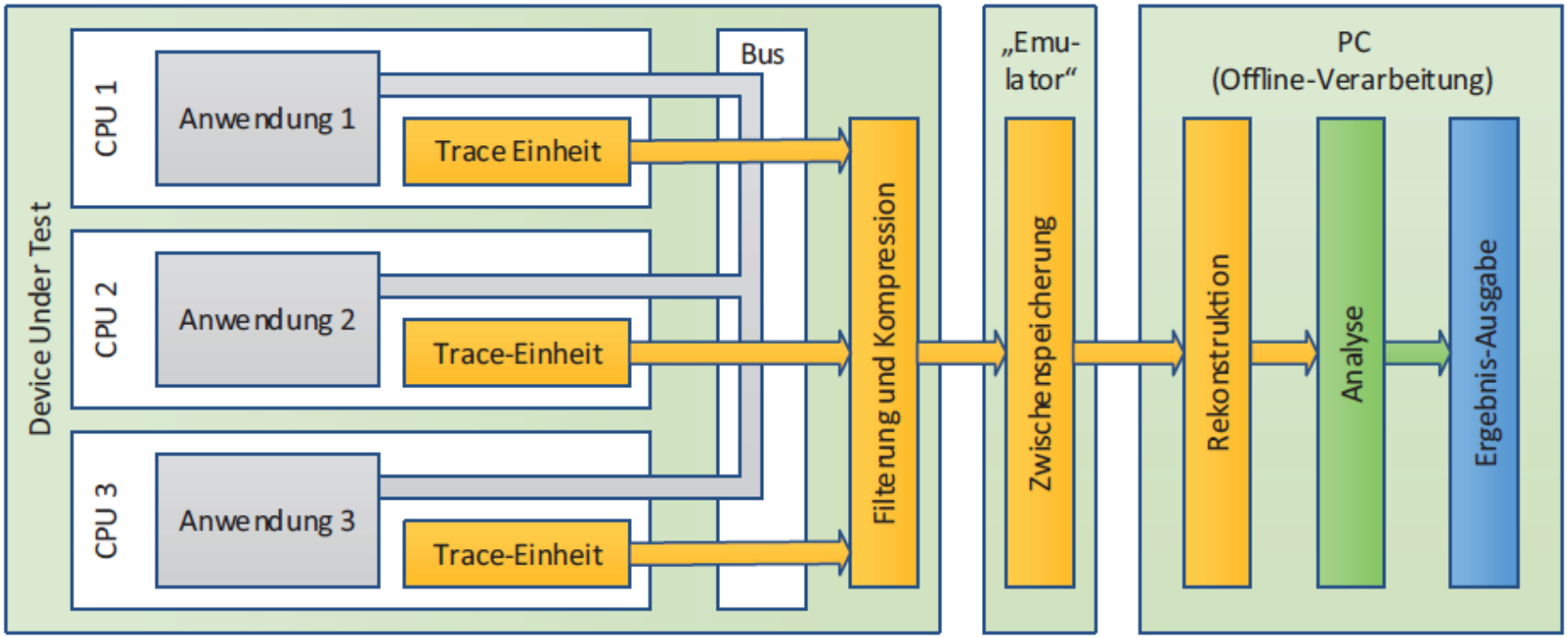
- **Wir können uns durch Werkzeuge bei der Unit-Test-Erstellung unterstützen lassen, sollten uns aber nicht durch die Werkzeuge in die Irre führen lassen. Werkzeuge können uns zur Zeit nur Tipparbeit abnehmen, nicht das Denken.**
- **Das Erzielen von 100% struktureller Testabdeckung (Coverage) darf auf keinen Fall überbewertet werden. Da gibt es noch mehr ... !**
- **Der Nachweis der Testabdeckung ist aber für Software mit hohem Integritätsanspruch aber auf jeden Fall notwendig ...**

Test Coverage (struktrurelle Testabdeckung)

- **... wird meist mit Hilfe von Instrumentierung des Codes erfasst,**
- **kann aber auch ohne Instrumentierung über das Trace-Port erfasst werden, wenn dem Testwerkzeug die CPU hinlänglich „bekannt“ ist.**

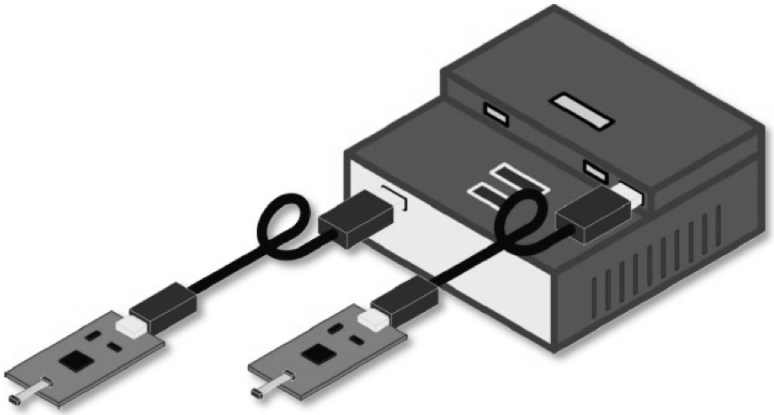


Offline-Analyse über Embedded-Trace-Schnittstelle

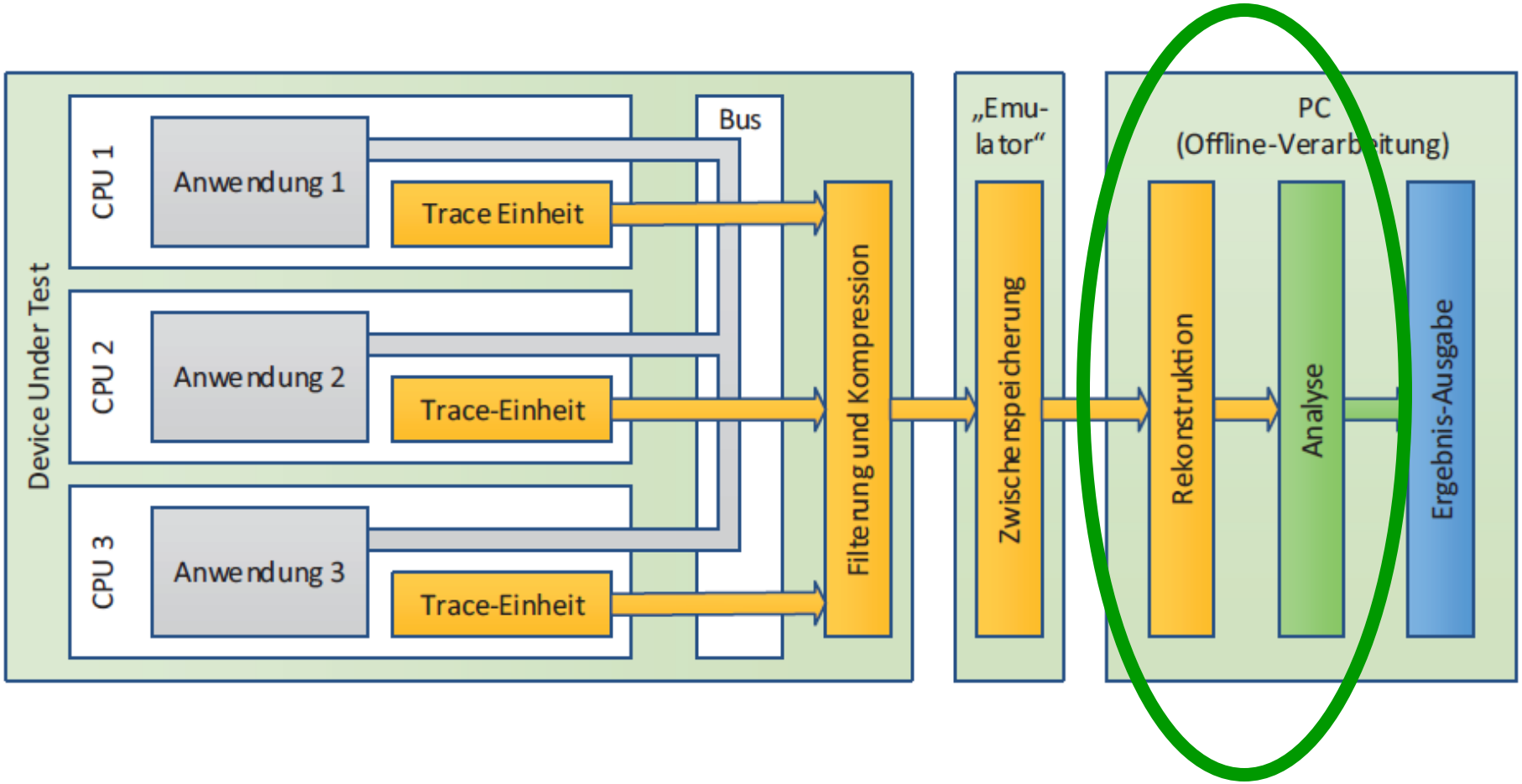


Randbedingung

- **Ist in Ihrem Projekt das Trace-Port im Endgerät zugänglich?**



Möglicher nächster Schritt: prüfe Korrektheitseigenschaften



Aufspüren von Data Races mit Eraser (Lockset Algorithmus)

(1) Für jede shared Variable *v*, $C(v) = all\ locks$.

(2) Für jd. Zugriff durch Thread *t* setze
$C(v) := C(v) \cap locks\_aktiv(t)$.

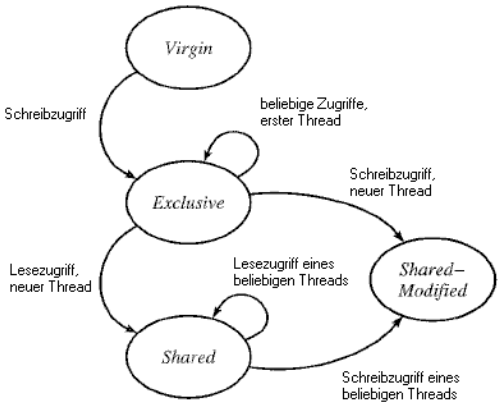
(3) Wenn während Programmausführung $C(v) = \{\}$,
dann gib eine Warnung aus.

Aufspüren von Data Races mit Eraser (Lockset Algorithmus)

Thread 1, CPU-1	Thread 2, CPU-2	*Locks_aktiv*	*C(v)*
		{ }	{mtx1, mtx2}
lock(mtx1);	Irgendwas();	{mtx1}	{mtx1, mtx2}
v = v + 1;	Irgendwas();	{mtx1}	{mtx1}
unlock(mtx1);	Irgendwas();	{ }	{mtx1}
Irgendwas();	lock(mtx2);	{mtx2}	{mtx1}
Irgendwas();	v = v + 1;	{mtx2}	**{ }**
Irgendwas();	unlock(mtx2);	{ }	{ }

Bewertung von Eraser (Stand der Technik)

- Ein Data Race muss nicht auftreten um als solches erkannt zu werden, aber der betroffene Code-Teil muss zumindest einmal durchlaufen werden.
- Eraser kann nur mit Locks umgehen und erfordert eine „Locking Discipline“.
- Zur Ausführung muss der Code instrumentiert werden und/oder Debug-Info zur Laufzeit interpretiert werden.



Aufspüren von Data Races durch Verletzung der Happens-Before-Relation

- Protokolliere Zugriffe auf Variablen während der Programmausführung
- Wenn zwei Zugriffe auf eine shared Variable bezüglich Happens-Before nicht geordnet sind, wird eine Warnung ausgegeben.

Zwei Operationen sind gemäß dieser transitiven Halbordnung geordnet, wenn

- sie in ein und demselben Thread ausgeführt werden und so definitiv eine Operation vor einer anderen ausgeführt wird, oder
- wenn sie Synchronisationsoperationen sind und die Semantik dieser Operationen eine andere zeitliche Reihenfolge nicht gestattet.

Thread 1	Thread 2
`lock(mutex);`	
`x = x + 1;`	
`unlock(mutex);`	
	`lock(mutex);`
	`x = x + 1;`
	`unlock(mutex);`

Thread 1	Thread 2
y = y + 1;	
lock(mutex);	
x = x + 1;	
unlock(mutex);	
	lock(mutex);
	x = x + 1;
	unlock(mutex);
	y = y + 1;

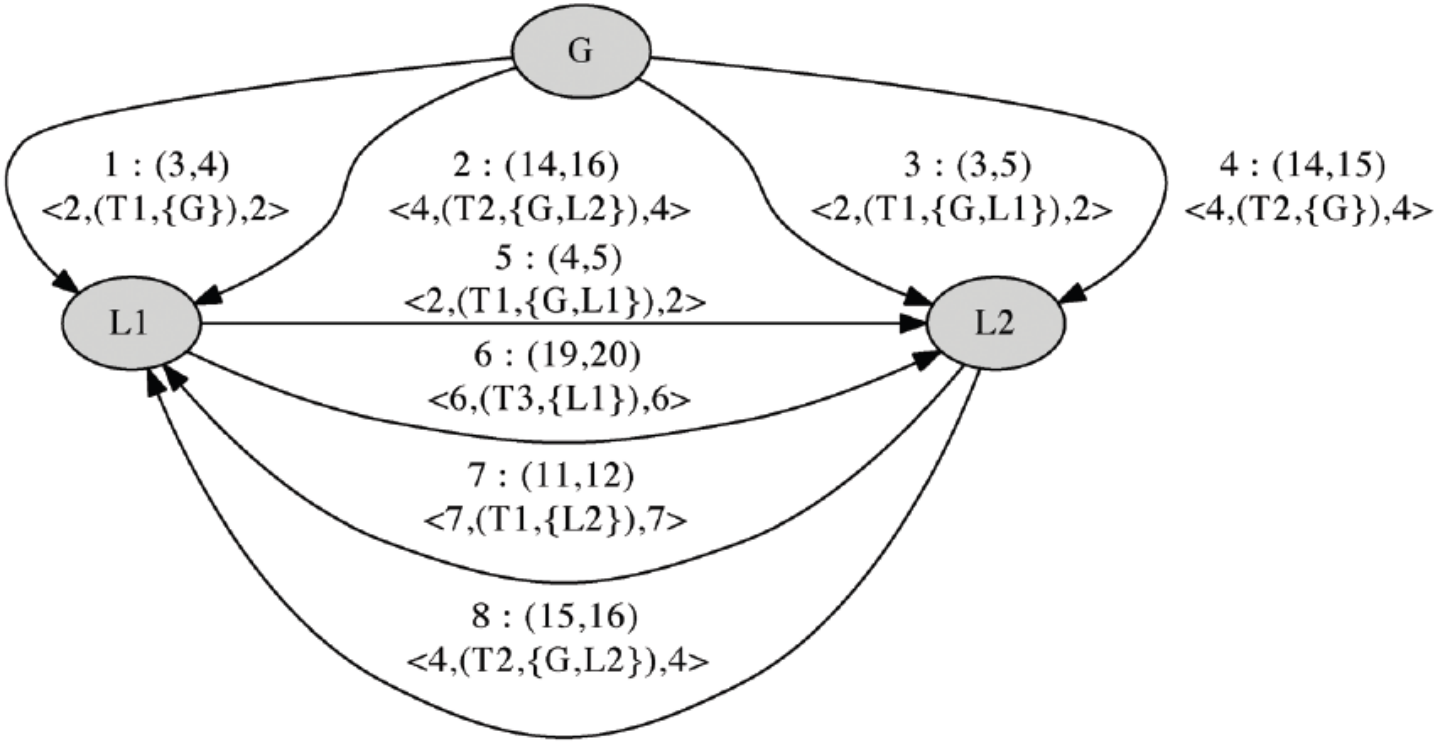
Bewertung der Analyse der Happens-Before-Relation

- Ein Data Race muss nicht auftreten um als solches erkannt zu werden, aber der betroffene Code-Teil muss zumindest einmal durchlaufen werden.
- Das Verfahren kann mit einer Vielzahl von Synchronisierungs-Mechanismen umgehen.
- Zur Ausführung muss der Code instrumentiert werden und/oder Debug-Info zur Laufzeit interpretiert werden.
- Zahlreiche Task-Durchläufe empfohlen!

Alternativen zu dynamischen Verfahren

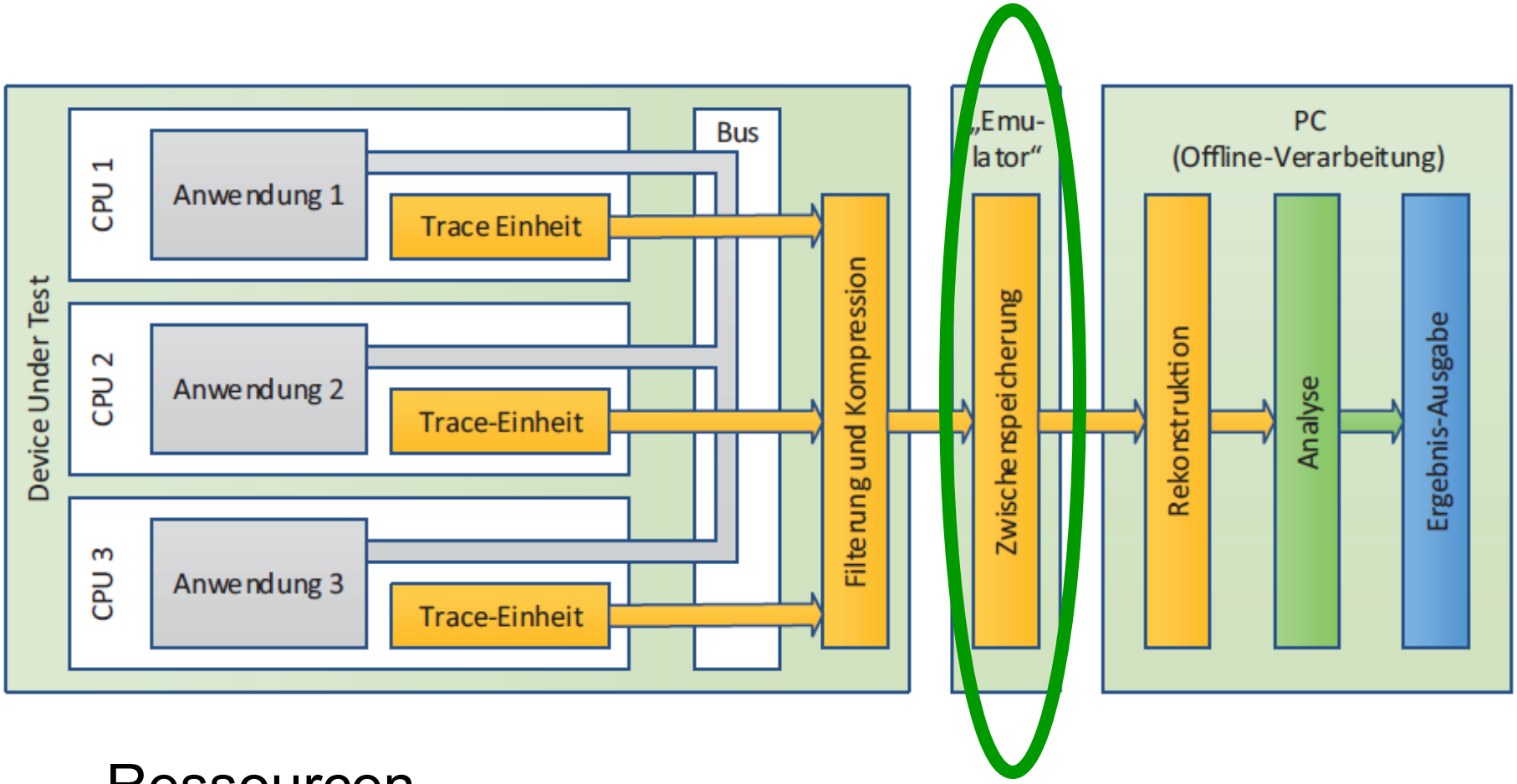
- Einsatz der beschriebenen Verfahren auf einem Modell des Codes in statischen Analyse-Werkzeugen. (Es müssen für Variablen-Zugriffe über Zeiger Heuristiken anwendet werden um einen Speicherzugriff verschiedener Threads auf eine gemeinsame Variable zu „erkennen“; daher Neigung zu Falschwarnungen).
- Abstrakte Interpretation (Möglichkeit des mathematischen Nachweises der Abwesenheit von Data Races).

Deadlock-Analyse



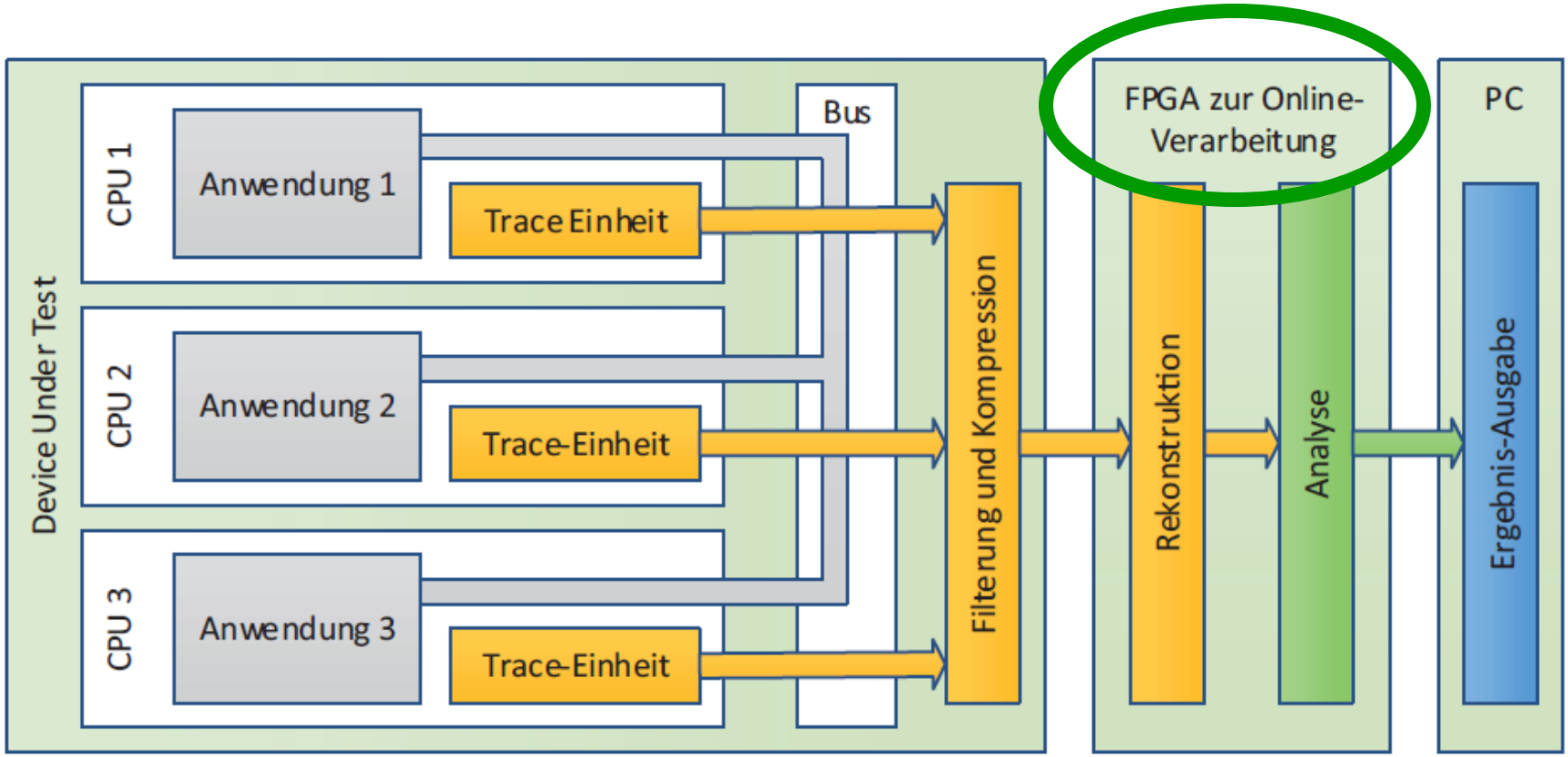
Wirklichkeit/Traum #2: Data Race Detection ohne Instrumentierung in Echtzeit

Problem beim Prüfen von Korrektheitseigenschaften über die Trace-Schnittstelle:



... Ressourcen

Lösungsmöglichkeit für das Problem



... Runtime Verification

Zusammenfassung

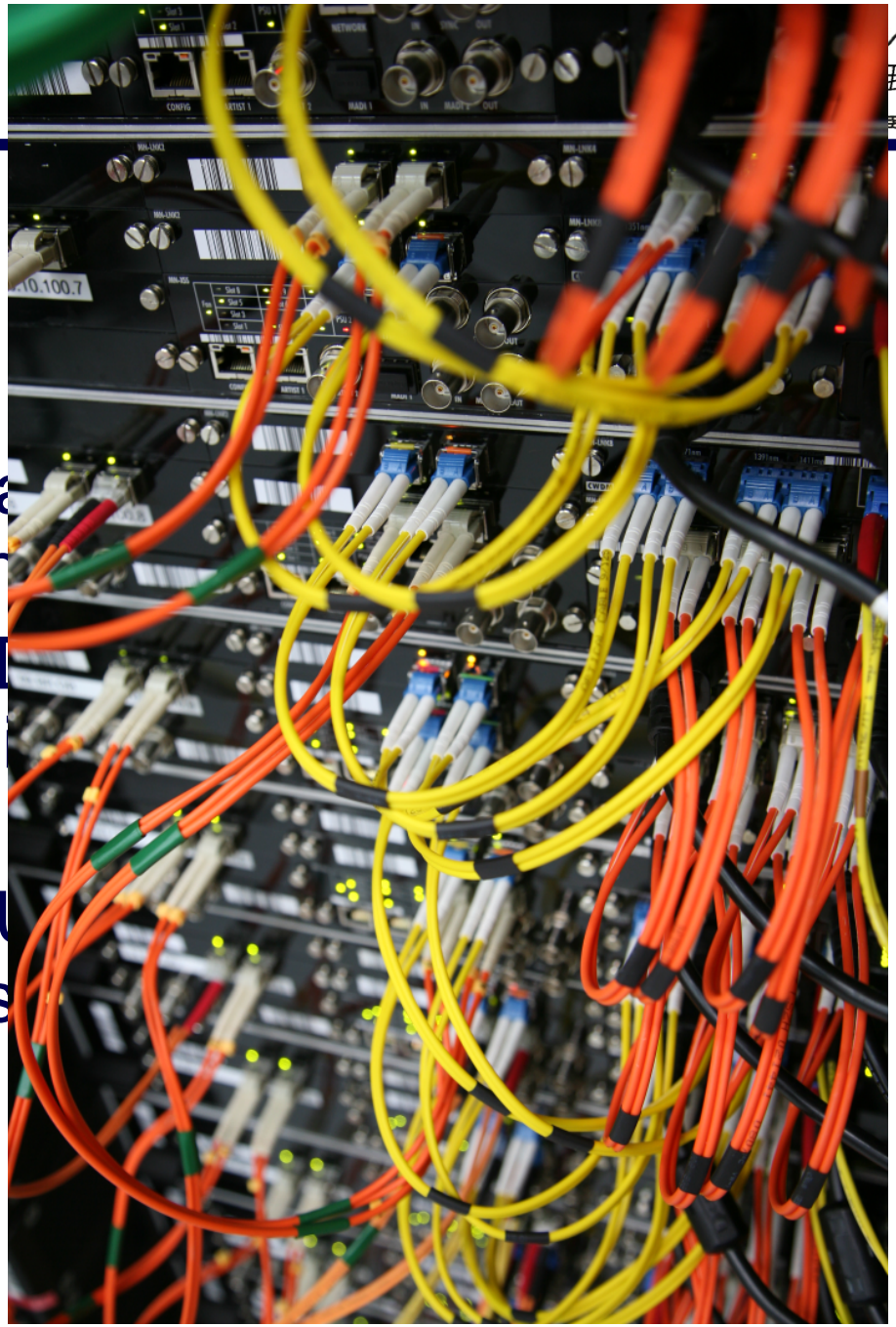
- Freuen Sie sich über automatische Erstellung von Unit Tests. Aber nicht zu sehr.
- Freuen Sie sich über Data Race und Deadlock-Erkennung in Echtzeit. Aber nicht zu früh.

Zusammenfassung

- Freuen Sie sich über automatische Erstellung von Unit Tests. Aber nicht zu sehr.
- Freuen Sie sich über Data Race und Deadlock-Erkennung in Echtzeit. Aber nicht zu früh.
- Freuen Sie sich über Unit-Tests ohne Instrumentierung. Das könnte Ihnen ein neues Projekt bringen. Aber nur, wenn Sie auch zu den Trace-Ports Ihrer CPU kommen.

Zusammenfassung

- Freuen Sie sich über a
von Unit Tests. Aber n
- Freuen Sie sich über D
Deadlock-Erkennung
früh.
- Freuen Sie sich über U
Instrumentierung. Das
Projekt bringen. Aber
den Trace-Ports Ihrer



Persönliche Tipps zum Abschluss

- Nehmen Sie die Zugänglichkeit des Trace-Ports bei neuen Projekten als eigenen Punkt ins Lastenheft auf.
- Auch wenn Sie heute damit nichts anfangen können: Vielleicht sind Sie in einigen Jahren sehr froh darüber!
- stephan.gruenfelder@gmx.at