Robuste Komponentensysteme durch Protokollprüfung

Wolf Zimmermann und Andreas Both

Martin-Luther-Universität Halle-Wittenberg

Workshop Protokoll-basierte Modellierung von Geschäftsinteraktionen
Informatik 2011
6. Oktober 2011

1. Einleitung

Komponentenbasierte Softwareentwicklung

Beobachtung: Zusammenstecken von Komponenten/Services ist (noch?) keine Routinetätigkeit

Symptome:

- Unerwartetes Verhalten von Komponenten/Services

Mögliche Ursachen

1. Einleitung

Komponentenbasierte Softwareentwicklung

Beobachtung: Zusammenstecken von Komponenten/Services ist (noch?) keine Routinetätigkeit

Symptome:

- Unerwartetes Verhalten von Komponenten/Services
- Verklemmungen wegen Nebenläufigkeit

Mögliche Ursachen

1. Einleitung

Komponentenbasierte Softwareentwicklung

Beobachtung: Zusammenstecken von Komponenten/Services ist (noch?) keine Routinetätigkeit

Symptome:

- Unerwartetes Verhalten von Komponenten/Services
- Verklemmungen wegen Nebenläufigkeit
- Unerwartete Abstürze in Komponenten
- Verletzung von Richtlinien (*Compliance*)
- Unerwartet schlechte Servicequalität

Mögliche Ursachen

1. Einleitung

Komponentenbasierte Softwareentwicklung

Beobachtung: Zusammenstecken von Komponenten/Services ist (noch?) keine Routinetätigkeit

Symptome:

- Unerwartetes Verhalten von Komponenten/Services
- Verklemmungen wegen Nebenläufigkeit
- Unerwartete Abstürze in Komponenten
- Verletzung von Richtlinien (*Compliance*)
- Unerwartet schlechte Servicequalität

Mögliche Ursachen

1. Einleitung

Komponentenbasierte Softwareentwicklung

Beobachtung: Zusammenstecken von Komponenten/Services ist (noch?) keine Routinetätigkeit

Symptome:

- Unerwartetes Verhalten von Komponenten/Services
- Verklemmungen wegen Nebenläufigkeit
- Unerwartete Abstürze in Komponenten
- Verletzung von Richtlinien (*Compliance*)
- Unerwartet schlechte Servicequalität

Mögliche Ursachen

1. Einleitung

Komponentenbasierte Softwareentwicklung

Beobachtung: Zusammenstecken von Komponenten/Services ist (noch?) keine Routinetätigkeit

Symptome:

- Unerwartetes Verhalten von Komponenten/Services
- Verklemmungen wegen Nebenläufigkeit
- Unerwartete Abstürze in Komponenten
- Verletzung von Richtlinien (*Compliance*)
- Unerwartet schlechte Servicequalität

Mögliche Ursachen

- Schnittstellen werden nicht eingehalten
- Synchronisation wird nicht eingehalten
- Komponenten nicht mehr zur Laufzeit
- Versionsinkompatibilitäten zwischen Komponenten

1. Einleitung

Komponentenbasierte Softwareentwicklung

Beobachtung: Zusammenstecken von Komponenten/Services ist (noch?) keine Routinetätigkeit

Symptome:

- Unerwartetes Verhalten von Komponenten/Services
- Verklemmungen wegen Nebenläufigkeit
- Unerwartete Abstürze in Komponenten
- Verletzung von Richtlinien (*Compliance*)
- Unerwartet schlechte Servicequalität

Mögliche Ursachen

- Schnittstellen werden ausschließlich über Signaturen beschrieben
- Komponenten/Services werden ausschließlich über Schnittstellen beschrieben
- Komponenten/Services werden ausschließlich über Schnittstellen und Signaturen beschrieben

1. Einleitung

Komponentenbasierte Softwareentwicklung

Beobachtung: Zusammenstecken von Komponenten/Services ist (noch?) keine Routinetätigkeit

Symptome:

- Unerwartetes Verhalten von Komponenten/Services
- Verklemmungen wegen Nebenläufigkeit
- Unerwartete Abstürze in Komponenten
- Verletzung von Richtlinien (*Compliance*)
- Unerwartet schlechte Servicequalität

Mögliche Ursachen

- Schnittstellen werden ausschließlich über Signaturen beschrieben

⇒ Syntaktische statt semantische Komposition

- Komponenten nicht zur Laufzeit ersetzbar

⇒ Keine Anpassung an geänderte Umgebung

1. Einleitung

Komponentenbasierte Softwareentwicklung

Beobachtung: Zusammenstecken von Komponenten/Services ist (noch?) keine Routinetätigkeit

Symptome:

- Unerwartetes Verhalten von Komponenten/Services
- Verklemmungen wegen Nebenläufigkeit
- Unerwartete Abstürze in Komponenten
- Verletzung von Richtlinien (*Compliance*)
- Unerwartet schlechte Servicequalität

Mögliche Ursachen

- Schnittstellen werden ausschließlich über Signaturen beschrieben

⇒ Syntaktische statt semantische Komposition

- Komponenten häufig zustandsbehaftet

⇒ Ausführungsreihenfolge der Dienste kann Fehler verursachen

1. Einleitung

Komponentenbasierte Softwareentwicklung

Beobachtung: Zusammenstecken von Komponenten/Services ist (noch?) keine Routinetätigkeit

Symptome:

- Unerwartetes Verhalten von Komponenten/Services
- Verklemmungen wegen Nebenläufigkeit
- Unerwartete Abstürze in Komponenten
- Verletzung von Richtlinien (*Compliance*)
- Unerwartet schlechte Servicequalität

Mögliche Ursachen

- Schnittstellen werden ausschließlich über Signaturen beschrieben

⇒ Syntaktische statt semantische Komposition

- Komponenten häufig zustandsbehaftet

⇒ Ausführungsreihenfolge der Dienste kann Fehler verursachen

1. Einleitung

Komponentenbasierte Softwareentwicklung

Beobachtung: Zusammenstecken von Komponenten/Services ist (noch?) keine Routinetätigkeit

Symptome:

- Unerwartetes Verhalten von Komponenten/Services
- Verklemmungen wegen Nebenläufigkeit
- Unerwartete Abstürze in Komponenten
- Verletzung von Richtlinien (*Compliance*)
- Unerwartet schlechte Servicequalität

Mögliche Ursachen

- Schnittstellen werden ausschließlich über Signaturen beschrieben

⇒ Syntaktische statt semantische Komposition

- Komponenten häufig zustandsbehaftet

⇒ Ausführungsreihenfolge der Dienste kann Fehler verursachen

1. Einleitung

Komponentenbasierte Softwareentwicklung

Beobachtung: Zusammenstecken von Komponenten/Services ist (noch?) keine Routinetätigkeit

Symptome:

- Unerwartetes Verhalten von Komponenten/Services
- Verklemmungen wegen Nebenläufigkeit
- Unerwartete Abstürze in Komponenten
- Verletzung von Richtlinien (*Compliance*)
- Unerwartet schlechte Servicequalität

Mögliche Ursachen

- Schnittstellen werden ausschließlich über Signaturen beschrieben

⇒ Syntaktische statt semantische Komposition

- Komponenten häufig zustandsbehaftet

⇒ Ausführungsreihenfolge der Dienste kann Fehler verursachen

1. Einleitung

Komponentenbasierte Softwareentwicklung

Beobachtung: Zusammenstecken von Komponenten/Services ist (noch?) keine Routinetätigkeit

Symptome:

- Unerwartetes Verhalten von Komponenten/Services
- Verklemmungen wegen Nebenläufigkeit
- Unerwartete Abstürze in Komponenten
- Verletzung von Richtlinien (*Compliance*)
- Unerwartet schlechte Servicequalität

Mögliche Ursachen

- Schnittstellen werden ausschließlich über Signaturen beschrieben

$\Rightarrow$ Syntaktische statt semantische Komposition

- Komponenten häufig zustandsbehaftet

$\Rightarrow$ Ausführungsreihenfolge der Dienste kann Fehler verursachen

Diskussion

Mögliche Abhilfe

Formale Spezifikation: Verträge für Komponenten

- Transport in die Praxis?
- Vollautomatische Werkzeugunterstützung?
- Stabilität der Verträge bei Pflege, Änderung des Systems?
- Nicht-funktionale Eigenschaften, Compliance?

Kritischer Blick auf Verträge am Beispiel von Web-Services

Fazit

Verträge sind hilfreich, reichen aber nicht aus, um Kompositionsprobleme zu lösen.

Diskussion

Mögliche Abhilfe

Formale Spezifikation: Verträge für Komponenten

- Transport in die Praxis?
- Vollautomatische Werkzeugunterstützung?
- Stabilität der Verträge bei Pflege, Änderung des Systems?
- Nicht-funktionale Eigenschaften, Compliance?

Kritischer Blick auf Verträge am Beispiel von Web-Services

Fazit

Verträge sind hilfreich, reichen aber nicht aus, um Kompositionsprobleme zu lösen.

Diskussion

Mögliche Abhilfe

Formale Spezifikation: Verträge für Komponenten

- Transport in die Praxis?
- Vollautomatische Werkzeugunterstützung?
- Stabilität der Verträge bei Pflege, Änderung des Systems?
- Nicht-funktionale Eigenschaften, Compliance?

Kritischer Blick auf Verträge am Beispiel von Web-Services

Fazit

Verträge sind hilfreich, reichen aber nicht aus, um Kompositionsprobleme zu lösen.

Diskussion

Mögliche Abhilfe

Formale Spezifikation: Verträge für Komponenten

- Transport in die Praxis?
- Vollautomatische Werkzeugunterstützung?
- Stabilität der Verträge bei Pflege, Änderung des Systems?
- Nicht-funktionale Eigenschaften, Compliance?

Kritischer Blick auf Verträge am Beispiel von Web-Services

Fazit

Verträge sind hilfreich, reichen aber nicht aus, um Kompositionsprobleme zu lösen.

Diskussion

Mögliche Abhilfe

Formale Spezifikation: Verträge für Komponenten

- Transport in die Praxis?
- Vollautomatische Werkzeugunterstützung?
- Stabilität der Verträge bei Pflege, Änderung des Systems?
- Nicht-funktionale Eigenschaften, Compliance?

Kritischer Blick auf Verträge am Beispiel von Web-Services

Fazit

Verträge sind hilfreich, reichen aber nicht aus, um Kompositionsprobleme zu lösen.

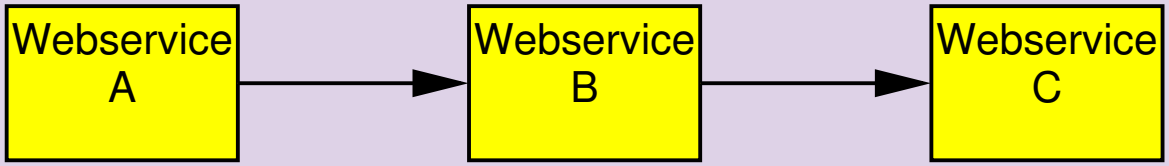
Diskussion

Mögliche Abhilfe

Formale Spezifikation: Verträge für Komponenten

- Transport in die Praxis?
- Vollautomatische Werkzeugunterstützung?
- Stabilität der Verträge bei Pflege, Änderung des Systems?
- Nicht-funktionale Eigenschaften, Compliance?

Kritischer Blick auf Verträge am Beispiel von Web-Services



Fazit

Verträge sind hilfreich, reichen aber nicht aus, um Kompositionsprobleme zu lösen.

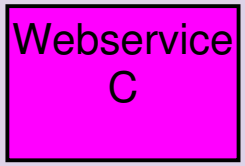
Diskussion

Mögliche Abhilfe

Formale Spezifikation: Verträge für Komponenten

- Transport in die Praxis?
- Vollautomatische Werkzeugunterstützung?
- Stabilität der Verträge bei Pflege, Änderung des Systems?
- Nicht-funktionale Eigenschaften, Compliance?

Kritischer Blick auf Verträge am Beispiel von Web-Services



Fazit

Verträge sind hilfreich, reichen aber nicht aus, um Kompositionsprobleme zu lösen.

Diskussion

Mögliche Abhilfe

Formale Spezifikation: Verträge für Komponenten

- Transport in die Praxis?
- Vollautomatische Werkzeugunterstützung?
- Stabilität der Verträge bei Pflege, Änderung des Systems?
- Nicht-funktionale Eigenschaften, Compliance?

Kritischer Blick auf Verträge am Beispiel von Web-Services

Fazit

Verträge sind hilfreich, reichen aber nicht aus, um Kompositionsprobleme zu lösen.

Diskussion

Mögliche Abhilfe

Formale Spezifikation: Verträge für Komponenten

- Transport in die Praxis?
- Vollautomatische Werkzeugunterstützung?
- Stabilität der Verträge bei Pflege, Änderung des Systems?
- Nicht-funktionale Eigenschaften, Compliance?

Kritischer Blick auf Verträge am Beispiel von Web-Services

Fazit

Verträge sind hilfreich, reichen aber nicht aus, um Kompositionsprobleme zu lösen.

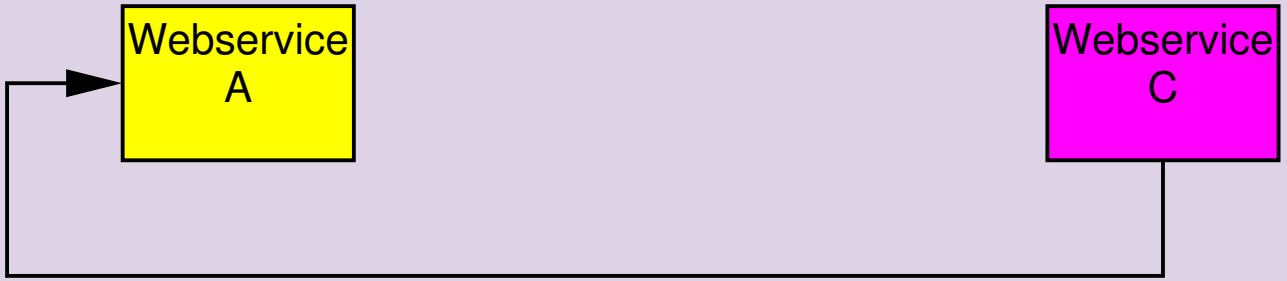
Diskussion

Mögliche Abhilfe

Formale Spezifikation: Verträge für Komponenten

- Transport in die Praxis?
- Vollautomatische Werkzeugunterstützung?
- Stabilität der Verträge bei Pflege, Änderung des Systems?
- Nicht-funktionale Eigenschaften, Compliance?

Kritischer Blick auf Verträge am Beispiel von Web-Services



Fazit

Verträge sind hilfreich, reichen aber nicht aus, um Kompositionsprobleme zu lösen.

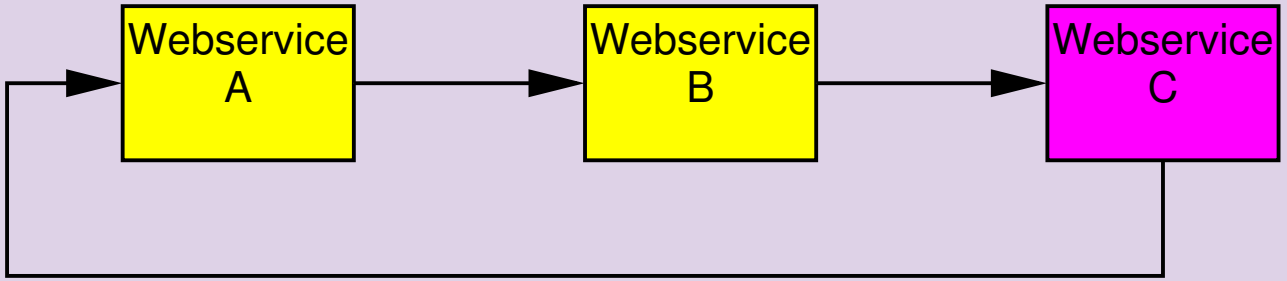
Diskussion

Mögliche Abhilfe

Formale Spezifikation: Verträge für Komponenten

- Transport in die Praxis?
- Vollautomatische Werkzeugunterstützung?
- Stabilität der Verträge bei Pflege, Änderung des Systems?
- Nicht-funktionale Eigenschaften, Compliance?

Kritischer Blick auf Verträge am Beispiel von Web-Services



Fazit

Verträge sind hilfreich, reichen aber nicht aus, um Kompositionsprobleme zu lösen.

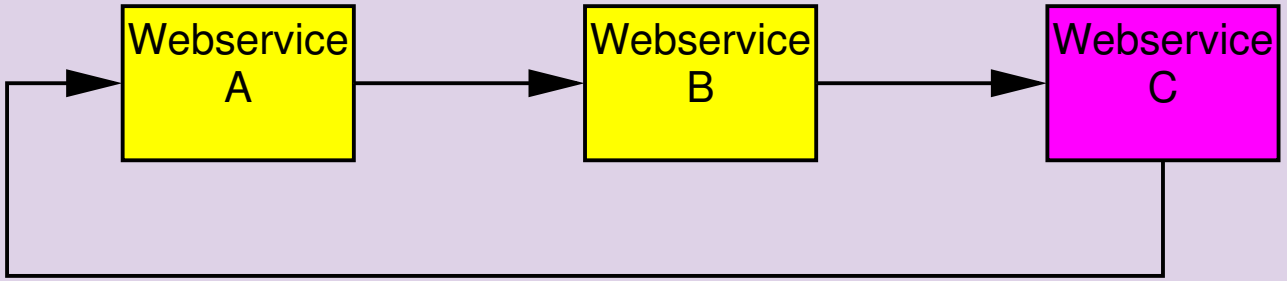
Diskussion

Mögliche Abhilfe

Formale Spezifikation: Verträge für Komponenten

- Transport in die Praxis?
- Vollautomatische Werkzeugunterstützung?
- Stabilität der Verträge bei Pflege, Änderung des Systems?
- Nicht-funktionale Eigenschaften, Compliance?

Kritischer Blick auf Verträge am Beispiel von Web-Services



Fazit

Verträge sind hilfreich, reichen aber nicht aus, um Kompositionsprobleme zu lösen.

Abhilfe durch Protokolle

Beispiel: Dateien

$open(mode : \{read, write, append\});$
$put(x : ITEM);$
$get : ITEM);$
$app(x : ITEM);$
$close;$

Beispiel: Vieraugenprinzip bei Überweisung

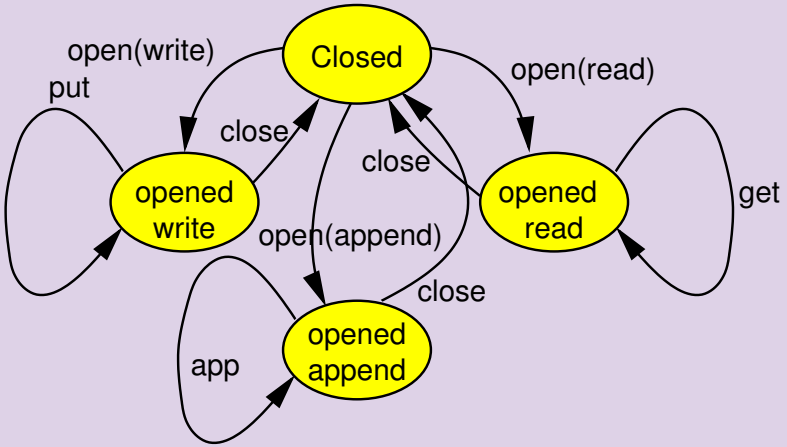
$order(x : ITEM);$
$check(x : ITEM, b : BILL);$
$received(x : ITEM, b : BILL);$

$sign_1(p : PERSON_1);$
$sign_2(p : PERSON_2);$
$pay(a : INT);$
$reject(x : ITEM, b : BILL);$

Abhilfe durch Protokolle

Beispiel: Dateien

$open(mode : \{read, write, append\});$
$put(x : ITEM);$
$get : ITEM);$
$app(x : ITEM);$
$close;$



Beispiel: Vieraugenprinzip bei Überweisung

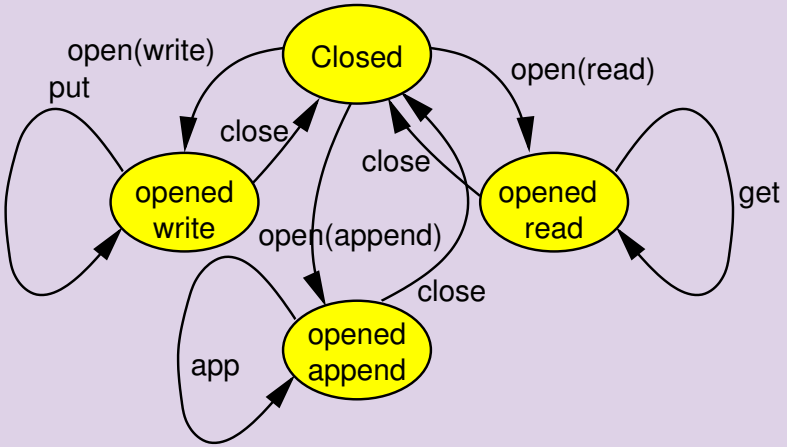
$order(x : ITEM);$
$check(x : ITEM, b : BILL);$
$received(x : ITEM, b : BILL);$

$sign_1(p : PERSON_1);$
$sign_2(p : PERSON_2);$
$pay(a : INT);$
$reject(x : ITEM, b : BILL);$

Abhilfe durch Protokolle

Beispiel: Dateien

$open(mode : \{read, write, append\});$
$put(x : ITEM);$
$get : ITEM);$
$app(x : ITEM);$
$close;$



Beispiel: Vieraugenprinzip bei Überweisung

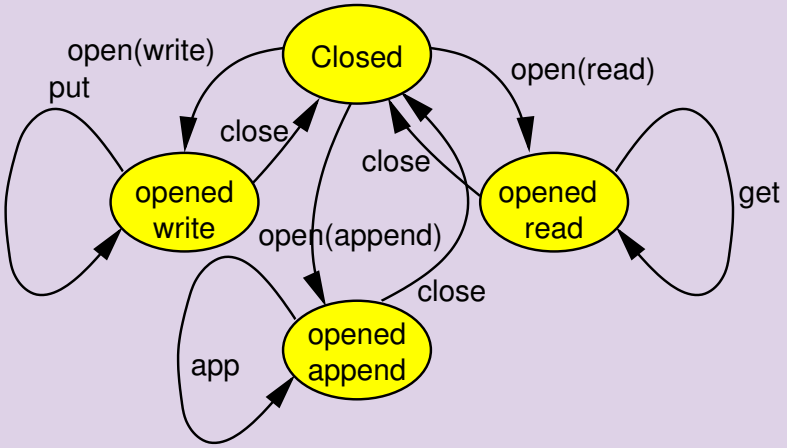
$order(x : ITEM);$
$check(x : ITEM, b : BILL);$
$received(x : ITEM, b : BILL);$

$sign_1(p : PERSON_1);$
$sign_2(p : PERSON_2);$
$pay(a : INT);$
$reject(x : ITEM, b : BILL);$

Abhilfe durch Protokolle

Beispiel: Dateien

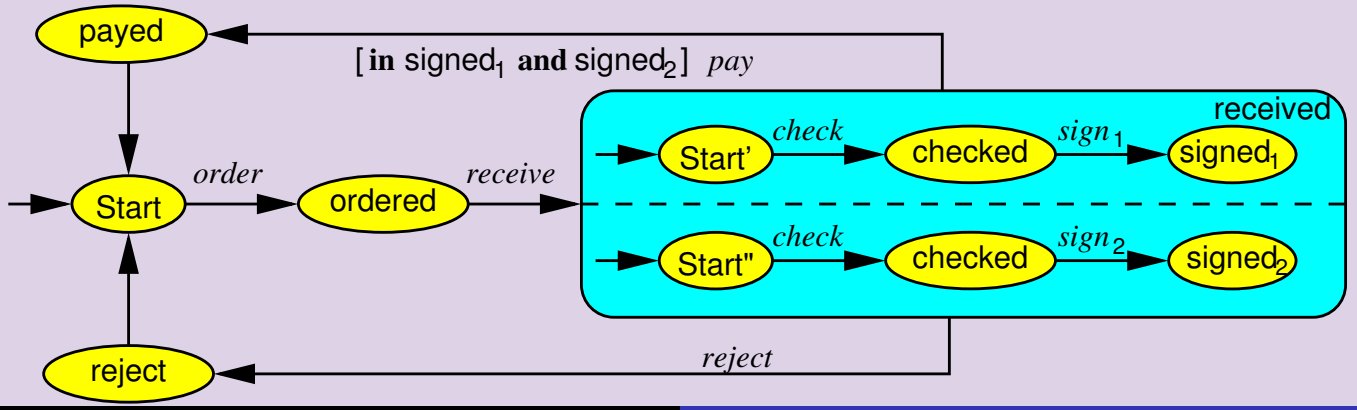
$open(mode : \{read, write, append\});$
$put(x : ITEM);$
$get : ITEM);$
$app(x : ITEM);$
$close;$



Beispiel: Vieraugenprinzip bei Überweisung

$order(x : ITEM);$
$check(x : ITEM, b : BILL);$
$received(x : ITEM, b : BILL);$

$sign_1(p : PERSON_1);$
$sign_2(p : PERSON_2);$
$pay(a : INT);$
$reject(x : ITEM, b : BILL);$



Komponentenprotokollprüfung

Protokoll einer Komponente C: Legale Folgen von Aufrufreihenfolgen der Dienste

- Formalisiert als formale Sprache $P(C)$
- Häufig definiert durch endlichen Automaten

Benutzung einer Komponente C: Tatsächlich mögliche Aufrufreihenfolgen der Dienste an C

- Formalisiert als formale Sprache $U(C)$
- Konservative Approximation des Verhaltens des Komponentensystems

Protokollkonformität: $U(C) \subseteq P(C)$.

Komponentenprotokollprüfung

Protokoll einer Komponente C: Legale Folgen von Aufrufreihenfolgen der Dienste

- Formalisiert als formale Sprache $P(C)$
- Häufig definiert durch endlichen Automaten

Benutzung einer Komponente C: Tatsächlich mögliche Aufrufreihenfolgen der Dienste an C

- Formalisiert als formale Sprache $U(C)$
- Konservative Approximation des Verhaltens des Komponentensystems

Protokollkonformität: $U(C) \subseteq P(C)$.

Komponentenprotokollprüfung

Protokoll einer Komponente C: Legale Folgen von Aufrufreihenfolgen der Dienste

- Formalisiert als formale Sprache $P(C)$
- Häufig definiert durch endlichen Automaten

Benutzung einer Komponente C: Tatsächlich mögliche Aufrufreihenfolgen der Dienste an C

- Formalisiert als formale Sprache $U(C)$
- Konservative Approximation des Verhaltens des Komponentensystems

Protokollkonformität: $U(C) \subseteq P(C)$.

Komponentenprotokollprüfung

Protokoll einer Komponente C: Legale Folgen von Aufrufreihenfolgen der Dienste

- Formalisiert als formale Sprache $P(C)$
- Häufig definiert durch endlichen Automaten

Benutzung einer Komponente C: Tatsächlich mögliche Aufrufreihenfolgen der Dienste an C

- Formalisiert als formale Sprache $U(C)$
- Konservative Approximation des Verhaltens des Komponentensystems

Protokollkonformität: $U(C) \subseteq P(C)$.

Komponentenprotokollprüfung

Protokoll einer Komponente C: Legale Folgen von Aufrufreihenfolgen der Dienste

- Formalisiert als formale Sprache $P(C)$
- Häufig definiert durch endlichen Automaten

Benutzung einer Komponente C: Tatsächlich mögliche Aufrufreihenfolgen der Dienste an C

- Formalisiert als formale Sprache $U(C)$
- Konservative Approximation des Verhaltens des Komponentensystems

Protokollkonformität: $U(C) \subseteq P(C)$.

Komponentenprotokollprüfung

Protokoll einer Komponente C: Legale Folgen von Aufrufreihenfolgen der Dienste

- Formalisiert als formale Sprache $P(C)$
- Häufig definiert durch endlichen Automaten

Benutzung einer Komponente C: Tatsächlich mögliche Aufrufreihenfolgen der Dienste an C

- Formalisiert als formale Sprache $U(C)$
- Konservative Approximation des Verhaltens des Komponentensystems

Protokollkonformität: $U(C) \subseteq P(C)$.

Komponentenprotokollprüfung

Protokoll einer Komponente C: Legale Folgen von Aufrufreihenfolgen der Dienste

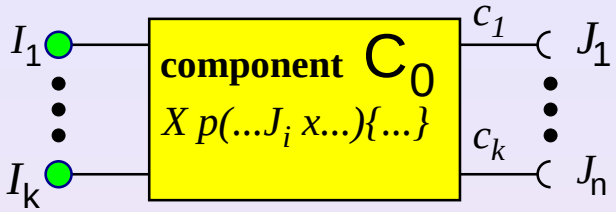
- Formalisiert als formale Sprache $P(C)$
- Häufig definiert durch endlichen Automaten

Benutzung einer Komponente C: Tatsächlich mögliche Aufrufreihenfolgen der Dienste an C

- Formalisiert als formale Sprache $U(C)$
- Konservative Approximation des Verhaltens des Komponentensystems

Protokollkonformität: $U(C) \subseteq P(C)$.

Unser Komponentenmodell



- Angebotsschnittstellen $I_1, \ldots, I_k$
- Nutzungsschnittstellen $J_1, \ldots, J_n$

Komponente: Implementierung der Angebotsschnittstellen

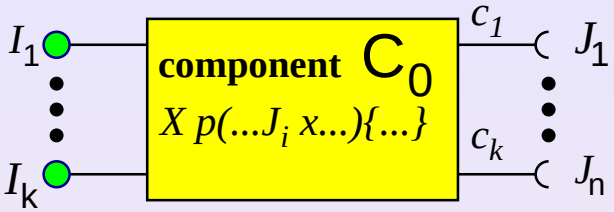
- unter Verwendung der Nutzungsschnittstellen
- sprach–, orts– und plattformunabhängige Auslieferung
- u.U. kein Zugriff auf Implementierung und auf Komponentenzustand

Komponentensystem: Menge von über Schnittstellen verbundene Komponenten

$\Rightarrow$ Kommunikation erfolgt nur über Schnittstellen

- Schnittstellen sind Mengen von Prozeduren/Funktionen (**Dienste**)
- Bei Aufruf **synchroner Dienste** wartet Aufrufer auf Fertigstellung
- Bei Aufruf **asynchroner Dienste** fährt Aufrufer fort ohne auf Fertigstellung zu warten.

Unser Komponentenmodell



- Angebotsschnittstellen $I_1, \ldots, I_k$
- Nutzungsschnittstellen $J_1, \ldots, J_n$

Komponente: Implementierung der Angebotsschnittstellen

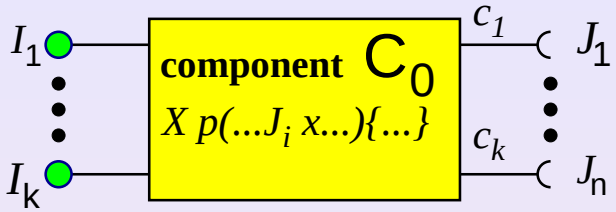
- unter Verwendung der Nutzungsschnittstellen
- sprach–, orts– und plattformunabhängige Auslieferung
- u.U. kein Zugriff auf Implementierung (auch nicht Binärcode)

Komponentensystem: Menge von über Schnittstellen verbundene Komponenten

$\Rightarrow$ Kommunikation erfolgt nur über Schnittstellen

- Schnittstellen sind Mengen von Prozeduren/Funktionen (**Dienste**)
- Bei Aufruf **synchroner Dienste** wartet Aufrufer auf Fertigstellung
- Bei Aufruf **asynchroner Dienste** fährt Aufrufer fort ohne auf Fertigstellung zu warten.

Unser Komponentenmodell



- Angebotsschnittstellen $I_1, \ldots, I_k$
- Nutzungsschnittstellen $J_1, \ldots, J_n$

Komponente: Implementierung der Angebotsschnittstellen

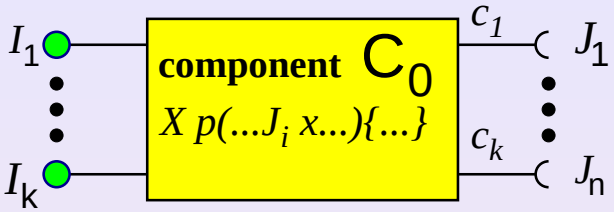
- unter Verwendung der Nutzungsschnittstellen
- sprach–, orts– und plattformunabhängige Auslieferung
- u.U. kein Zugriff auf Implementierung (auch nicht Binärcode)

Komponentensystem: Menge von über Schnittstellen verbundene Komponenten

$\Rightarrow$ Kommunikation erfolgt nur über Schnittstellen

- Schnittstellen sind Mengen von Prozeduren/Funktionen (**Dienste**)
- Bei Aufruf **synchroner Dienste** wartet Aufrufer auf Fertigstellung
- Bei Aufruf **asynchroner Dienste** fährt Aufrufer fort ohne auf Fertigstellung zu warten.

Unser Komponentenmodell



- Angebotsschnittstellen $I_1, \ldots, I_k$
- Nutzungsschnittstellen $J_1, \ldots, J_n$

Komponente: Implementierung der Angebotsschnittstellen

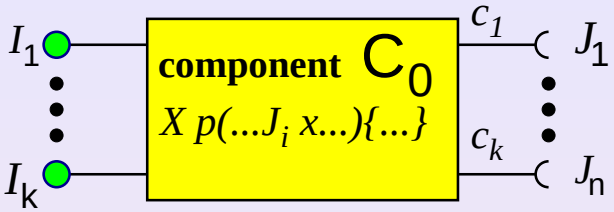
- unter Verwendung der Nutzungsschnittstellen
- sprach–, orts– und plattformunabhängige Auslieferung
- u.U. kein Zugriff auf Implementierung (auch nicht Binärcode)

Komponentensystem: Menge von über Schnittstellen verbundene Komponenten

$\Rightarrow$ Kommunikation erfolgt nur über Schnittstellen

- Schnittstellen sind Mengen von Prozeduren/Funktionen (Dienste)
- Bei Aufruf **synchroner Dienste** wartet Aufrufer auf Fertigstellung
- Bei Aufruf **asynchroner Dienste** fährt Aufrufer fort ohne auf Fertigstellung zu warten.

Unser Komponentenmodell



- Angebotsschnittstellen $I_1, \ldots, I_k$
- Nutzungsschnittstellen $J_1, \ldots, J_n$

Komponente: Implementierung der Angebotsschnittstellen

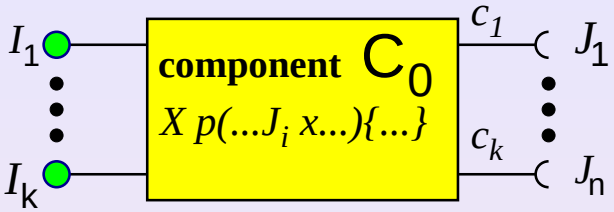
- unter Verwendung der Nutzungsschnittstellen
- sprach–, orts– und plattformunabhängige Auslieferung
- u.U. kein Zugriff auf Implementierung (auch nicht Binärcode)

Komponentensystem: Menge von über Schnittstellen verbundene Komponenten

$\Rightarrow$ Kommunikation erfolgt nur über Schnittstellen

- Schnittstellen sind Mengen von Prozeduren/Funktionen (**Dienste**)
- Bei Aufruf **synchroner Dienste** wartet Aufrufer auf Fertigstellung
- Bei Aufruf **asynchroner Dienste** fährt Aufrufer fort ohne auf Fertigstellung zu warten.

Unser Komponentenmodell



- Angebotsschnittstellen $I_1, \ldots, I_k$
- Nutzungsschnittstellen $J_1, \ldots, J_n$

Komponente: Implementierung der Angebotsschnittstellen

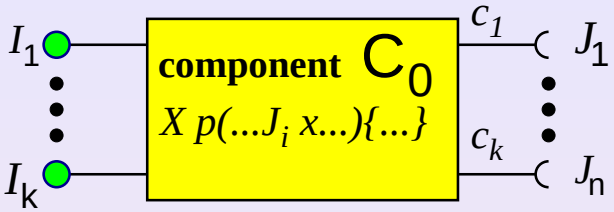
- unter Verwendung der Nutzungsschnittstellen
- sprach–, orts– und plattformunabhängige Auslieferung
- u.U. kein Zugriff auf Implementierung (auch nicht Binärcode)

Komponentensystem: Menge von über Schnittstellen verbundene Komponenten

⇒ Kommunikation erfolgt nur über Schnittstellen

- Schnittstellen sind Mengen von Prozeduren/Funktionen **Dienste**)
- Bei Aufruf **synchroner Dienste** wartet Aufrufer auf Fertigstellung
- Bei Aufruf **asynchroner Dienste** fährt Aufrufer fort ohne auf Fertigstellung zu warten.

Unser Komponentenmodell



- Angebotsschnittstellen $I_1, \ldots, I_k$
- Nutzungsschnittstellen $J_1, \ldots, J_n$

Komponente: Implementierung der Angebotsschnittstellen

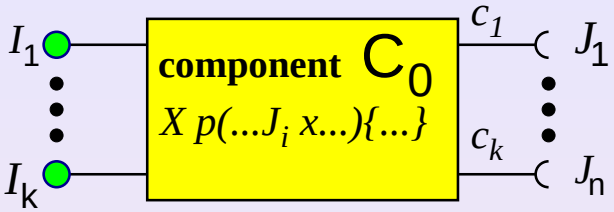
- unter Verwendung der Nutzungsschnittstellen
- sprach–, orts– und plattformunabhängige Auslieferung
- u.U. kein Zugriff auf Implementierung (auch nicht Binärcode)

Komponentensystem: Menge von über Schnittstellen verbundene Komponenten

$\Rightarrow$ Kommunikation erfolgt nur über Schnittstellen

- Schnittstellen sind Mengen von Prozeduren/Funktionen **Dienste**)
- Bei Aufruf **synchroner Dienste** wartet Aufrufer auf Fertigstellung
- Bei Aufruf **asynchroner Dienste** fährt Aufrufer fort ohne auf Fertigstellung zu warten.

Unser Komponentenmodell



- Angebotsschnittstellen $I_1, \ldots, I_k$
- Nutzungsschnittstellen $J_1, \ldots, J_n$

Komponente: Implementierung der Angebotsschnittstellen

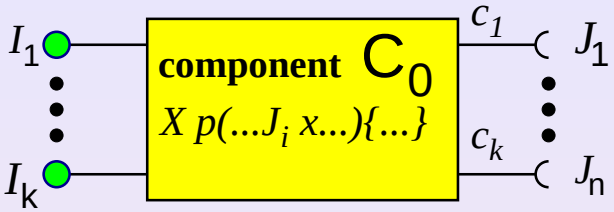
- unter Verwendung der Nutzungsschnittstellen
- sprach–, orts– und plattformunabhängige Auslieferung
- u.U. kein Zugriff auf Implementierung (auch nicht Binärcode)

Komponentensystem: Menge von über Schnittstellen verbundene Komponenten

$\Rightarrow$ Kommunikation erfolgt nur über Schnittstellen

- Schnittstellen sind Mengen von Prozeduren/Funktionen **Dienste**)
- Bei Aufruf **synchroner Dienste** wartet Aufrufer auf Fertigstellung
- Bei Aufruf **asynchroner Dienste** fährt Aufrufer fort ohne auf Fertigstellung zu warten.

Unser Komponentenmodell



- Angebotsschnittstellen $I_1, \ldots, I_k$
- Nutzungsschnittstellen $J_1, \ldots, J_n$

Komponente: Implementierung der Angebotsschnittstellen

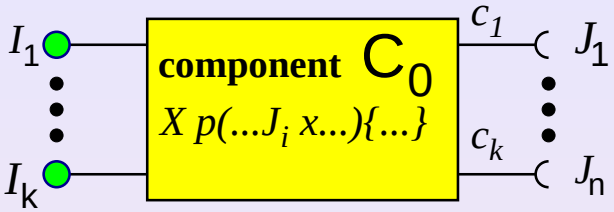
- unter Verwendung der Nutzungsschnittstellen
- sprach–, orts– und plattformunabhängige Auslieferung
- u.U. kein Zugriff auf Implementierung (auch nicht Binärcode)

Komponentensystem: Menge von über Schnittstellen verbundene Komponenten

$\Rightarrow$ Kommunikation erfolgt nur über Schnittstellen

- Schnittstellen sind Mengen von Prozeduren/Funktionen (**Dienste**)
- Bei Aufruf **synchroner Dienste** wartet Aufrufer auf Fertigstellung
- Bei Aufruf **asynchroner Dienste** fährt Aufrufer fort ohne auf Fertigstellung zu warten.

Unser Komponentenmodell



- Angebotsschnittstellen $I_1, \ldots, I_k$
- Nutzungsschnittstellen $J_1, \ldots, J_n$

Komponente: Implementierung der Angebotsschnittstellen

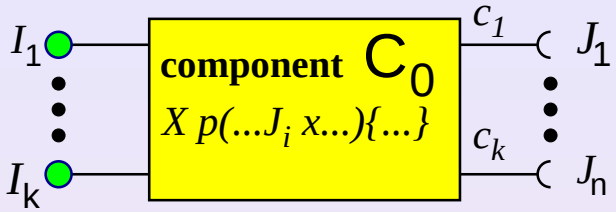
- unter Verwendung der Nutzungsschnittstellen
- sprach–, orts– und plattformunabhängige Auslieferung
- u.U. kein Zugriff auf Implementierung (auch nicht Binärcode)

Komponentensystem: Menge von über Schnittstellen verbundene Komponenten

$\Rightarrow$ Kommunikation erfolgt nur über Schnittstellen

- Schnittstellen sind Mengen von Prozeduren/Funktionen (**Dienste**)
- Bei Aufruf **synchroner Dienste** wartet Aufrufer auf Fertigstellung
- Bei Aufruf **asynchroner Dienste** fährt Aufrufer fort ohne auf Fertigstellung zu warten.

Unser Komponentenmodell



- Angebotsschnittstellen $I_1, \ldots, I_k$
- Nutzungsschnittstellen $J_1, \ldots, J_n$

Komponente: Implementierung der Angebotsschnittstellen

- unter Verwendung der Nutzungsschnittstellen
- sprach–, orts– und plattformunabhängige Auslieferung
- u.U. kein Zugriff auf Implementierung (auch nicht Binärcode)

Komponentensystem: Menge von über Schnittstellen verbundene Komponenten

$\Rightarrow$ Kommunikation erfolgt nur über Schnittstellen

- Schnittstellen sind Mengen von Prozeduren/Funktionen (**Dienste**)
- Bei Aufruf **synchroner Dienste** wartet Aufrufer auf Fertigstellung
- Bei Aufruf **asynchroner Dienste** fährt Aufrufer fort ohne auf Fertigstellung zu warten.

2. Verwandte Arbeiten

Lokale Protokollprüfung (u.A. Nierstrasz, Plasil,...)

- Prüfung jeder Leitung von Nutzungs– zu Angebotsschnittstelle
- Sowohl Nutzung als auch Protokoll als reguläre Sprache modelliert

$\Rightarrow$ Protokollkonfirmitätsprüfung $\triangleq$ Teilmengenbeziehung regulärer Sprachen

- Nebenläufigkeit durch Shuffle-Automaten

Problem

Lokale Protokollprüfung kann zu falschen Positivaussagen führen.

Ursachen (Zimmermann/Schaarschmidt 2006

2. Verwandte Arbeiten

Lokale Protokollprüfung (u.A. Nierstrasz, Plasil,...)

- Prüfung jeder Leitung von Nutzungs– zu Angebotsschnittstelle
- Sowohl Nutzung als auch Protokoll als reguläre Sprache modelliert

$\Rightarrow$ Protokollkonfirmitätsprüfung $\triangleq$ Teilmengenbeziehung regulärer Sprachen

- Nebenläufigkeit durch Shuffle-Automaten

Problem

Lokale Protokollprüfung kann zu falschen Positivaussagen führen.

Ursachen (Zimmermann/Schaarschmidt 2006

2. Verwandte Arbeiten

Lokale Protokollprüfung (u.A. Nierstrasz, Plasil,...)

- Prüfung jeder Leitung von Nutzungs– zu Angebotsschnittstelle
- Sowohl Nutzung als auch Protokoll als reguläre Sprache modelliert

$\Rightarrow$ Protokollkonfirmitätsprüfung $\triangleq$ Teilmengenbeziehung regulärer Sprachen

- Nebenläufigkeit durch Shuffle-Automaten

Problem

Lokale Protokollprüfung kann zu falschen Positivaussagen führen.

Ursachen (Zimmermann/Schaarschmidt 2006

2. Verwandte Arbeiten

Lokale Protokollprüfung (u.A. Nierstrasz, Plasil,...)

- Prüfung jeder Leitung von Nutzungs– zu Angebotsschnittstelle
- Sowohl Nutzung als auch Protokoll als reguläre Sprache modelliert

$\Rightarrow$ Protokollkonfirmitätsprüfung $\triangleq$ Teilmengenbeziehung regulärer Sprachen

- Nebenläufigkeit durch Shuffle-Automaten

Problem

Lokale Protokollprüfung kann zu falschen Positivaussagen führen.

Ursachen (Zimmermann/Schaarschmidt 2006

- Dienste-Schnittstellen können nicht durch Protokolle vollständig beschrieben werden
- Zustandsinformationen können nicht durch Protokolle beschrieben werden
- Zeitbedingungen können nicht durch Protokolle beschrieben werden
- Verwendung von Kontrollflussanweisungen (Verzweigungen, Schleifen, Rekursionen) in Protokollen

2. Verwandte Arbeiten

Lokale Protokollprüfung (u.A. Nierstrasz, Plasil,...)

- Prüfung jeder Leitung von Nutzungs– zu Angebotsschnittstelle
- Sowohl Nutzung als auch Protokoll als reguläre Sprache modelliert

$\Rightarrow$ Protokollkonfirmitätsprüfung $\triangleq$ Teilmengenbeziehung regulärer Sprachen

- Nebenläufigkeit durch Shuffle-Automaten

Problem

Lokale Protokollprüfung kann zu falschen Positivaussagen führen.

Ursachen (Zimmermann/Schaarschmidt 2006

- Mehrere Schnittstellen können nicht unabhängig voneinander benutzt werden

2. Verwandte Arbeiten

Lokale Protokollprüfung (u.A. Nierstrasz, Plasil,...)

- Prüfung jeder Leitung von Nutzungs– zu Angebotsschnittstelle
- Sowohl Nutzung als auch Protokoll als reguläre Sprache modelliert

$\Rightarrow$ Protokollkonfirmitätsprüfung $\triangleq$ Teilmengenbeziehung regulärer Sprachen

- Nebenläufigkeit durch Shuffle-Automaten

Problem

Lokale Protokollprüfung kann zu falschen Positivaussagen führen.

Ursachen (Zimmermann/Schaarschmidt 2006

- Mehrere Schnittstellen können nicht unabhängig voneinander benutzt werden

$\Rightarrow$ Komponentenprotokoll statt Angebotsschnittstellenprotokoll

$\Rightarrow$ Blackbox-Verzahnung

$\Rightarrow$ Verhaltensverzahnung (Kontrollflussverzahnung)

$\Rightarrow$ Schnittstellenverzahnung (Datenflussverzahnung)

2. Verwandte Arbeiten

Lokale Protokollprüfung (u.A. Nierstrasz, Plasil,...)

- Prüfung jeder Leitung von Nutzungs– zu Angebotsschnittstelle
- Sowohl Nutzung als auch Protokoll als reguläre Sprache modelliert

$\Rightarrow$ Protokollkonfirmitätsprüfung $\triangleq$ Teilmengenbeziehung regulärer Sprachen

- Nebenläufigkeit durch Shuffle-Automaten

Problem

Lokale Protokollprüfung kann zu falschen Positivaussagen führen.

Ursachen (Zimmermann/Schaarschmidt 2006

- Mehrere Schnittstellen können nicht unabhängig voneinander benutzt werden

$\Rightarrow$ Komponentenprotokoll statt Angebotsschnittstellenprotokoll

- Rekursive Rückaufrufe
- Verhaltensbeschreibung über Kellerautomaten

$\Rightarrow$ Berücksichtigt keine Nebenläufigkeit

2. Verwandte Arbeiten

Lokale Protokollprüfung (u.A. Nierstrasz, Plasil,...)

- Prüfung jeder Leitung von Nutzungs– zu Angebotsschnittstelle
- Sowohl Nutzung als auch Protokoll als reguläre Sprache modelliert

$\Rightarrow$ Protokollkonfirmitätsprüfung $\triangleq$ Teilmengenbeziehung regulärer Sprachen

- Nebenläufigkeit durch Shuffle-Automaten

Problem

Lokale Protokollprüfung kann zu falschen Positivaussagen führen.

Ursachen (Zimmermann/Schaarschmidt 2006

- Mehrere Schnittstellen können nicht unabhängig voneinander benutzt werden

$\Rightarrow$ Komponentenprotokoll statt Angebotsschnittstellenprotokoll

- Rekursive Rückaufrufe
- Verhaltensbeschreibung über Kellerautomaten

$\Rightarrow$ Berücksichtigt keine Nebenläufigkeit

2. Verwandte Arbeiten

Lokale Protokollprüfung (u.A. Nierstrasz, Plasil,...)

- Prüfung jeder Leitung von Nutzungs– zu Angebotsschnittstelle
- Sowohl Nutzung als auch Protokoll als reguläre Sprache modelliert

$\Rightarrow$ Protokollkonfirmitätsprüfung $\triangleq$ Teilmengenbeziehung regulärer Sprachen

- Nebenläufigkeit durch Shuffle-Automaten

Problem

Lokale Protokollprüfung kann zu falschen Positivaussagen führen.

Ursachen (Zimmermann/Schaarschmidt 2006

- Mehrere Schnittstellen können nicht unabhängig voneinander benutzt werden

$\Rightarrow$ Komponentenprotokoll statt Angebotsschnittstellenprotokoll

- Rekursive Rückaufrufe
- Verhaltensbeschreibung über Kellerautomaten

$\Rightarrow$ Berücksichtigt keine Nebenläufigkeit

2. Verwandte Arbeiten

Lokale Protokollprüfung (u.A. Nierstrasz, Plasil,...)

- Prüfung jeder Leitung von Nutzungs– zu Angebotsschnittstelle
- Sowohl Nutzung als auch Protokoll als reguläre Sprache modelliert

$\Rightarrow$ Protokollkonfirmitätsprüfung $\triangleq$ Teilmengenbeziehung regulärer Sprachen

- Nebenläufigkeit durch Shuffle-Automaten

Problem

Lokale Protokollprüfung kann zu falschen Positivaussagen führen.

Ursachen (Zimmermann/Schaarschmidt 2006

- Mehrere Schnittstellen können nicht unabhängig voneinander benutzt werden

$\Rightarrow$ Komponentenprotokoll statt Angebotsschnittstellenprotokoll

- Rekursive Rückaufrufe
- Verhaltensbeschreibung über Kellerautomaten

$\Rightarrow$ Berücksichtigt keine Nebenläufigkeit

2. Verwandte Arbeiten

Lokale Protokollprüfung (u.A. Nierstrasz, Plasil,...)

- Prüfung jeder Leitung von Nutzungs– zu Angebotsschnittstelle
- Sowohl Nutzung als auch Protokoll als reguläre Sprache modelliert

$\Rightarrow$ Protokollkonfirmitätsprüfung $\triangleq$ Teilmengenbeziehung regulärer Sprachen

- Nebenläufigkeit durch Shuffle-Automaten

Problem

Lokale Protokollprüfung kann zu falschen Positivaussagen führen.

Ursachen (Zimmermann/Schaarschmidt 2006

- Mehrere Schnittstellen können nicht unabhängig voneinander benutzt werden

$\Rightarrow$ Komponentenprotokoll statt Angebotsschnittstellenprotokoll

- Rekursive Rückaufrufe
- Verhaltensbeschreibung über Kellerautomaten

$\Rightarrow$ Berücksichtigt keine Nebenläufigkeit

2. Verwandte Arbeiten

Lokale Protokollprüfung (u.A. Nierstrasz, Plasil,...)

- Prüfung jeder Leitung von Nutzungs– zu Angebotsschnittstelle
- Sowohl Nutzung als auch Protokoll als reguläre Sprache modelliert

$\Rightarrow$ Protokollkonfirmitätsprüfung $\triangleq$ Teilmengenbeziehung regulärer Sprachen

- Nebenläufigkeit durch Shuffle-Automaten

Problem

Lokale Protokollprüfung kann zu falschen Positivaussagen führen.

Ursachen (Zimmermann/Schaarschmidt 2006

- Mehrere Schnittstellen können nicht unabhängig voneinander benutzt werden

$\Rightarrow$ Komponentenprotokoll statt Angebotsschnittstellenprotokoll

- Rekursive Rückaufrufe
- Verhaltensbeschreibung über Kellerautomaten

$\Rightarrow$ Berücksichtigt keine Nebenläufigkeit

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kellerwalder

⇒ viele Programmpunkte

⇒ Kellerwalder wird übertragen in Kelleraustauscher

⇒ Viele Zustandsübergängen werden Serviceaufrufe zugeordnet

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kellerwalder

⇒ viele Programmpunkte

⇒ Kellerwalder wird überführt in Kellerbaum

⇒ Viele Zustandsübergängen werden Servicearbeit zugeordnet

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun k-Kreuzer

⇒ Kellerinhalt enthält k Programmpunkte

⇒ Verhalten ist nun Automat über k Kellern

⇒ Jeder Zustand entspricht einem Service, der mehrere Services aufrufen kann

```
void main() {      void b() {
q0: a();           q5: b();
q1: return;        q6: c();
}                  q7: return
void a() {         }
q2: b();           void c() {
q3: c();           q8: a();
q4: return;        q9: return
}                  }
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun k-Kreuzer

⇒ Kellerinhalt enthält Programmpunkte

⇒ Kellerinhalt wird als Liste in Kd-Baum

⇒ Jeder Zustand entspricht einem Service-Transitionszustand

```
void main() {
q0: a();
q1: return;
}
void a() {
q2: b();
q3: c();
q4: return;
}
```

```
void b() {
q5: b();
q6: c();
q7: return
}
void c() {
q8: a();
q9: return
}
```

q_0

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun k-Kellerautomat
- Jeder Programmpunkt
- Kellerautomat wird ausgeführt in Kontext
- Jeder Zustand entspricht einem Service, Transition entspricht

```
void main() {
q0: a();
q1: return;
}
void a() {
q2: b();
q3: c();
q4: return;
}
```

```
void b() {
q5: b();
q6: c();
q7: return
}
void c() {
q8: a();
q9: return
}
```

q_0

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

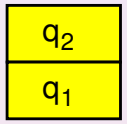
- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun k-Kellerwert
- Viele Programmpunkte
- Kellerwerte sind Kellerinhalte Kb Instanzen
- Viele Zustände entsprechen warten Service, Ereignis, Kommunikation




```
void main() {
q0: a();
q1: return;
}
void a() {
q2: b();
q3: c();
q4: return;
}
```

```
void b() {
q5: b();
q6: c();
q7: return
}
void c() {
q8: a();
q9: return
}
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

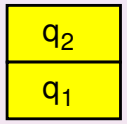
- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun k-Kreuzer
- Jeder Programmpunkt
- Kreuzer bewegt sich entlang Kreuzungen
- Jeder Zustand entspricht einem Serviceaufruf




```
void main() {      void b() {
q0: a();           q5: b();
q1: return;        q6: c();
}                  q7: return
void a() {         }
q2: b();           void c() {
q3: c();           q8: a();
q4: return;        q9: return
}                  }
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun k-Kreuzer
- Viele Programmpunkte
- Kellerinhalt wird durch Kellerinhalte des Kreuzers ersetzt

⇒ Viele Zustände können zu einem Zustand zusammengefasst werden (Servicebasiertes Testen)

q_5
q_3
q_1

```
void main() {      void b() {
q0: a();           q5: b();
q1: return;        q6: c();
}                  q7: return
void a() {         }
q2: b();           void c() {
q3: c();           q8: a();
q4: return;        q9: return
}                  }
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun k-Kellerautomat
- Jeder Programmpunkt
- Kellerinhalt wird Kellerinhalt in Ka Instanzen
- Jeder Zustand entspricht nun verschachtelten Serviceaufrufen

q_5
q_3
q_1

```
void main() {
q0: a();
q1: return;
}
void a() {
q2: b();
q3: c();
q4: return;
}
```

```
void b() {
q5: b();
q6: c();
q7: return
}
void c() {
q8: a();
q9: return
}
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

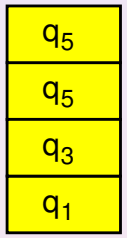
- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun k-Kreuzer
- Jeder Kreuz ist Programmpunkt
- Kreuzer bewegen sich entlang Kreuzen
- Ein Zustand entspricht einem Schnitt durch Kreuzer




```
void main() {      void b() {
q0: a();           q5: b();
q1: return;        q6: c();
}                  q7: return
void a() {         }
q2: b();           void c() {
q3: c();           q8: a();
q4: return;        q9: return
}                  }
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

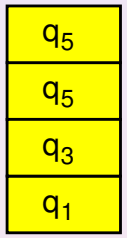
- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kellerinhalt
- Jeder Prozess hat eigenen Programmzähler
- Kellerinhalt und Programmzähler bilden Zustand
- Es gibt Zustandsübergänge von den Semantiken Fork und Join




```
void main() {     void b() {
q0: a();          q5: b();
q1: return;       q6: c();
}                 q7: return
void a() {        }
q2: b();          void c() {
q3: c();          q8: a();
q4: return;       q9: return
}                 }
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

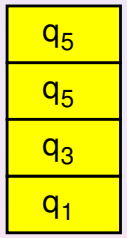
- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kellerinhalt
  und Wert der Variablen
- Kellerinhalt und Wert der Variablen bestimmen
  die Zustände aufgrund der von Semantik definierten Übergänge




```
void main() {      void b() {
q0: a();           q5: b();
q1: return;        q6: c();
}                  q7: return
void a() {         }
q2: b();           void c() {
q3: c();           q8: a();
q4: return;        q9: return
}                  }
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

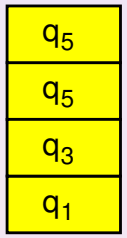
- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kellerinhalt
  und Programmpunkte
- Kellerinhalt ist nun Stack in Ausführung
  Es gibt zusätzlich Zustände von Semaphoren und Mutexen




```
void main() {      void b() {
q0: a();           q5: b();
q1: return;        q6: c();
}                  q7: return
void a() {         }
q2: b();           void c() {
q3: c();           q8: a();
q4: return;        q9: return
}                  }
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

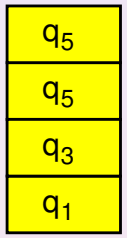
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {     void b() {
q0: a();          q5: b();
q1: return;       q6: c();
}                 q7: return
void a() {        }
q2: b();          void c() {
q3: c();          q8: a();
q4: return;       q9: return
}                 }
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt

```
void main() {          async void b() {
q0: a();               q5: b();
q1: return;            q6: c();
}                      q7: return
sync void a() {        }
q2: b();               snyc void c() {
q3: c();               q8: a();
q4: return;            q9: return
}                      }
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt

```
void main() {          async void b() {
q0: a();               q5: b();
q1: return;            q6: c();
}                      q7: return
sync void a() {        }
q2: b();               snyc void c() {
q3: c();               q8: a();
q4: return;            q9: return
}                      }
```

q_0

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt

```
void main() {
q0: a();
q1: return;
}
sync void a() {
q2: b();
q3: c();
q4: return;
}
```

```
async void b() {
q5: b();
q6: c();
q7: return
}
snyc void c() {
q8: a();
q9: return
}
```

q_0

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt

q_2
q_1

```
void main() {
q0: a();
q1: return;
}
sync void a() {
q2: b();
q3: c();
q4: return;
}
```

```
async void b() {
q5: b();
q6: c();
q7: return
}
snyc void c() {
q8: a();
q9: return
}
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt

q_2
q_1

```
void main() {
q0: a();
q1: return;
}
sync void a() {
q2: b();
q3: c();
q4: return;
}
```

```
async void b() {
q5: b();
q6: c();
q7: return
}
snyc void c() {
q8: a();
q9: return
}
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

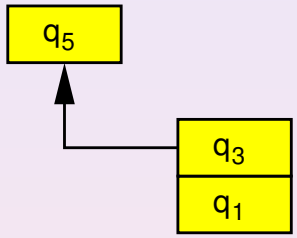
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {
q0: a();
q1: return;
}
sync void a() {
q2: b();
q3: c();
q4: return;
}
```

```
async void b() {
q5: b();
q6: c();
q7: return
}
snyc void c() {
q8: a();
q9: return
}
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

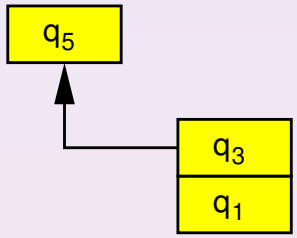
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {
q0: a();
q1: return;
}
sync void a() {
q2: b();
q3: c();
q4: return;
}
```

```
async void b() {
q5: b();
q6: c();
q7: return
}
snyc void c() {
q8: a();
q9: return
}
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

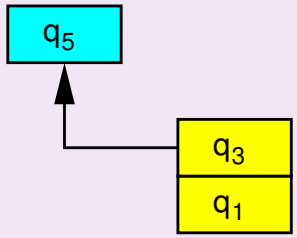
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {
q0: a();
q1: return;
}
sync void a() {
q2: b();
q3: c();
q4: return;
}
```

```
async void b() {
q5: b();
q6: c();
q7: return
}
snyc void c() {
q8: a();
q9: return
}
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

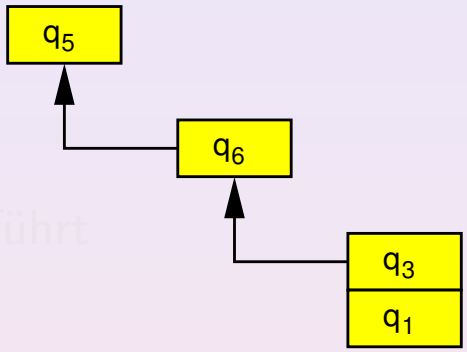
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {
q0: a();
q1: return;
}
sync void a() {
q2: b();
q3: c();
q4: return;
}
```

```
async void b() {
q5: b();
q6: c();
q7: return
}
snyc void c() {
q8: a();
q9: return
}
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

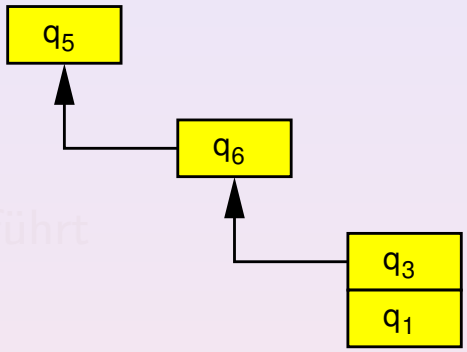
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {
q0: a();
q1: return;
}
sync void a() {
q2: b();
q3: c();
q4: return;
}
```

```
async void b() {
q5: b();
q6: c();
q7: return
}
snyc void c() {
q8: a();
q9: return
}
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

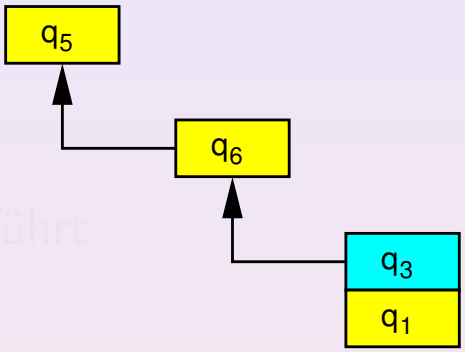
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {
q0: a();
q1: return;
}
sync void a() {
q2: b();
q3: c();
q4: return;
}
```

```
async void b() {
q5: b();
q6: c();
q7: return
}
snyc void c() {
q8: a();
q9: return
}
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

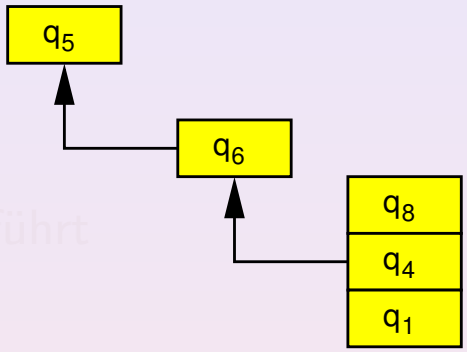
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {
q0: a();
q1: return;
}
sync void a() {
q2: b();
q3: c();
q4: return;
}
```

```
async void b() {
q5: b();
q6: c();
q7: return
}
snyc void c() {
q8: a();
q9: return
}
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

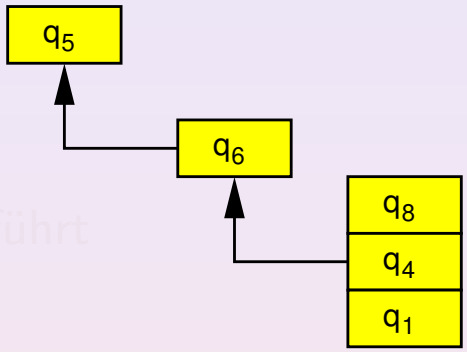
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {        async void b() {
q0: a();             q5: b();
q1: return;          q6: c();
}                    q7: return
sync void a() {      }
q2: b();             snyc void c() {
q3: c();             q8: a();
q4: return;          q9: return
}                    }
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

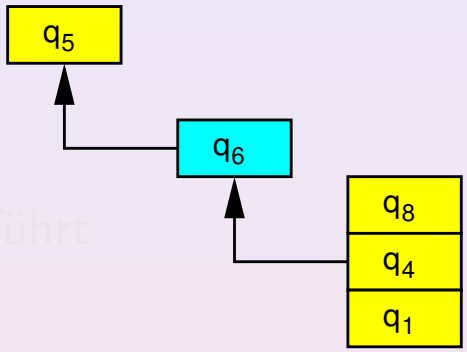
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {
q0: a();
q1: return;
}
sync void a() {
q2: b();
q3: c();
q4: return;
}
```

```
async void b() {
q5: b();
q6: c();
q7: return
}
snyc void c() {
q8: a();
q9: return
}
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

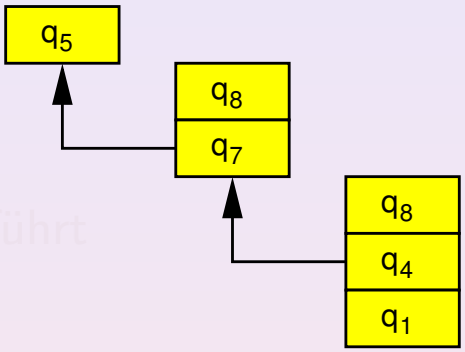
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {
q0: a();
q1: return;
}
sync void a() {
q2: b();
q3: c();
q4: return;
}
```

```
async void b() {
q5: b();
q6: c();
q7: return
}
snyc void c() {
q8: a();
q9: return
}
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

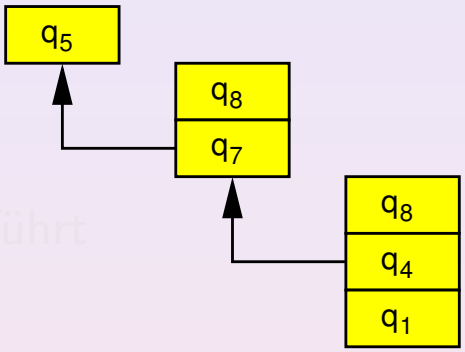
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {
q0: a();
q1: return;
}
sync void a() {
q2: b();
q3: c();
q4: return;
}
```

```
async void b() {
q5: b();
q6: c();
q7: return
}
snyc void c() {
q8: a();
q9: return
}
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

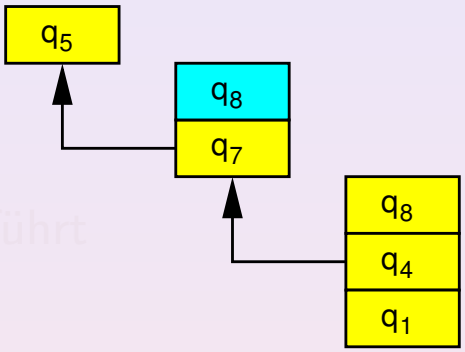
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {
q0: a();
q1: return;
}
sync void a() {
q2: b();
q3: c();
q4: return;
}
```

```
async void b() {
q5: b();
q6: c();
q7: return
}
snyc void c() {
q8: a();
q9: return
}
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

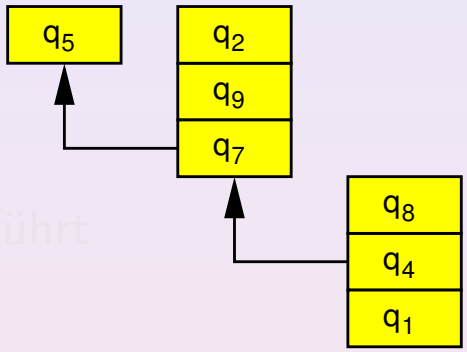
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {
q0: a();
q1: return;
}
sync void a() {
q2: b();
q3: c();
q4: return;
}
```

```
async void b() {
q5: b();
q6: c();
q7: return
}
snyc void c() {
q8: a();
q9: return
}
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

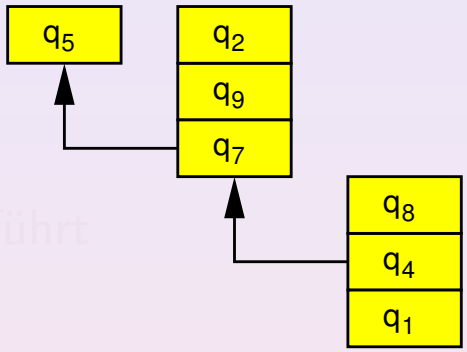
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {        async void b() {
q0: a();             q5: b();
q1: return;          q6: c();
}                    q7: return
sync void a() {      }
q2: b();             snyc void c() {
q3: c();             q8: a();
q4: return;          q9: return
}                    }
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

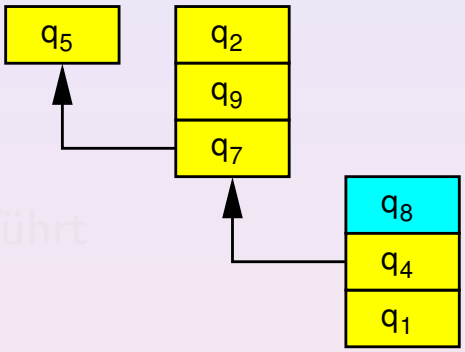
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {        async void b() {
q0: a();             q5: b();
q1: return;          q6: c();
}                    q7: return
sync void a() {      }
q2: b();             snyc void c() {
q3: c();             q8: a();
q4: return;          q9: return
}                    }
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

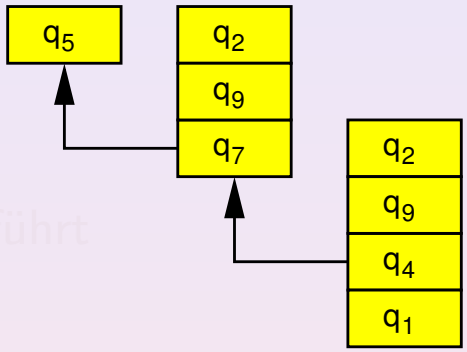
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {
q0: a();
q1: return;
}
sync void a() {
q2: b();
q3: c();
q4: return;
}
```

```
async void b() {
q5: b();
q6: c();
q7: return
}
snyc void c() {
q8: a();
q9: return
}
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

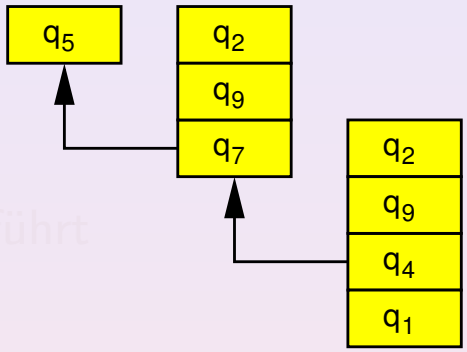
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {          async void b() {
q0: a();               q5: b();
q1: return;            q6: c();
}                      q7: return
sync void a() {        }
q2: b();               snyc void c() {
q3: c();               q8: a();
q4: return;            q9: return
}                      }
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

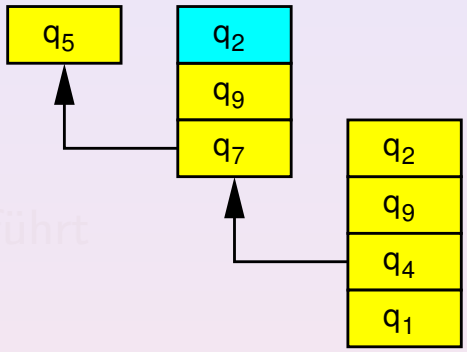
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {
q0: a();
q1: return;
}
sync void a() {
q2: b();
q3: c();
q4: return;
}
```

```
async void b() {
q5: b();
q6: c();
q7: return
}
snyc void c() {
q8: a();
q9: return
}
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

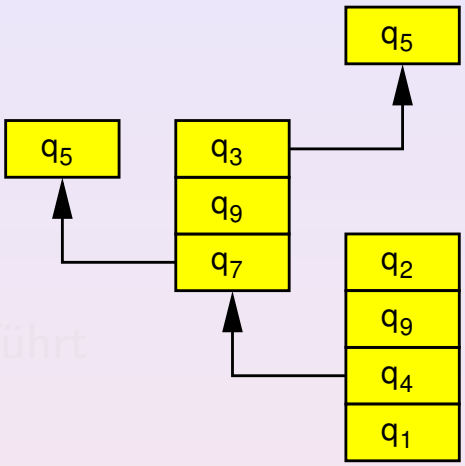
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {          async void b() {
q0: a();               q5: b();
q1: return;            q6: c();
}                      q7: return
sync void a() {        }
q2: b();               snyc void c() {
q3: c();               q8: a();
q4: return;            q9: return
}                      }
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

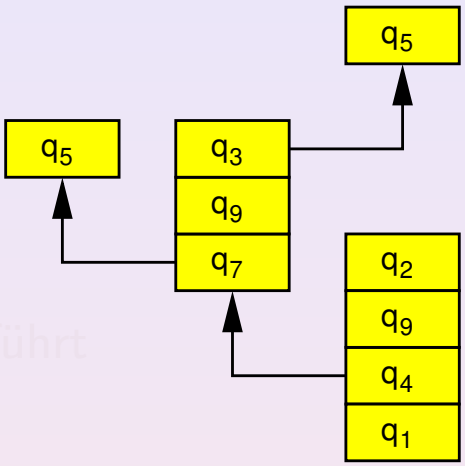
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {          async void b() {
q0: a();               q5: b();
q1: return;            q6: c();
}                      q7: return
sync void a() {        }
q2: b();               snyc void c() {
q3: c();               q8: a();
q4: return;            q9: return
}                      }
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

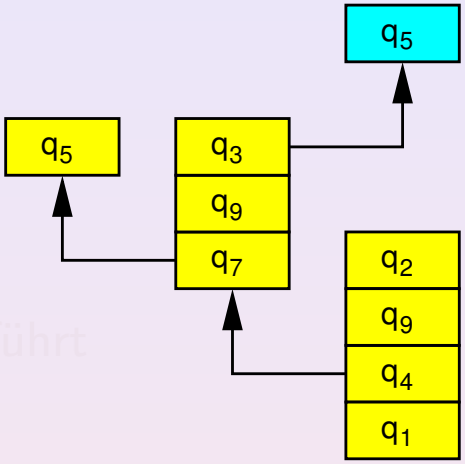
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {
q0: a();
q1: return;
}
sync void a() {
q2: b();
q3: c();
q4: return;
}
```

```
async void b() {
q5: b();
q6: c();
q7: return
}
snyc void c() {
q8: a();
q9: return
}
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

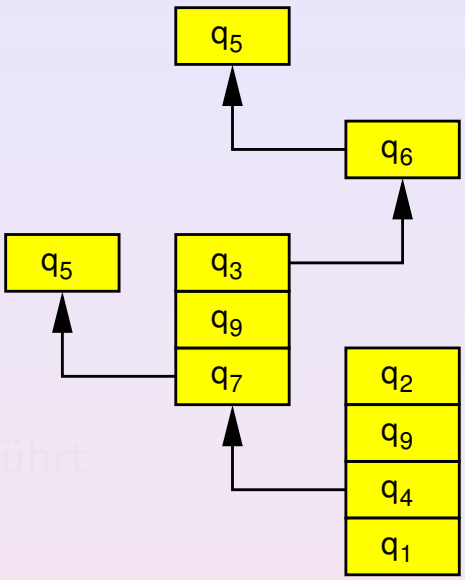
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {
q0: a();
q1: return;
}
sync void a() {
q2: b();
q3: c();
q4: return;
}
```

```
async void b() {
q5: b();
q6: c();
q7: return
}
snyc void c() {
q8: a();
q9: return
}
```

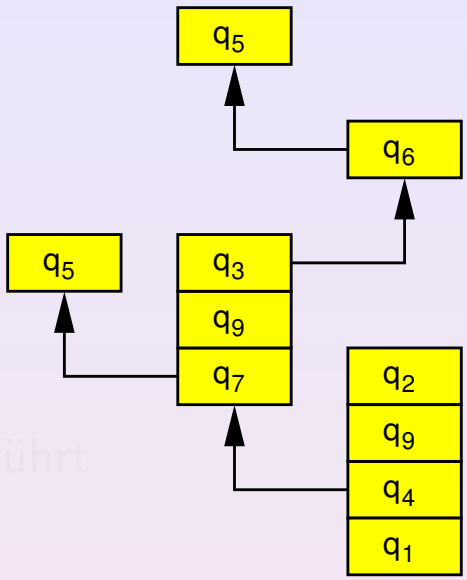
3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

⇒ Zustand (Keller) ist Folge von Programmpunkten



Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt

```
void main() {
q0: a();
q1: return;
}
sync void a() {
q2: b();
q3: c();
q4: return;
}
```

```
async void b() {
q5: b();
q6: c();
q7: return
}
snyc void c() {
q8: a();
q9: return
}
```

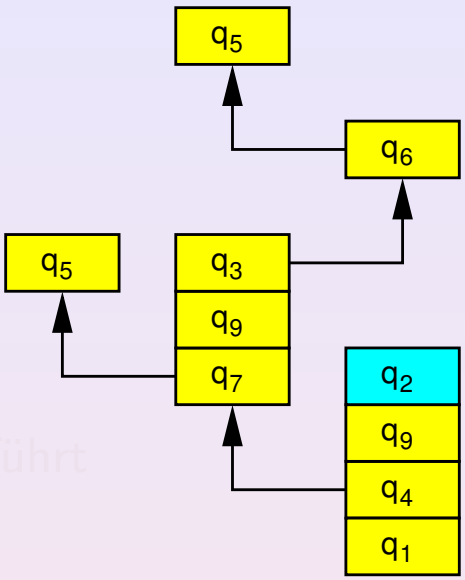
3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

⇒ Zustand (Keller) ist Folge von Programmpunkten



Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt

```
void main() {          async void b() {
q0: a();               q5: b();
q1: return;            q6: c();
}                      q7: return
sync void a() {        }
q2: b();               snyc void c() {
q3: c();               q8: a();
q4: return;            q9: return
}                      }
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

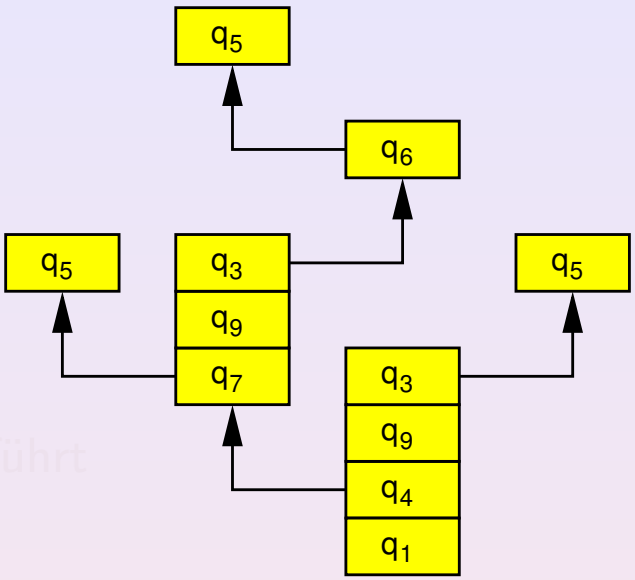
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {
q0: a();
q1: return;
}
sync void a() {
q2: b();
q3: c();
q4: return;
}
```

```
async void b() {
q5: b();
q6: c();
q7: return
}
snyc void c() {
q8: a();
q9: return
}
```

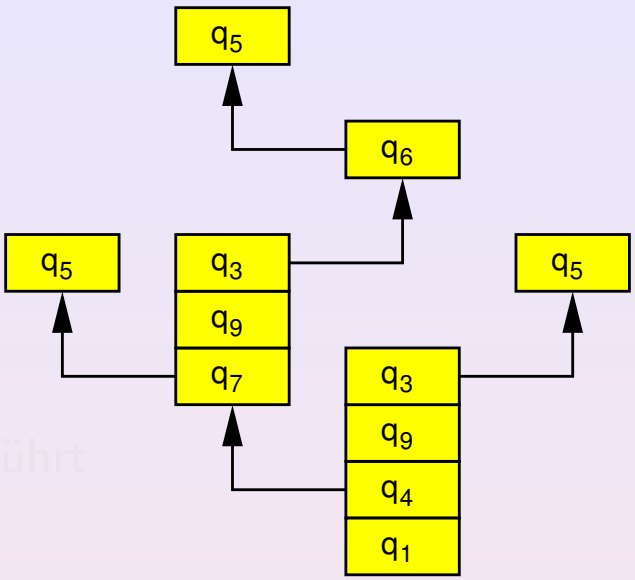
3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

⇒ Zustand (Keller) ist Folge von Programmpunkten



Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt

```
void main() {          async void b() {
q0: a();               q5: b();
q1: return;            q6: c();
}                      q7: return
sync void a() {        }
q2: b();               snyc void c() {
q3: c();               q8: a();
q4: return;            q9: return
}                      }
```

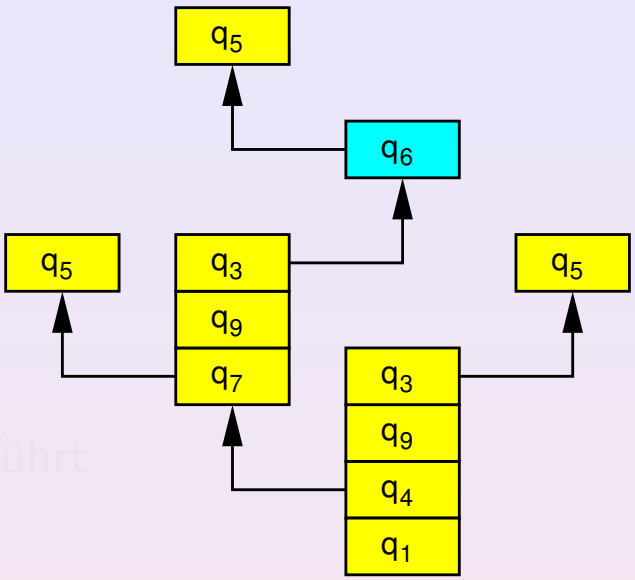
3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

⇒ Zustand (Keller) ist Folge von Programmpunkten



Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt

```
void main() {
q0: a();
q1: return;
}
sync void a() {
q2: b();
q3: c();
q4: return;
}
```

```
async void b() {
q5: b();
q6: c();
q7: return
}
snyc void c() {
q8: a();
q9: return
}
```

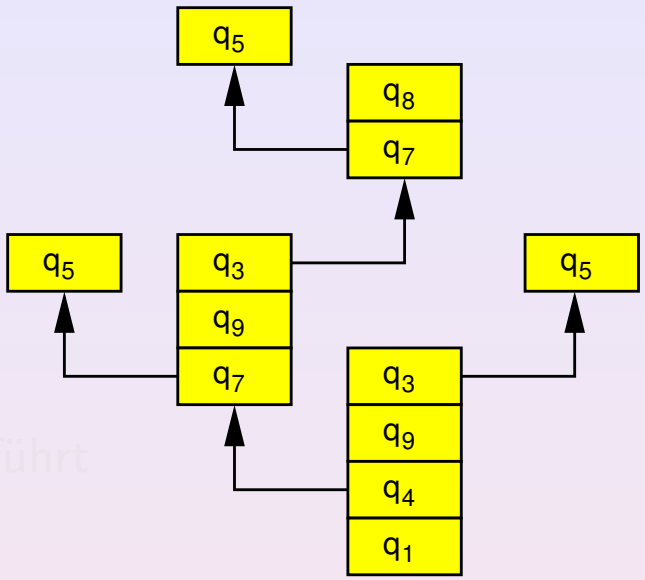
3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

⇒ Zustand (Keller) ist Folge von Programmpunkten



Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt

```
void main() {          async void b() {
q0: a();               q5: b();
q1: return;            q6: c();
}                      q7: return
sync void a() {        }
q2: b();               snyc void c() {
q3: c();               q8: a();
q4: return;            q9: return
}                      }
```

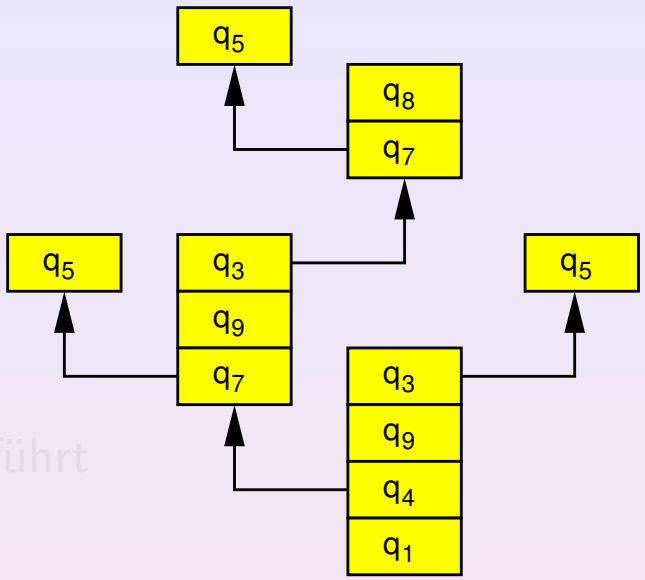
3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

⇒ Zustand (Keller) ist Folge von Programmpunkten



Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt

```
void main() {
q0: a();
q1: return;
}
sync void a() {
q2: b();
q3: c();
q4: return;
}
```

```
async void b() {
q5: b();
q6: c();
q7: return
}
snyc void c() {
q8: a();
q9: return
}
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

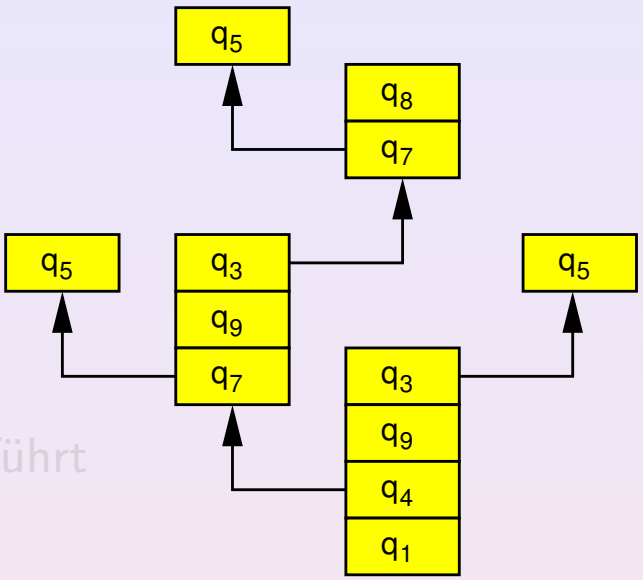
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {
q0: a();
q1: return;
}
sync void a() {
q2: b();
q3: c();
q4: return;
}
```

```
async void b() {
q5: b();
q6: c();
q7: return
}
snyc void c() {
q8: a();
q9: return
}
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

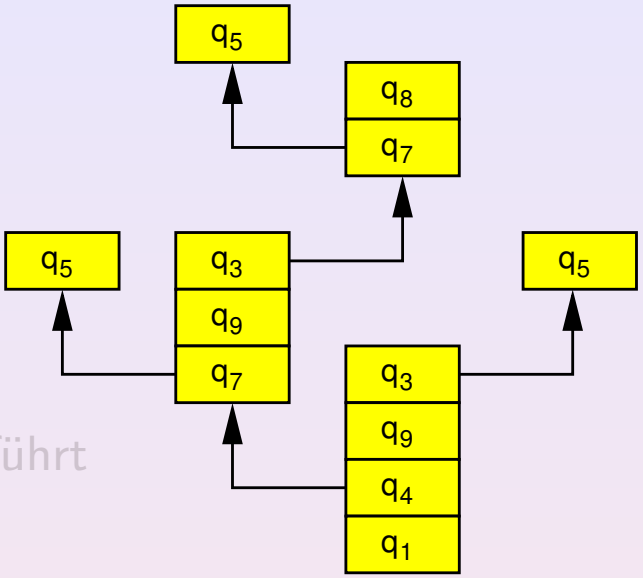
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {          async void b() {
q0: a();               q5: b();
q1: return;            q6: c();
}                      q7: return
sync void a() {        }
q2: b();               snyc void c() {
q3: c();               q8: a();
q4: return;            q9: return
}                      }
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

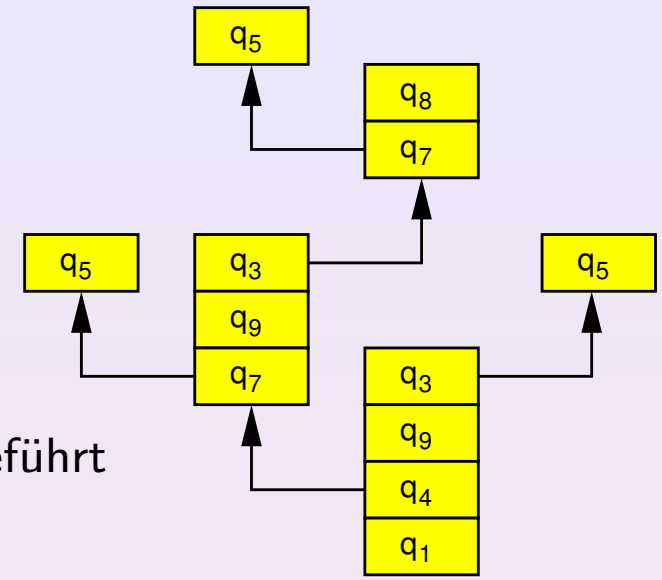
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {
q0: a();
q1: return;
}
sync void a() {
q2: b();
q3: c();
q4: return;
}
```

```
async void b() {
q5: b();
q6: c();
q7: return
}
snyc void c() {
q8: a();
q9: return
}
```

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

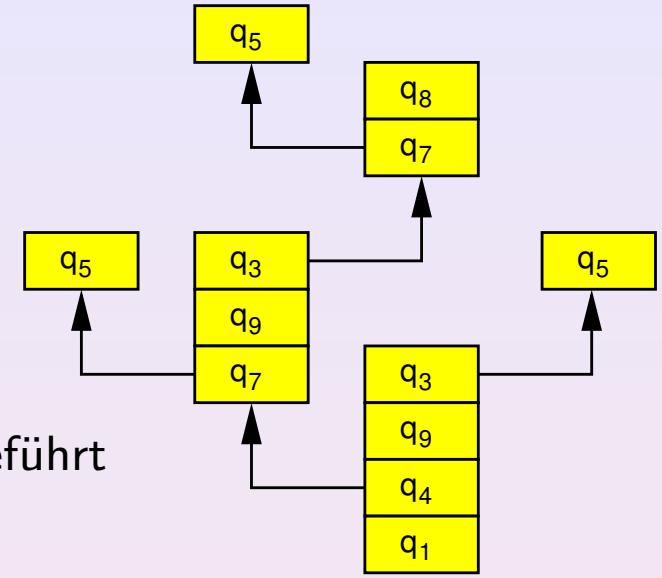
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {        async void b() {
q0: a();             q5: b();
q1: return;          q6: c();
}                    q7: return
sync void a() {      }
q2: b();             snyc void c() {
```

Beobachtungen

- Zustandsübergangssystem Π definiert formale Sprache: $L(\Pi) \triangleq \{w \in \Sigma^* : q_0 \overset{w}{\Rightarrow} \varepsilon\}$ wobei q_0 Anfangszustand und Σ Signaturen aller Komponenten-/Serviceschnittstellen
- Für Komponentenprotokollprüfung betrachtet man nur die Schnittstellen einer Komponente C mit Protokoll A_C und prüft, ob $L(\Pi) \subseteq L(A)$

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

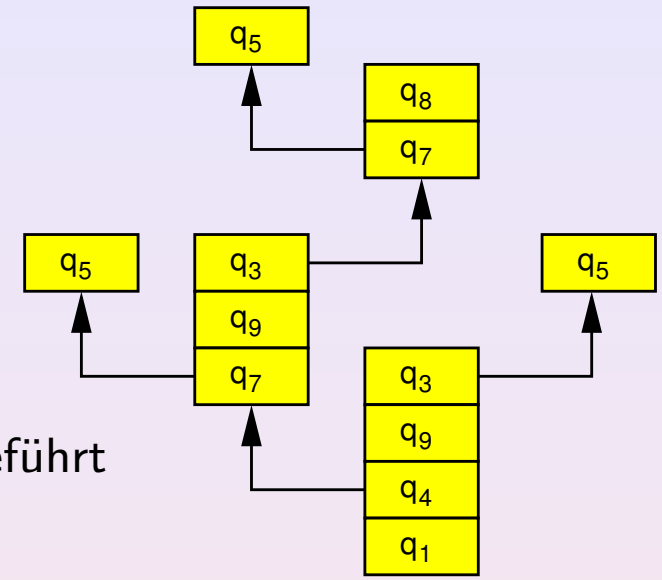
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt




```
void main() {        async void b() {
q0: a();             q5: b();
q1: return;          q6: c();
}                    q7: return
sync void a() {      }
q2: b();             snyc void c() {
```

Beobachtungen

- Zustandsübergangssystem Π definiert formale Sprache: $L(\Pi) \triangleq \{w \in \Sigma^* : q_0 \overset{w}{\Rightarrow} \varepsilon\}$ wobei q_0 Anfangszustand und Σ Signaturen aller Komponenten-/Serviceschnittstellen
- Für Komponentenprotokollprüfung betrachtet man nur die Schnittstellen einer Komponente C mit Protokoll A_C und prüft, ob $L(\Pi) \subseteq L(A)$

3. Modellierung von Rekursion und Nebenläufigkeit

Rekursive Programme: Verhaltensmodellierung durch Kellerautomaten

- Wir betrachten nur Steuerfluss

⇒ Kellerinhalt enthält Programmpunkte

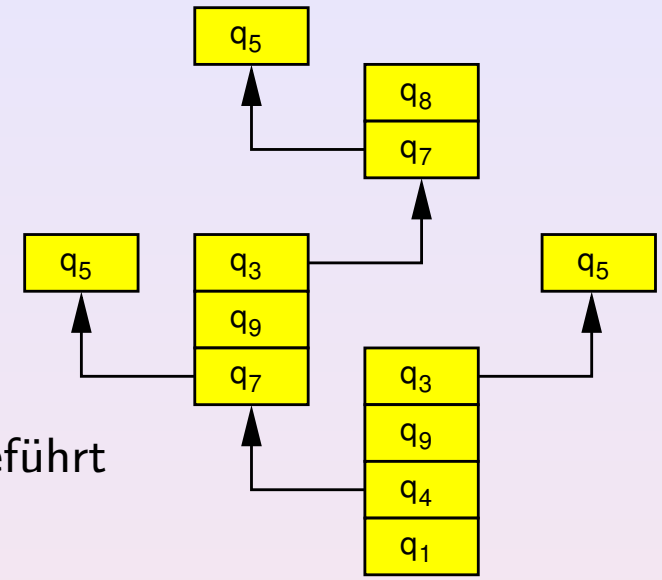
⇒ Zustand (Keller) ist Folge von Programmpunkten

Rekursive und Nebenläufige Programme:

- Zustand ist nun Kaktuskeller von Programmpunkten

⇒ Kaktuskeller wird überführt in Kaktuskeller

☞ Bei Zustandsübergängen werden Serviceaufrufe mitgeführt



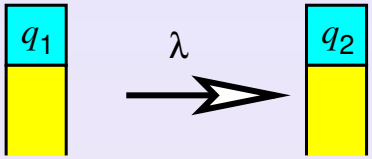

```
void main() {        async void b() {
q0: a();             q5: b();
q1: return;          q6: c();
}                    q7: return
sync void a() {      }
q2: b();             snyc void c() {
```

Beobachtungen

- Zustandsübergangssystem Π definiert formale Sprache: $L(\Pi) \triangleq \{w \in \Sigma^* : q_0 \overset{w}{\Rightarrow} \varepsilon\}$ wobei q_0 Anfangszustand und Σ Signaturen aller Komponenten-/Serviceschnittstellen
- Für Komponentenprotokollprüfung betrachtet man nur die Schnittstellen einer Komponente C mit Protokoll A_C und prüft, ob $L(\Pi) \subseteq L(A)$

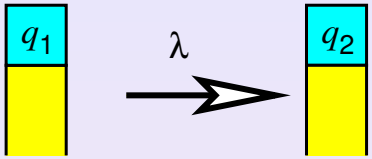
Klassifikation der Zustandsübergänge

Anweisungen ohne Prozeduraufruf/ –rückkehr

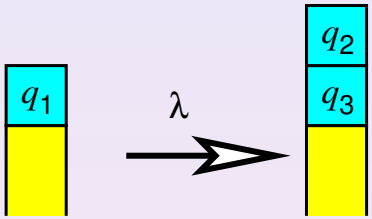


Klassifikation der Zustandsübergänge

Anweisungen ohne Prozeduraufruf/ –rückkehr

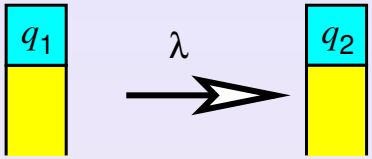


synchroner Prozeduraufruf (intern)

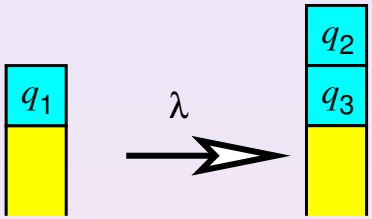


Klassifikation der Zustandsübergänge

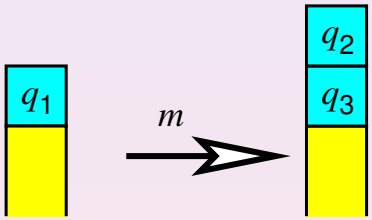
Anweisungen ohne Prozeduraufruf/ –rückkehr



synchroner Prozeduraufruf (intern)

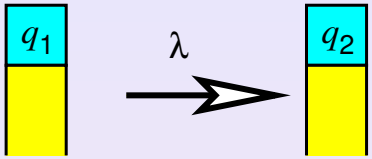


synchroner Prozeduraufruf (extern)

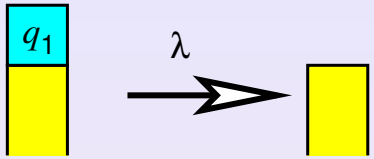


Klassifikation der Zustandsübergänge

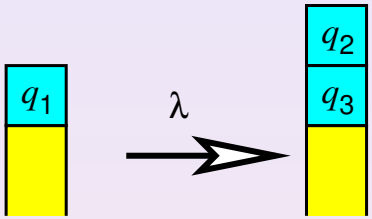
Anweisungen ohne Prozeduraufruf/ –rückkehr



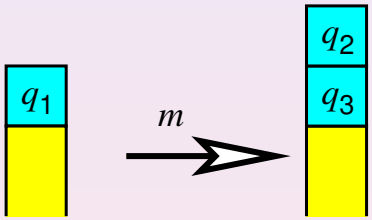
Prozedur– rückkehr



synchroner Prozeduraufruf (intern)

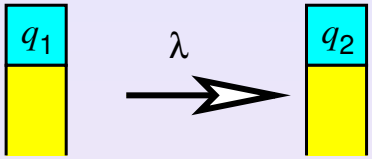


synchroner Prozeduraufruf (extern)

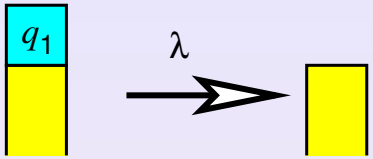


Klassifikation der Zustandsübergänge

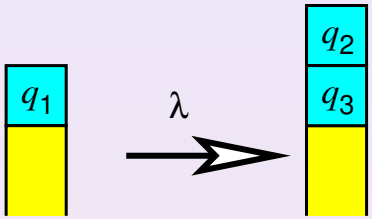
Anweisungen ohne Prozeduraufruf/ –rückkehr



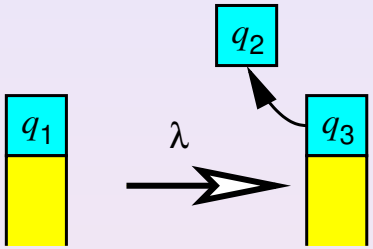
Prozedur– rückkehr



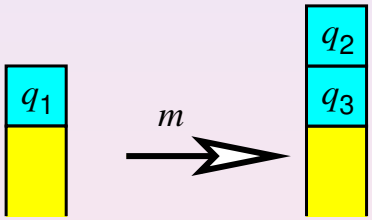
synchroner Prozeduraufruf (intern)



asynchroner Prozeduraufruf (intern) oder Prozessstart

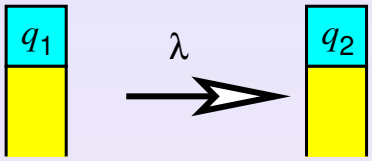


synchroner Prozeduraufruf (extern)

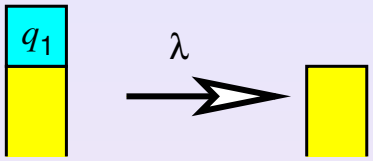


Klassifikation der Zustandsübergänge

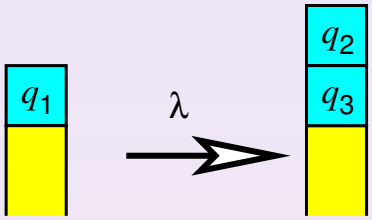
Anweisungen ohne Prozeduraufruf/ –rückkehr



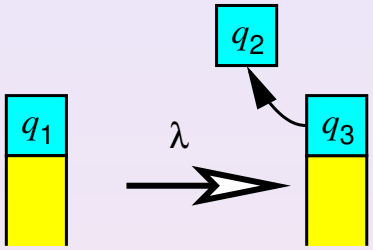
Prozedur– rückkehr



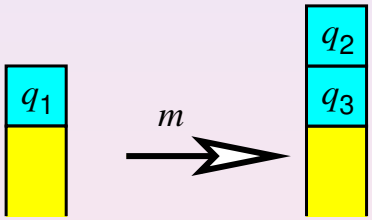
synchroner Prozeduraufruf (intern)



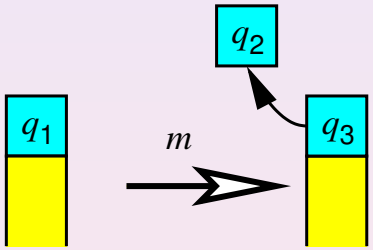
asynchroner Prozeduraufruf (intern) oder Prozessstart



synchroner Prozeduraufruf (extern)

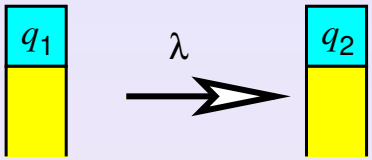


asynchroner Prozeduraufruf (extern)



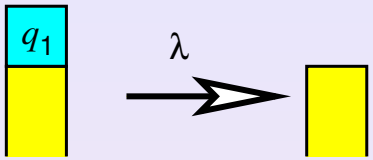
Klassifikation der Zustandsübergänge

Anweisungen ohne Prozeduraufruf/ –rückkehr



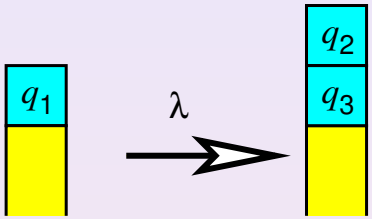
Klasse (1,1)

Prozedur– rückkehr

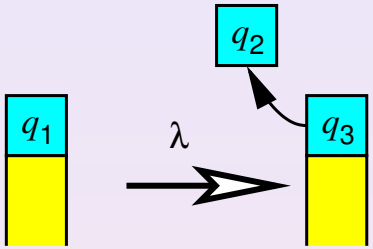


Klasse (1,1)

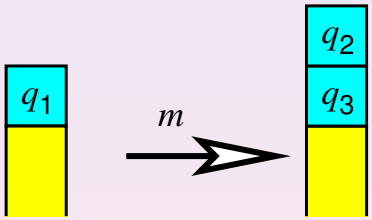
synchroner Prozeduraufruf (intern)



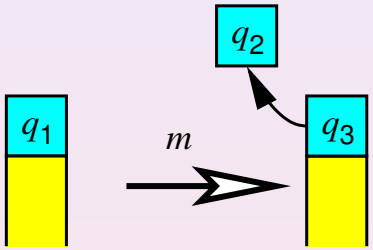
asynchroner Prozeduraufruf (intern) oder Prozessstart



synchroner Prozeduraufruf (extern)

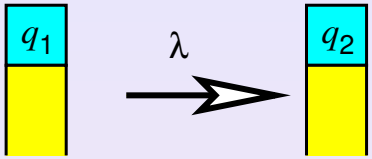


asynchroner Prozeduraufruf (extern)



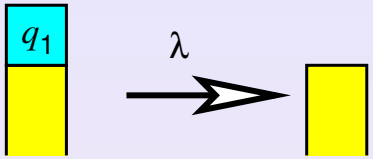
Klassifikation der Zustandsübergänge

Anweisungen ohne Prozeduraufruf/ –rückkehr



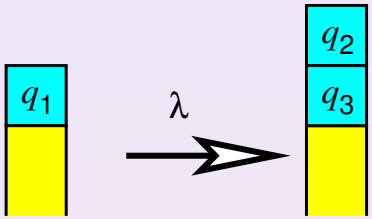
Klasse (1,1)

Prozedur– rückkehr



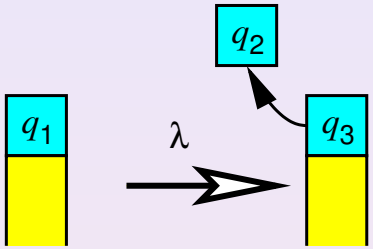
Klasse (1,1)

synchroner Prozeduraufruf (intern)

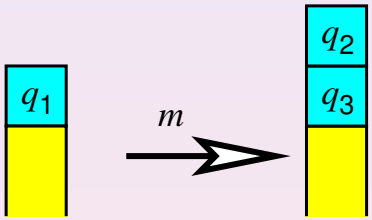


Klasse (1,S)

asynchroner Prozeduraufruf (intern) oder Prozessstart

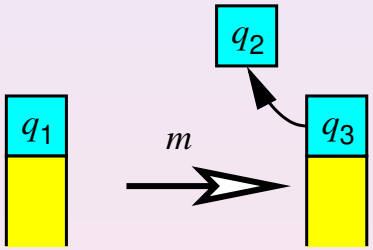


synchroner Prozeduraufruf (extern)



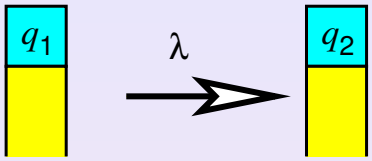
Klasse (1,S)

asynchroner Prozeduraufruf (extern)



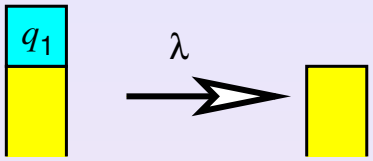
Klassifikation der Zustandsübergänge

Anweisungen ohne Prozeduraufruf/ –rückkehr



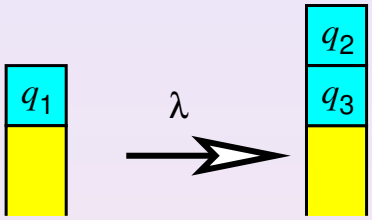
Klasse (1,1)

Prozedur– rückkehr



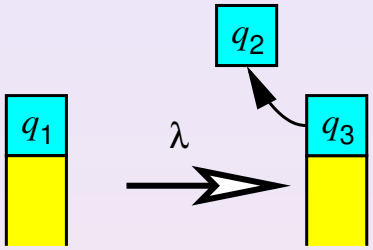
Klasse (1,1)

synchroner Prozeduraufruf (intern)



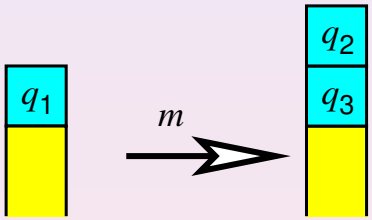
Klasse (1,S)

asynchroner Prozeduraufruf (intern) oder Prozessstart



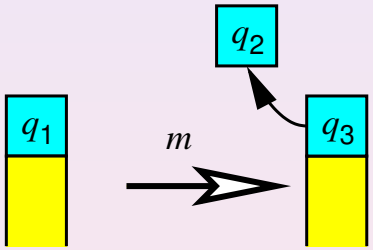
Klasse (1,P)

synchroner Prozeduraufruf (extern)



Klasse (1,S)

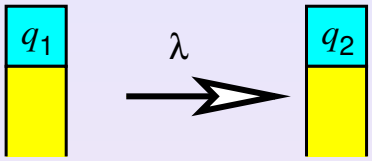
asynchroner Prozeduraufruf (extern)



Klasse (1,P)

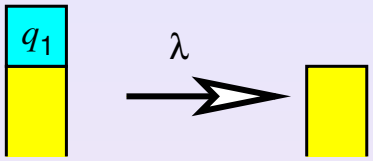
Klassifikation der Zustandsübergänge

Anweisungen ohne Prozeduraufruf/ –rückkehr



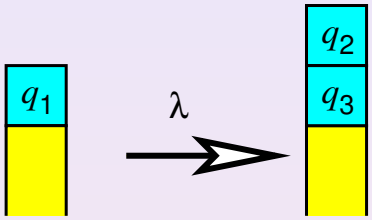
Klasse (1,1)

Prozedur– rückkehr



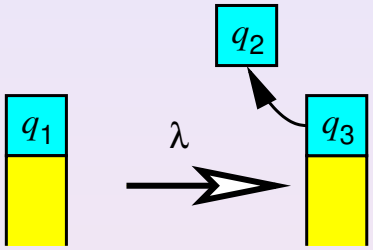
Klasse (1,1)

synchroner Prozeduraufruf (intern)



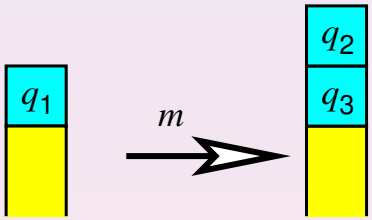
Klasse (1,S)

asynchroner Prozeduraufruf (intern) oder Prozessstart



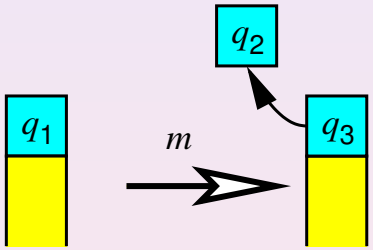
Klasse (1,P)

synchroner Prozeduraufruf (extern)



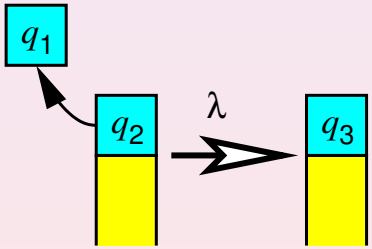
Klasse (1,S)

asynchroner Prozeduraufruf (extern)



Klasse (1,P)

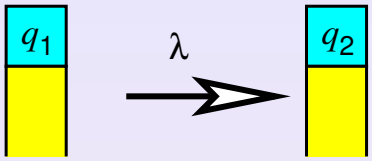
Synchronisation



Klasse (P,1)

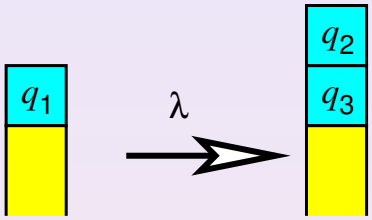
Klassifikation der Zustandsübergänge

Anweisungen ohne Prozeduraufruf/ –rückkehr



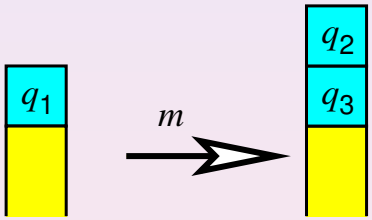
Klasse (1,1)

synchroner Prozeduraufruf (intern)



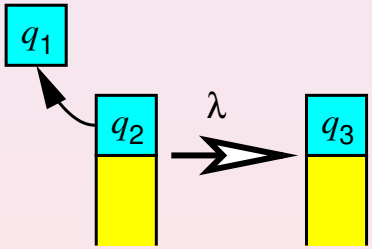
Klasse (1,S)

synchroner Prozeduraufruf (extern)



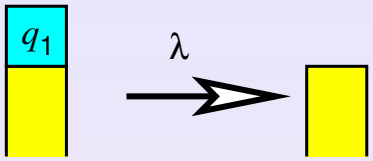
Klasse (1,S)

Synchronisation



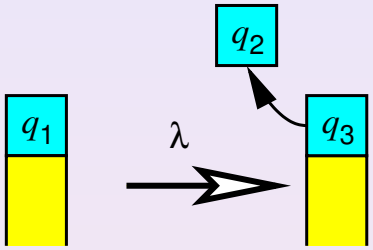
Klasse (P,1)

Prozedur– rückkehr



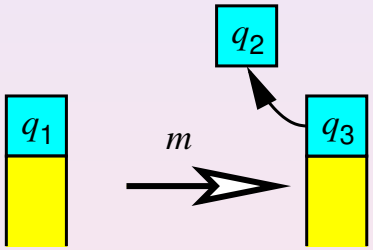
Klasse (1,1)

asynchroner Prozeduraufruf (intern) oder Prozessstart



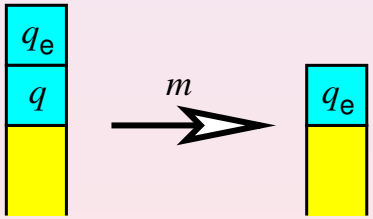
Klasse (1,P)

asynchroner Prozeduraufruf (extern)



Klasse (1,P)

Abbruch eines Aufrufs mit Ausnahme



Klasse (S,1)

Diskussion (Both, Zimmermann, Franke 2010)

Zustandsübergangssystem für Kaktuskeller entspricht Prozessersetzungssystemen nach Richard Mayr (1997)

(x, y)-PRS (Mayr 1997): Alle Zustandsübergänge gehören zu Klasse (x, y)
(G bedeutet keine Einschränkung)

Diskussion (Both, Zimmermann, Franke 2010)

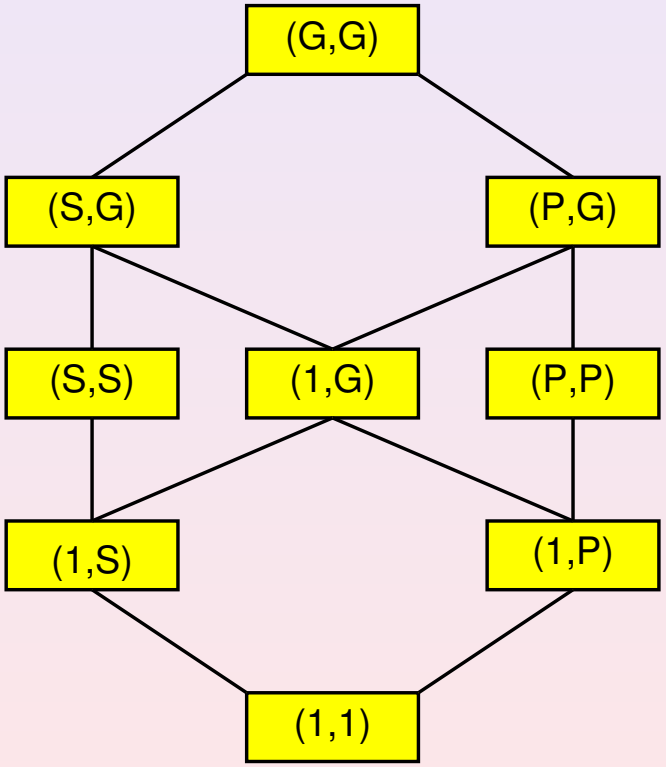
Zustandsübergangssystem für Kaktuskeller entspricht Prozessersetzungssystemen nach Richard Mayr (1997)

(x, y)-**PRS (Mayr 1997):** Alle Zustandsübergänge gehören zu Klasse (x, y)
(G bedeutet keine Einschränkung)

Diskussion (Both, Zimmermann, Franke 2010)

Zustandsübergangssystem für Kaktuskeller entspricht Prozessersetzungssystemen nach Richard Mayr (1997)

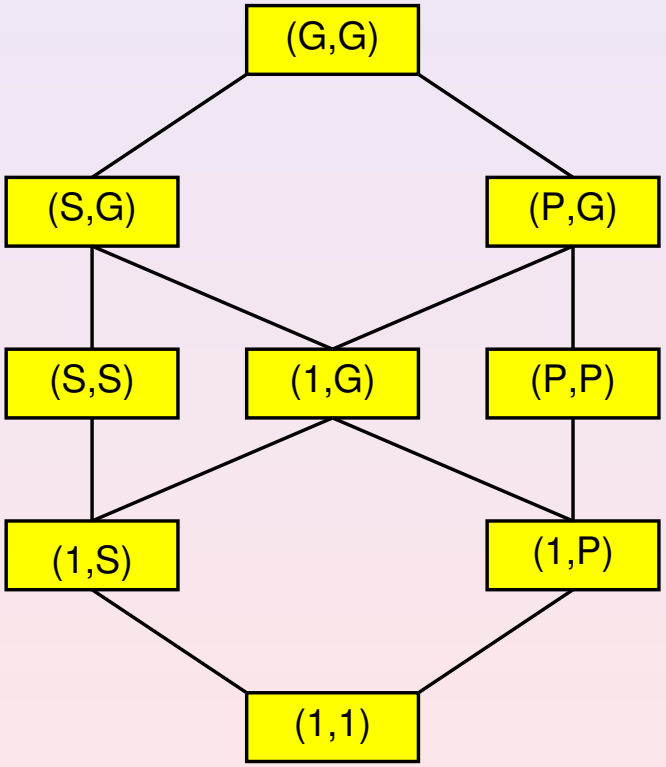
(x, y)-PRS (Mayr 1997): Alle Zustandsübergänge gehören zu Klasse (x, y)
(G bedeutet keine Einschränkung)



Diskussion (Both, Zimmermann, Franke 2010)

Zustandsübergangssystem für Kaktuskeller entspricht Prozessersetzungssystemen nach Richard Mayr (1997)

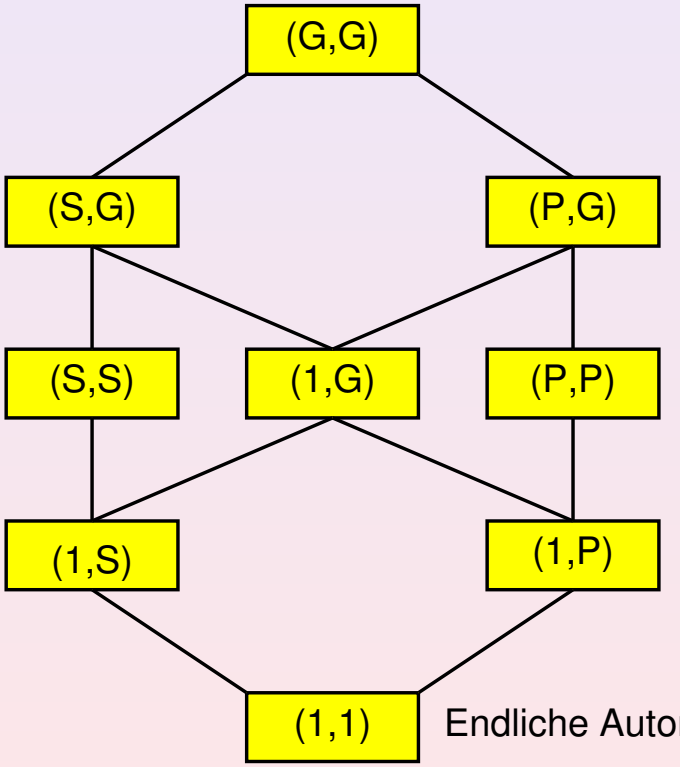
(x, y)-PRS (Mayr 1997): Alle Zustandsübergänge gehören zu Klasse (x, y) (G bedeutet keine Einschränkung)



Diskussion (Both, Zimmermann, Franke 2010)

Zustandsübergangssystem für Kaktuskeller entspricht Prozessersetzungssystemen nach Richard Mayr (1997)

(x, y)-PRS (Mayr 1997): Alle Zustandsübergänge gehören zu Klasse (x, y) (G bedeutet keine Einschränkung)

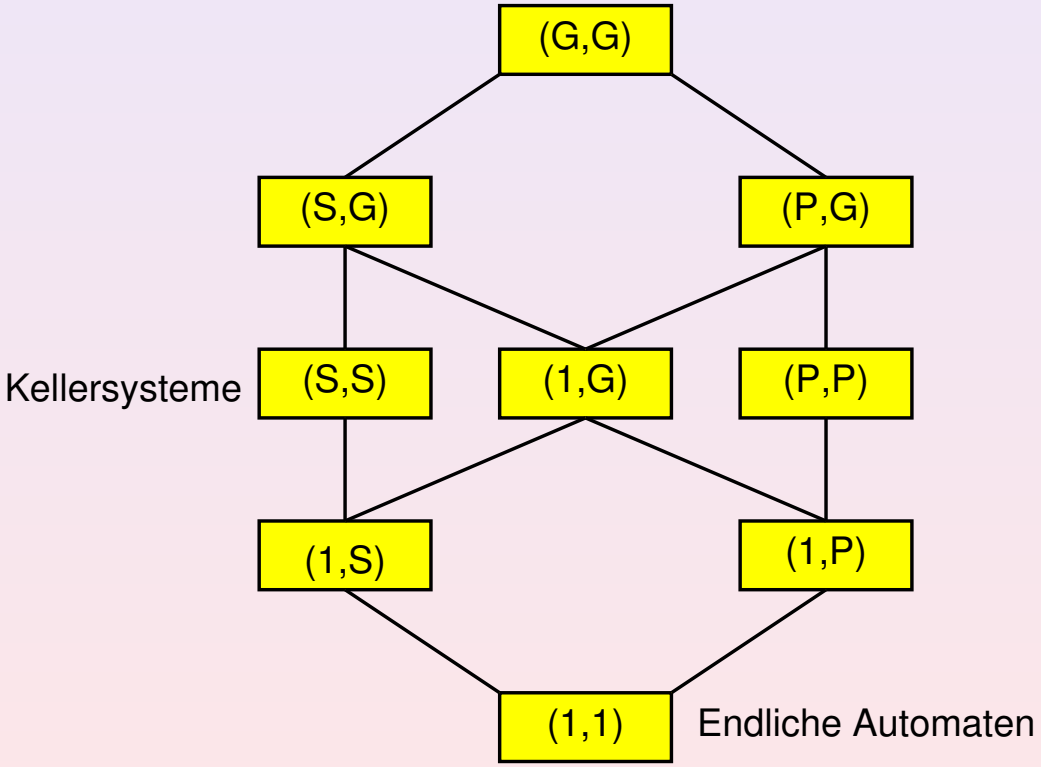


Endliche Automaten

Diskussion (Both, Zimmermann, Franke 2010)

Zustandsübergangssystem für Kaktuskeller entspricht Prozessersetzungssystemen nach Richard Mayr (1997)

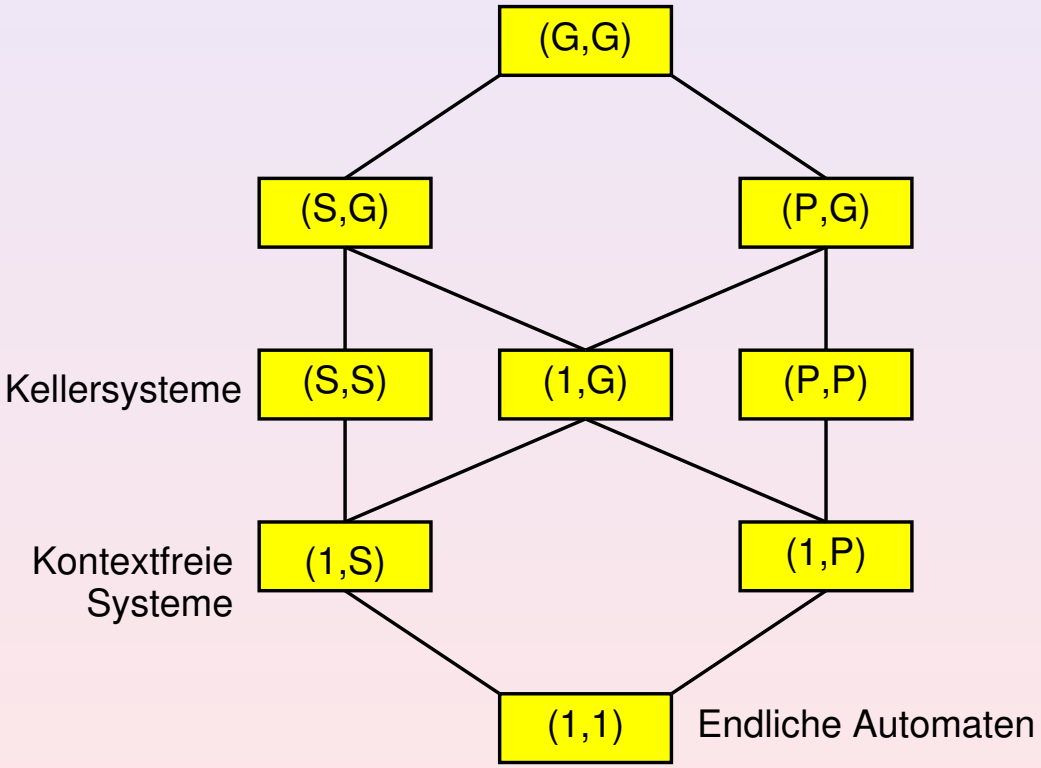
(x, y)-PRS (Mayr 1997): Alle Zustandsübergänge gehören zu Klasse (x, y) (G bedeutet keine Einschränkung)



Diskussion (Both, Zimmermann, Franke 2010)

Zustandsübergangssystem für Kaktuskeller entspricht Prozessersetzungssystemen nach Richard Mayr (1997)

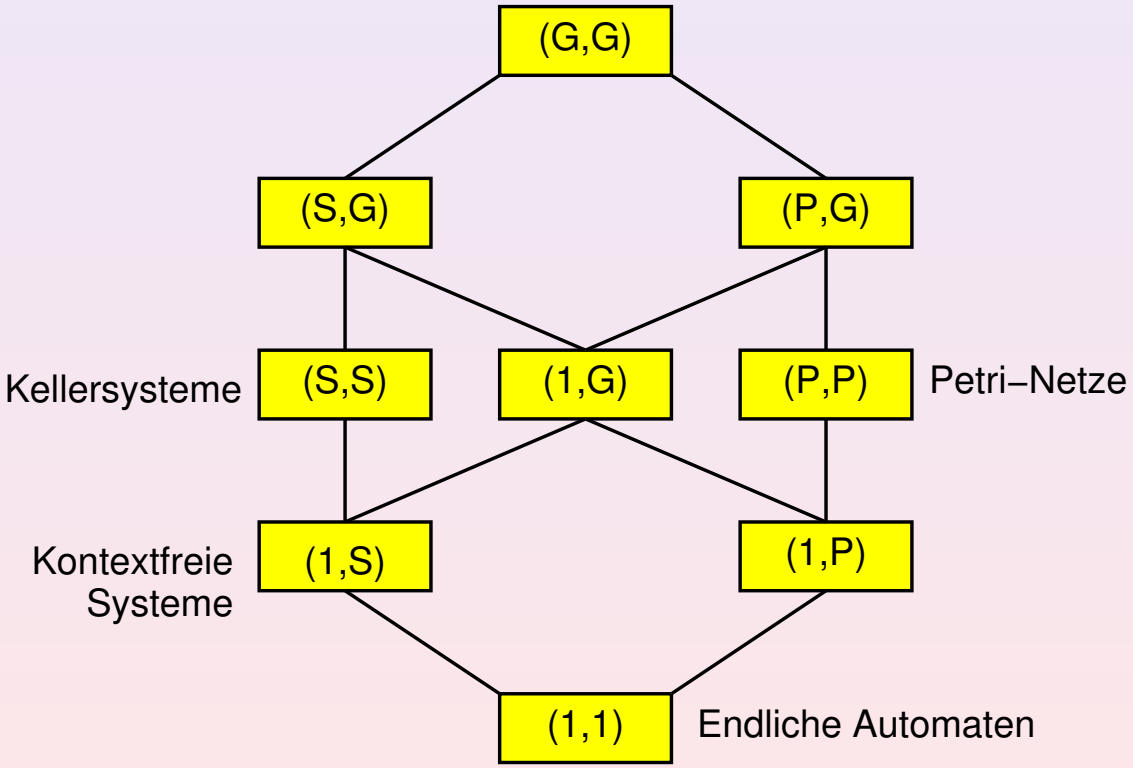
(x, y)-PRS (Mayr 1997): Alle Zustandsübergänge gehören zu Klasse (x, y) (G bedeutet keine Einschränkung)



Diskussion (Both, Zimmermann, Franke 2010)

Zustandsübergangssystem für Kaktuskeller entspricht Prozessersetzungssystemen nach Richard Mayr (1997)

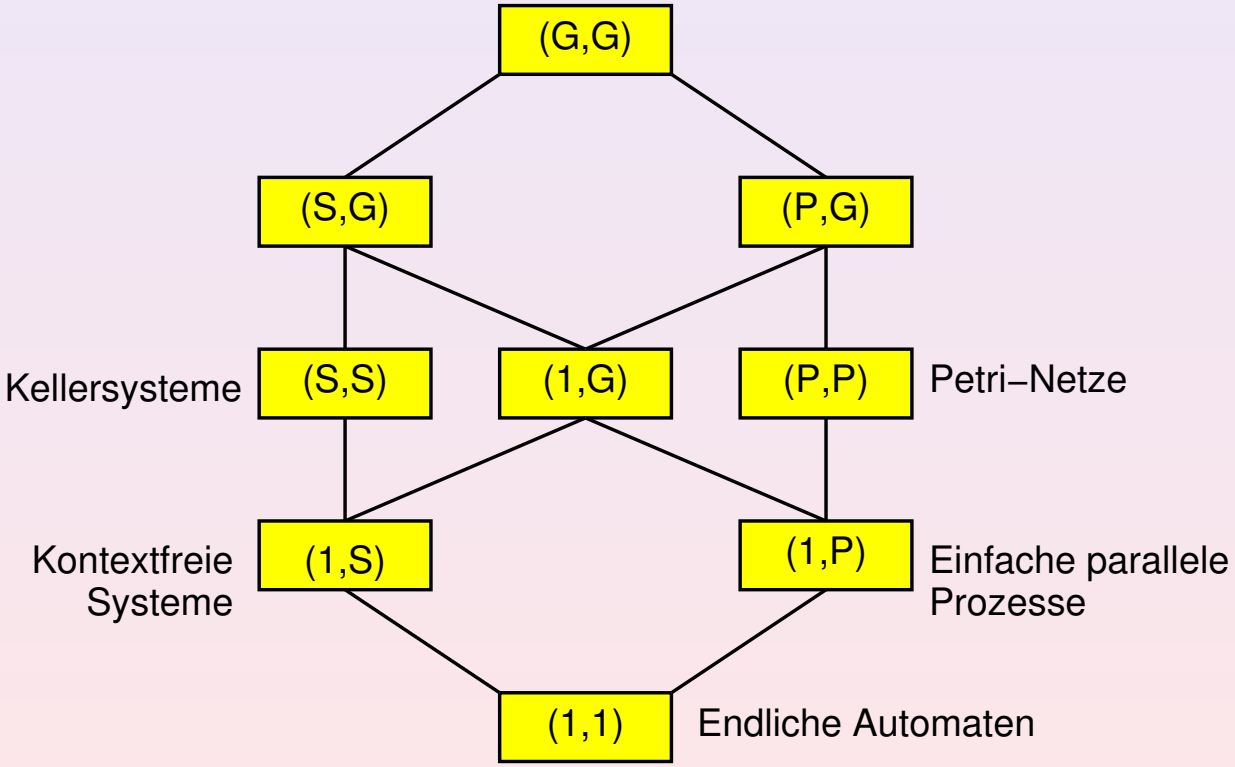
(x, y)-PRS (Mayr 1997): Alle Zustandsübergänge gehören zu Klasse (x, y) (G bedeutet keine Einschränkung)



Diskussion (Both, Zimmermann, Franke 2010)

Zustandsübergangssystem für Kaktuskeller entspricht Prozessersetzungssystemen nach Richard Mayr (1997)

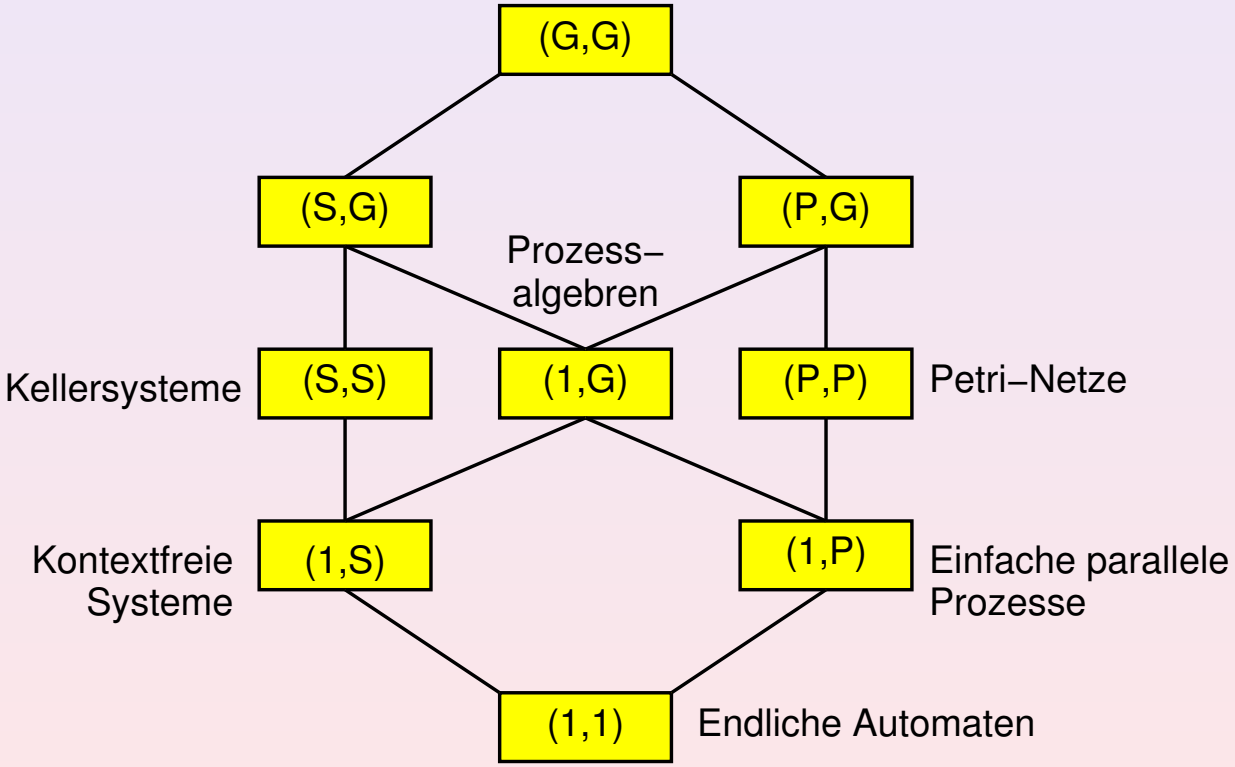
(x, y)-PRS (Mayr 1997): Alle Zustandsübergänge gehören zu Klasse (x, y) (G bedeutet keine Einschränkung)



Diskussion (Both, Zimmermann, Franke 2010)

Zustandsübergangssystem für Kaktuskeller entspricht Prozessersetzungssystemen nach Richard Mayr (1997)

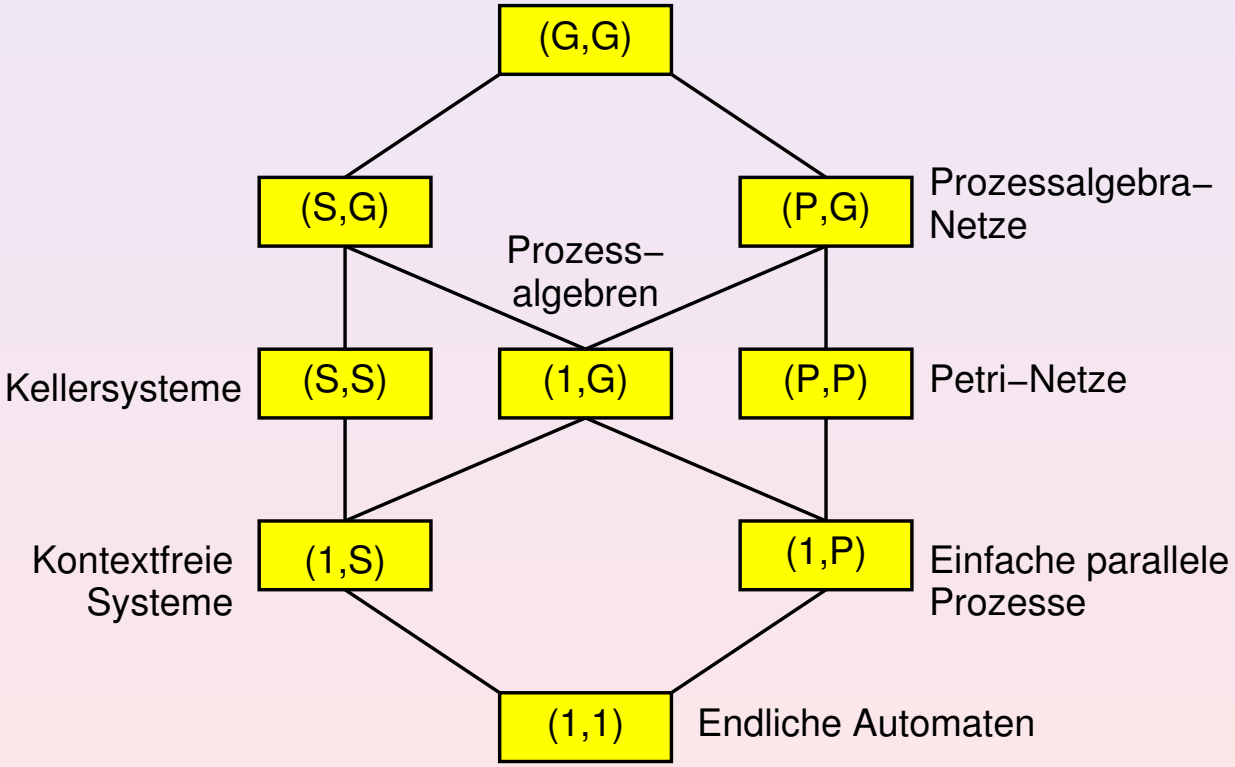
(x, y)-PRS (Mayr 1997): Alle Zustandsübergänge gehören zu Klasse (x, y) (G bedeutet keine Einschränkung)



Diskussion (Both, Zimmermann, Franke 2010)

Zustandsübergangssystem für Kaktuskeller entspricht Prozessersetzungssystemen nach Richard Mayr (1997)

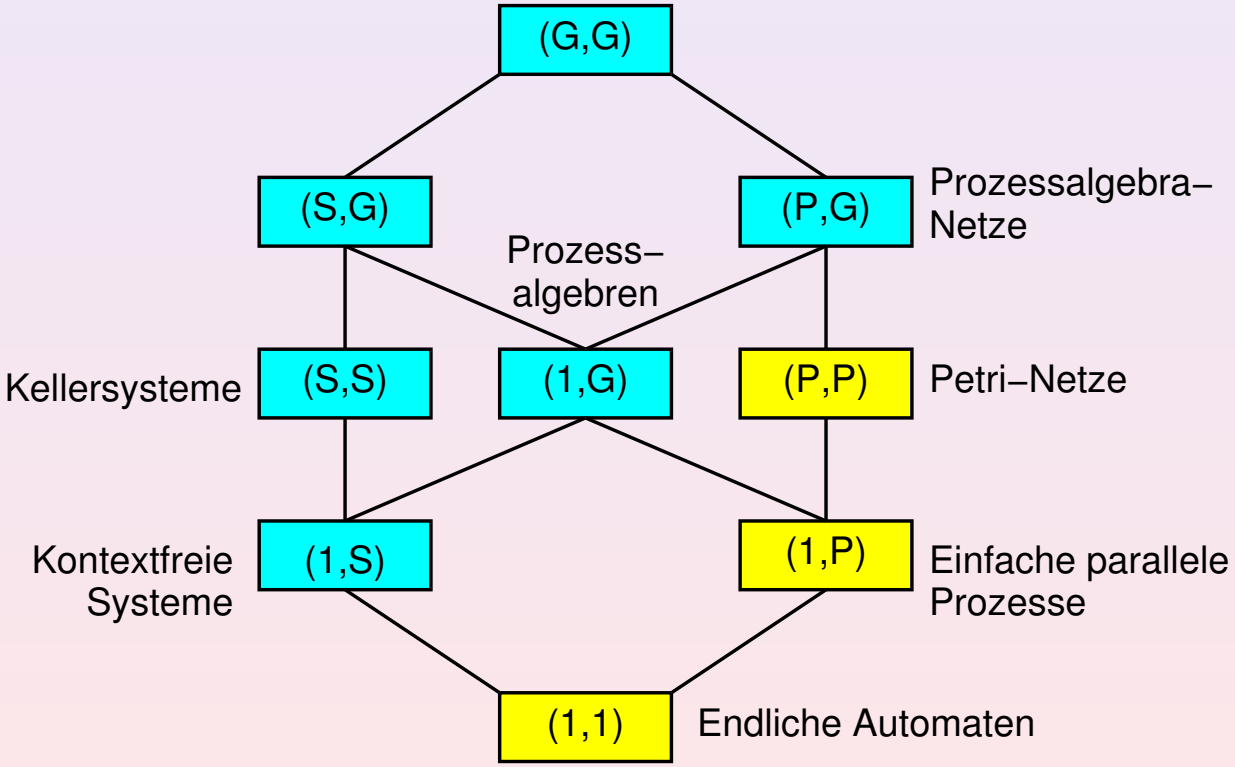
(x, y)-PRS (Mayr 1997): Alle Zustandsübergänge gehören zu Klasse (x, y) (G bedeutet keine Einschränkung)



Diskussion (Both, Zimmermann, Franke 2010)

Zustandsübergangssystem für Kaktuskeller entspricht Prozessersetzungssystemen nach Richard Mayr (1997)

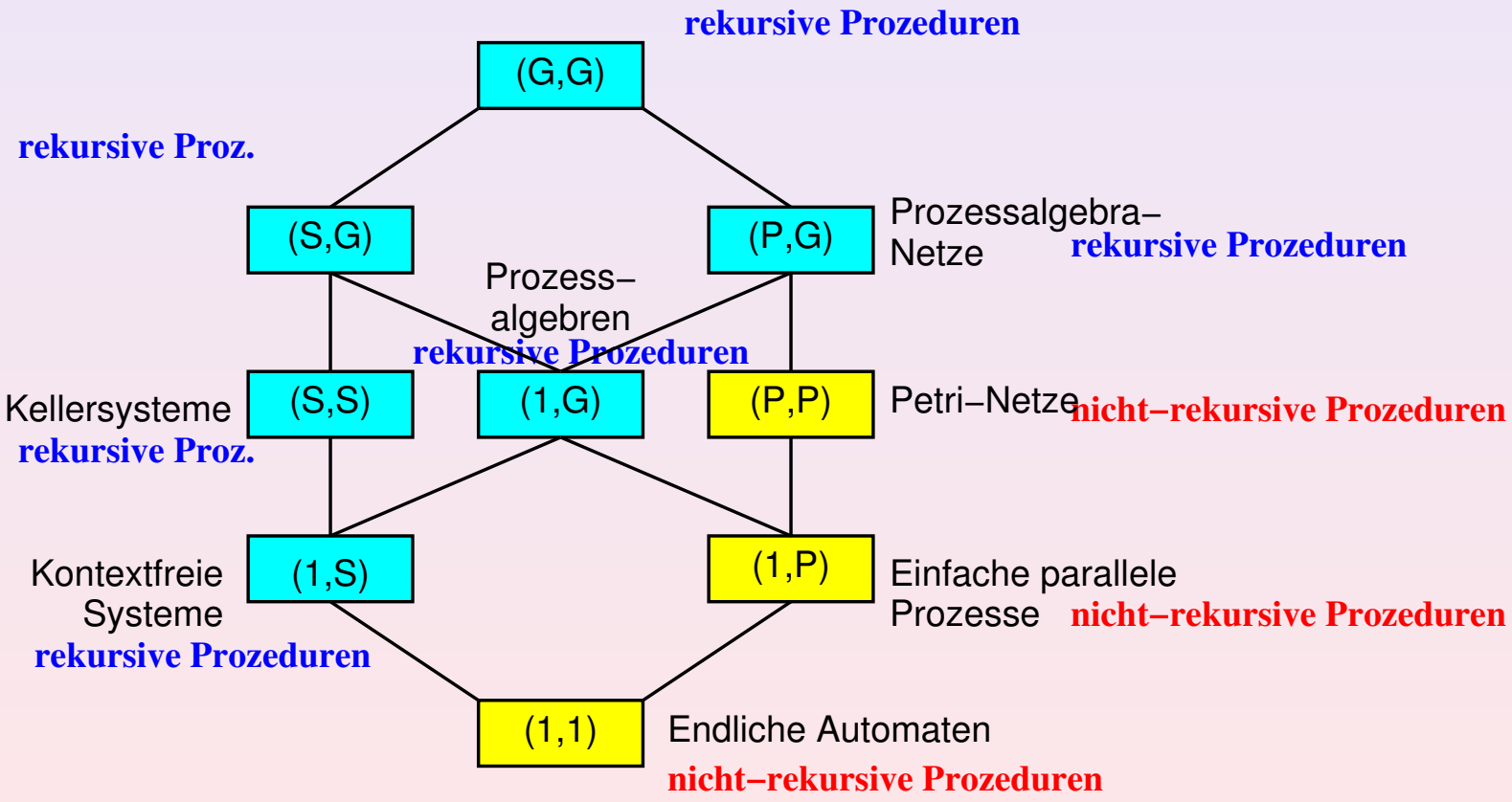
(x, y)-PRS (Mayr 1997): Alle Zustandsübergänge gehören zu Klasse (x, y) (G bedeutet keine Einschränkung)



Diskussion (Both, Zimmermann, Franke 2010)

Zustandsübergangssystem für Kaktuskeller entspricht Prozessersetzungssystemen nach Richard Mayr (1997)

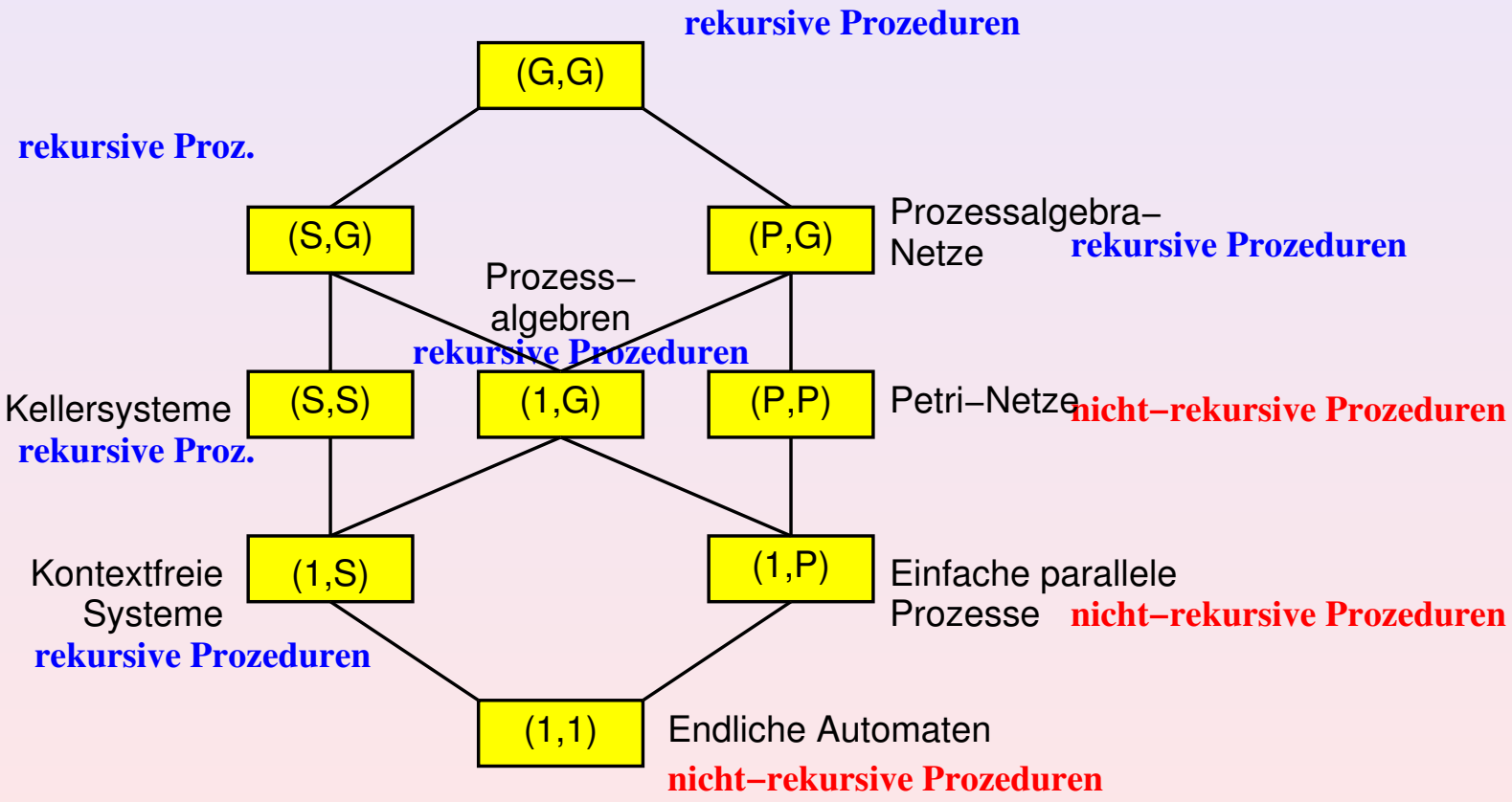
(x, y)-PRS (Mayr 1997): Alle Zustandsübergänge gehören zu Klasse (x, y) (G bedeutet keine Einschränkung)



Diskussion (Both, Zimmermann, Franke 2010)

Zustandsübergangssystem für Kaktuskeller entspricht Prozessersetzungssystemen nach Richard Mayr (1997)

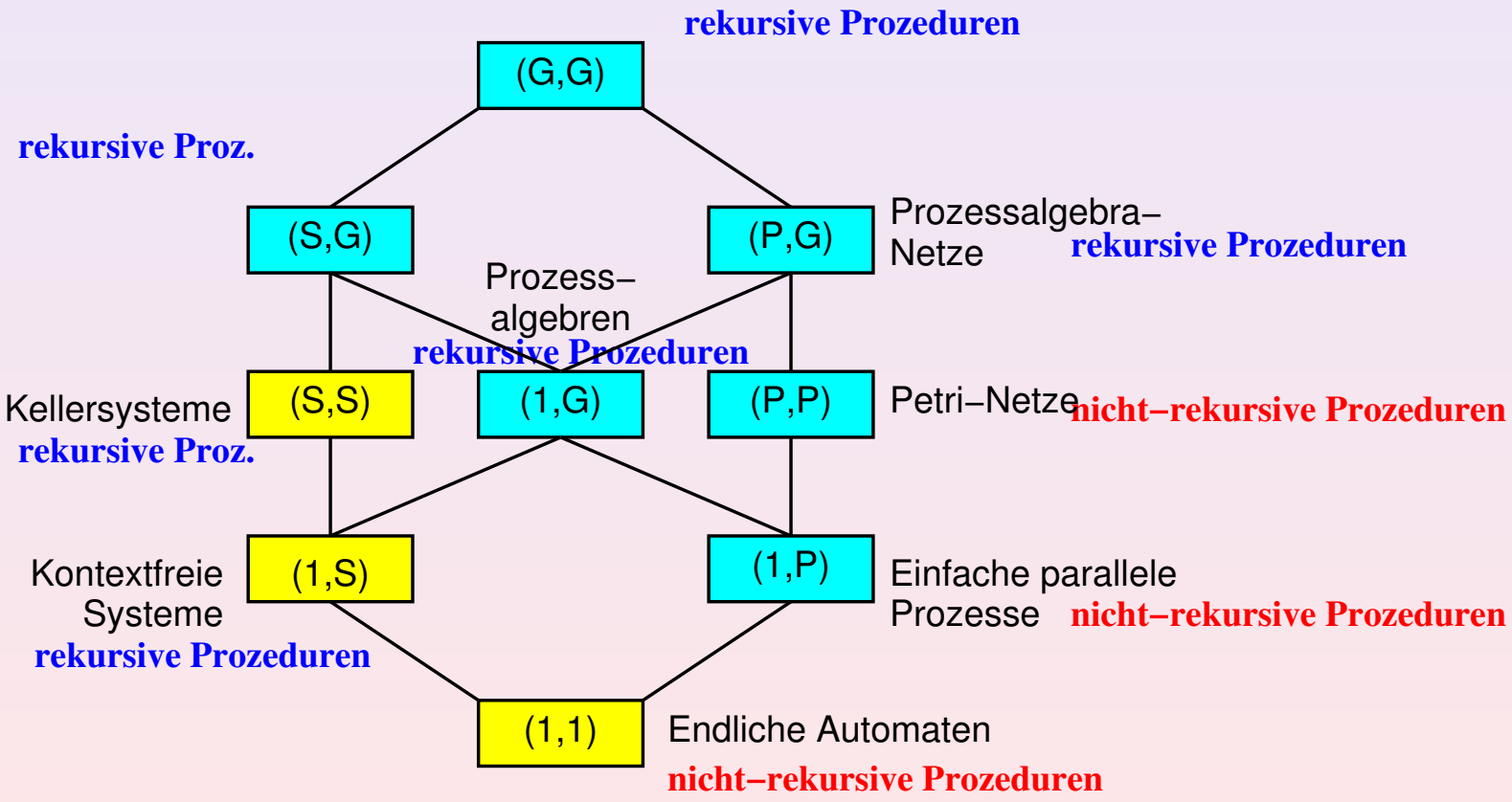
(x, y)-PRS (Mayr 1997): Alle Zustandsübergänge gehören zu Klasse (x, y) (G bedeutet keine Einschränkung)



Diskussion (Both, Zimmermann, Franke 2010)

Zustandsübergangssystem für Kaktuskeller entspricht Prozessersetzungssystemen nach Richard Mayr (1997)

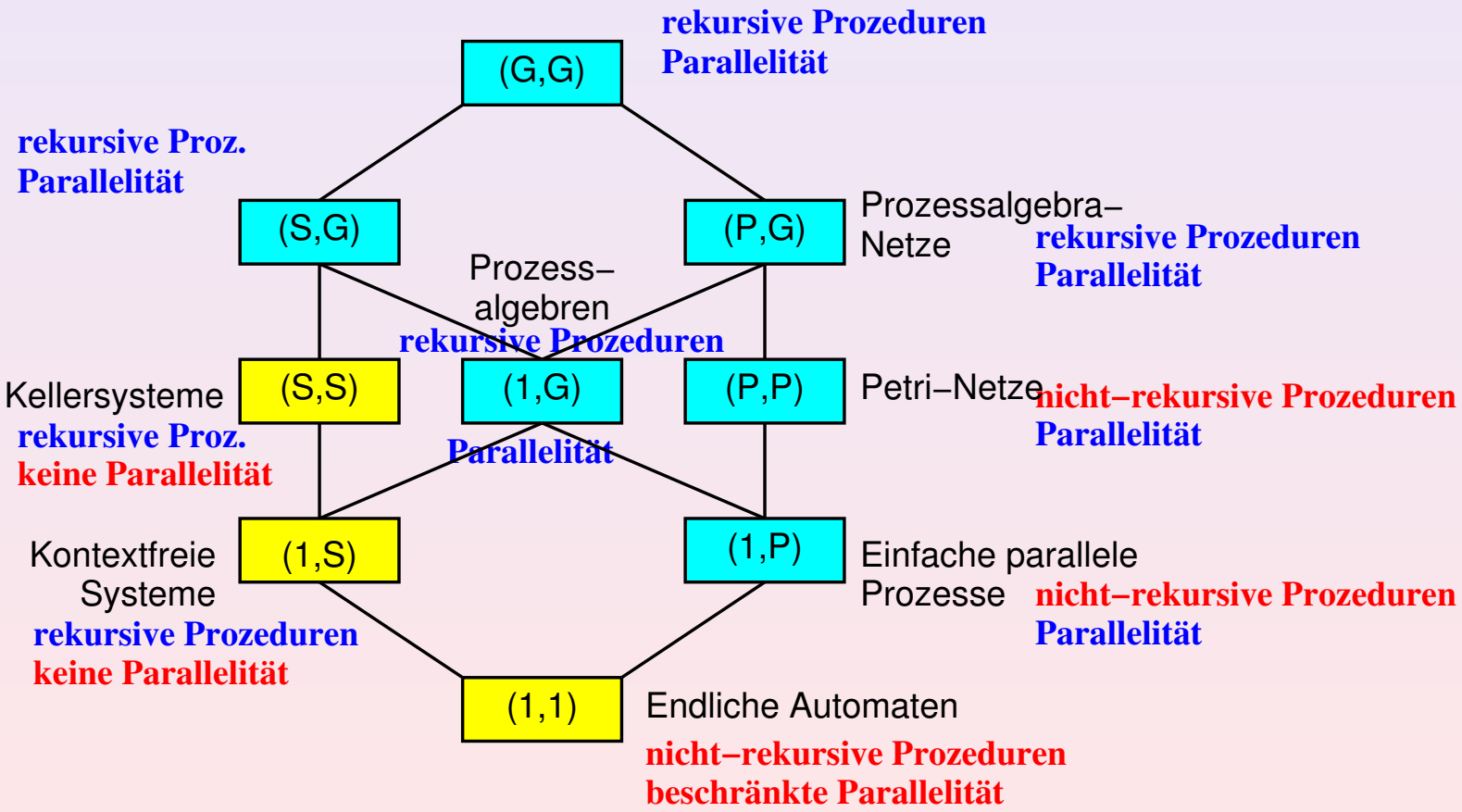
(x, y)-PRS (Mayr 1997): Alle Zustandsübergänge gehören zu Klasse (x, y) (G bedeutet keine Einschränkung)



Diskussion (Both, Zimmermann, Franke 2010)

Zustandsübergangssystem für Kaktuskeller entspricht Prozessersetzungssystemen nach Richard Mayr (1997)

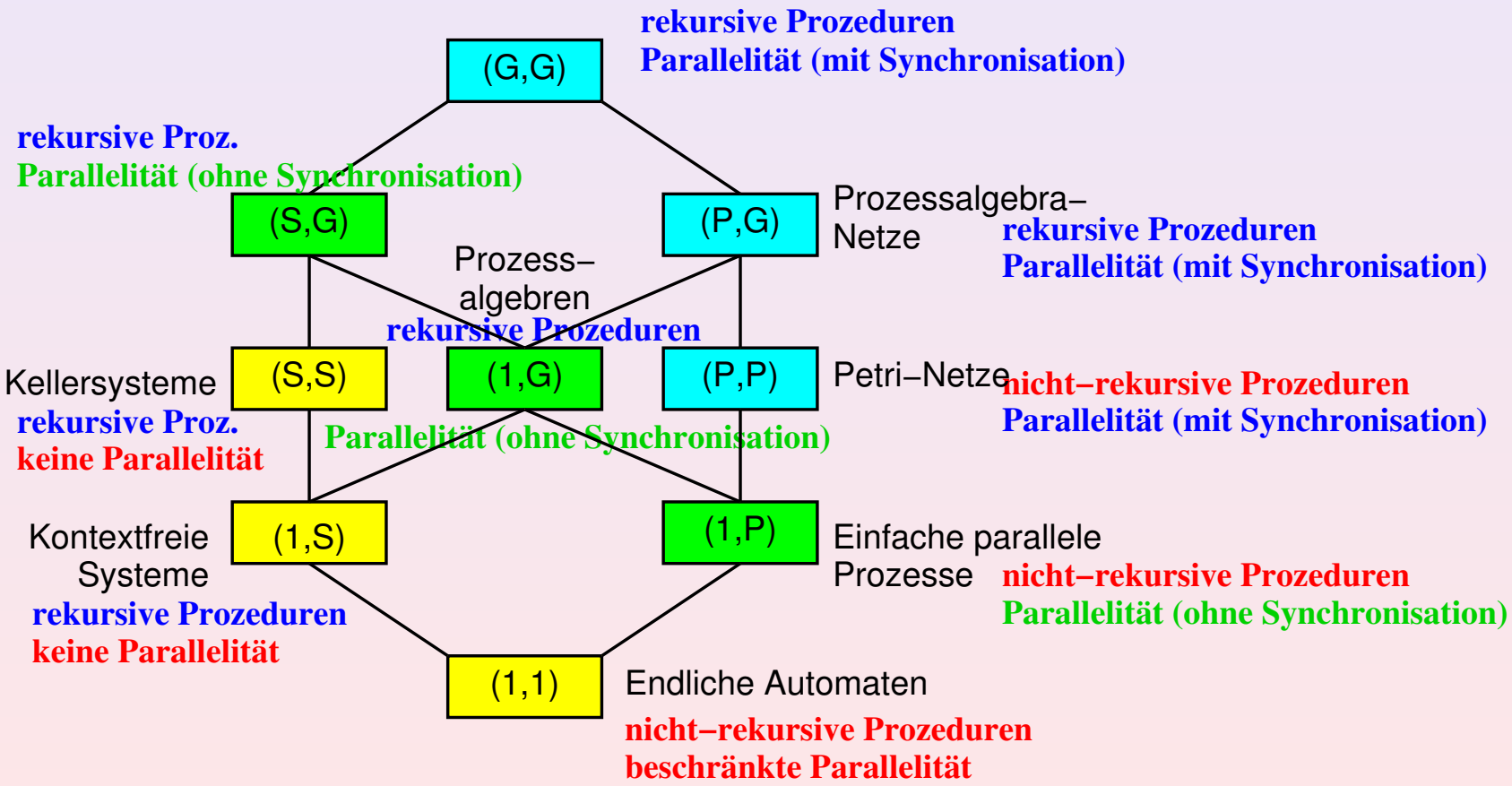
(x, y)-PRS (Mayr 1997): Alle Zustandsübergänge gehören zu Klasse (x, y) (G bedeutet keine Einschränkung)



Diskussion (Both, Zimmermann, Franke 2010)

Zustandsübergangssystem für Kaktuskeller entspricht Prozessersetzungssystemen nach Richard Mayr (1997)

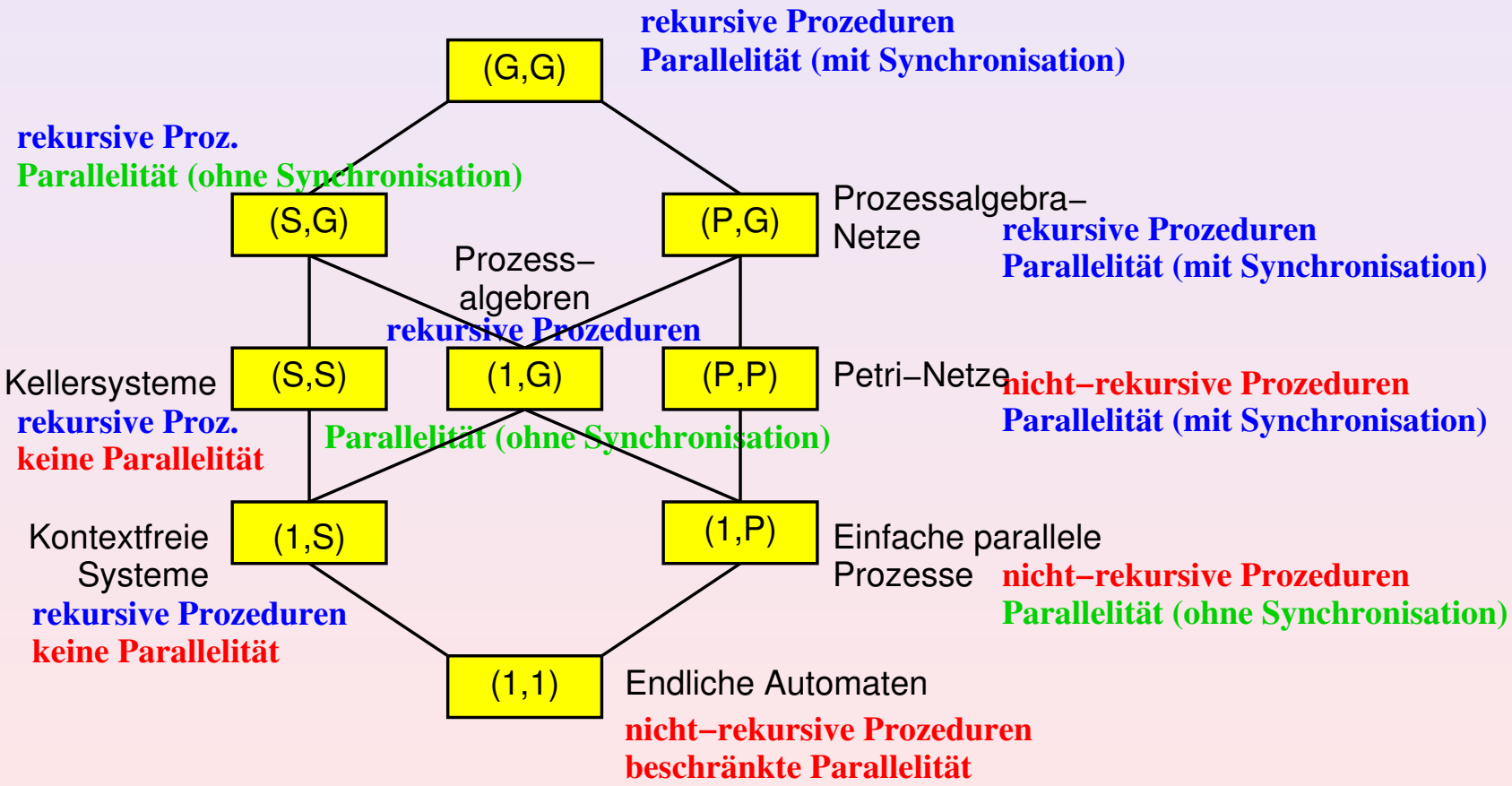
(x, y)-PRS (Mayr 1997): Alle Zustandsübergänge gehören zu Klasse (x, y) (G bedeutet keine Einschränkung)



Diskussion (Both, Zimmermann, Franke 2010)

Zustandsübergangssystem für Kaktuskeller entspricht Prozessersetzungssystemen nach Richard Mayr (1997)

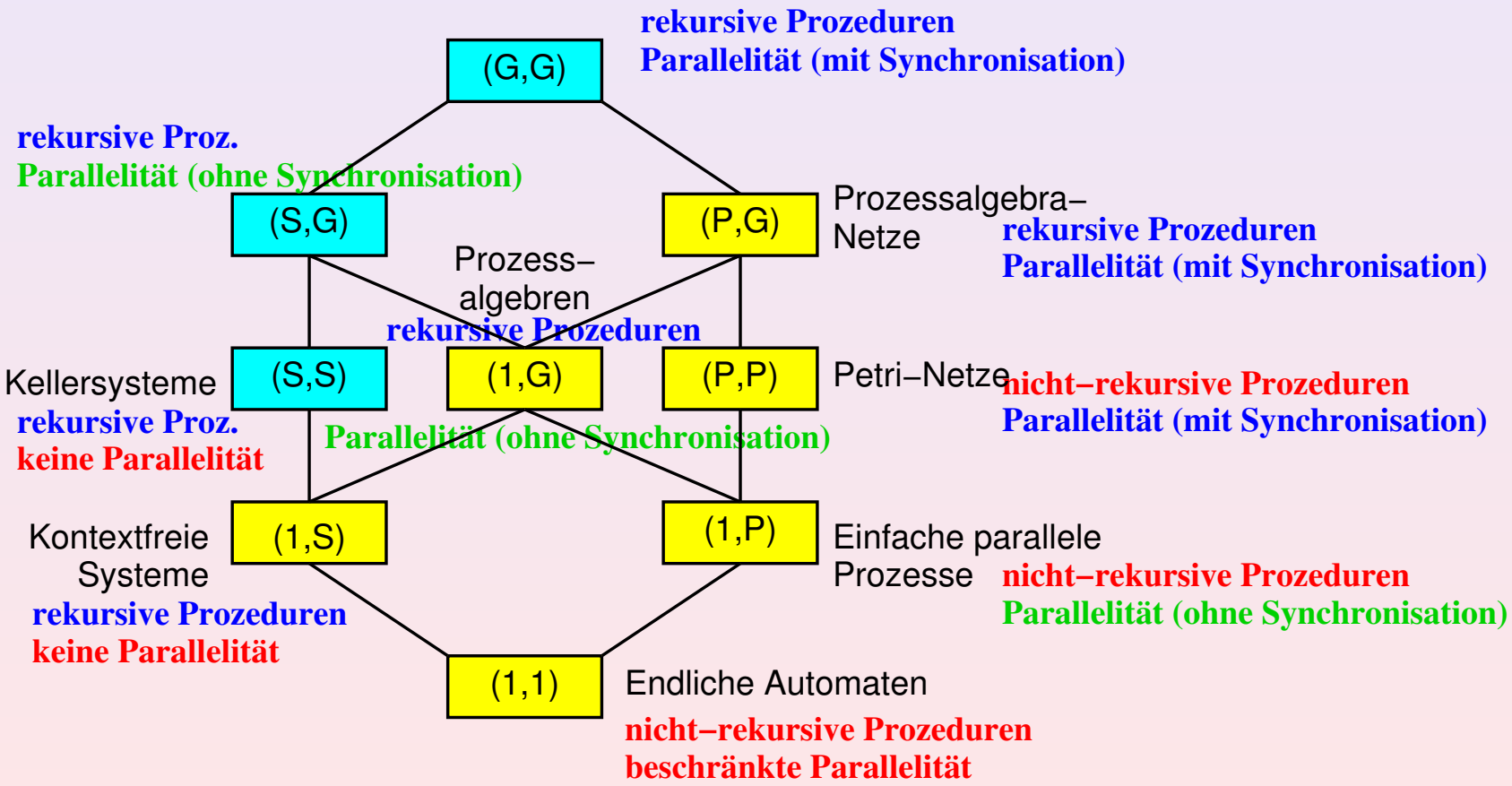
(x, y)-PRS (Mayr 1997): Alle Zustandsübergänge gehören zu Klasse (x, y) (G bedeutet keine Einschränkung)



Diskussion (Both, Zimmermann, Franke 2010)

Zustandsübergangssystem für Kaktuskeller entspricht Prozessersetzungssystemen nach Richard Mayr (1997)

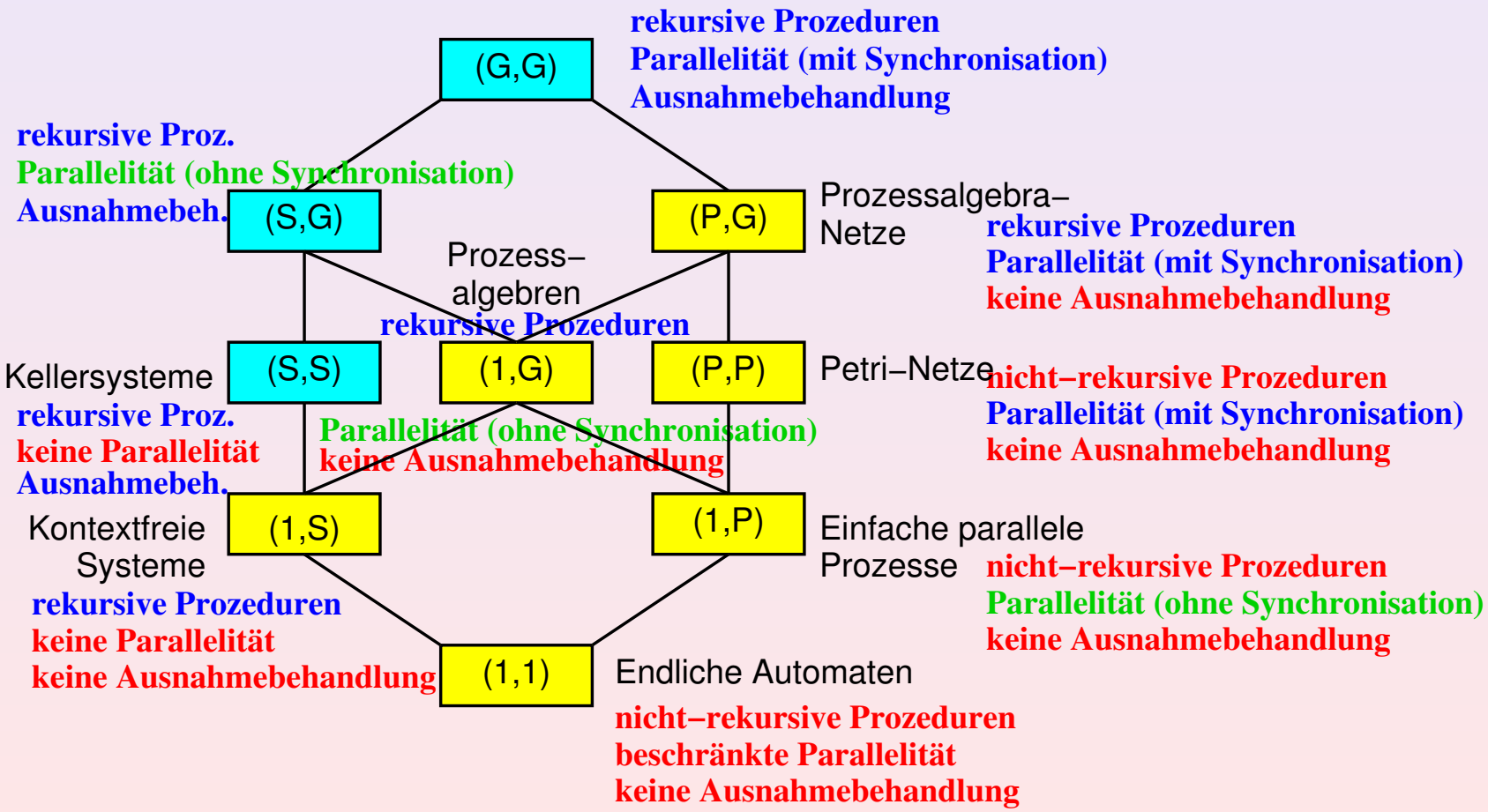
(x, y)-PRS (Mayr 1997): Alle Zustandsübergänge gehören zu Klasse (x, y) (G bedeutet keine Einschränkung)



Diskussion (Both, Zimmermann, Franke 2010)

Zustandsübergangssystem für Kaktuskeller entspricht Prozessersetzungssystemen nach Richard Mayr (1997)

(x, y)-PRS (Mayr 1997): Alle Zustandsübergänge gehören zu Klasse (x, y) (G bedeutet keine Einschränkung)



Entscheidbarkeitsresultate

Satz (Erreichbarkeit (Mayr 1997))

Für PRS Π ist entscheidbar,

- ob $l \overset{x}{\Rightarrow} u$
- ob $x \in L(\Pi)$, $L(\Pi) = \emptyset$

Entscheidbarkeitsresultate

Satz (Erreichbarkeit (Mayr 1997))

Für PRS Π ist entscheidbar,

- ob $l \overset{x}{\Rightarrow} u$
- ob $x \in L(\Pi)$, $L(\Pi) = \emptyset$

Entscheidbarkeitsresultate

Satz (Erreichbarkeit (Mayr 1997))

Für PRS Π ist entscheidbar,

- ob $I \overset{x}{\Rightarrow} u$
- ob $x \in L(\Pi)$, $L(\Pi) = \emptyset$

Entscheidbarkeitsresultate

Satz (Erreichbarkeit (Mayr 1997))

Für PRS Π ist entscheidbar,

- ob $l \overset{x}{\Rightarrow} u$
- ob $x \in L(\Pi)$, $L(\Pi) = \emptyset$

Satz (LTL-Modell-Prüfung (Mayr 1997))

Sei φ aussagenlogische LTL-Formel, die eine Sprache $L(\varphi) \subseteq \Sigma^*$ definiert.

- Für $(1,1)$-PRS, $(1,S)$-PRS,$(1,P)$-PRS, (S,S)-PRS und (P,P)-PRS ist entscheidbar, ob $L(\Pi) \subseteq L(\varphi)$
- Für $(1,G)$-PRS, (S,G)-PRS, (P,G)-PRS und (G,G)-PRS ist unentscheidbar, ob $L(\Pi) \subseteq L(\varphi)$

Entscheidbarkeitsresultate

Satz (Erreichbarkeit (Mayr 1997))

Für PRS Π ist entscheidbar,

- ob $l \overset{x}{\Rightarrow} u$
- ob $x \in L(\Pi)$, $L(\Pi) = \emptyset$

Satz (LTL-Modell-Prüfung (Mayr 1997))

Sei φ aussagenlogische LTL-Formel, die eine Sprache $L(\varphi) \subseteq \Sigma^*$ definiert.

- Für $(1,1)$-PRS, $(1,S)$-PRS,$(1,P)$-PRS, (S,S)-PRS und (P,P)-PRS ist entscheidbar, ob $L(\Pi) \subseteq L(\varphi)$
- Für $(1,G)$-PRS, (S,G)-PRS, (P,G)-PRS und (G,G)-PRS ist unentscheidbar, ob $L(\Pi) \subseteq L(\varphi)$

Entscheidbarkeitsresultate

Satz (Erreichbarkeit (Mayr 1997))

Für PRS Π ist entscheidbar,

- ob $I \overset{x}{\Rightarrow} u$
- ob $x \in L(\Pi)$, $L(\Pi) = \emptyset$

Satz (LTL-Modell-Prüfung (Mayr 1997))

Sei φ aussagenlogische LTL-Formel, die eine Sprache $L(\varphi) \subseteq \Sigma^*$ definiert.

- Für $(1,1)$-PRS, $(1,S)$-PRS,$(1,P)$-PRS, (S,S)-PRS und (P,P)-PRS ist entscheidbar, ob $L(\Pi) \subseteq L(\varphi)$
- Für $(1,G)$-PRS, (S,G)-PRS, (P,G)-PRS und (G,G)-PRS ist unentscheidbar, ob $L(\Pi) \subseteq L(\varphi)$

Entscheidbarkeitsresultate

Satz (Erreichbarkeit (Mayr 1997))

Für PRS Π ist entscheidbar,

- ob $l \overset{x}{\Rightarrow} u$
- ob $x \in L(\Pi)$, $L(\Pi) = \emptyset$

Satz (LTL-Modell-Prüfung (Mayr 1997))

Sei φ aussagenlogische LTL-Formel, die eine Sprache $L(\varphi) \subseteq \Sigma^*$ definiert.

- Für $(1,1)$-PRS, $(1,S)$-PRS,$(1,P)$-PRS, (S,S)-PRS und (P,P)-PRS ist entscheidbar, ob $L(\Pi) \subseteq L(\varphi)$
- Für $(1,G)$-PRS, (S,G)-PRS, (P,G)-PRS und (G,G)-PRS ist unentscheidbar, ob $L(\Pi) \subseteq L(\varphi)$

Korollar (Protokollkonformitätsprüfung)

Für $(1,G)$-PRS, (S,G), (P,G)-PRS und (G,G)-PRS Π und eine reguläre Sprache L ist unentscheidbar, ob $L(\Pi) \subseteq L$

4. Protokollkonformitätsprüfung (CBSE2008, ECOWS2008)

Beobachtung

$L(\Pi) \subseteq L$ genau dann, wenn $L(\Pi) \cap L^c = \emptyset$

- Unentscheidbar für PRS Π und reguläres L, falls rechte Seiten von Regeln keiner Einschränkung unterliegen

$\Rightarrow$ Konstruktion einer Sprache $L_\approx \supseteq L(\Pi) \cap L^c$, so dass $L_\approx \stackrel{?}{=} \emptyset$ entscheidbar wird

☞ $L_\approx = L(\Pi_\approx)$ für ein PRS $\Pi_\approx$ ist ausreichend (**Kombinierte Abstraktion**)

- Ziel: $\Pi_\approx$ soll zur selben Klasse wie Π gehören und im entscheidbaren Fall soll $L_\approx = L(\Pi) \cap L^c$ sein.

Idee

Verallgemeinerung der Textbuchmethode zur Berechnung des Durchschnitts einer kontextfreien Sprache mit einer regulären Sprache

4. Protokollkonformitätsprüfung (CBSE2008, ECOWS2008)

Beobachtung

$L(\Pi) \subseteq L$ genau dann, wenn $L(\Pi) \cap L^c = \emptyset$

- Unentscheidbar für PRS Π und reguläres L, falls rechte Seiten von Regeln keiner Einschränkung unterliegen

$\Rightarrow$ Konstruktion einer Sprache $L_\approx \supseteq L(\Pi) \cap L^c$, so dass $L_\approx \stackrel{?}{=} \emptyset$ entscheidbar wird

☞ $L_\approx = L(\Pi_\approx)$ für ein PRS $\Pi_\approx$ ist ausreichend (**Kombinierte Abstraktion**)

- Ziel: $\Pi_\approx$ soll zur selben Klasse wie Π gehören und im entscheidbaren Fall soll $L_\approx = L(\Pi) \cap L^c$ sein.

Idee

Verallgemeinerung der Textbuchmethode zur Berechnung des Durchschnitts einer kontextfreien Sprache mit einer regulären Sprache

4. Protokollkonformitätsprüfung (CBSE2008, ECOWS2008)

Beobachtung

$L(\Pi) \subseteq L$ genau dann, wenn $L(\Pi) \cap L^c = \emptyset$

- Unentscheidbar für PRS Π und reguläres L, falls rechte Seiten von Regeln keiner Einschränkung unterliegen

$\Rightarrow$ Konstruktion einer Sprache $L_\approx \supseteq L(\Pi) \cap L^c$, so dass $L_\approx \stackrel{?}{=} \emptyset$ entscheidbar wird

☞ $L_\approx = L(\Pi_\approx)$ für ein PRS $\Pi_\approx$ ist ausreichend (**Kombinierte Abstraktion**)

- Ziel: $\Pi_\approx$ soll zur selben Klasse wie Π gehören und im entscheidbaren Fall soll $L_\approx = L(\Pi) \cap L^c$ sein.

Idee

Verallgemeinerung der Textbuchmethode zur Berechnung des Durchschnitts einer kontextfreien Sprache mit einer regulären Sprache

4. Protokollkonformitätsprüfung (CBSE2008, ECOWS2008)

Beobachtung

$L(\Pi) \subseteq L$ genau dann, wenn $L(\Pi) \cap L^c = \emptyset$

- Unentscheidbar für PRS Π und reguläres L, falls rechte Seiten von Regeln keiner Einschränkung unterliegen

$\Rightarrow$ Konstruktion einer Sprache $L_\approx \supseteq L(\Pi) \cap L^c$, so dass $L_\approx \stackrel{?}{=} \emptyset$ entscheidbar wird

☞ $L_\approx = L(\Pi_\approx)$ für ein PRS $\Pi_\approx$ ist ausreichend (**Kombinierte Abstraktion**)

- Ziel: $\Pi_\approx$ soll zur selben Klasse wie Π gehören und im entscheidbaren Fall soll $L_\approx = L(\Pi) \cap L^c$ sein.

Idee

Verallgemeinerung der Textbuchmethode zur Berechnung des Durchschnitts einer kontextfreien Sprache mit einer regulären Sprache

4. Protokollkonformitätsprüfung (CBSE2008, ECOWS2008)

Beobachtung

$L(\Pi) \subseteq L$ genau dann, wenn $L(\Pi) \cap L^c = \emptyset$

- Unentscheidbar für PRS Π und reguläres L, falls rechte Seiten von Regeln keiner Einschränkung unterliegen

$\Rightarrow$ Konstruktion einer Sprache $L_\approx \supseteq L(\Pi) \cap L^c$, so dass $L_\approx \stackrel{?}{=} \emptyset$ entscheidbar wird

☞ $L_\approx = L(\Pi_\approx)$ für ein PRS $\Pi_\approx$ ist ausreichend (**Kombinierte Abstraktion**)

- Ziel: $\Pi_\approx$ soll zur selben Klasse wie Π gehören und im entscheidbaren Fall soll $L_\approx = L(\Pi) \cap L^c$ sein.

Idee

Verallgemeinerung der Textbuchmethode zur Berechnung des Durchschnitts einer kontextfreien Sprache mit einer regulären Sprache

4. Protokollkonformitätsprüfung (CBSE2008, ECOWS2008)

Beobachtung

$L(\Pi) \subseteq L$ genau dann, wenn $L(\Pi) \cap L^c = \emptyset$

- Unentscheidbar für PRS Π und reguläres L, falls rechte Seiten von Regeln keiner Einschränkung unterliegen

$\Rightarrow$ Konstruktion einer Sprache $L_\approx \supseteq L(\Pi) \cap L^c$, so dass $L_\approx \stackrel{?}{=} \emptyset$ entscheidbar wird

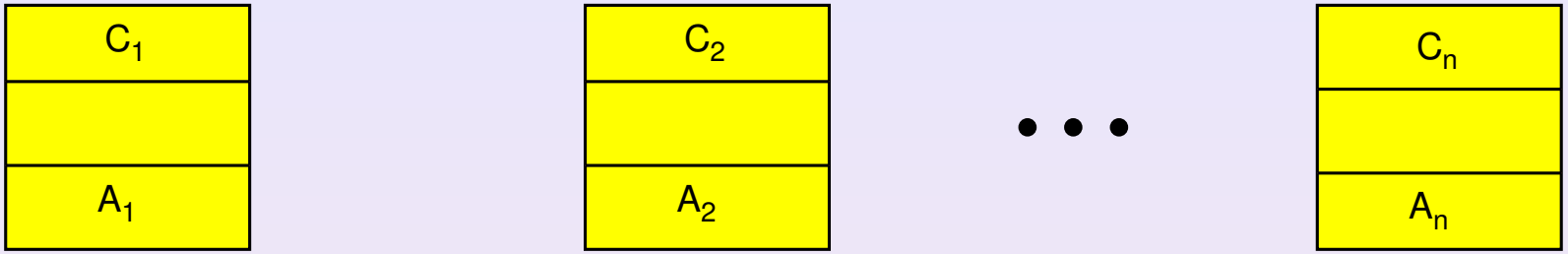
☞ $L_\approx = L(\Pi_\approx)$ für ein PRS $\Pi_\approx$ ist ausreichend (**Kombinierte Abstraktion**)

- Ziel: $\Pi_\approx$ soll zur selben Klasse wie Π gehören und im entscheidbaren Fall soll $L_\approx = L(\Pi) \cap L^c$ sein.

Idee

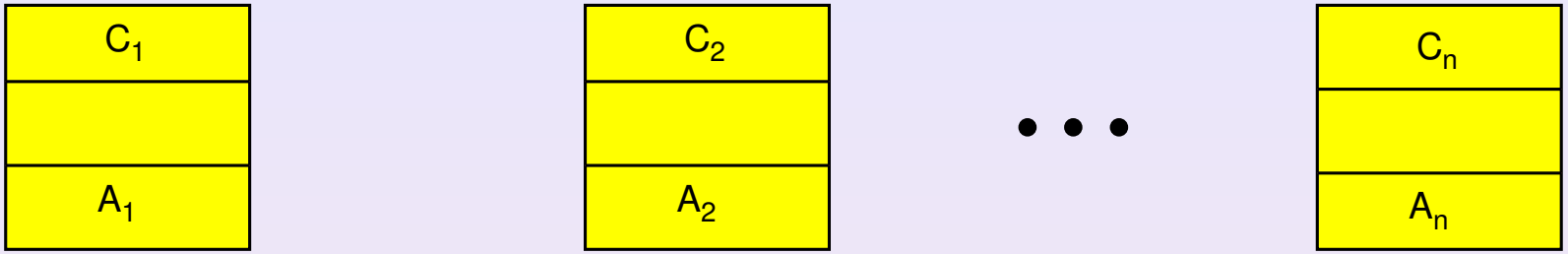
Verallgemeinerung der Textbuchmethode zur Berechnung des Durchschnitts einer kontextfreien Sprache mit einer regulären Sprache

Methodik (VeriCap)



- Komponenten werden manuell mit Protokollen versehen
- Bei Auslieferung bzw. Veröffentlichung wird Kontrollflussabstraktion (automatisch) generiert und mitgeliefert
  - ☞ Derzeit für Python, PHP, BPEL und .NET
- Bestimmung der kombinierten Abstraktion,
- Bei negativem Resultat wird Gegenbeispiel erzeugt

Methodik (VeriCap)

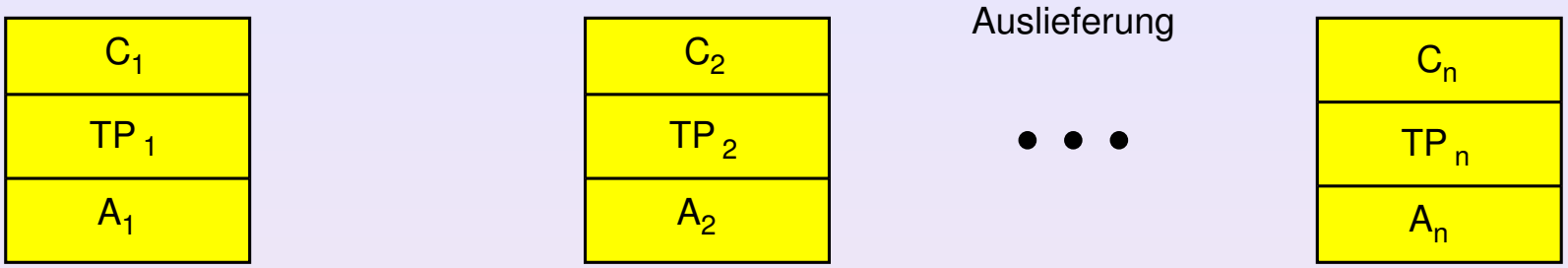


- Komponenten werden manuell mit Protokollen versehen
- Bei Auslieferung bzw. Veröffentlichung wird Kontrollflussabstraktion (automatisch) generiert und mitgeliefert

☞ Derzeit für Python, PHP, BPEL und .NET

- Bestimmung der kombinierten Abstraktion,
- Bei negativem Resultat wird Gegenbeispiel erzeugt

Methodik (VeriCap)

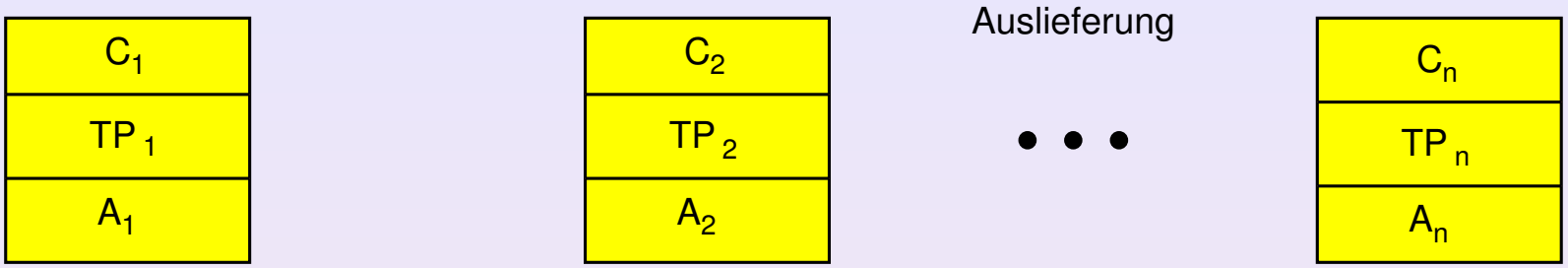


- Komponenten werden manuell mit Protokollen versehen
- Bei Auslieferung bzw. Veröffentlichung wird Kontrollflussabstraktion (automatisch) generiert und mitgeliefert

☞ Derzeit für Python, PHP, BPEL und .NET

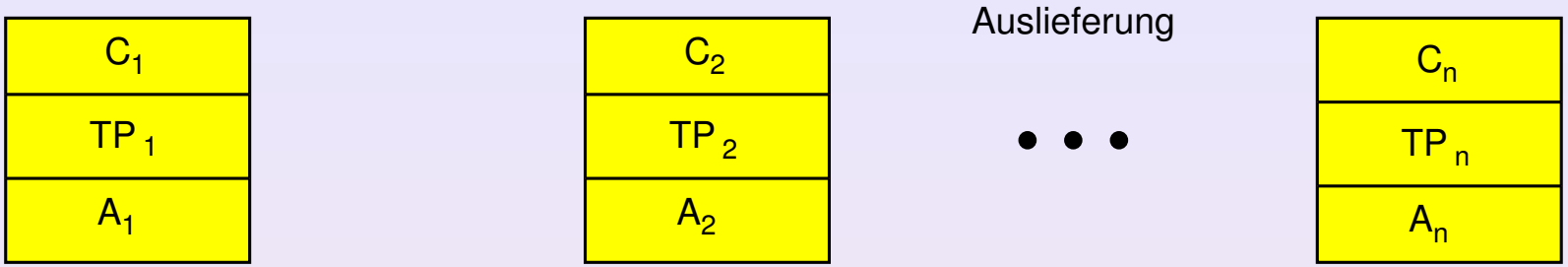
- Bestimmung der kombinierten Abstraktion,
- Bei negativem Resultat wird Gegenbeispiel erzeugt

Methodik (VeriCap)



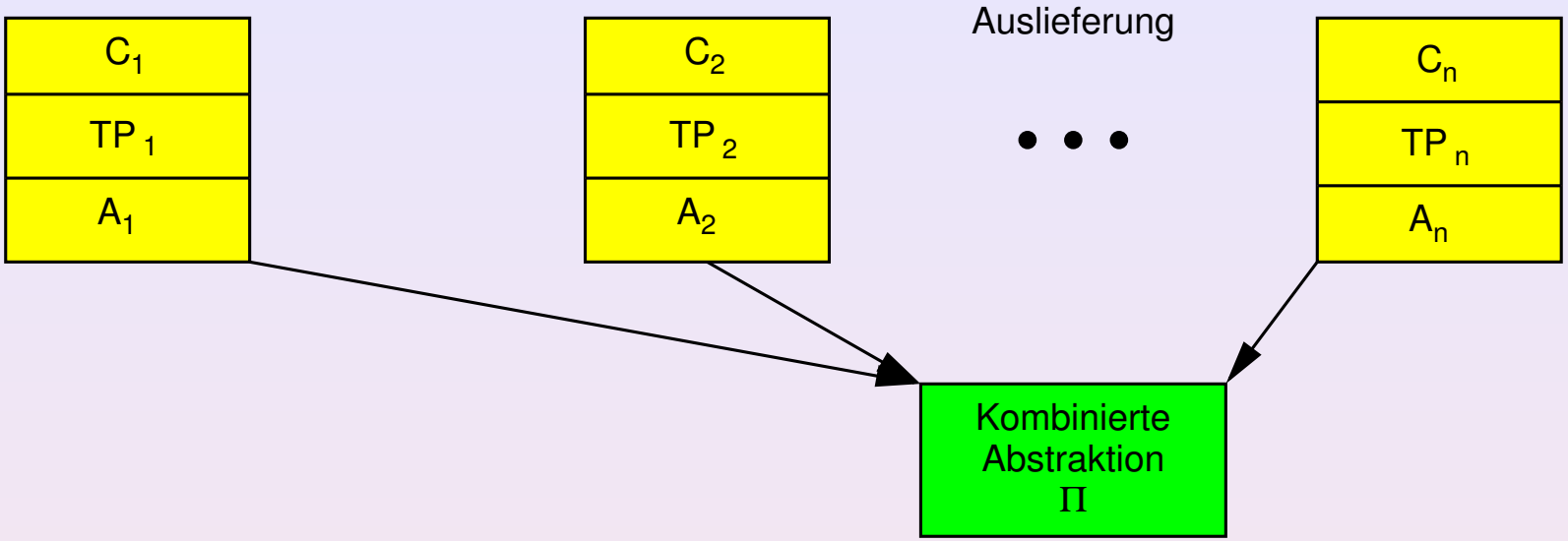
- Komponenten werden manuell mit Protokollen versehen
- Bei Auslieferung bzw. Veröffentlichung wird Kontrollflussabstraktion (automatisch) generiert und mitgeliefert
- ☞ Derzeit für Python, PHP, BPEL und .NET
- Bestimmung der kombinierten Abstraktion,
- Bei negativem Resultat wird Gegenbeispiel erzeugt

Methodik (VeriCap)



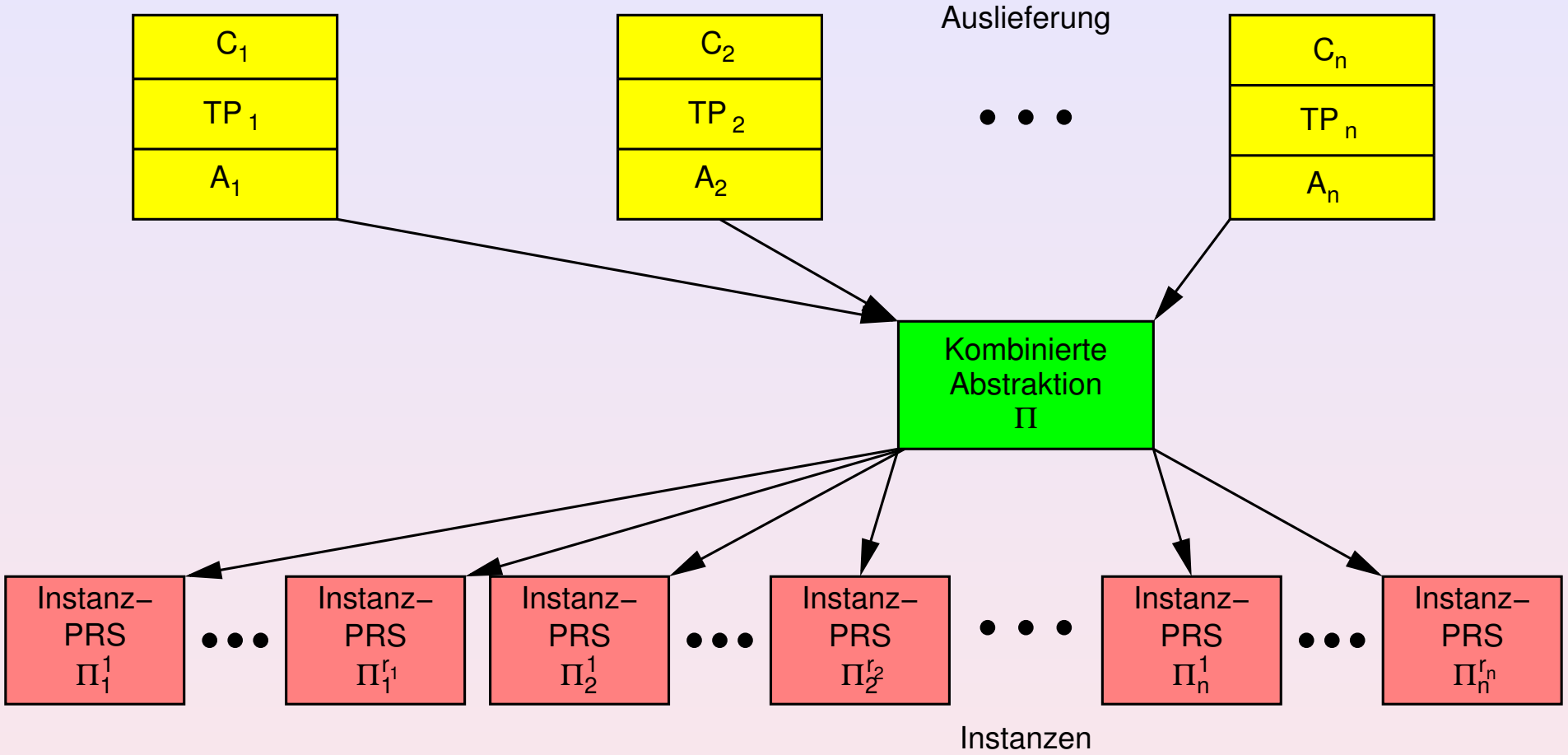
- Komponenten werden manuell mit Protokollen versehen
- Bei Auslieferung bzw. Veröffentlichung wird Kontrollflussabstraktion (automatisch) generiert und mitgeliefert
- ☞ Derzeit für Python, PHP, BPEL und .NET
- Bestimmung der kombinierten Abstraktion,
- Bei negativem Resultat wird Gegenbeispiel erzeugt

Methodik (VeriCap)



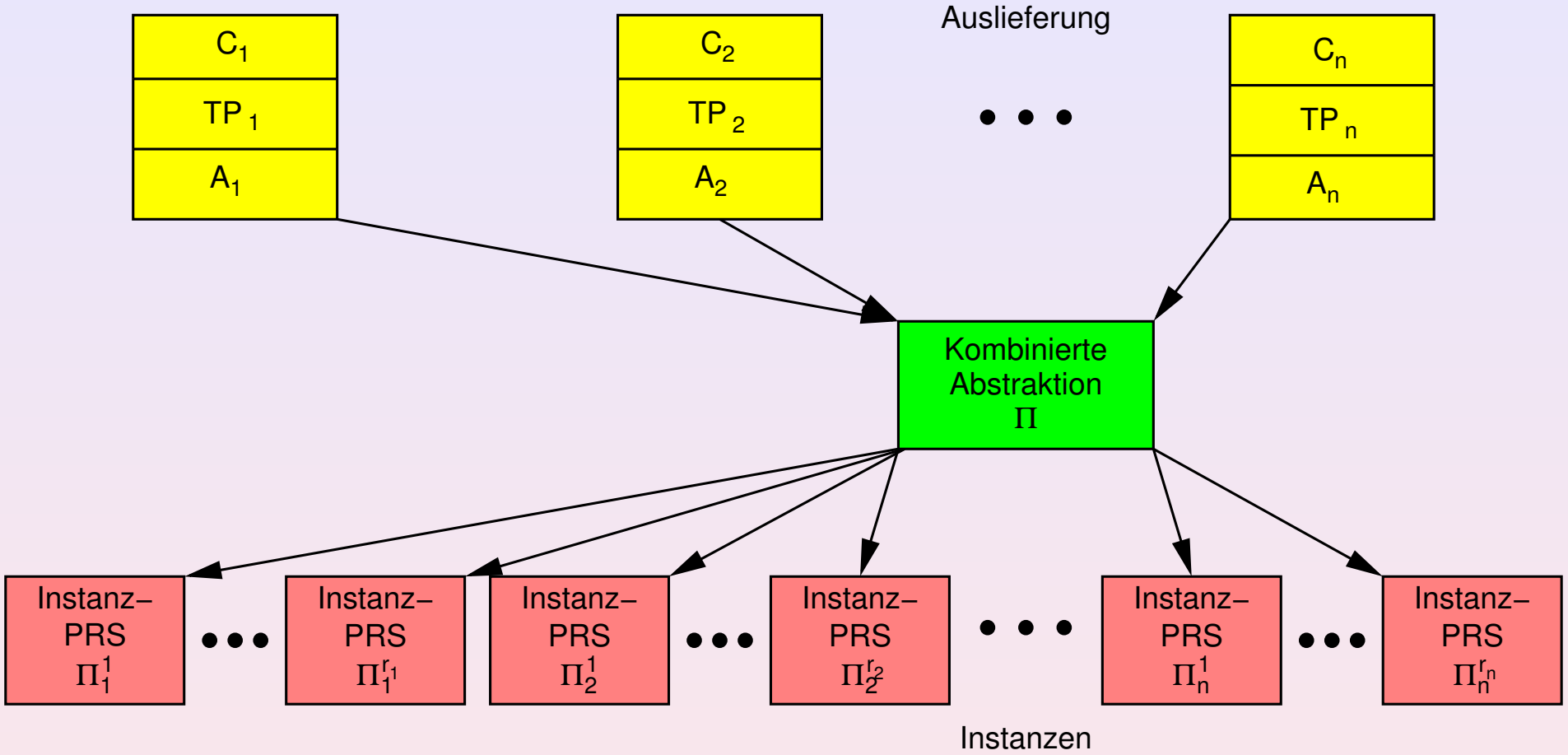
- Komponenten werden manuell mit Protokollen versehen
- Bei Auslieferung bzw. Veröffentlichung wird Kontrollflussabstraktion (automatisch) generiert und mitgeliefert
- ☞ Derzeit für Python, PHP, BPEL und .NET
- Bestimmung der kombinierten Abstraktion,
- Bei negativem Resultat wird Gegenbeispiel erzeugt

Methodik (VeriCap)



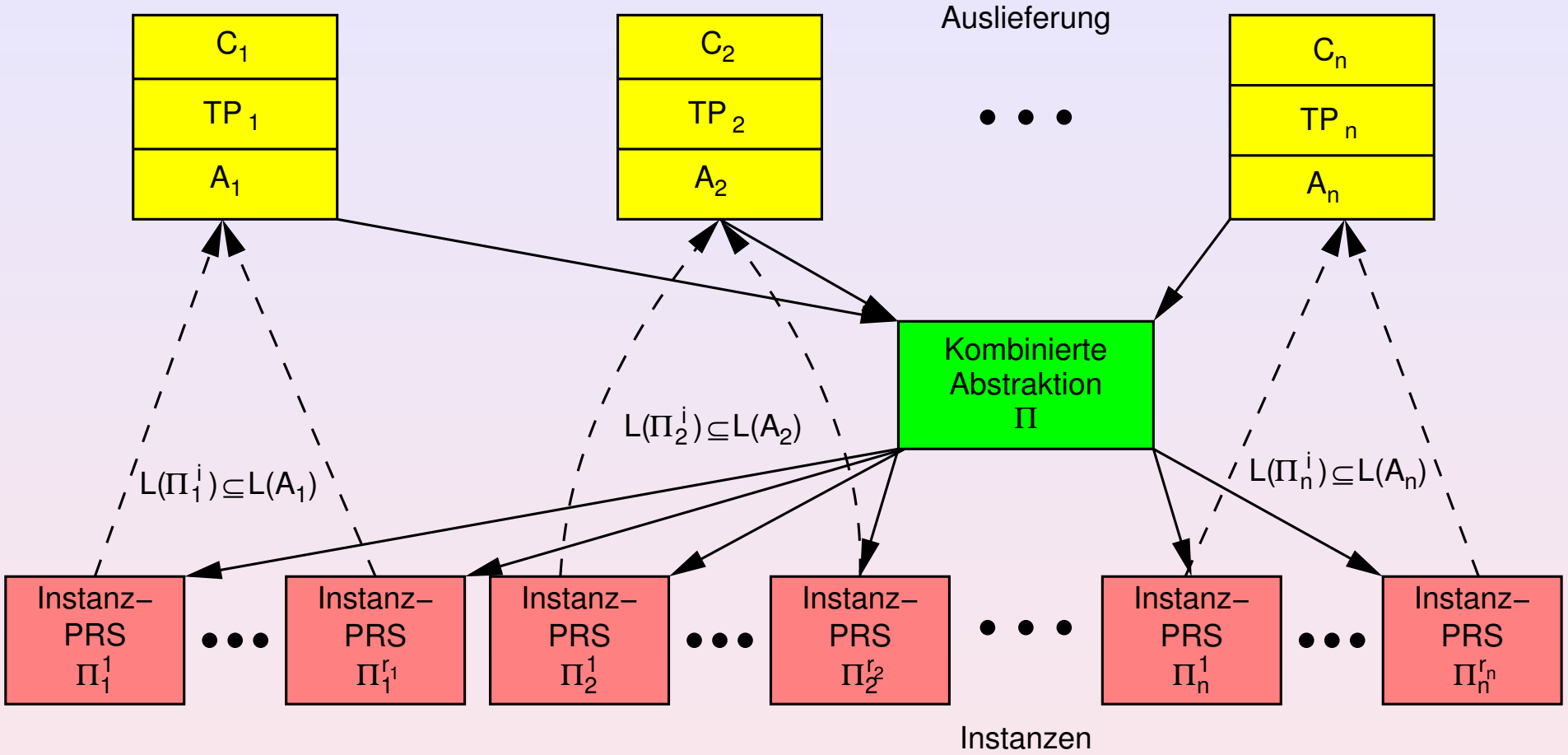
- Komponenten werden manuell mit Protokollen versehen
- Bei Auslieferung bzw. Veröffentlichung wird Kontrollflussabstraktion (automatisch) generiert und mitgeliefert
- ☞ Derzeit für Python, PHP, BPEL und .NET
- Bestimmung der kombinierten Abstraktion,
- Bei negativem Resultat wird Gegenbeispiel erzeugt

Methodik (VeriCap)



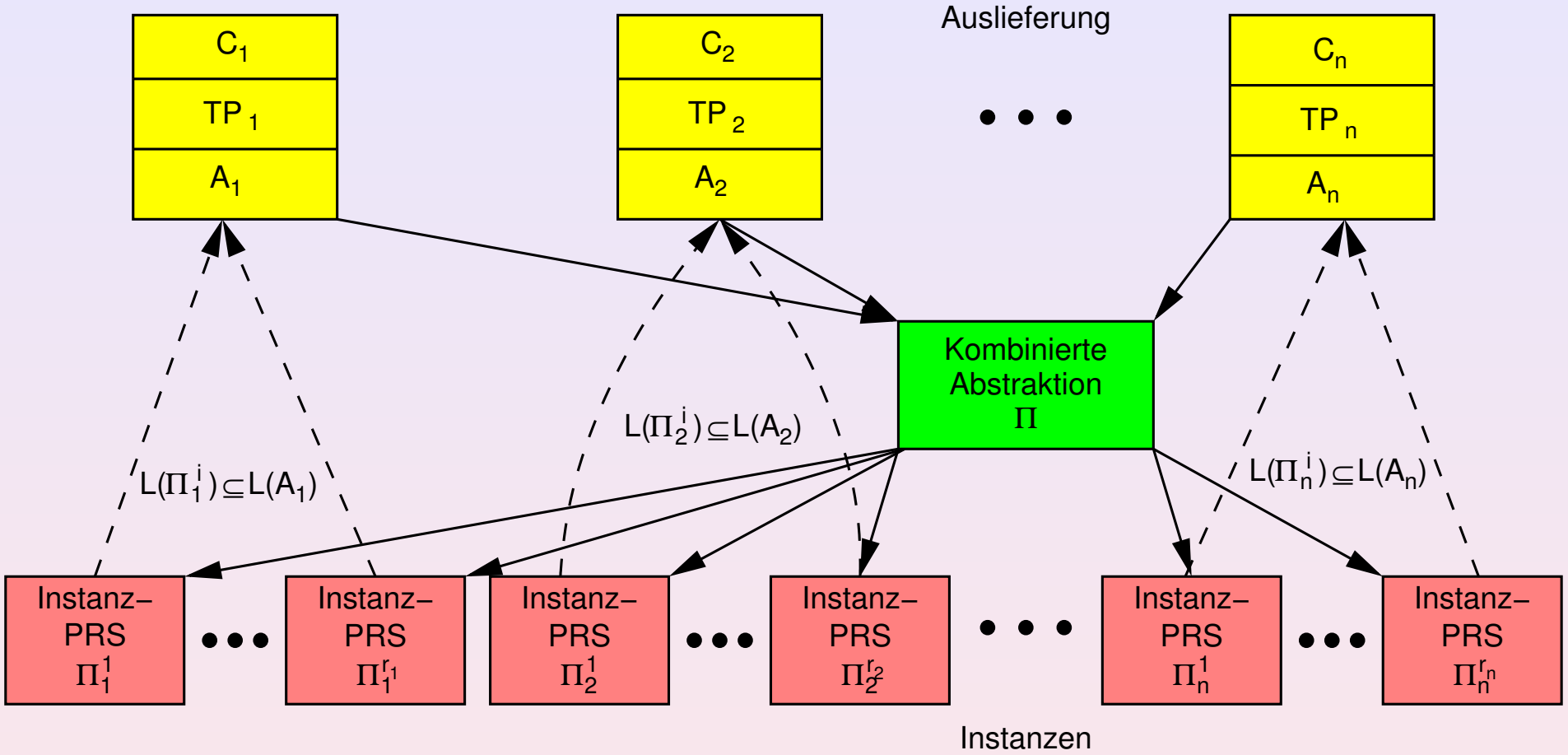
- Komponenten werden manuell mit Protokollen versehen
- Bei Auslieferung bzw. Veröffentlichung wird Kontrollflussabstraktion (automatisch) generiert und mitgeliefert
- ☞ Derzeit für Python, PHP, BPEL und .NET
- Bestimmung der kombinierten Abstraktion, Prüfung jeder Instanz
- Bei negativem Resultat wird Gegenbeispiel erzeugt

Methodik (VeriCap)



- Komponenten werden manuell mit Protokollen versehen
- Bei Auslieferung bzw. Veröffentlichung wird Kontrollflussabstraktion (automatisch) generiert und mitgeliefert
- ☞ Derzeit für Python, PHP, BPEL und .NET
- Bestimmung der kombinierten Abstraktion, Prüfung jeder Instanz
- Bei negativem Resultat wird Gegenbeispiel erzeugt

Methodik (VeriCap)



- Komponenten werden manuell mit Protokollen versehen
- Bei Auslieferung bzw. Veröffentlichung wird Kontrollflussabstraktion (automatisch) generiert und mitgeliefert
- ☞ Derzeit für Python, PHP, BPEL und .NET
- Bestimmung der kombinierten Abstraktion, Prüfung jeder Instanz
- Bei negativem Resultat wird Gegenbeispiel erzeugt

5. Schlussfolgerungen

- Protokollprüfung ist analog Modellprüfung
- Erreichbarkeitsprobleme auch für unbeschränkte Parallelität und unbeschränkte Rekursionstiefe entscheidbar
- Protokollprüfungsproblem wird unentscheidbar

⇒ Konservative Approximation des Protokollprüfungsproblems

- Sowohl a-priori als auch a-posteriori Ausschluss (SEAA 2009) unechter falscher Negative möglich
- Kontrollfluss kann kompositional ermittelt werden (CBSE2008)
- Abstraktionen zu PRS für Python, PHP, BPEL (ECOWS 2008), .NET und Teile von C++
- Einbindung in Integrationsprozess (GI 2009, ECOWS 2009)
- Bereits auf industriellen Fallstudien angewendet

5. Schlussfolgerungen

- Protokollprüfung ist analog Modellprüfung
- Erreichbarkeitsprobleme auch für unbeschränkte Parallelität und unbeschränkte Rekursionstiefe entscheidbar
- Protokollprüfungsproblem wird unentscheidbar

⇒ Konservative Approximation des Protokollprüfungsproblems

- Sowohl a-priori als auch a-posteriori Ausschluss (SEAA 2009) unechter falscher Negative möglich
- Kontrollfluss kann kompositional ermittelt werden (CBSE2008)
- Abstraktionen zu PRS für Python, PHP, BPEL (ECOWS 2008), .NET und Teile von C++
- Einbindung in Integrationsprozess (GI 2009, ECOWS 2009)
- Bereits auf industriellen Fallstudien angewendet

5. Schlussfolgerungen

- Protokollprüfung ist analog Modellprüfung
- Erreichbarkeitsprobleme auch für unbeschränkte Parallelität und unbeschränkte Rekursionstiefe entscheidbar
- Protokollprüfungsproblem wird unentscheidbar

⇒ Konservative Approximation des Protokollprüfungsproblems

- Sowohl a-priori als auch a-posteriori Ausschluss (SEAA 2009) unechter falscher Negative möglich
- Kontrollfluss kann kompositional ermittelt werden (CBSE2008)
- Abstraktionen zu PRS für Python, PHP, BPEL (ECOWS 2008), .NET und Teile von C++
- Einbindung in Integrationsprozess (GI 2009, ECOWS 2009)
- Bereits auf industriellen Fallstudien angewendet

5. Schlussfolgerungen

- Protokollprüfung ist analog Modellprüfung
- Erreichbarkeitsprobleme auch für unbeschränkte Parallelität und unbeschränkte Rekursionstiefe entscheidbar
- Protokollprüfungsproblem wird unentscheidbar

⇒ Konservative Approximation des Protokollprüfungsproblems

- Sowohl a-priori als auch a-posteriori Ausschluss (SEAA 2009) unechter falscher Negative möglich
- Kontrollfluss kann kompositional ermittelt werden (CBSE2008)
- Abstraktionen zu PRS für Python, PHP, BPEL (ECOWS 2008), .NET und Teile von C++
- Einbindung in Integrationsprozess (GI 2009, ECOWS 2009)
- Bereits auf industriellen Fallstudien angewendet

5. Schlussfolgerungen

- Protokollprüfung ist analog Modellprüfung
- Erreichbarkeitsprobleme auch für unbeschränkte Parallelität und unbeschränkte Rekursionstiefe entscheidbar
- Protokollprüfungsproblem wird unentscheidbar

⇒ Konservative Approximation des Protokollprüfungsproblems

- Sowohl a-priori als auch a-posteriori Ausschluss (SEAA 2009) unechter falscher Negative möglich
- Kontrollfluss kann kompositional ermittelt werden (CBSE2008)
- Abstraktionen zu PRS für Python, PHP, BPEL (ECOWS 2008), .NET und Teile von C++
- Einbindung in Integrationsprozess (GI 2009, ECOWS 2009)
- Bereits auf industriellen Fallstudien angewendet

5. Schlussfolgerungen

- Protokollprüfung ist analog Modellprüfung
- Erreichbarkeitsprobleme auch für unbeschränkte Parallelität und unbeschränkte Rekursionstiefe entscheidbar
- Protokollprüfungsproblem wird unentscheidbar

⇒ Konservative Approximation des Protokollprüfungsproblems

- Sowohl a-priori als auch a-posteriori Ausschluss (SEAA 2009) unechter falscher Negative möglich
- Kontrollfluss kann kompositional ermittelt werden (CBSE2008)
- Abstraktionen zu PRS für Python, PHP, BPEL (ECOWS 2008), .NET und Teile von C++
- Einbindung in Integrationsprozess (GI 2009, ECOWS 2009)
- Bereits auf industriellen Fallstudien angewendet

5. Schlussfolgerungen

- Protokollprüfung ist analog Modellprüfung
- Erreichbarkeitsprobleme auch für unbeschränkte Parallelität und unbeschränkte Rekursionstiefe entscheidbar
- Protokollprüfungsproblem wird unentscheidbar

⇒ Konservative Approximation des Protokollprüfungsproblems

- Sowohl a-priori als auch a-posteriori Ausschluss (SEAA 2009) unechter falscher Negative möglich
- Kontrollfluss kann kompositional ermittelt werden (CBSE2008)
- Abstraktionen zu PRS für Python, PHP, BPEL (ECOWS 2008), .NET und Teile von C++
- Einbindung in Integrationsprozess (GI 2009, ECOWS 2009)
- Bereits auf industriellen Fallstudien angewendet

5. Schlussfolgerungen

- Protokollprüfung ist analog Modellprüfung
- Erreichbarkeitsprobleme auch für unbeschränkte Parallelität und unbeschränkte Rekursionstiefe entscheidbar
- Protokollprüfungsproblem wird unentscheidbar

⇒ Konservative Approximation des Protokollprüfungsproblems

- Sowohl a-priori als auch a-posteriori Ausschluss (SEAA 2009) unechter falscher Negative möglich
- Kontrollfluss kann kompositional ermittelt werden (CBSE2008)
- Abstraktionen zu PRS für Python, PHP, BPEL (ECOWS 2008), .NET und Teile von C++
- Einbindung in Integrationsprozess (GI 2009, ECOWS 2009)
- Bereits auf industriellen Fallstudien angewendet

5. Schlussfolgerungen

- Protokollprüfung ist analog Modellprüfung
- Erreichbarkeitsprobleme auch für unbeschränkte Parallelität und unbeschränkte Rekursionstiefe entscheidbar
- Protokollprüfungsproblem wird unentscheidbar

⇒ Konservative Approximation des Protokollprüfungsproblems

- Sowohl a-priori als auch a-posteriori Ausschluss (SEAA 2009) unechter falscher Negative möglich
- Kontrollfluss kann kompositional ermittelt werden (CBSE2008)
- Abstraktionen zu PRS für Python, PHP, BPEL (ECOWS 2008), .NET und Teile von C++
- Einbindung in Integrationsprozess (GI 2009, ECOWS 2009)
- Bereits auf industriellen Fallstudien angewendet

Aktuelle und nächste Schritte

Protokollprüfung

- Weitere Anwendung auf industrielle Fallstudien (z.Z. in Arbeit)
- Abstraktionen für weitere Sprachen (z.Z. in Arbeit Java, C++)
- Effizientere Protokollprüfungsalgorithmen
- Berücksichtigung von Variableninhalten (analog Softwaremodellprüfung)
- Hierarchische Komponenten (Herleitung von Protokollen)
- Erweiterung auf objekt-orientierte Systeme (Protokolle für Klassen, Vererbung)
- Ausdrucksmächtigere Protokolle, z.B. constraint Automaten, counter Automaten

Andere nicht-funktionale Eigenschaften

Aktuelle und nächste Schritte

Protokollprüfung

- Weitere Anwendung auf industrielle Fallstudien (z.Z. in Arbeit)
- Abstraktionen für weitere Sprachen (z.Z. in Arbeit Java, C++)
- Effizientere Protokollprüfungsalgorithmen
- Berücksichtigung von Variableninhalten (analog Softwaremodellprüfung)
- Hierarchische Komponenten (Herleitung von Protokollen)
- Erweiterung auf objekt-orientierte Systeme (Protokolle für Klassen, Vererbung)
- Ausdrucksmächtigere Protokolle, z.B. constraint Automaten, counter Automaten

Andere nicht-funktionale Eigenschaften

Aktuelle und nächste Schritte

Protokollprüfung

- Weitere Anwendung auf industrielle Fallstudien (z.Z. in Arbeit)
- Abstraktionen für weitere Sprachen (z.Z. in Arbeit Java, C++)
- Effizientere Protokollprüfungsalgorithmen
- Berücksichtigung von Variableninhalten (analog Softwaremodellprüfung)
- Hierarchische Komponenten (Herleitung von Protokollen)
- Erweiterung auf objekt-orientierte Systeme (Protokolle für Klassen, Vererbung)
- Ausdrucksmächtigere Protokolle, z.B. constraint Automaten, counter Automaten

Andere nicht-funktionale Eigenschaften

Aktuelle und nächste Schritte

Protokollprüfung

- Weitere Anwendung auf industrielle Fallstudien (z.Z. in Arbeit)
- Abstraktionen für weitere Sprachen (z.Z. in Arbeit Java, C++)
- Effizientere Protokollprüfungsalgorithmen
- Berücksichtigung von Variableninhalten (analog Softwaremodellprüfung)
- Hierarchische Komponenten (Herleitung von Protokollen)
- Erweiterung auf objekt-orientierte Systeme (Protokolle für Klassen, Vererbung)
- Ausdrucksmächtigere Protokolle, z.B. constraint Automaten, counter Automaten

Andere nicht-funktionale Eigenschaften

Aktuelle und nächste Schritte

Protokollprüfung

- Weitere Anwendung auf industrielle Fallstudien (z.Z. in Arbeit)
- Abstraktionen für weitere Sprachen (z.Z. in Arbeit Java, C++)
- Effizientere Protokollprüfungsalgorithmen
- Berücksichtigung von Variableninhalten (analog Softwaremodellprüfung)
- Hierarchische Komponenten (Herleitung von Protokollen)
- Erweiterung auf objekt-orientierte Systeme (Protokolle für Klassen, Vererbung)
- Ausdrucksmächtigere Protokolle, z.B. constraint Automaten, counter Automaten

Andere nicht-funktionale Eigenschaften

Aktuelle und nächste Schritte

Protokollprüfung

- Weitere Anwendung auf industrielle Fallstudien (z.Z. in Arbeit)
- Abstraktionen für weitere Sprachen (z.Z. in Arbeit Java, C++)
- Effizientere Protokollprüfungsalgorithmen
- Berücksichtigung von Variableninhalten (analog Softwaremodellprüfung)
- Hierarchische Komponenten (Herleitung von Protokollen)
- Erweiterung auf objekt-orientierte Systeme (Protokolle für Klassen, Vererbung)
- Ausdrucksmächtigere Protokolle, z.B. constraint Automaten, counter Automaten

Andere nicht-funktionale Eigenschaften

- Termination/Deadlock/Lebendigkeit
- Zeitanforderungen (z.B. Schnittstellen müssen innerhalb einer Zeitspanne aktiv sein)
- Ressourcenverbrauch (z.B. Speicherverbrauch, Energieverbrauch)

Aktuelle und nächste Schritte

Protokollprüfung

- Weitere Anwendung auf industrielle Fallstudien (z.Z. in Arbeit)
- Abstraktionen für weitere Sprachen (z.Z. in Arbeit Java, C++)
- Effizientere Protokollprüfungsalgorithmen
- Berücksichtigung von Variableninhalten (analog Softwaremodellprüfung)
- Hierarchische Komponenten (Herleitung von Protokollen)
- Erweiterung auf objekt-orientierte Systeme (Protokolle für Klassen, Vererbung)
- Ausdrucksmächtigere Protokolle, z.B. constraint Automaten, counter Automaten

Andere nicht-funktionale Eigenschaften

- Spezialfall Termination (z.B. (Mandler) / Zimmermann 2006)
- Speicherverbrauch (z.B. (Zimmermann / Mandler 2006))

Aktuelle und nächste Schritte

Protokollprüfung

- Weitere Anwendung auf industrielle Fallstudien (z.Z. in Arbeit)
- Abstraktionen für weitere Sprachen (z.Z. in Arbeit Java, C++)
- Effizientere Protokollprüfungsalgorithmen
- Berücksichtigung von Variableninhalten (analog Softwaremodellprüfung)
- Hierarchische Komponenten (Herleitung von Protokollen)
- Erweiterung auf objekt-orientierte Systeme (Protokolle für Klassen, Vererbung)
- Ausdrucksmächtigere Protokolle, z.B. constraint Automaten, counter Automaten

Andere nicht-funktionale Eigenschaften

- Spezialfall Terminierungsanalyse (Weißbach/Zimmermann 2010)
- Terminierungsanalyse als SLA-Methode (Weißbach/Zimmermann 2011)
- Analyse von Ressourcenverbrauch (z.B. Speicherverbrauch)

Aktuelle und nächste Schritte

Protokollprüfung

- Weitere Anwendung auf industrielle Fallstudien (z.Z. in Arbeit)
- Abstraktionen für weitere Sprachen (z.Z. in Arbeit Java, C++)
- Effizientere Protokollprüfungsalgorithmen
- Berücksichtigung von Variableninhalten (analog Softwaremodellprüfung)
- Hierarchische Komponenten (Herleitung von Protokollen)
- Erweiterung auf objekt-orientierte Systeme (Protokolle für Klassen, Vererbung)
- Ausdrucksmächtigere Protokolle, z.B. constraint Automaten, counter Automaten

Andere nicht-funktionale Eigenschaften

- Spezialfall Terminierungsanalyse (Weißbach/Zimmermann 2010)
- Terminierungsanalyse als SLA-Methode (Weißbach/Zimmermann 2011)
- Analyse von Lebendigkeitsbedingungen (z.B. Verklemmungsfreiheit)
- Analyse von Zeit-, Speicher-, Energieaufwand etc.

Aktuelle und nächste Schritte

Protokollprüfung

- Weitere Anwendung auf industrielle Fallstudien (z.Z. in Arbeit)
- Abstraktionen für weitere Sprachen (z.Z. in Arbeit Java, C++)
- Effizientere Protokollprüfungsalgorithmen
- Berücksichtigung von Variableninhalten (analog Softwaremodellprüfung)
- Hierarchische Komponenten (Herleitung von Protokollen)
- Erweiterung auf objekt-orientierte Systeme (Protokolle für Klassen, Vererbung)
- Ausdrucksmächtigere Protokolle, z.B. constraint Automaten, counter Automaten

Andere nicht-funktionale Eigenschaften

- Spezialfall Terminierungsanalyse (Weißbach/Zimmermann 2010)
- Terminierungsanalyse als SLA-Methode (Weißbach/Zimmermann 2011)
- Analyse von Lebendigkeitsbedingungen (z.B. Verklemmungsfreiheit)
- Analyse von Zeit-, Speicher-, Energieaufwand etc.

Aktuelle und nächste Schritte

Protokollprüfung

- Weitere Anwendung auf industrielle Fallstudien (z.Z. in Arbeit)
- Abstraktionen für weitere Sprachen (z.Z. in Arbeit Java, C++)
- Effizientere Protokollprüfungsalgorithmen
- Berücksichtigung von Variableninhalten (analog Softwaremodellprüfung)
- Hierarchische Komponenten (Herleitung von Protokollen)
- Erweiterung auf objekt-orientierte Systeme (Protokolle für Klassen, Vererbung)
- Ausdrucksmächtigere Protokolle, z.B. constraint Automaten, counter Automaten

Andere nicht-funktionale Eigenschaften

- Spezialfall Terminierungsanalyse (Weißbach/Zimmermann 2010)
- Terminierungsanalyse als SLA-Methode (Weißbach/Zimmermann 2011)
- Analyse von Lebendigkeitsbedingungen (z.B. Verklemmungsfreiheit)
- Analyse von Zeit-, Speicher-, Energieaufwand etc.

Aktuelle und nächste Schritte

Protokollprüfung

- Weitere Anwendung auf industrielle Fallstudien (z.Z. in Arbeit)
- Abstraktionen für weitere Sprachen (z.Z. in Arbeit Java, C++)
- Effizientere Protokollprüfungsalgorithmen
- Berücksichtigung von Variableninhalten (analog Softwaremodellprüfung)
- Hierarchische Komponenten (Herleitung von Protokollen)
- Erweiterung auf objekt-orientierte Systeme (Protokolle für Klassen, Vererbung)
- Ausdrucksmächtigere Protokolle, z.B. constraint Automaten, counter Automaten

Andere nicht-funktionale Eigenschaften

- Spezialfall Terminierungsanalyse (Weißbach/Zimmermann 2010)
- Terminierungsanalyse als SLA-Methode (Weißbach/Zimmermann 2011)
- Analyse von Lebendigkeitsbedingungen (z.B. Verklemmungsfreiheit)
- Analyse von Zeit-, Speicher-, Energieaufwand etc.

Aktuelle und nächste Schritte

Protokollprüfung

- Weitere Anwendung auf industrielle Fallstudien (z.Z. in Arbeit)
- Abstraktionen für weitere Sprachen (z.Z. in Arbeit Java, C++)
- Effizientere Protokollprüfungsalgorithmen
- Berücksichtigung von Variableninhalten (analog Softwaremodellprüfung)
- Hierarchische Komponenten (Herleitung von Protokollen)
- Erweiterung auf objekt-orientierte Systeme (Protokolle für Klassen, Vererbung)
- Ausdrucksmächtigere Protokolle, z.B. constraint Automaten, counter Automaten

Andere nicht-funktionale Eigenschaften

- Spezialfall Terminierungsanalyse (Weißbach/Zimmermann 2010)
- Terminierungsanalyse als SLA-Methode (Weißbach/Zimmermann 2011)
- Analyse von Lebendigkeitsbedingungen (z.B. Verklemmungsfreiheit)
- Analyse von Zeit-, Speicher-, Energieaufwand etc.

Danke an unsere Studenten

Monique Argus
Christian Blaar
Rene Franke
Franceso Freder
Christian Heike
Hannes Hellmich
Johannes Jahn
Johannes-Edgar Krause
Stefan Prätsch
Christoph Panser
Rebecca Pötschke
Andreas Rudolph