# Einsatz von Debuggern im Hardware-in-the-Loop-Test

## Test vom Modultest bis zum Systemtest

SAEC Days 21.07.2020 Kristian Trenkel

# Gliederung

I. Einleitung
II. Hardware-in-the-Loop-Test
III. Integration von Debuggern in die Testumgebung
IV. Anwendungsbeispiele
  a. Test eines Lenkradschloss-Steuergerätes
  b. Codecoverage-Messungen bei einem Steuergerät für eine Wankstabilisierung
V. Zusammenfassung und Ausblick

# Einleitung

## Herausforderungen

Oft mehr als 50 Steuergeräte (ECUs) und bis zu 100 Mio Code-Zeilen

Requirements Traceability

Komplexe Systeme

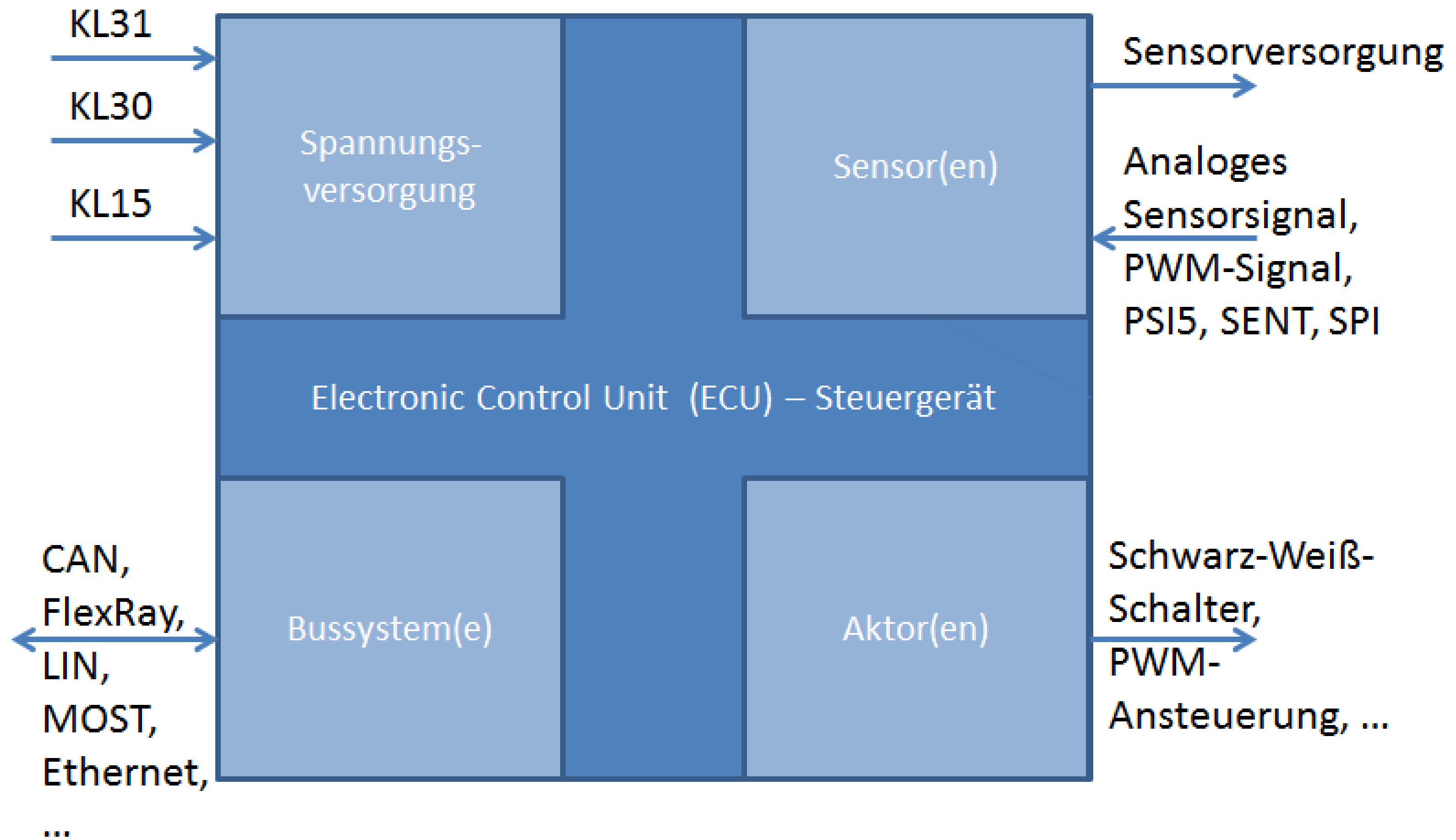
Kurze Entwicklungs-zyklen

Meist mehr als 3000 Requirements pro ECU

Steigende Safety-Anforderungen

# Einleitung

## Device under Test – DUT



# Einleitung

## Anforderungen an den Test

- Automatisierter Test komplexer Steuergeräte
- Absicherung sicherheitskritischer Funktionen (z. B. Fahrerassistenzsysteme)
- Erhöhung der Testabdeckung
  → White Box- und Grey Box-Tests
- Durchführung der Tests mit realer Hardware und Software unter Echtzeitbedingungen
- Bestimmung der Codecoverage ohne Einfluss auf die Software

# Gliederung

I. Einleitung
II. Hardware-in-the-Loop-Test
III. Integration von Debuggern in die Testumgebung
IV. Anwendungsbeispiele
   a. Test eines Lenkradschloss-Steuergerätes
   b. Codecoverage-Messungen bei einem Steuergerät für eine Wankstabilisierung
V. Zusammenfassung und Ausblick

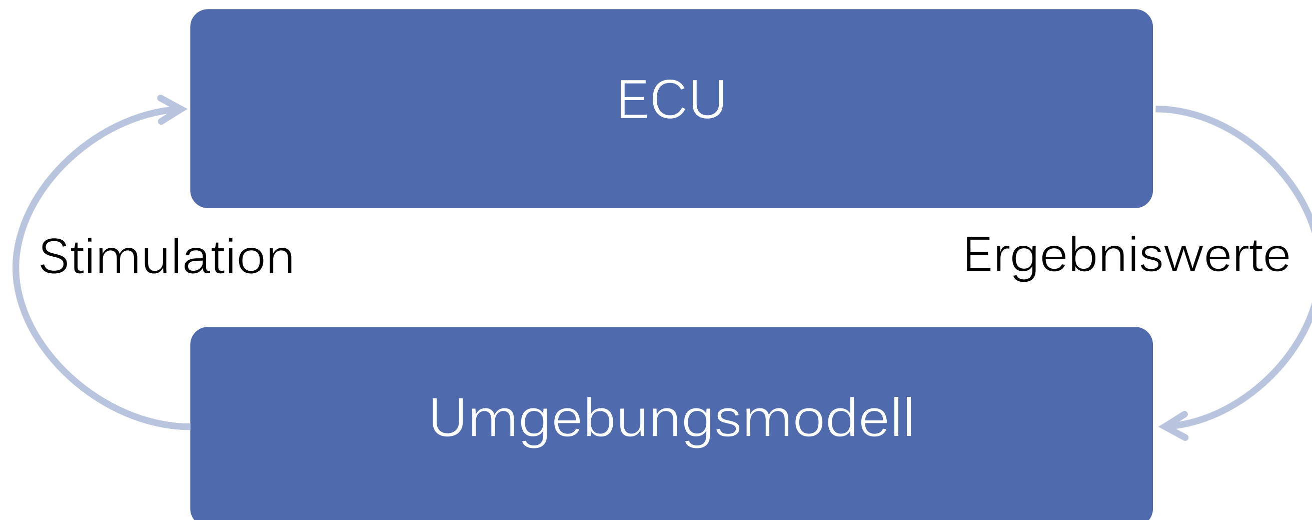
# Hardware-in-the-Loop-Test

## Einführung

- Begriffsklärung:

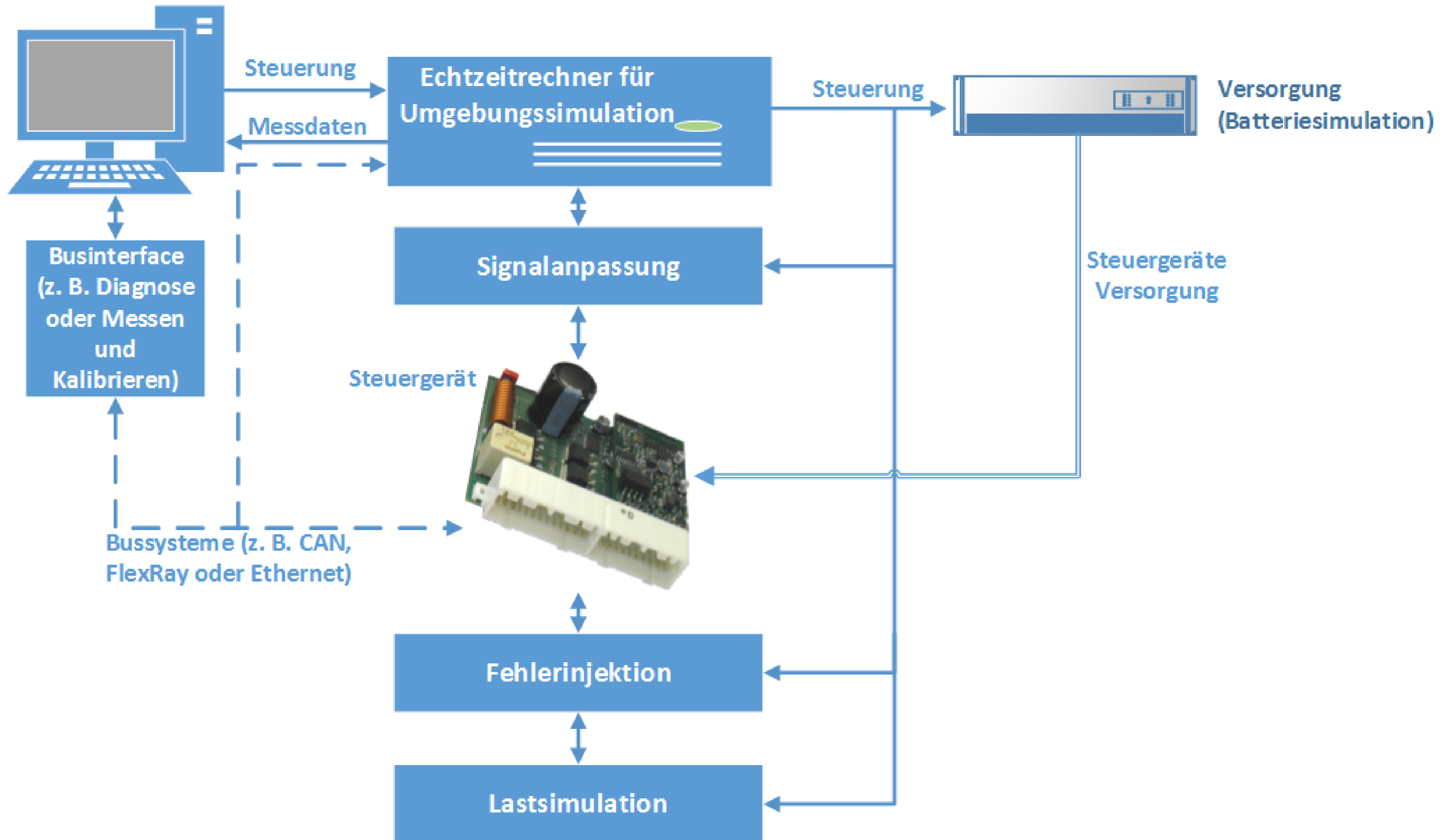
  *„bezeichnet ein Verfahren, bei dem ein eingebettetes System (z. B. reales elektronisches Steuergerät oder reale mechatronische Komponente) über seine Ein- und Ausgänge an ein angepasstes Gegenstück, das im allgemeinen HIL-Simulator genannt wird und als Nachbildung der realen Umgebung des Systems dient, angeschlossen wird."*

- Entwicklungsbegleitendes Testverfahren für Embedded Systems



# Hardware-in-the-Loop-Test

## Aufbau eines HIL-Systems






TESTHAUS AUS ÜBERZEUGUNG

# Hardware-in-the-Loop-Test

## Aufbau eines HIL-Systems

# Hardware-in-the-Loop-Test

**White Box- und Grey Box-Tests – Bisheriges Vorgehen**

- Zugriff auf Variablen des DUT über das Protokoll CCP[1]/XCP[2]
  - Lesender und schreibender Zugriff
- Verwendung bereits vorhandener Bussysteme
- Einfluss auf Ressourcenbedarf und Laufzeit der Software
- Notwendige Ressourcen (z. B. RAM) zum Teil nicht verfügbar

➔ Einfluss auf Test bzw. Testergebnisse möglich

[1] CAN Calibration Protocol
[2] Extended Calibration Protocol – Universal Measurement and Calibration Protocol


TESTHAUS AUS ÜBERZEUGUNG

# Hardware-in-the-Loop-Test

## Codecoverage-Messungen – Bisheriges Vorgehen

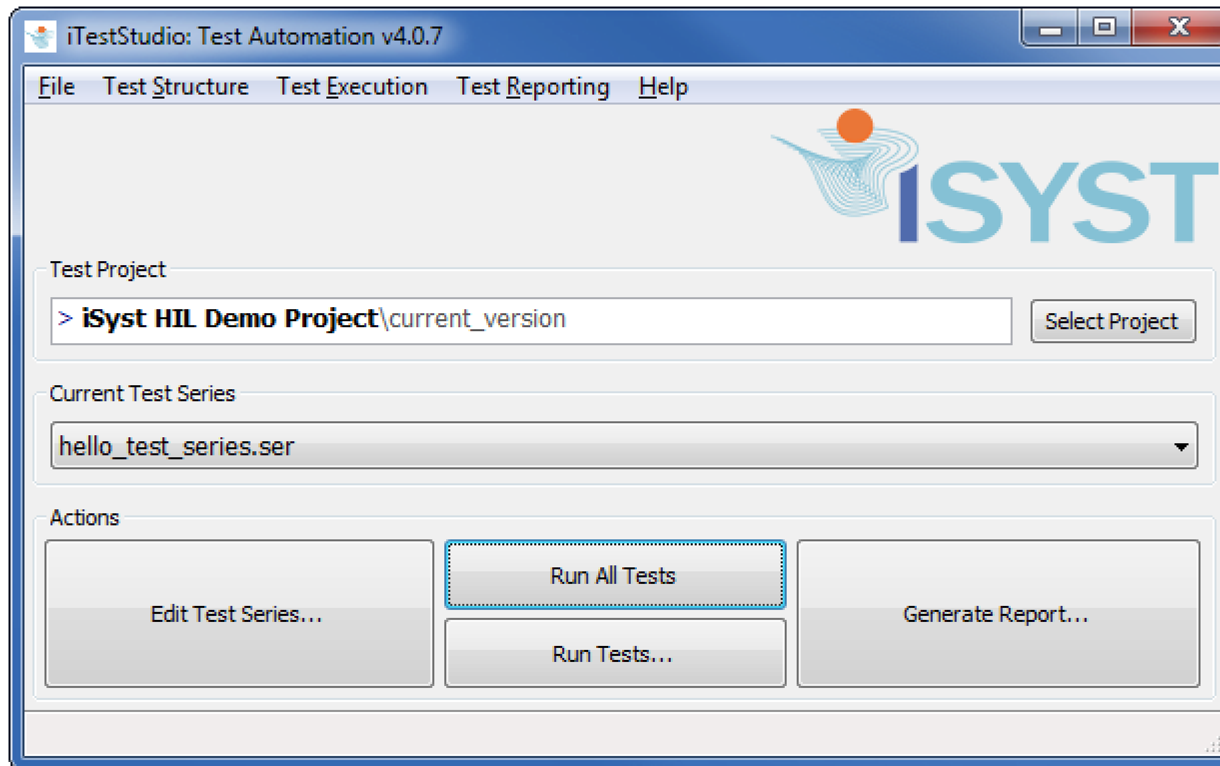
- Instrumentierung des Codes
- Einfluss auf Ressourcenbedarf und Laufzeit der Software
- Notwendige Ressourcen (z. B. RAM) oft nicht verfügbar

➔ Einfluss auf Test bzw. Testergebnisse möglich

[1] CAN Calibration Protocol
[2] Extended Calibration Protocol – Universal Measurement and Calibration Protocol

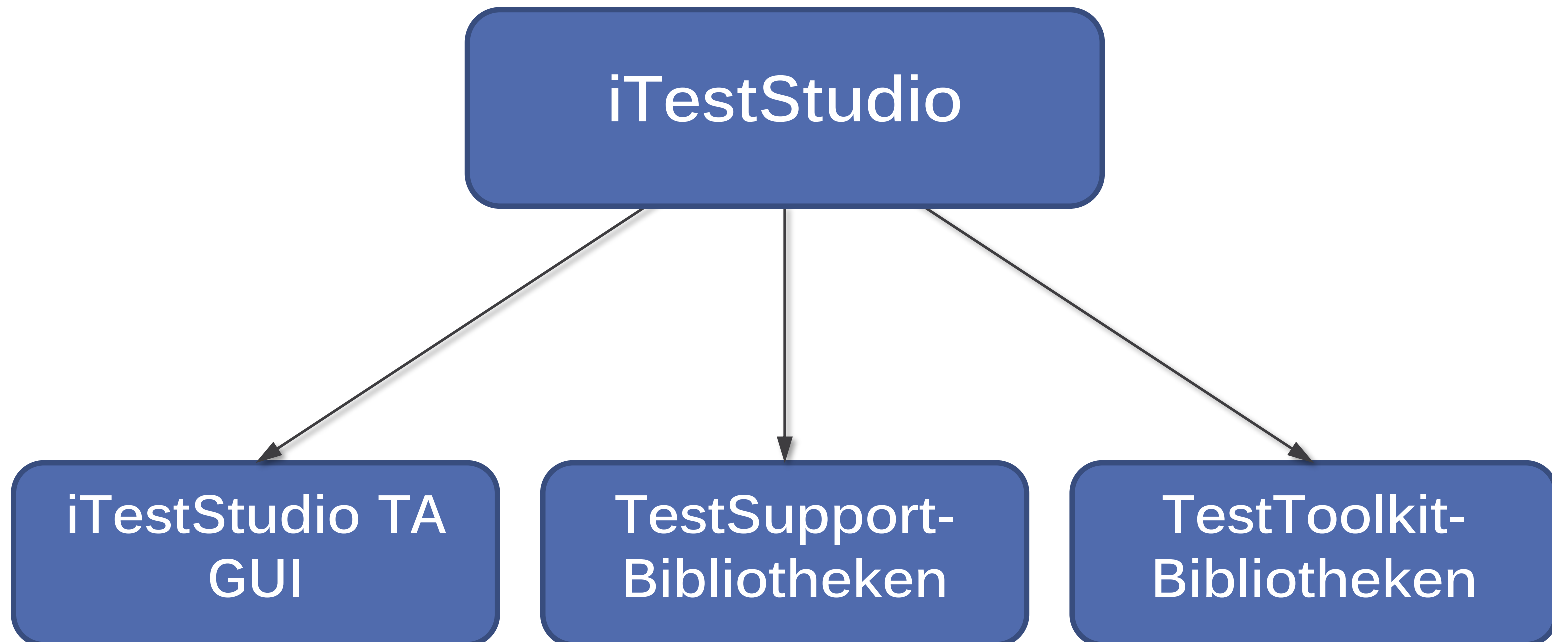
# Hardware-in-the-Loop-Test

## Testautomatisierung – iTestStudio



# Hardware-in-the-Loop-Test

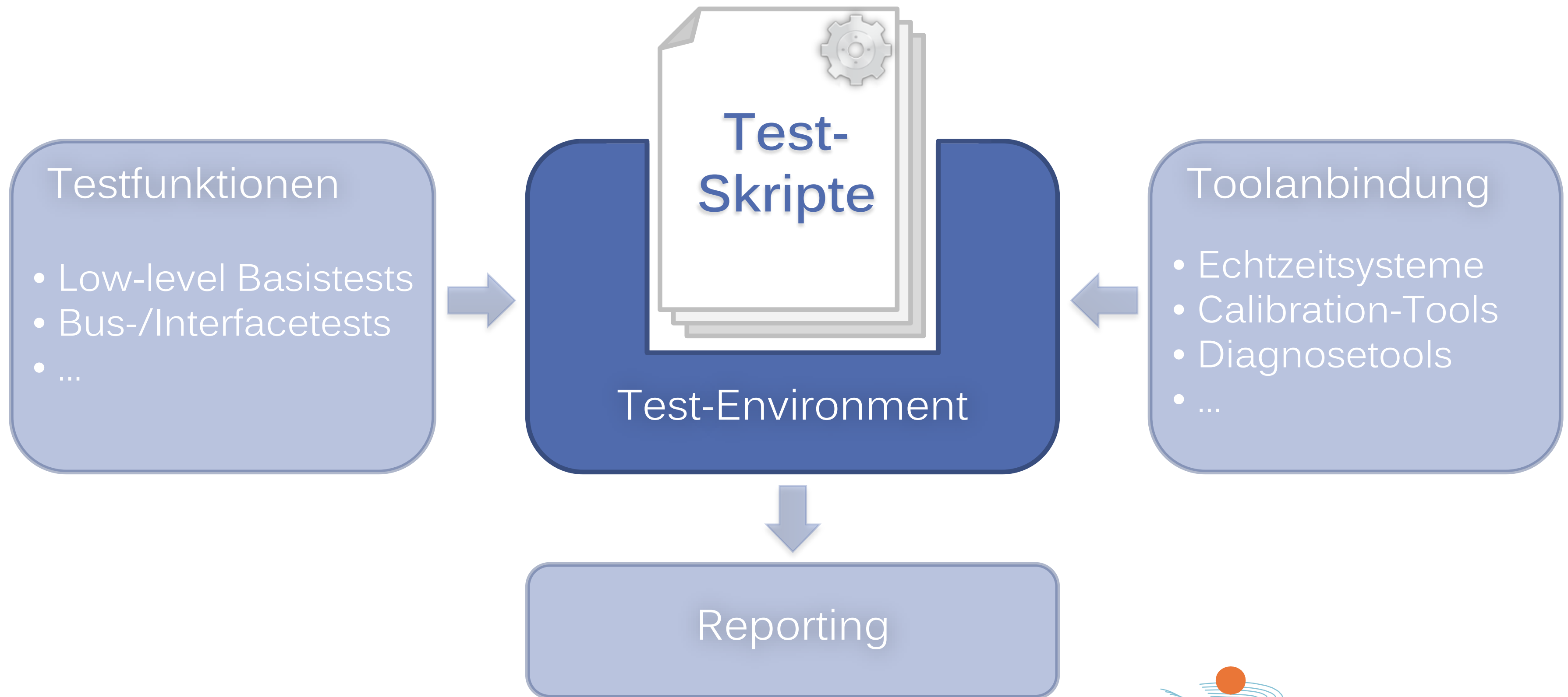
## Testautomatisierung – iTestStudio




ISYST

# Beispiel-Tool – iTestStudio

## Testautomatisierung – iTestStudio




ISYST

# Gliederung


I. Einleitung

II. Hardware-in-the-Loop-Test

III. Integration von Debuggern in die Testumgebung

IV. Anwendungsbeispiele

  a. Test eines Lenkradschloss-Steuergerätes

  b. Codecoverage-Messungen bei einem Steuergerät für eine Wankstabilisierung

V. Zusammenfassung und Ausblick

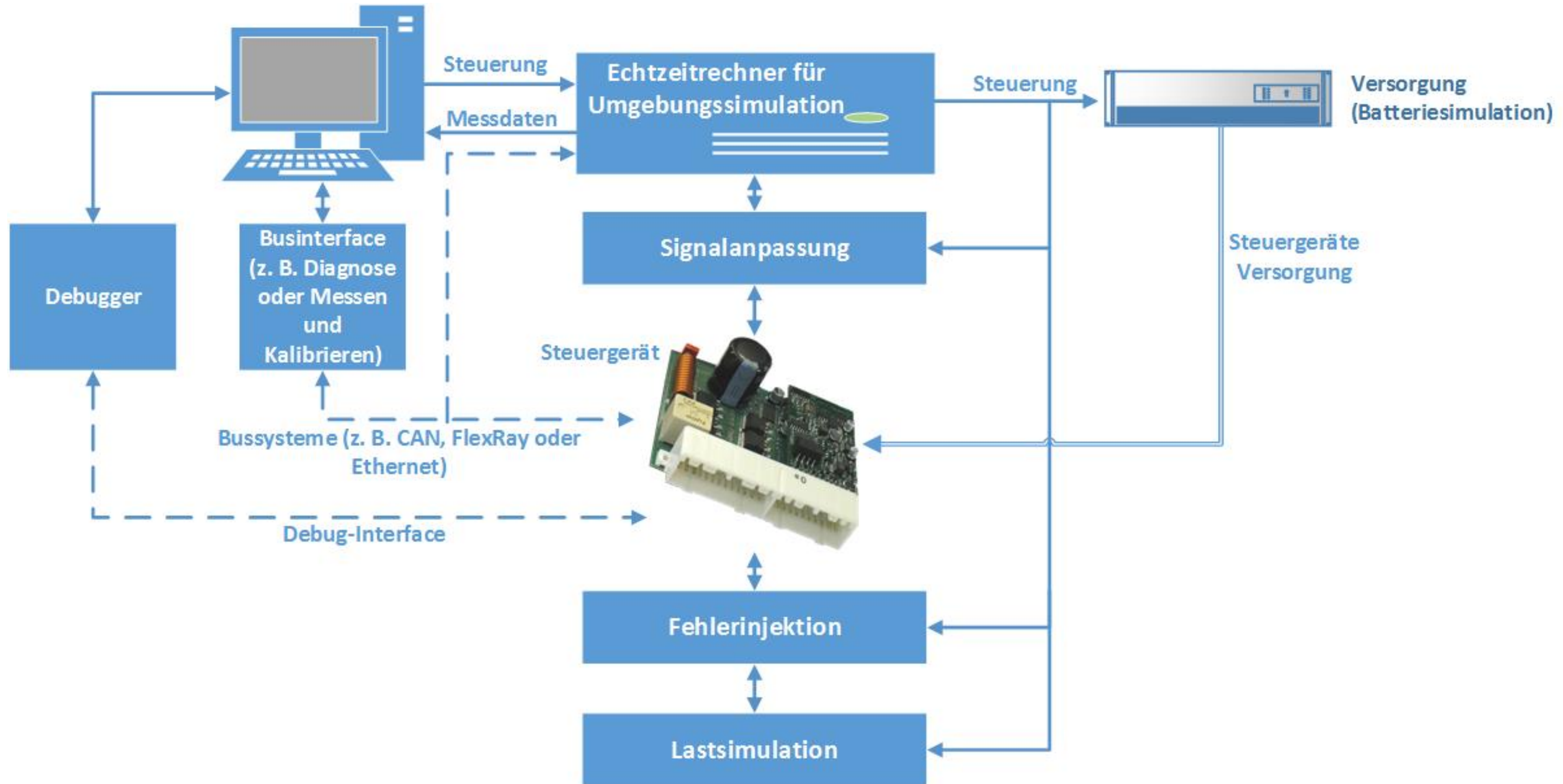
# Integration von Debuggern in die Testumgebung

**Ziele**

- Lese- und Schreibzugriffe auf (globale) Variablen des DUT ohne Einfluss auf das Laufzeitverhalten
- Bestimmung der Code-Abdeckung ohne Einfluss auf das Laufzeitverhalten
- Universelle Schnittstelle zu Debuggern über Abstraktionsschicht
- Keine Abhängigkeit der Testfälle von Toolschnittstelle bzw. Debugger
- Wiederverwendbarkeit der Schnittstellen in verschiedenen Projekten

# Integration von Debuggern in die Testumgebung

## Aufbau eines HIL-Systems



# Integration von Debuggern in die Testumgebung

**Variablenzugriff**

- Zweistufige Abstraktion zwischen Debugger/Debug-Tool und Testautomatisierung

- 1. Stufe:
  - Klasse, welche eine einzelne Variable repräsentiert
  - Realisierung der spezifischen Anbindung an den Debugger für eine Variable
  - Implementierung der Zugriffsmethoden *set* und *get*

# Integration von Debuggern in die Testumgebung

## Variablenzugriff

```
class EcuVar(VarBase):
    """ Simple wrapper for debugger variables """

    # ##########################################################################
    def __init__(self, context, identifier,
                 unit="", alias="", descr="",
                 lookup=None, default=None, resettable=True,
                 cpl_id=None):
```

- context – Referenz auf Schnittstelle des Debuggers
- identifier – Variablenname/-bezeichner



TESTHAUS AUS ÜBERZEUGUNG

# Integration von Debuggern in die Testumgebung

## Variablenzugriff

- 2. Stufe:
  - Container-Klasse, welche alle Variablen für ein Testprojekt bzw. ein DUT sammelt
  - Handling toolspezifischer Eigenschaften (z. B. Starten des Tools)

```
class EcuVarContainer(VarContainerBase):
    """ Base class for ECU variable containers """
    def __init__(self, debugger_api):
        print "loading BlueBox variables:"
```

# Integration von Debuggern in die Testumgebung

## Variablenzugriff – Anwendung

```
class EcuVars(EcuVarContainer):
    def __init__(self, debugger_api):

        EcuVarContainer.__init__(self, debugger_api)

        self.state_en    = EcuVar(debugger_api, "(((TDFM).dummy_module).dummy_structure).State_en",
                                  lookup={1: "UNDEFINED", 2: "INACTIVE", 3: "ACTIVE"}, resettable=False)
```

```
# get debugger variables
ecu = testenv.getDebug()

# set state_en to 1
ecu.state_en.set(1)

# get state_en value
state_en_value = ecu.state_en.get()
```

# Integration von Debuggern in die Testumgebung

## Codecoverage

- Codecoverage-Messung über Debug-Schnittstelle der Mikrokontroller auf Binärcode-Ebene
  - Mapping auf C-Code über Tools der Debugger
- Umsetzung unabhängig von Testfällen
- Integration in Standard-Methoden des TestEnvironment für Startup und Shutdown des DUT
- Wrapper-Klasse zur Adaption der verschiedenen Schnittstellen der Debugger(-Tools)

# Gliederung

I. Einleitung
II. Hardware-in-the-Loop-Test
III. Integration von Debuggern in die Testumgebung
IV. Anwendungsbeispiele
    a. Test eines Lenkradschloss-Steuergerätes
    b. Codecoverage-Messungen bei einem Steuergerät für eine Wankstabilisierung
V. Zusammenfassung und Ausblick

# Anwendungsbeispiele

**Test eines Lenkradschloss-Steuergerätes**

- HIL-Testprojekt für das Steuergerät eines Lenkradschlosses
- Sicherheitseinstufung ASIL D
- Keine XCP-Unterstützung (Ressourcenmangel)
- Variablenzugriff für nötige Testabdeckung erforderlich
- Lösung:
  - Einsatz/Anbindung des BlueBox-Debuggers und der Software winIDEA von iSYSTEM

➔ Problemlose Integration in HIL-Umgebung

➔ Herstellung der Testbarkeit in wenigen Tagen

➔ Wiederverwendung von Testfällen

# Gliederung

I. Einleitung
II. Hardware-in-the-Loop-Test
III. Integration von Debuggern in die Testumgebung
IV. Anwendungsbeispiele
a. Test eines Lenkradschloss-Steuergerätes
b. Codecoverage-Messungen bei einem Steuergerät für eine Wankstabilisierung
V. Zusammenfassung und Ausblick

# Anwendungsbeispiele

**Codecoverage-Messungen bei einem Steuergerät für eine Wankstabilisierung**

- HIL-Testprojekt für eine Wankstabilisierung
- Einstufung als sicherheitskritisches System
- Codecoverage-Messung für Tests notwendig
- Instrumentierung des Codes nicht möglich/zulässig
- Lösung
  - Codecoverage-Messung mittels Debugger
  - iSYSTEM winIDEA/BlueBox und Lauterbach Trace32/PowerDebug

➔ Problemlose Integration in HIL-Umgebung

➔ Austausch der Debugger ohne Anpassung an Testfällen

# Gliederung


I. Einleitung
II. Hardware-in-the-Loop-Test
III. Integration von Debuggern in die Testumgebung
IV. Anwendungsbeispiele
    a. Test eines Lenkradschloss-Steuergerätes
    b. Codecoverage-Messungen bei einem Steuergerät für eine Wankstabilisierung
V. Zusammenfassung und Ausblick

# Zusammenfassung und Ausblick

- Einbindung von Debuggern in den HIL-Test möglich und sinnvoll
- Variablenzugriff ohne Einfluss auf das DUT
- Exakte Codecoverage-Messung ohne Einfluss auf das DUT
- Erweiterung der Testmöglichkeiten des HIL-Tests
- Austausch der Debugger ohne Anpassung der Tests
- Wiederverwendbarkeit der Schnittstellen in anderen Testprojekten

Vielen Dank für die Aufmerksamkeit!

Fragen oder Anregungen?