

Verlässliche Echtzeitsysteme

Verifikation nicht-funktionaler Eigenschaften

Wintersemester 2024/25

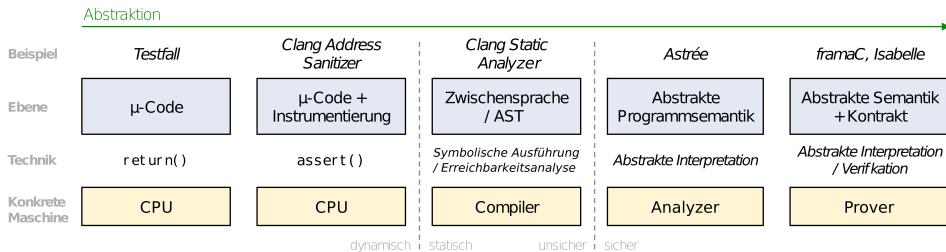
Peter Wägemann

Lehrstuhl für Systemsoftware

Friedrich-Alexander-Universität Erlangen-Nürnberg

<https://sys.cs.fau.de>

Wiederholung: Verifikationsverfahren



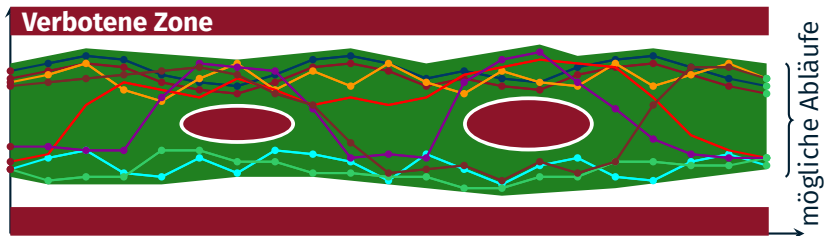
■ *Statisch* versus *dynamisch*

- Nutzung der konkreten/abstrakten Programmsemantik (siehe Folien VIII/15 ff)
- Konkrete Ausführung (Maschine) hängt von Betrachtungsebene ab!

■ *Sicher* versus *unsicher*

- Vollständigkeit der Analyse (sicher \mapsto 100 %, siehe Folien VIII/20 ff)
- Steht im Bezug zu bestimmter Spezifikation (z.B. C-Standard bei Astrée)

Wiederholung: Abstrakte Interpretation (vgl. VIII/20 ff)



- *Abstrakte Interpretation* (engl. *abstract interpretation*)
 - Betrachtet eine *abstrakte Semantik* (engl. *abstract semantics*)
 - Sie umfasst **alle Fälle der konkreten Programmsemantik**
 - Sicherheitszonen beschreiben fehlerhafte Zustände
 - abstrakte Semantik sicher \Rightarrow konkrete Semantik ist sicher
 - Eigenschaften: Vollständigkeit, Genauigkeit, Effizienz

Übersicht und Problemstellung

Abstrakte Interpretation für Laufzeitfehler ist nicht genug:

- Bisher stand Verifikation des korrekten Verhaltens im Vordergrund
 - *Abstrakte Interpretation:*
Abwesenheit von Laufzeitfehlern (Sprachstandard, nicht-funktional)

Übersicht und Problemstellung

Abstrakte Interpretation für Laufzeitfehler ist nicht genug:

- Bislang stand Verifikation des korrekten Verhaltens im Vordergrund
 - *Abstrakte Interpretation:*
Abwesenheit von Laufzeitfehlern (Sprachstandard, nicht-funktional)

- ⚠ Dies ist **notwendig**, jedoch **nicht hinreichend**
 - Einfluss nicht-funktionaler Eigenschaften der Ausführungsumgebung
 - Anwendung ist in die Umwelt eingebettet!
 - Exemplarisch: **Speicherverbrauch** und **Laufzeit**

Übersicht und Problemstellung

Abstrakte Interpretation für Laufzeitfehler ist nicht genug:

- Bislang stand Verifikation des korrekten Verhaltens im Vordergrund
 - *Abstrakte Interpretation:*
Abwesenheit von Laufzeitfehlern (Sprachstandard, nicht-funktional)

- ⚠ Dies ist **notwendig**, jedoch **nicht hinreichend**
 - Einfluss nicht-funktionaler Eigenschaften der Ausführungsumgebung
 - Anwendung ist in die Umwelt eingebettet!
 - Exemplarisch: **Speicherverbrauch** und **Laufzeit**
- 👉 **Einhaltung** bestimmter *nicht-funktionaler Eigenschaften* garantieren?
 - Speicherverbrauch: *Worst-Case Stack Demand* (WCSD, siehe 6 ff)
 - Laufzeit: *Worst-Case Execution Time/Response Time* (WCET/WCRT, siehe 16 ff)
 - Messung versus statische Analyse

- 1 Speicherverbrauch
 - Überblick
 - Messbasierte Bestimmung
 - Analytische Bestimmung
 - Systemweite Stackverbrauchsanalyse
- 2 WCET- & WCRT-Analyse
 - Systemweite Kontrollflüsse
 - Implicit Path Enumeration Technique (IPET)
 - Integrierte, systemweite IPET
- 3 Zusammenfassung

- ☞ Betrachtung des Speicherverbrauchs nach Lokalität
 - *Festwertspeicher* (engl. *Read Only Memory, ROM*)
 - Umfasst die Übersetzungseinheiten (*Funktionen* und *Konstanten*)
 - **Architekturabhängig** (Wortbreite, Optimierungsstufe, Inlining, ...)
 - Größe ist dem Compiler/Linker **statisch bekannt**:

```
gcc -WL,-Map,PROGRAM.map *.o -o PROGRAM
```


- ☞ Betrachtung des Speicherverbrauchs nach Lokalität
 - *Festwertspeicher* (engl. *Read Only Memory, ROM*)
 - Umfasst die Übersetzungseinheiten (*Funktionen* und *Konstanten*)
 - **Architekturabhängig** (Wortbreite, Optimierungsstufe, Inlining, ...)
 - Größe ist dem Compiler/Linker **statisch bekannt**:

```
gcc -WL,-Map,PROGRAM.map *.o -o PROGRAM
```
 - *Direktzugriffsspeicher* (engl. *Random Access Memory, RAM*)
 - In eingebetteten Systemen typischerweise statisch allokiert (*globale Variablen* & *Stapelspeicher*-Konfiguration)
 - Permanenter Verbrauch (**architekturabhängig**) ebenso **statisch bekannt**

- ☞ Betrachtung des Speicherverbrauchs nach Lokalität
 - *Festwertspeicher* (engl. *Read Only Memory, ROM*)
 - Umfasst die Übersetzungseinheiten (*Funktionen* und *Konstanten*)
 - **Architekturabhängig** (Wortbreite, Optimierungsstufe, Inlining, ...)
 - Größe ist dem Compiler/Linker **statisch bekannt**:

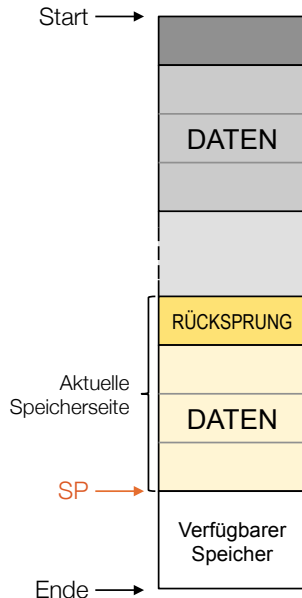
```
gcc -WL,-Map,PROGRAM.map *.o -o PROGRAM
```
 - *Direktzugriffsspeicher* (engl. *Random Access Memory, RAM*)
 - In eingebetteten Systemen typischerweise statisch allokiert (*globale Variablen* & *Stapelspeicher*-Konfiguration)
 - Permanenter Verbrauch (**architekturabhängig**) ebenso **statisch bekannt**

Dynamischer Speicher (`malloc()`) in eingebetteten Systemen

wird typischerweise auf den *Stapelspeicher* (engl. *Stack*) abgebildet

Der Stapelspeicher (Stack) Dyn. Nutzung von Speicher in eingebetteten Systemen

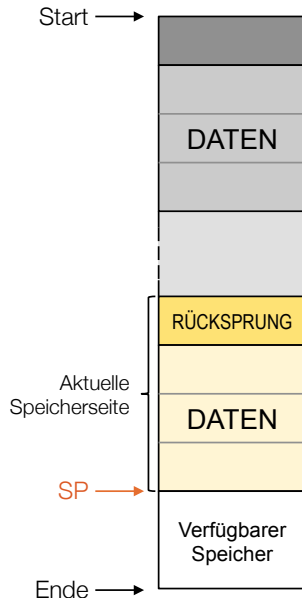
- ☞ Stapelspeicher wird verwendet für:
 - Lokale Variablen und Zwischenwerte
 - Funktionsparameter
 - Rücksprungadressen



Der Stapelspeicher (Stack) Dyn. Nutzung von Speicher in eingebetteten Systemen

- ☞ Stapelspeicher wird verwendet für:
 - Lokale Variablen und Zwischenwerte
 - Funktionsparameter
 - Rücksprungadressen

⚠ Größe wird (üblicherweise) zur *Übersetzungszeit* festgelegt



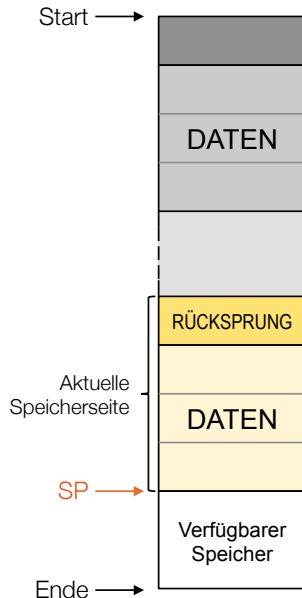
Der Stapelspeicher (Stack) Dyn. Nutzung von Speicher in eingebetteten Systemen

- ☞ Stapelspeicher wird verwendet für:
 - Lokale Variablen und Zwischenwerte
 - Funktionsparameter
 - Rücksprungadressen

⚠ Größe wird (üblicherweise) zur *Übersetzungszeit* festgelegt

Fehlerquelle Stapelspeicher

- Unterdimensionierung \leadsto **Überlauf**
- Interrupt Service Routine (ISR) auf Stack ausgeführt
- Größenbestimmung \approx Halteproblem



Problem: Maximaler Speicherverbrauch

Fallbeispiel: Stellwerk Hamburg-Altona [8]



■ *Elektronisches Stellwerk*

- Hersteller: Siemens
- Simis-3216 (i486)
- Inbetriebnahme: 12. März 1995
- Kosten: 62,6 Mio DM
- Ersetzte 8 Stellwerke (1911-52)

Problem: Maximaler Speicherverbrauch

Fallbeispiel: Stellwerk Hamburg-Altona [8]



■ *Elektronisches Stellwerk*

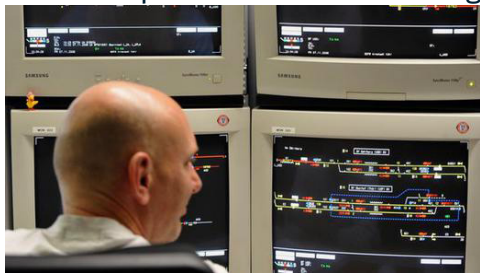
- Hersteller: Siemens
- Simis-3216 (i486)
- Inbetriebnahme: 12. März 1995
- Kosten: 62,6 Mio DM
- Ersetzte 8 Stellwerke (1911-52)

⚠ Dynamische Verwaltung der Stellbefehle auf dem Stapelspeicher

- Initial 3.5 KiB \leadsto **zu klein** schon für normalen Verkehr
- Fehlerbehandlungsroutine fehlerhaft (nicht getestet) \leadsto **Endlosschleife**
- Notabschaltung durch Sicherungsmaßnahmen (fail-stop)

Problem: Maximaler Speicherverbrauch

Fallbeispiel: Stellwerk Hamburg-Altona [8]



■ Elektronisches Stellwerk

- Hersteller: Siemens
- Simis-3216 (i486)
- Inbetriebnahme: 12. März 1995
- Kosten: 62,6 Mio DM
- Ersetzte 8 Stellwerke (1911-52)

⚠ Dynamische Verwaltung der Stellbefehle auf dem Stapelspeicher

- Initial 3.5 KiB \leadsto zu klein schon für normalen Verkehr
- Fehlerbehandlungsroutine fehlerhaft (nicht getestet) \leadsto Endlosschleife
- Notabschaltung durch Sicherungsmaßnahmen (fail-stop)

Ausfall am Tag der Inbetriebnahme

Kein Schienenverkehr für **2 Tage**, 2 Monate Notfahrplan

Bestimmung des Stapelspeicherverbrauchs

☞ Überabschätzung führt zu unnötigen Kosten

Bestimmung des Stapelspeicherverbrauchs

- ☞ **Überabschätzung** führt zu **unnötigen Kosten**
- ⚠ **Unterabschätzung** des Speicherverbrauchs führt zu **Stapelüberlauf**
 - Schwerwiegendes und komplexes Fehlermuster
 - undefiniertes Verhalten, **Datenfehler** oder Programmabsturz→ Schwer zu finden, reproduzieren und beheben!

Bestimmung des Stapelspeicherverbrauchs

- ☞ **Überabschätzung** führt zu **unnötigen Kosten**
- ⚠ **Unterabschätzung** des Speicherverbrauchs führt zu **Stapelüberlauf**
 - Schwerwiegendes und komplexes Fehlermuster
 - undefiniertes Verhalten, **Datenfehler** oder Programmabsturz
 - Schwer zu finden, reproduzieren und beheben!
- Voraussetzungen für sinnvolle Analyse
 - Zyklische Ausführungspfade vermeiden
 - Keine **Rekursion**, **Funktionszeiger**, **dynamischer Speicher**

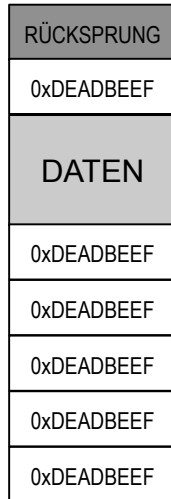
Bestimmung des Stapelspeicherverbrauchs

- ☞ **Überabschätzung** führt zu **unnötigen Kosten**
- ⚠ **Unterabschätzung** des Speicherverbrauchs führt zu **Stapelüberlauf**
 - Schwerwiegendes und komplexes Fehlermuster
 - undefiniertes Verhalten, **Datenfehler** oder Programmabsturz
 - Schwer zu finden, reproduzieren und beheben!
- Voraussetzungen für sinnvolle Analyse
 - Zyklische Ausführungspfade vermeiden
 - Keine **Rekursion**, **Funktionszeiger**, **dynamischer Speicher**
- ⚠ Analyse gängiger Compiler
 - `gcc -fstack-usage` ist **nicht genug**
 - Richtwert bei der Entwicklung einzelner Funktionen

Messung des Stapelspeicherverbrauchs

Analog zum dynamischen Testen (siehe Folie VII/18 ff.):

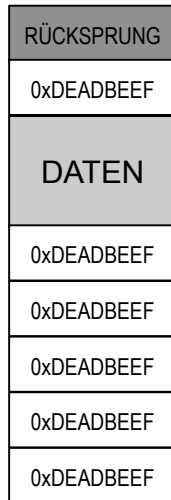
- *Messung* (Water-Marking, Stack Canaries)
 - Stapelspeicher wird *vorinitialisiert* (z.B. 0xDEADBEEF)
 - Maximaler Verbrauch **der Ausführung**
 - ↪ höchste Speicherstelle ohne Wasserzeichen
 - Auf Rücksprungadressen anwendbar



Messung des Stapelspeicherverbrauchs

Analog zum dynamischen Testen (siehe Folie VII/18 ff.):

- *Messung* (Water-Marking, Stack Canaries)
 - Stapelspeicher wird *vorinitialisiert* (z.B. 0xDEADBEEF)
 - Maximaler Verbrauch **der Ausführung**
 - ↪ höchste Speicherstelle ohne Wasserzeichen
 - Auf Rücksprungadressen anwendbar
- Systemüberwachung zur Laufzeit
 - Verfahren gut geeignet zur dynamischen Fehlererkennung
 - *Stack Check* in AUTOSAR, OSEK, ...



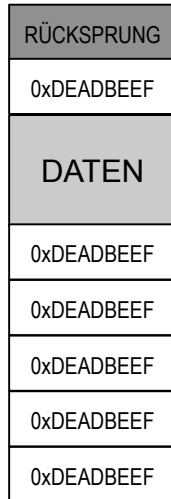
Messung des Stapelspeicherverbrauchs

Analog zum dynamischen Testen (siehe Folie VII/18 ff.):


- *Messung* (Water-Marking, Stack Canaries)
 - Stapelspeicher wird *vorinitialisiert* (z.B. 0xDEADBEEF)
 - Maximaler Verbrauch **der Ausführung**
 - ↪ höchste Speicherstelle ohne Wasserzeichen
 - Auf Rücksprungadressen anwendbar
- Systemüberwachung zur Laufzeit
 - Verfahren gut geeignet zur dynamischen Fehlererkennung
 - *Stack Check* in AUTOSAR, OSEK, ...

⚠ Keine Aussagen zum maximalen Speicherverbrauch

- Liefert nur den konkreten Verbrauch der Messungen
- Fehleranfällig und aufwendig
- **Keine Garantien möglich!**



```
1 unsigned int function(unsigned char a, unsigned char b) {  
2     unsigned int c;  
3     unsigned char d;  
4     /* code */  
5     return c;  
6 }
```

 Ausführungsbedingungen bestimmen tatsächlichen Speicherbedarf


```
1 unsigned int function(unsigned char a, unsigned char b) {  
2     unsigned int c;  
3     unsigned char d;  
4     /* code */  
5     return c;  
6 }
```

 Ausführungsbedingungen bestimmen tatsächlichen Speicherbedarf

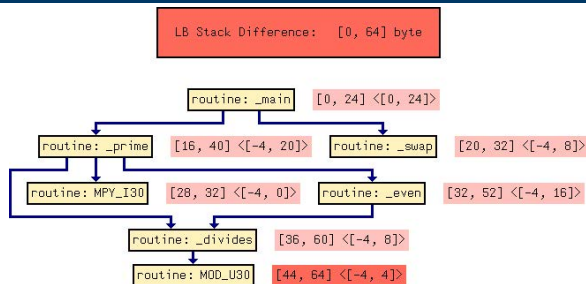
- *Speicherausrichtung* (engl. *alignment*) von Variablen und Parametern
 - Abhängig von *Binärschnittstelle* (engl. *Application Binary Interface, ABI*)
 - In diesem Beispiel 16 Byte (und mehr)

```
1 unsigned int function(unsigned char a, unsigned char b) {  
2     unsigned int c;  
3     unsigned char d;  
4     /* code */  
5     return c;  
6 }
```

 Ausführungsbedingungen bestimmen tatsächlichen Speicherbedarf

- *Speicherausrichtung* (engl. *alignment*) von Variablen und Parametern
 - Abhängig von *Binärschnittstelle* (engl. *Application Binary Interface, ABI*)
 - In diesem Beispiel 16 Byte (und mehr)
- Inline-Ersetzung der Funktion (kein Stapelverbrauch für Aufruf)

Bestimmung des maximalen Stapelspeicherverbrauchs



Durch abstrakte Interpretation des Programmcodes [3]:

- Statische Analyse des **Aufrufgraphen**
 - Pufferüberlauf als weitere Form von Laufzeitfehlern
 - Vorgehen analog zum Korrektheitsnachweis
- Weist **Abwesenheit** von Pufferüberläufen nach
 - Ausrollen von Schleifen
 - Pfadanalyse \leadsto maximaler WCSD
 - **allerdings noch ohne Berücksichtigung von Interrupts ...**

Systemweite Stackverbrauchsanalyse

- ⚠ Problem: systemweiter Stackverbrauchsanalyse mit Interrupts
 - Eigenschaften der Hardwarearchitektur
 - Interrupts werden auf *aktuellem Stack* ausgeführt
 - **Verschachtelte (engl. nested) Interrupts** je nach Architektur möglich

Systemweite Stackverbrauchsanalyse

- ⚠ Problem: systemweiter Stackverbrauchsanalyse mit Interrupts
 - Eigenschaften der Hardwarearchitektur
 - Interrupts werden auf *aktuellem Stack* ausgeführt
 - **Verschachtelte (engl. nested) Interrupts** je nach Architektur möglich
 - Beispiel (8-Bit AVR):

```
static uint32_t events = 0;
ISR(int1){ events++ } // Interrupt Service Routine (ISR) für Interrupt 1
ISR(int2){ ... }
ISR(int3){ ... }

int main(int a){
  while(true) {
    if( a > func0() ){ // datenflussabhängiger Kontrollfluss
      EIMSK &= ~(1 << INT1); // Interrupt 1 deaktivieren
      // nicht-atomarer Zugriff auf Variable events: hier Abarbeitung von Interrupt 1 verhindern
      process_events();
      EIMSK |= (1 << INT1); // Interrupt 1 aktivieren
    }
  }
}
```

Systemweite Stackverbrauchsanalyse

- ⚠ Problem: systemweiter Stackverbrauchsanalyse mit Interrupts
 - Eigenschaften der Hardwarearchitektur
 - Interrupts werden auf *aktuellem Stack* ausgeführt
 - **Verschachtelte (engl. nested) Interrupts** je nach Architektur möglich

- Beispiel (8-Bit AVR):

```
static uint32_t events = 0;
ISR(int1){ events++ } // Interrupt Service Routine (ISR) für Interrupt 1
ISR(int2){ ... }
ISR(int3){ ... }

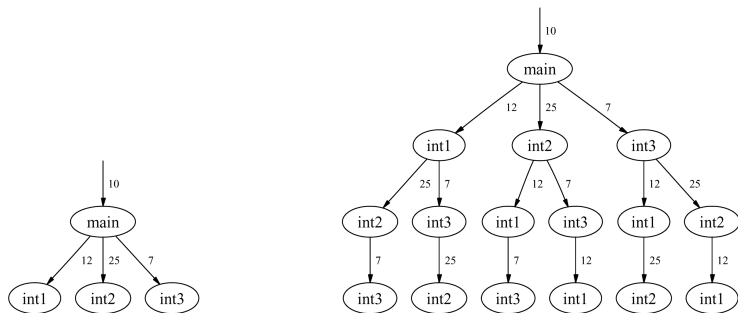
int main(int a){
    while(true) {
        if( a > func0() ){ // datenflussabhängiger Kontrollfluss
            EIMSK &= ~(1 << INT1); // Interrupt 1 deaktivieren
            // nicht-atomarer Zugriff auf Variable events: hier Abarbeitung von Interrupt 1 verhindern
            process_events();
            EIMSK |= (1 << INT1); // Interrupt 1 aktivieren
        }
    }
}
```

- **Wird `process_events()` von Interrupt 1 hier unterbrochen?**
- **Kann Speicherbedarf von `process_events()` und Interrupt 1 hier gleichzeitig vorhanden sein auf dem Stack?**

Abstrakte Interpretation mittels Interrupt-Preemption-Graph

- Lösung
 - abstrakte Interpretation von Systemzuständen zur Berücksichtigung von Interrupts mittels **Interrupt-Preemption-Graph** (Regehr et al. [7])
- Bestandteile der abstrakten Domäne/Zustände des Interrupt-Preemption-Graphs
 1. Register der Interrupt-Maskierung
 2. Status-Register
 3. Stackpointer
- Semantische Unterschiede zwischen WCRT und WCSD bei Interrupts
- Abstrakte Interpretation hat Kenntnis über Operationen auf Bitebene (hier: $1 \ll INT1$)
 - ↪ Abschätzung der Registerinhalten für pfadsensitive Interrupt-Maskierung (hier: EIMSK)

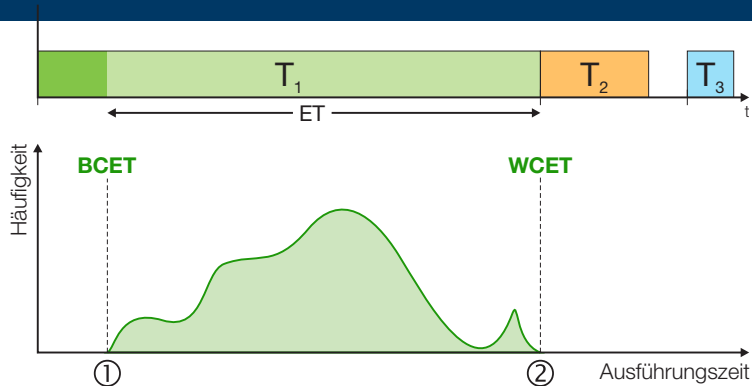
Interrupt-Preemption-Graph: Verschachtelte Interrupts



- Semantik ohne (links) und mit (rechts) verschachtelten Interrupts [7]
- Wissen über Datenfluss \leadsto Vermeidung von Interrupt-Kanten im Graph möglich möglich
 - Bei Ausführung von `process_events()` kann zusätzlicher Interrupt ausgeschlossen werden \leadsto **reduziert Analysepessimismus**
- Traversierung des Interrupt-Preemption-Graph \leadsto *WCSD*

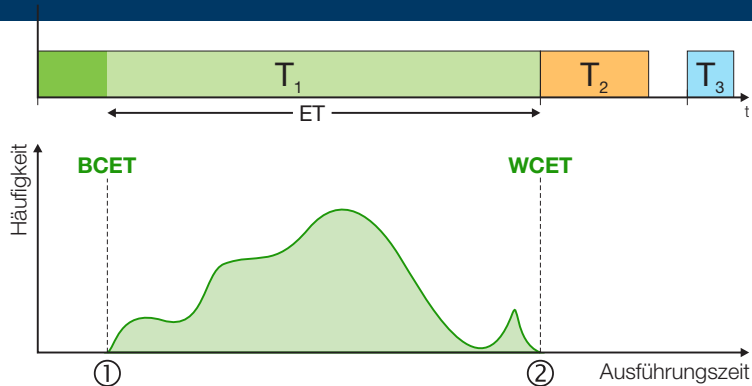
- 1 Speicherverbrauch
 - Überblick
 - Messbasierte Bestimmung
 - Analytische Bestimmung
 - Systemweite Stackverbrauchsanalyse
- 2 WCET- & WCRT-Analyse
 - Systemweite Kontrollflüsse
 - Implicit Path Enumeration Technique (IPET)
 - Integrierte, systemweite IPET
- 3 Zusammenfassung

Die maximale Ausführungszeit

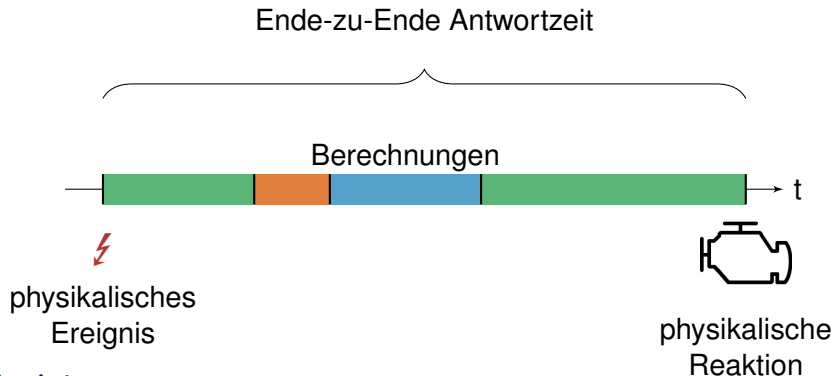


- *Maximale Ausführungszeit*
 - Worst-Case Execution Time (WCET)
 - Unabdingbares Maß für *zulässigen Ablaufplan*
- Tatsächliche Ausführungszeit zwischen BCET und WCET

Die maximale Ausführungszeit



- *Maximale Ausführungszeit*
 - Worst-Case Execution Time ($WCET$)
 - Unabdingbares Maß für *zulässigen Ablaufplan*
- Tatsächliche Ausführungszeit zwischen $BCET$ und $WCET$
- Viele Systeme (z.B. OSEK) erlauben **Ereignisse & Verdrängungen**
 - ☞ Analyse der **Worst-Case Response Time** notwendig

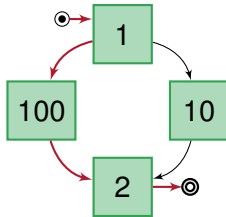


■ Beispiel

- Ereignis: Drücken eines Tasters
- Ergebnis: Setzen von Stellwert eines Ventils

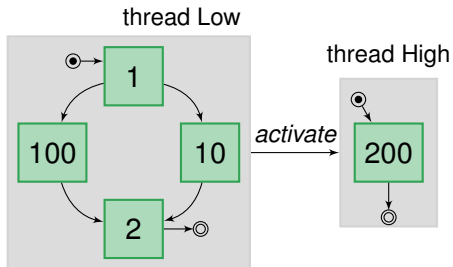
■ *Verdrängungen* von Aufgaben mit höherer Priorität

Beispiel: WCRT



Worst-Case Response Time: 103 Zyklen

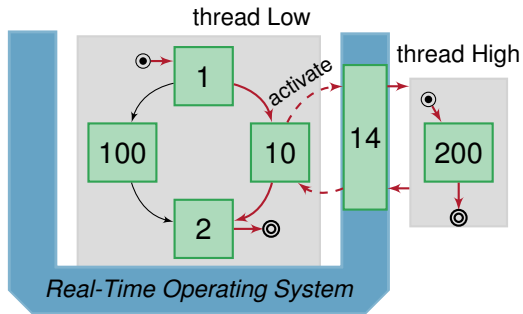
Beispiel: WCRT



Worst-Case Response Time: $103 + 200 + t(\text{RTOS})$ Zyklen?

- Real-Time Operating System (RTOS)

Beispiel: WCRT



Worst-Case Response Time: 241 Zyklen

Beispiel: WCRT

ISR

thread Low

Lokale WCET-Analyse stoppt an der Systemaufrufchnittstelle.

Threads und RTOS interagieren miteinander.

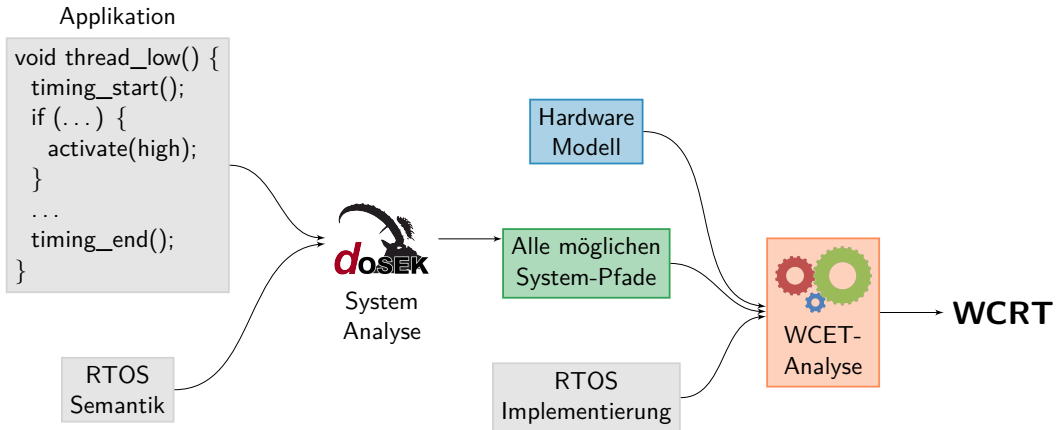
Threads sind sehr oft **nicht unabhängig**.

Worst-Case Response Time: 445 Zyklen

Probleme mit zusammengesetzter Antwortzeitanalyse

- Die verbreiteten Ansätze arbeiten **kompositionell**
 - WCET wird für jede Komponente (z.B. Systemaufruf) **pessimistisch** in Isolation bestimmt
 - Aggregation von WCETs anhand ihrer *möglichen Interaktionen*
 - Jede WCET für sich muss pessimistisch sein um Korrektheit (engl. soundness) zu bieten
 - Immer den **längsten Pfad** in jedem Thread
 - Immer die **längste Dauer** durch den Kern beim Systemaufruf
- ⇒ *Integrierte Formulierung* einer systemweiten WCRT-Analyse
- Erfassen der Maschienenebene und der Scheduling-Analyse
 - RTOS-Verhalten und Umweltmodell müssen integriert werden
 - Formulierung thread-übergreifender Flussfakten (z.B. gegenseitiger Ausschluss)

SysWCET Werkzeugkette im Überblick [2]



C. Dietrich, P. Wagemann, P. Ulbrich, D. Lohmann

SysWCET: Whole-System Response-Time Analysis for Fixed-Priority Real-Time Systems

23rd Real-Time and Embedded Technology and Applications Symposium (RTAS '17)

- 1 Speicherverbrauch
- 2 WCET- & WCRT-Analyse
 - Systemweite Kontrollflüsse
 - Implicit Path Enumeration Technique (IPET)
 - Integrierte, systemweite IPET
- 3 Zusammenfassung

- Systemmodell
 - Ereignisgesteuertes Ausführungsmodell: Threads, ISRs, etc.
 - Fixed-priority Scheduling Semantik
 - Kenntnis über das statisches Systemwissen
 - Kernobjekte (Thread, Locks, periodische Signale) und ihre Konfiguration
 - Anwendungsstruktur (Kontrollflussgraph) und Systemaufrufe (Ort, Argumente)

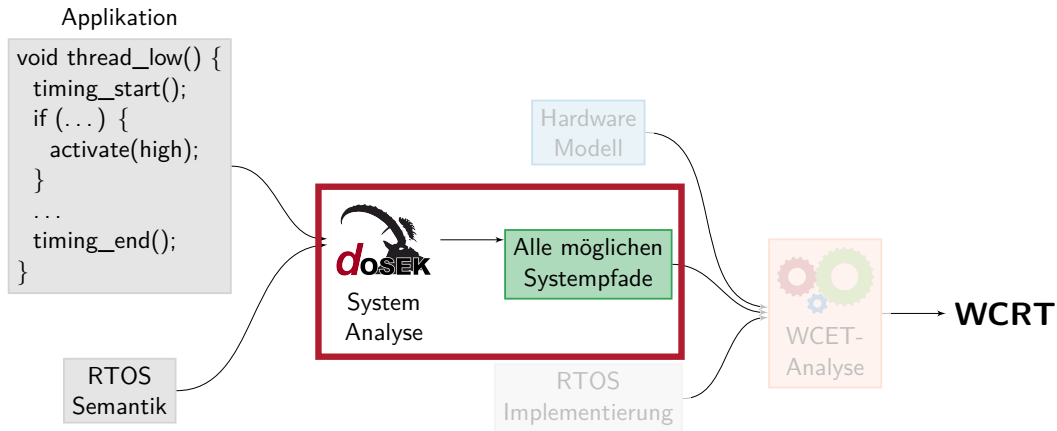
■ Systemmodell

- Ereignisgesteuertes Ausführungsmodell: Threads, ISRs, etc.
- Fixed-priority Scheduling Semantik
- Kenntnis über das statisches Systemwissen
 - Kernobjekte (Thread, Locks, periodische Signale) und ihre Konfiguration
 - Anwendungsstruktur (Kontrollflussgraph) und Systemaufrufe (Ort, Argumente)



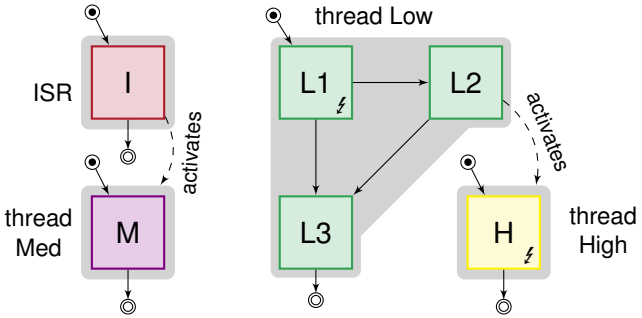
- Systeme die unserem Systemmodell entsprechen: OSEK, AUTOSAR
 - Industriestandard aus der Automobilbranche
 - *Statische Konfiguration* zur Übersetzungszeit
 - Fixed-priority Scheduling mit Threads und ISRs
 - Stapelbasiertes Priority-Ceiling-Protokoll (PCP) für Locks

Bestimmung systemweiter Programmpfade

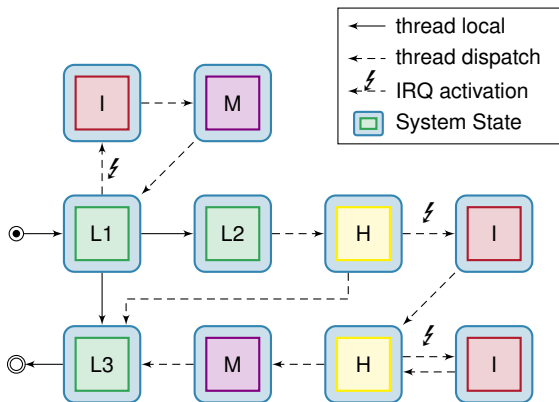
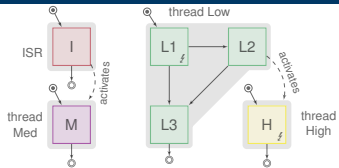


- Bestimmung aller möglichen Programmpfade des Gesamtsystems
- Berücksichtigung von
 1. Applikationscode
 2. RTOS Semantik

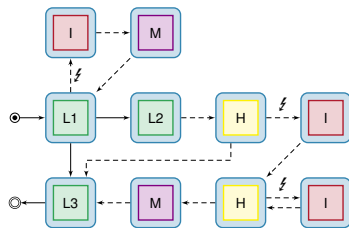
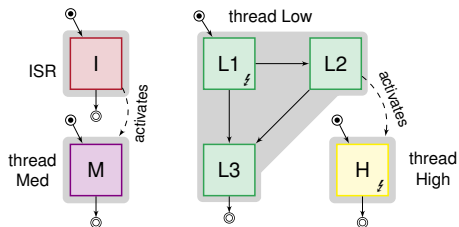
Operating-System State Transition Graph [1]



Operating-System State Transition Graph [1]

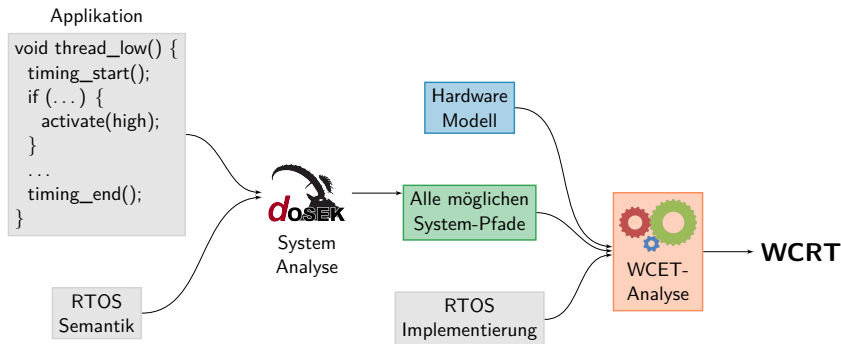


Operating-System State Transition Graph [1]



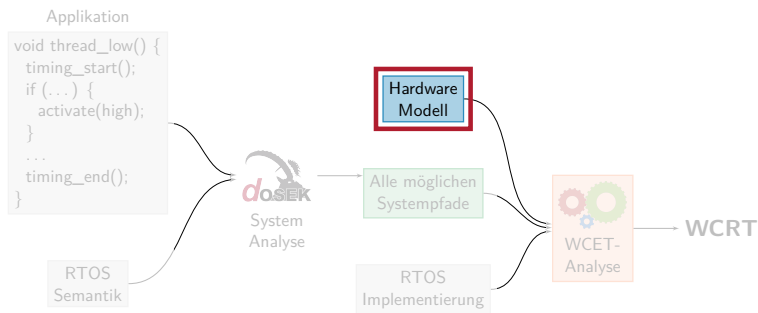
- Explizite Aufzählung aller Systemzustände
- Betriebssystemkern ↔ Anwendung ↔ Umgebungsmodell
- Beinhaltet: Prioritäten, Interruptaktivierungen, Lockprotokolle, ISR-Verdrängung
- Mit diesen Graphen können wir umgehen ...

- 1 Speicherverbrauch
- 2 WCET- & WCRT-Analyse
 - Systemweite Kontrollflüsse
 - **Implicit Path Enumeration Technique (IPET)**
 - Integrierte, systemweite IPET
- 3 Zusammenfassung



■ Bestandteile der WCET-Analyse (IPET)

1. Pfad-Analyse: Hardware-unabhängige Analyse
2. Hardware-Analyse: abhängig von Zielplattform



■ Bestandteile der WCET-Analyse (IPET)

1. Pfad-Analyse: Hardware-unabhängige Analyse
2. Hardware-Analyse: abhängig von Zielplattform

- ☞ Ausführungszeit von Maschineninstruktionen ist essentiell

- Beispiel

```
/* x = a + b */  
LOAD r2, _a  
LOAD r1, _b  
ADD r3, r2, r1
```

- ☞ Ausführungszeit von Maschineninstruktionen ist essentiell

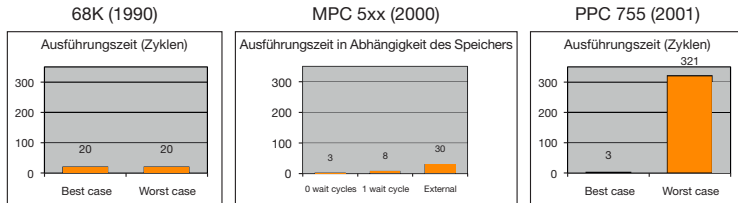
- Beispiel

```
/* x = a + b */
```

```
LOAD r2, _a
```

```
LOAD r1, _b
```

```
ADD r3, r2, r1
```

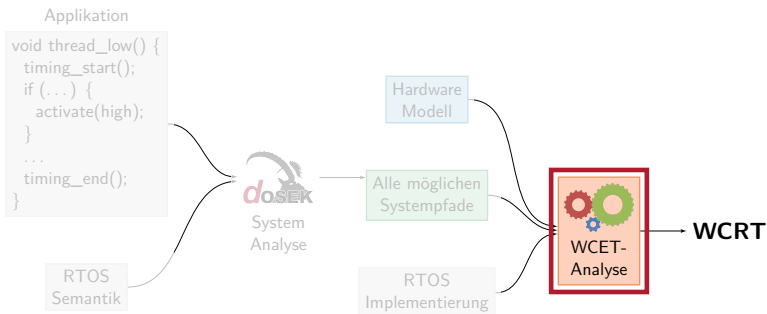


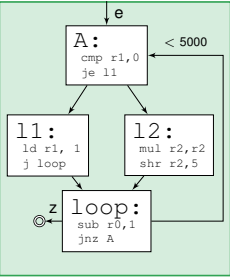
Quelle: C. Ferdinand [4]

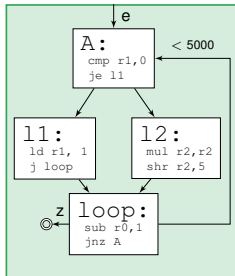
- Hardware-Analyse

- Umfangreich untersucht: Standard-Techniken
- Ergebnis: WCET für Sequenzen (Basisblöcke)

WCET-Analyse mittels Impliziter Pfadaufzählung

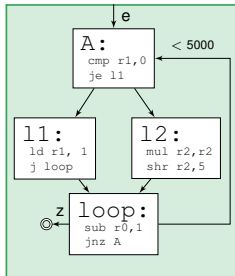






Problem: Erfordert explizite Aufzählung aller Pfade

- Auf Ebene der Systemzustände ist explizite Aufzählung handhabbar ✓
- Auf Basisblock-Ebene algorithmisch nicht möglich (Beispiel: 2^{5000})

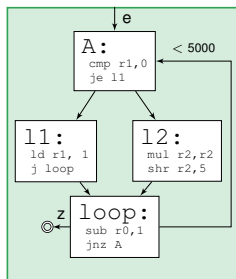


Problem: Erfordert explizite Aufzählung aller Pfade

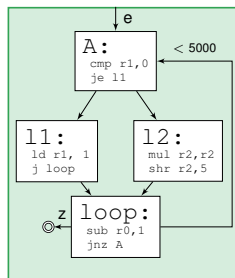
- Auf Ebene der Systemzustände ist explizite Aufzählung handhabbar ✓
- Auf Basisblock-Ebene algorithmisch nicht möglich (Beispiel: 2^{5000})

Lösung: Vereinfachung der konkreten Pfadsemantik

- Abstraktion und Abbildung auf ein Flussproblem/Integer Linear Program (ILP)
- Flussprobleme sind mathematisch gut untersucht
- **Implicit Path Enumeration Technique (IPET)** [5, 6]



- **Flusserhaltung** für Basisblöcke (eingehender = ausgehender Fluss)
- **Nebenbedingungen** für Variablen aus Kontrollfluss ableitbar, z.B.:
 1. $A = e + \text{loop}$
 2. $A = l1 + l2$
- **Schleifeniteration** benötigen obere Schranke
 4. $A < 5000 + e$
- Einmalige Analyse des Fragments
 5. $e = 1; z = 1$



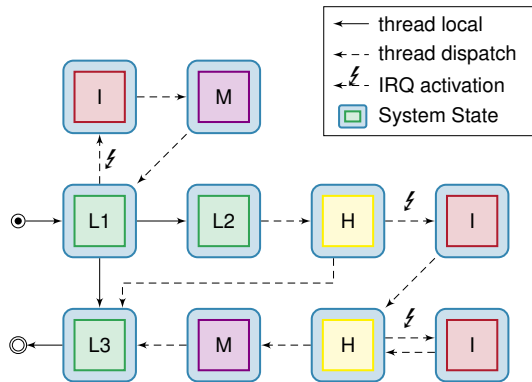
■ **Kosten der Blöcke** aus Hardware-Analyse, z. B.

- A: 4 Zyklen
- l1: 2 Zyklen

■ Zielfunktion (unter Einhaltung aller Nebenbedingungen)

- Maximiere Summe von (Ausführungshäufigkeit · Kosten) aller Blöcke
- $\max: A \cdot 4 + l1 \cdot 2 + \dots$
- Lösung der Zielfunktion mittels mathematischer Optimierungsprogramme (lp_solve, Gurobi)

- 1 Speicherverbrauch
- 2 WCET- & WCRT-Analyse
 - Systemweite Kontrollflüsse
 - Implicit Path Enumeration Technique (IPET)
 - Integrierte, systemweite IPET
- 3 Zusammenfassung



■ SysWCET Idee kurz zusammengefasst

1. IPET für den State Transition Graph: *Zustandshäufigkeit*
2. Ausserdem: ein IPET Fragment für jeden Programmblock
3. Ableitung einer Blockhäufigkeit aus der Zustandshäufigkeit

Geschichtete IPET Kontruktion

■ Systemschicht

- IPET-Variablen für Zustände & Zustandsübergänge
- Wie häufig ist das System in S_1 im Fall der WCRT?

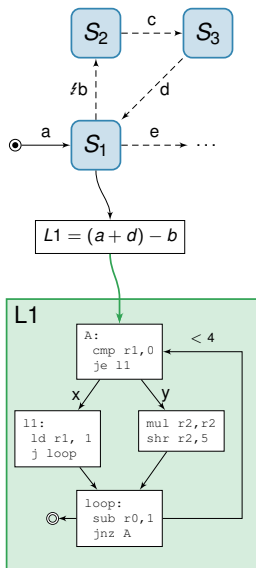
■ Leimschicht

- Blockhäufigkeiten aus Zustandshäufigkeiten

■ Maschinenschicht

- Ein IPET-Fragment aus jedem Codeblock
- RTOS Code wird gleich behandelt
- Blockhäufigkeiten aktivieren das Fragment
- Eine IPET-Formulierung: ermöglicht Flussfakten innerhalb und zwischen Codeblöcken

👉 Lösung der Formulierung \rightsquigarrow WCRT



- 1 Speicherverbrauch
 - Überblick
 - Messbasierte Bestimmung
 - Analytische Bestimmung
 - Systemweite Stackverbrauchsanalyse
- 2 WCET- & WCRT-Analyse
 - Systemweite Kontrollflüsse
 - Implicit Path Enumeration Technique (IPET)
 - Integrierte, systemweite IPET
- 3 Zusammenfassung

- *Dynamische Messung* \rightsquigarrow Beobachtung
 - Speicherverbrauch
 - Water-Marking \rightsquigarrow Füllstand des statischen Stapels zur Laufzeit
 - Semantik der Interaktion von Aufgaben/Betriebssystem relevant
 - Ausführungszeit
 - Durch (strukturiertes) Testen der Echtzeitanwendung

- *Dynamische Messung* \leadsto Beobachtung
 - Speicherverbrauch
 - Water-Marking \leadsto Füllstand des statischen Stapels zur Laufzeit
 - Semantik der Interaktion von Aufgaben/Betriebssystem relevant
 - Ausführungszeit
 - Durch (strukturiertes) Testen der Echtzeitanwendung
- *Statische Analyse* \leadsto Bestimmung einer **oberen Schranke**
 - Speicherverbrauch
 - Analyse des Kontroll- und Aufrufgraphen
 - Systemweite Analyse: Interrupt-Preemption-Graph
 - Ausführungszeit
 - *Makroskopisch: Was macht das Programm?*
 - *Mikroskopisch: Was passiert in der Hardware?*
 - Semantik der Interaktion von Aufgaben/Betriebssystem relevant

- [1] Dietrich, C. ; Hoffmann, M. ; Lohmann, D. :
Global Optimization of Fixed-Priority Real-Time Systems by RTOS-Aware Control-Flow Analysis.
In: *ACM Transactions on Embedded Computing Systems (TECS)* 16 (2017),
S. 35:1–35:25
- [2] Dietrich, C. ; Wägemann, P. ; Ulbrich, P. ; Lohmann, D. :
SysWCET: Whole-System Response-Time Analysis for Fixed-Priority Real-Time Systems.
In: *Proceedings of the 23rd Real-Time and Embedded Technology and Applications Symposium (RTAS '17)*), 37–48

- [3] Ferdinand, C. ; Heckmann, R. ; Franzen, B. :
Static memory and timing analysis of embedded systems code.
In: *Proceedings of the 3rd European Symposium on Verification and Validation of Software Systems*, 2007, S. 07–04
- [4] Ferdinand, C. ; Heckmann, R. ; Wolff, H.-J. ; Renz, C. ; Parshin, O. ; Wilhelm, R. :
Towards model-driven development of hard real-time systems.
In: *Model-Driven Development of Reliable Automotive Services*.
Springer, 2008, S. 145–160

[5] Li, Y.-T. S. ; Malik, S. :

Performance Analysis of Embedded Software Using Implicit Path Enumeration.

In: *ACM SIGPLAN Notices* Bd. 30 ACM, 1995, S. 88–98

[6] Puschner, P. ; Schedl, A. :

Computing Maximum Task Execution Times: A Graph-Based Approach.

In: *Real-Time Systems* 13 (1997), S. 67–91

[7] Regehr, J. ; Reid, A. ; Webb, K. :

Eliminating Stack Overflow by Abstract Interpretation.

In: *ACM Transactions on Embedded Computing Systems* 4 (2005), Nr. 4, S. 751–778.

<http://dx.doi.org/10.1145/1113830.1113833>. –

DOI 10.1145/1113830.1113833. –

ISSN 1539-9087

[8] Weber-Wulff, D. :

More on German Train Problems.

<http://catless.ncl.ac.uk/Risks/17.02.html>.

Version: 04 1995