Software Product Lines

Concepts, Analysis and Implementation

Variabilitätsmodellierung im Lösungsraum

Dr. Malte Lochau

Malte.Lochau@es.tu-darmstadt.de

© author(s) of these slides 2008 including research results of the research network ES and TU Darmstadt otherwise as specified at the respective slide

Inhalt

I. Einführung

- Motivation und Grundlagen
- Feature-orientierte Produktlinien

II. Produktlinien-Engineering

- Feature-Modelle und Produktkonfiguration
- Variabilitätsmodellierung im Lösungsraum
- Programmierparadigmen für Produktlinien

- Annotativ
- Komposition
- Transformativ
- Intrinsisch
- CVL

III. Produktlinien-Analyse

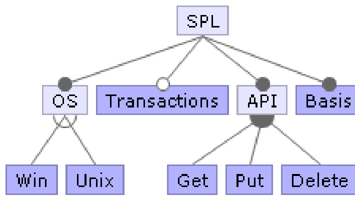
- Feature-Interaktion
- Testen von Produktlinien
- Verifikation von Produktlinien

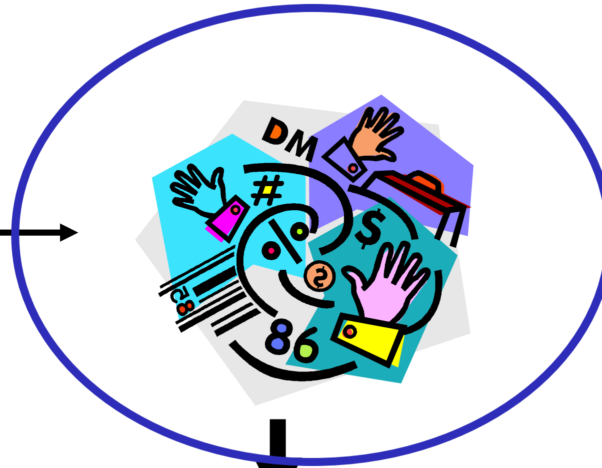
IV. Fallbeispiele und aktuelle Forschungsthemen

Software-Product-Line Engineering

Domain Eng.

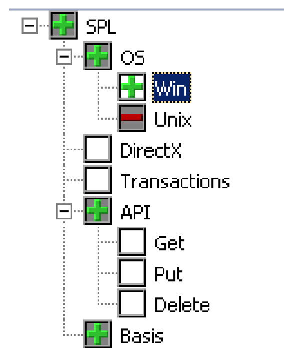
Feature-Modell





Wiederverwendbare Implementierungs-artefakte

Application Eng.

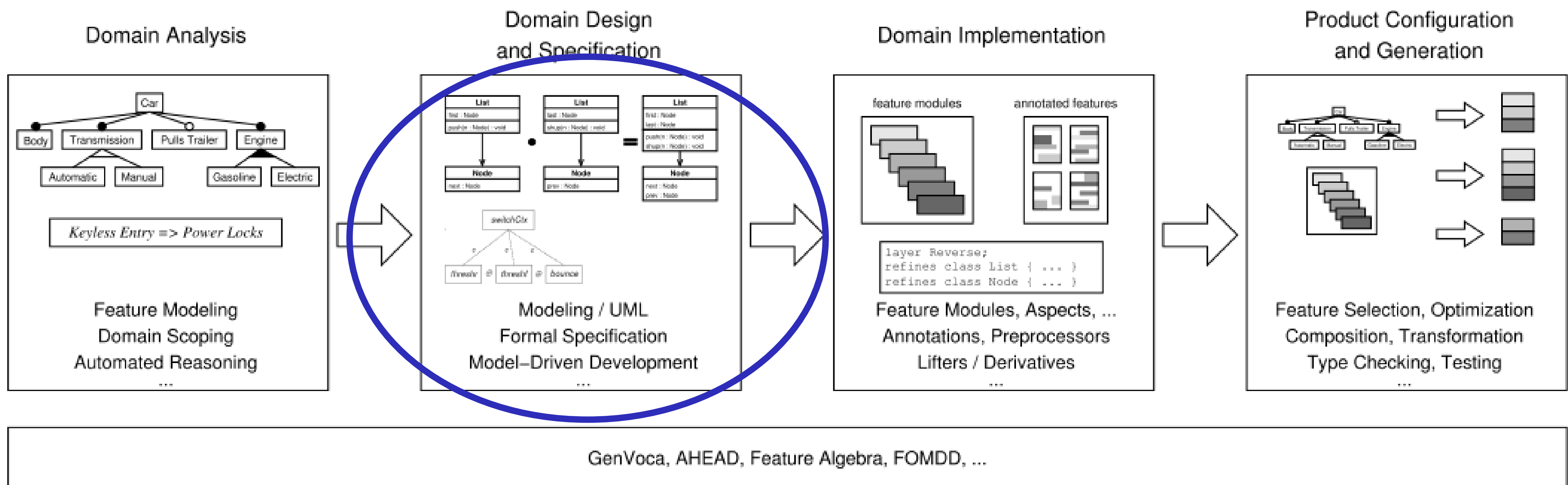


Feature-Auswahl

Generator

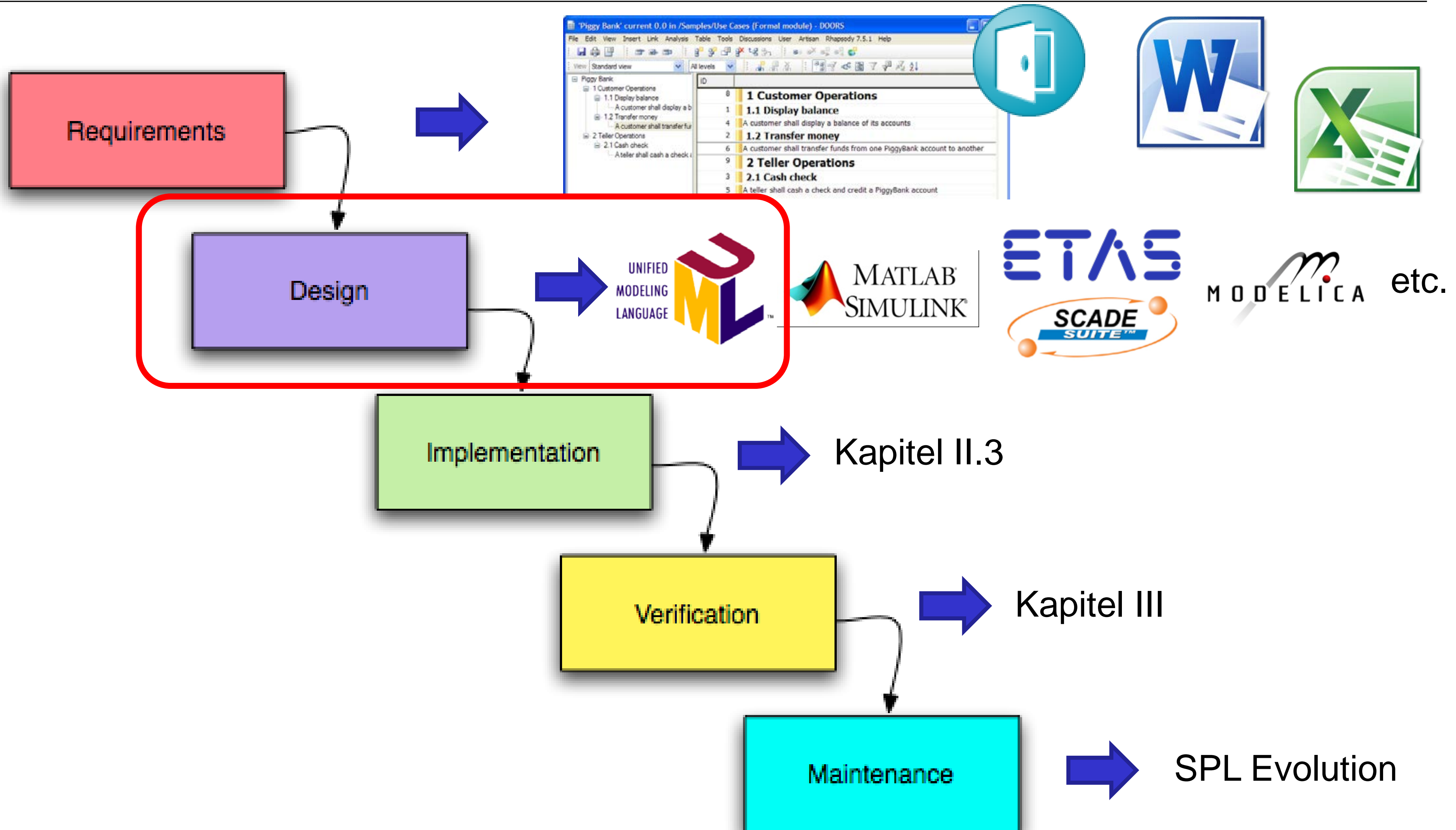
Fertiges Program

FOSD - Modellierung



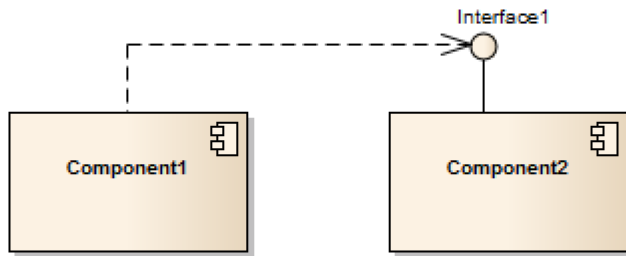
- Modelle als Abstraktionssicht auf das zu entwerfende System
- Features als Konzept zur Variabilitätsmodellierung

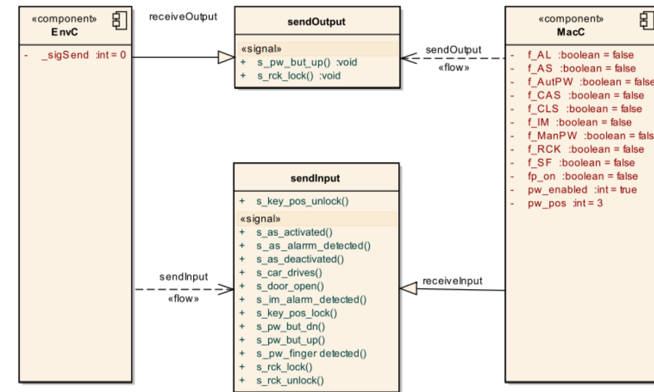
(Grafische) Modellierungssprachen



Grafische Design-Modelle

Struktur



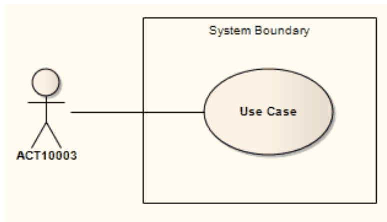


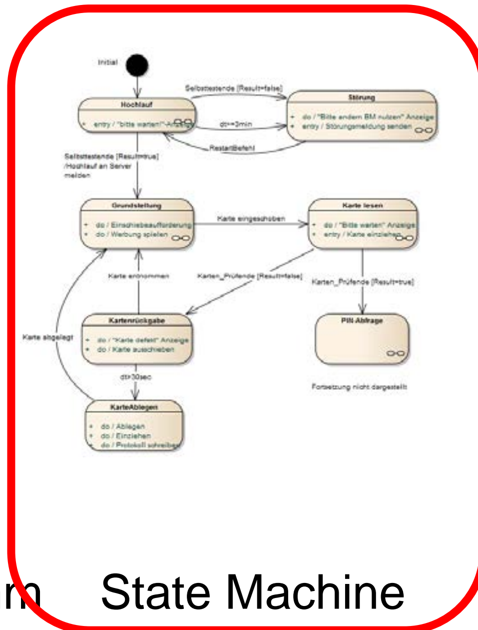
Komponentendiagramm

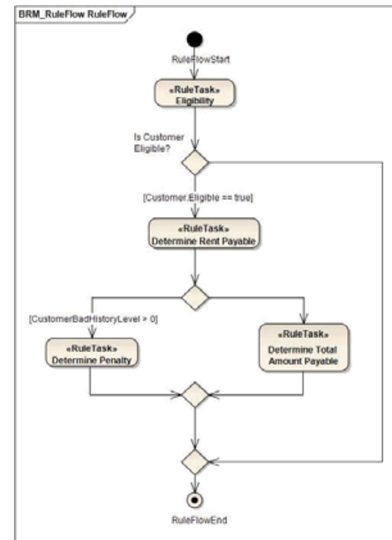
Klassendiagramm

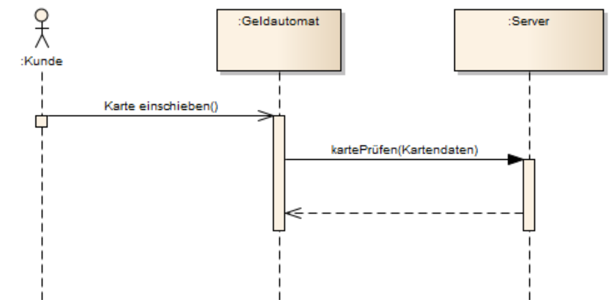
etc.

Verhalten









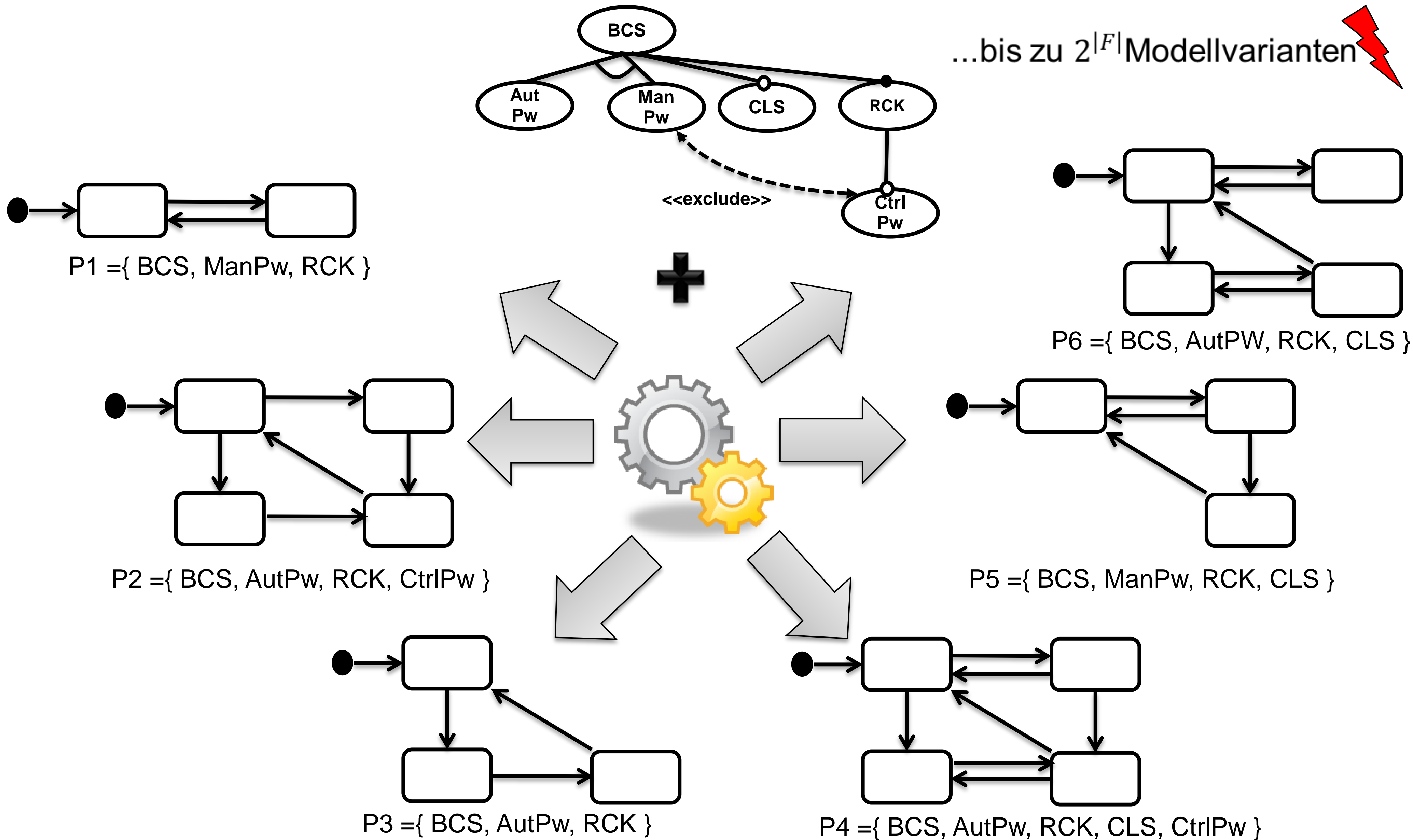
Use Case Diagramm

State Machine

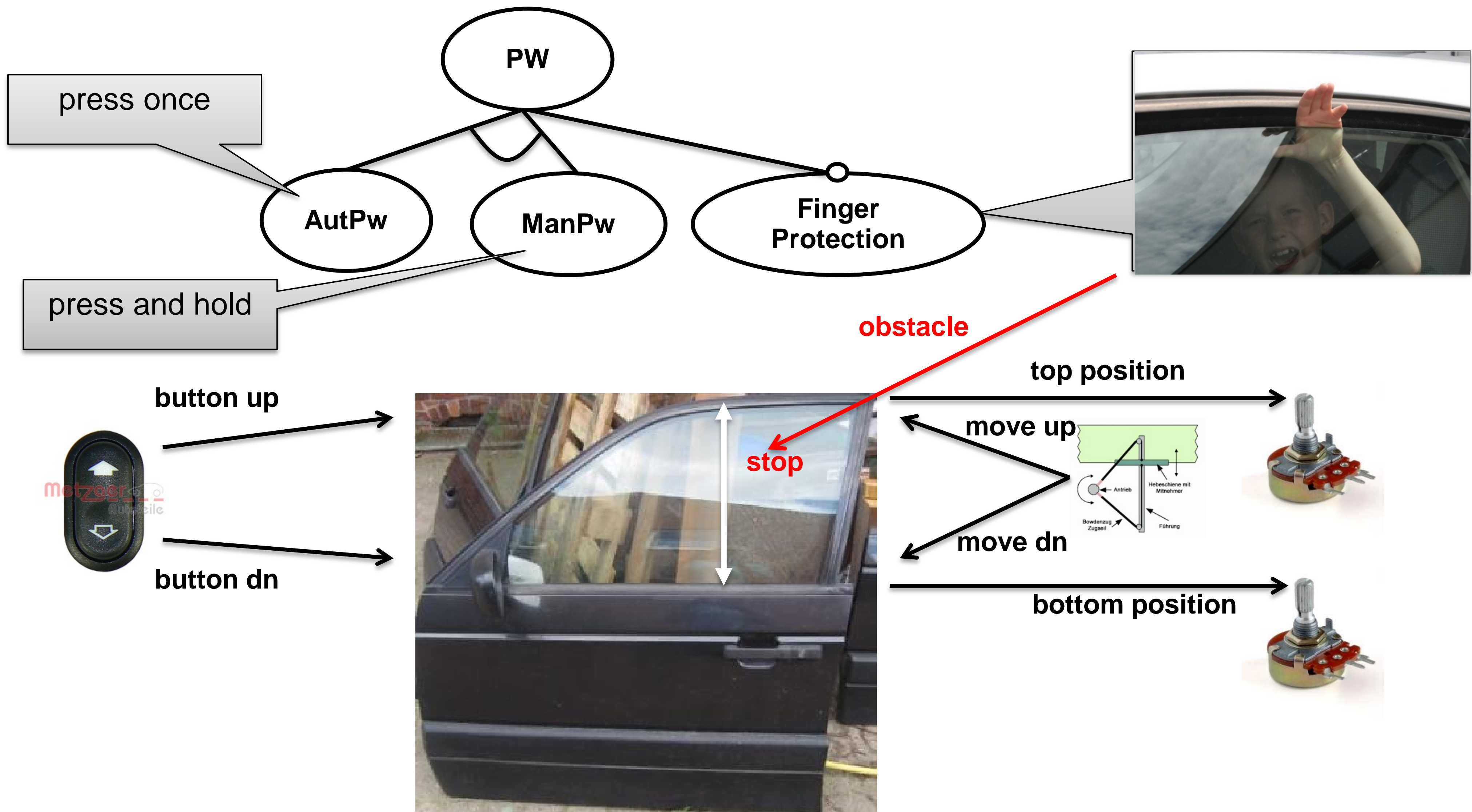
Aktivitätsdiagramm

Sequenzdiagramm

Modellvarianten

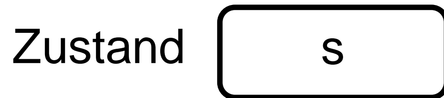


Durchgängiges Beispiel: BCS-Small

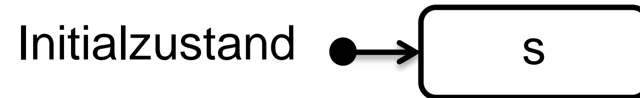


Exkurs: UML State Machines (1/2)

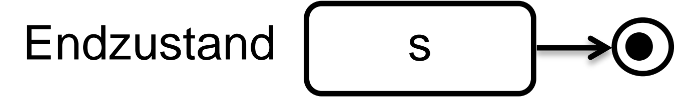
Zustand



Initialzustand

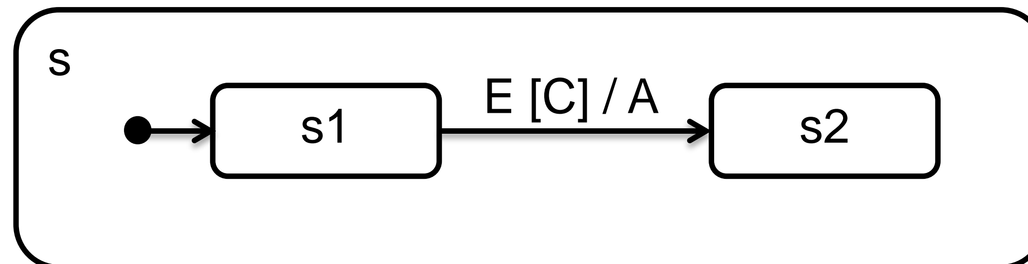


Endzustand

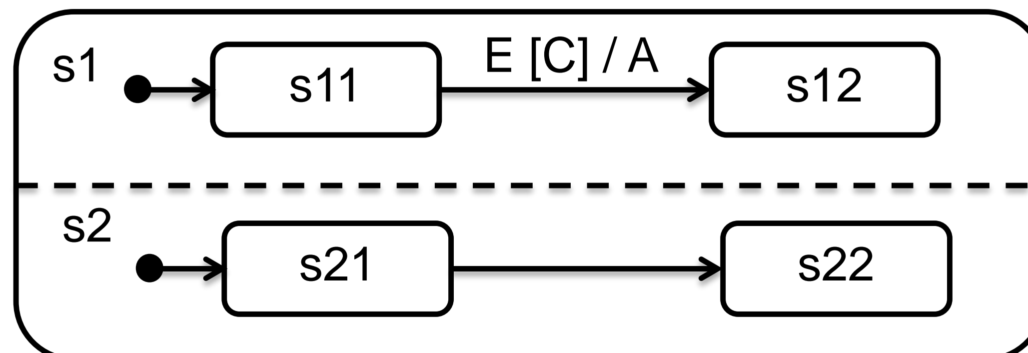


Transition

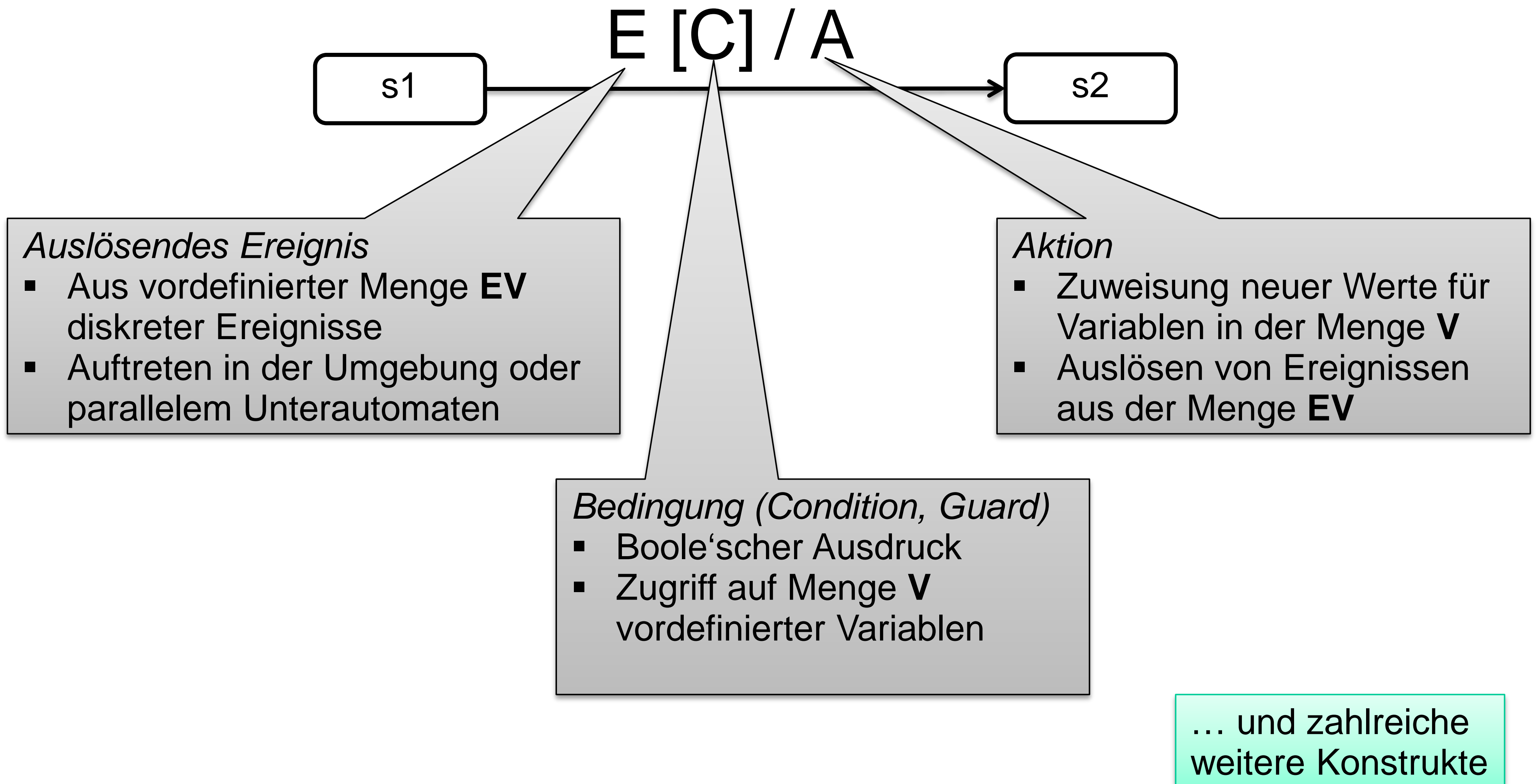
XOR Zustand



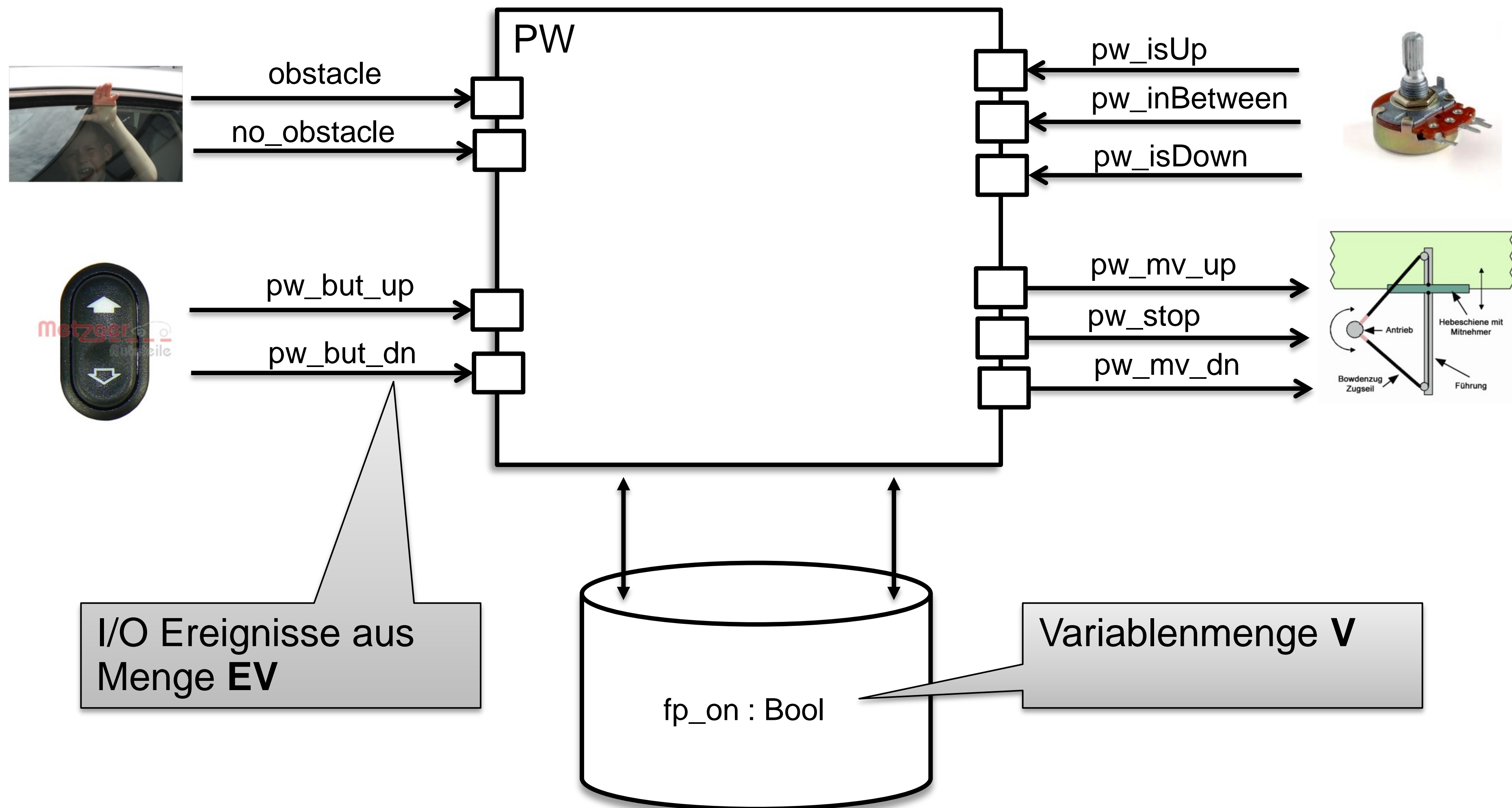
AND Zustand



Exkurs: UML State Machines (1/2)

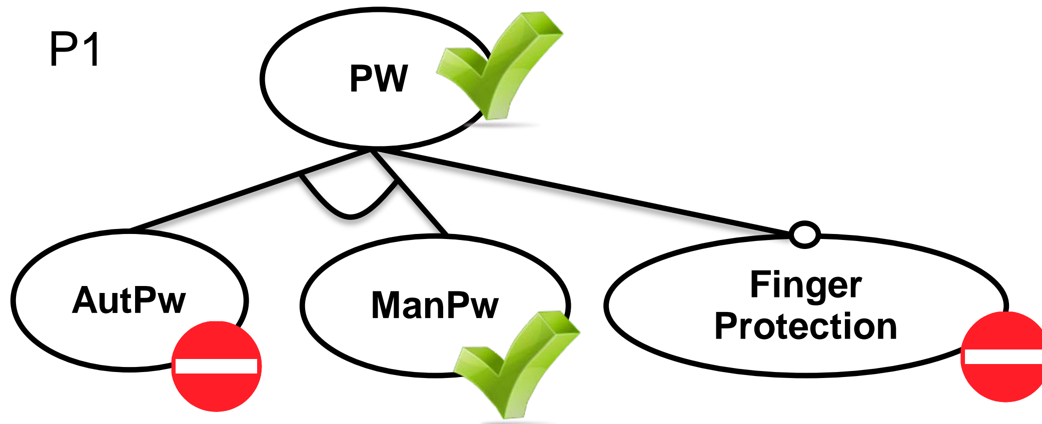


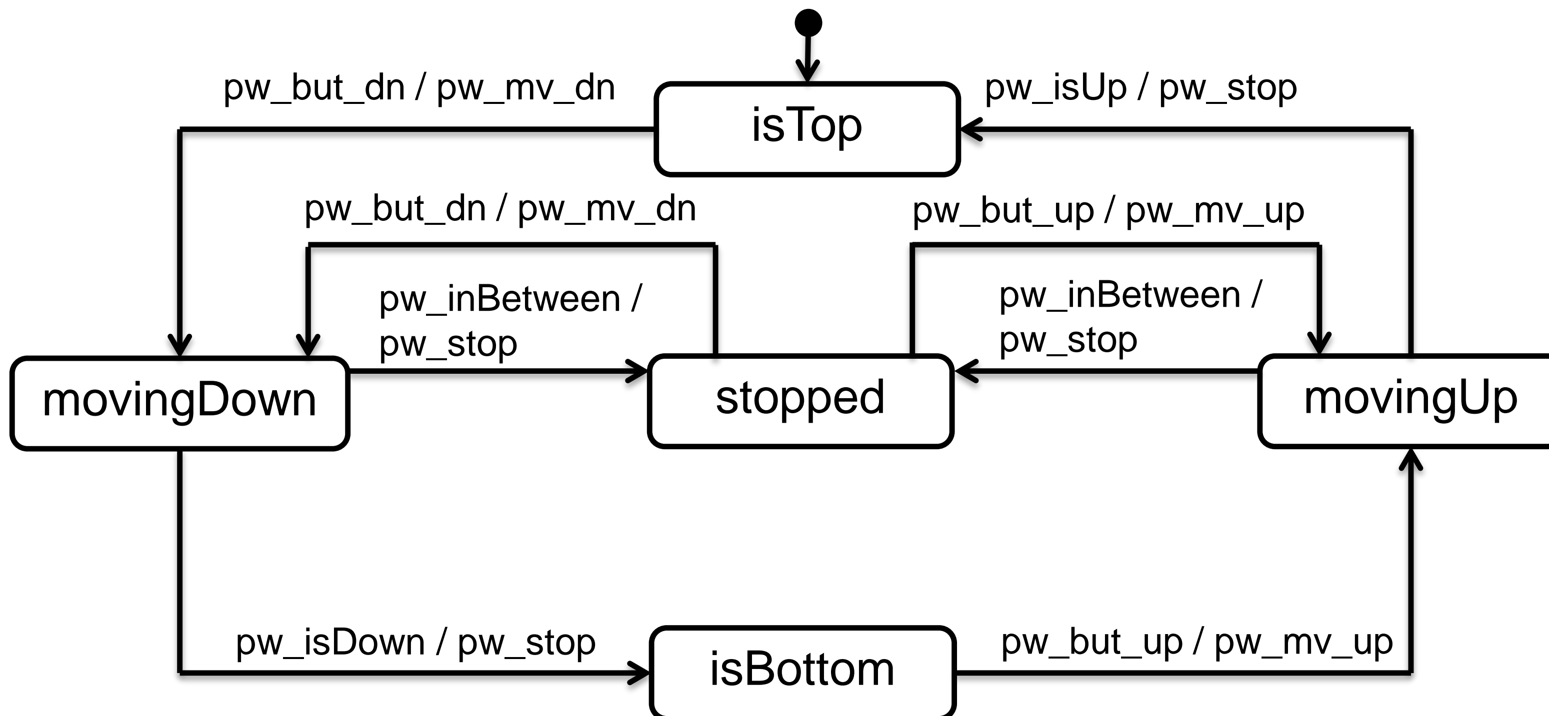
Beispiel: BCS-Small



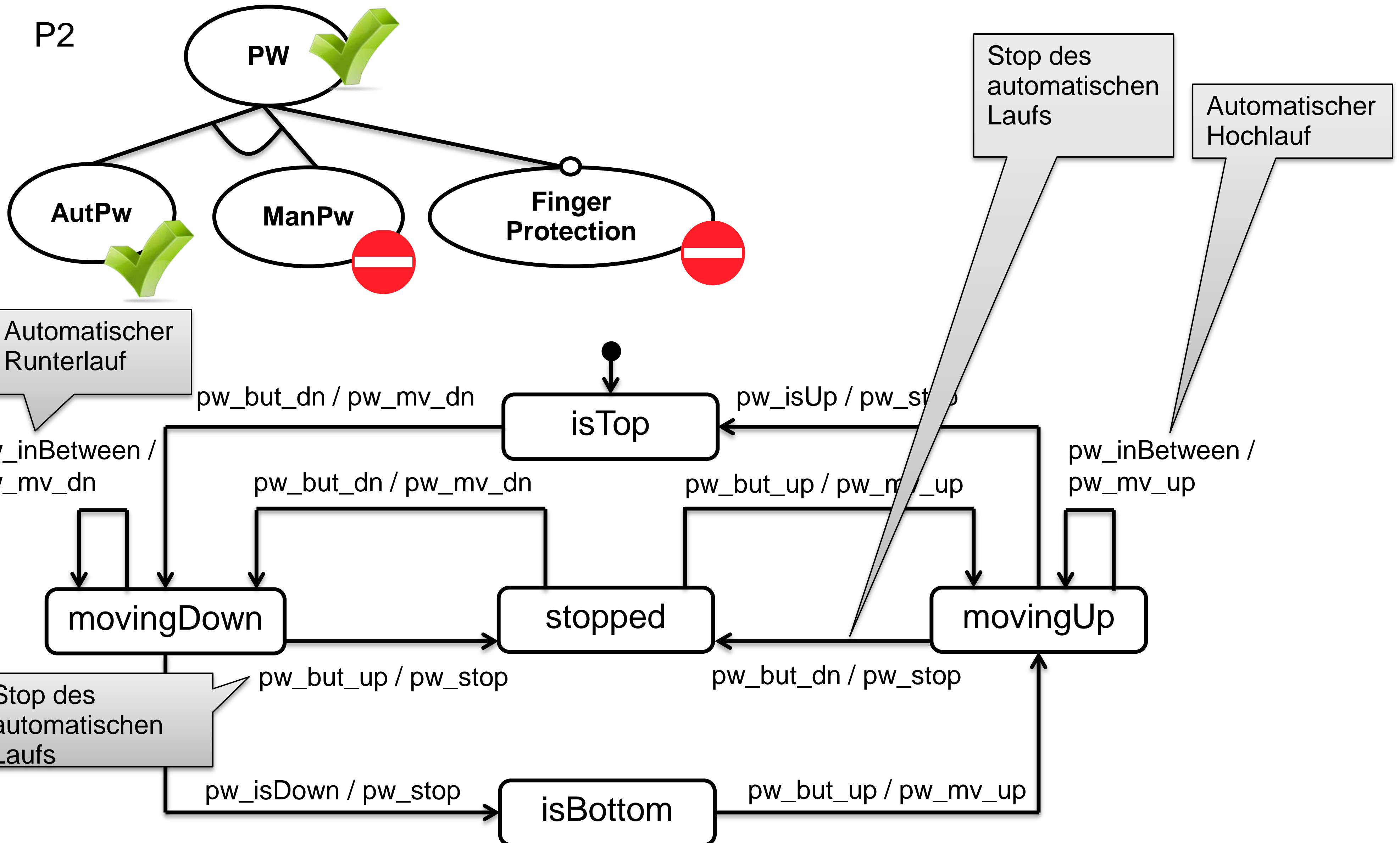
State Machine Varianten von BCS-Small

P1



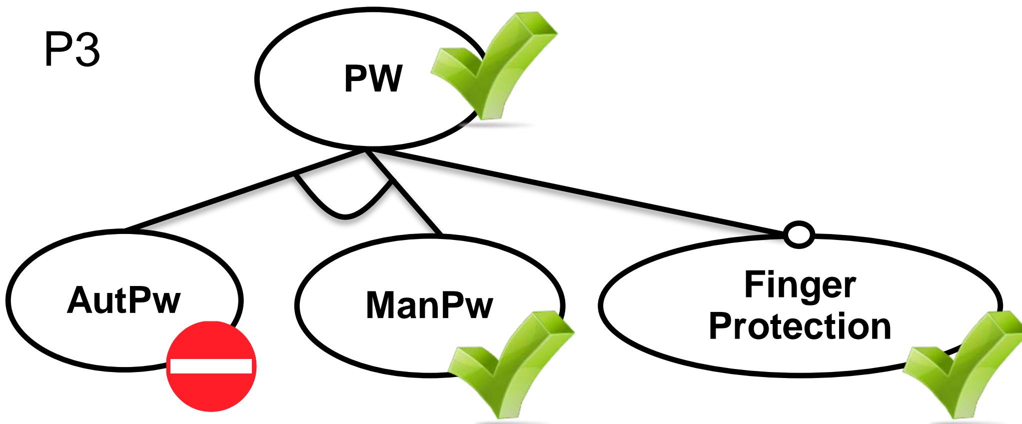


State Machine Varianten von BCS-Small



State Machine Varianten von BCS-Small

P3



Status des Einklemmschutzes

Hochfahren blockieren

pw_but_dn / pw_mv_dn

isTop

pw_isUp / pw_stop

pw_but_dn / pw_mv_dn

pw_but_up [NOT fp_on] / pw_mv_up

pw_inBetween / pw_stop

pw_inBetween / pw_stop

movingDown

stopped

movingUp

Hochfahren blockieren

pw_isDown / pw_stop

isBottom

pw_but_up [NOT fp_on] / pw_mv_up

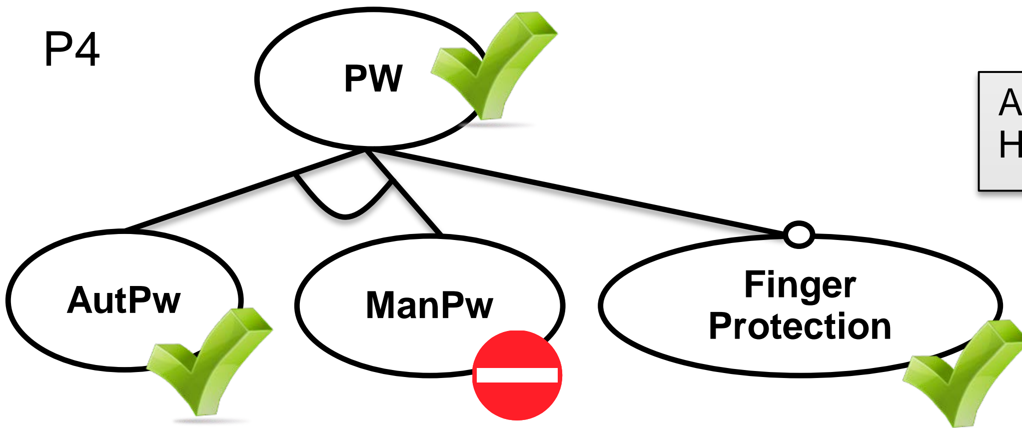
FP_off

no_obstacle / fp_on :=false

obstacle / fp_on :=true

FP_on

State Machine Varianten von BCS-Small



Automatisches Hochfahren stoppen

Automatisches Hochfahren blockieren

isTop
movingDown
stopped
movingUp
isBottom
FP_off
FP_on

pw_but_dn / pw_mv_dn

pw_inBetween / pw_mv_dn

pw_but_dn / pw_mv_dn

pw_but_up / pw_stop

pw_isDown / pw_stop

pw_is / pw_stop

pw_but_up [NOT fp_on] / pw_mv_up

[fp_on] / pw_stop

pw_but_dn / pw_stop

pw_but_up [NOT fp_on] / pw_mv_up

pw_inBetween [NOT fp_on] / pw_mv_up

no_obstacle / fp_on :=false

obstacle / fp_on :=true

Anforderungen an Konzepte zur Variabilitätsmodellierung (1/2)

Modellierungskonzept: Wie wird Variabilität spezifiert?

- Durch existierende Konstrukte der Modellierungssprache oder durch neue Konstrukte?
- Sprachunabhängig oder sprachspezifisch?

Repräsentation: Wie werden gemeinsame und variable Teile dargestellt?

- Vorhandene Syntax, spezielle Syntax, separate Darstellung, ….
- Wie wird eine Variante abgeleitet und materialisiert?
- Müssen existierende Modellierungs-Tools erweitert werden?

Granularität: Welche Modellelemente können variieren?

- Auf welcher Detailebene können Modellelemente variieren?
- Strukturierung der Variabilität modular/separiert oder integriert?

Anforderungen an Konzepte zur Variabilitätsmodellierung (2/2)

Flexibilität: Wieviel Freiheit hat der Modellierer?

- Verschiedene Darstellungsmöglichkeiten?
- Definition von (Anti-)Pattern und Best Practices?
- Transformation in Normalformen?

Nutzbarkeit und Analysierbarkeit

- Intuitiv lernbar und nutzbar?
- Sind existierende Modelle einfach zu verstehen?
- Tool-Support und Skalierbarkeit von Analysetechniken?

Anpassbarkeit und Wartung

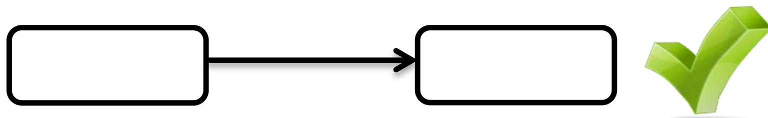
- Von Modellen?
- Der Sprache selbst?

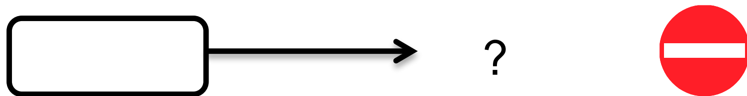
Wohlgeformtheit von Modellvarianten (1/3)

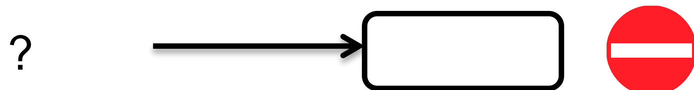
- Modellelemente stehen in Beziehung zu anderen Modellelementen.
- Variabilitätsmodellierungsansatz muss sicherstellen, dass alle ableitbaren Varianten **wohlgeformt** sind

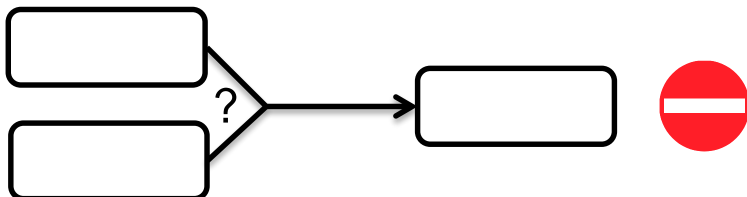
Beispiel: State Machine Modelle

1. Transitionen haben genau einen Start- und einen Zielzustand



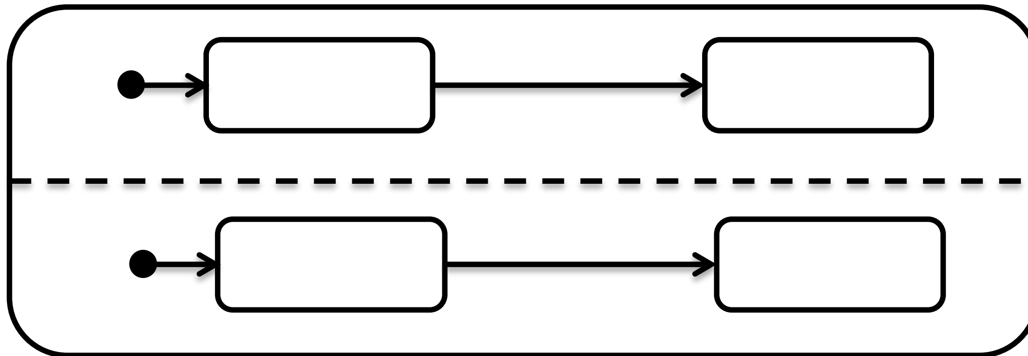


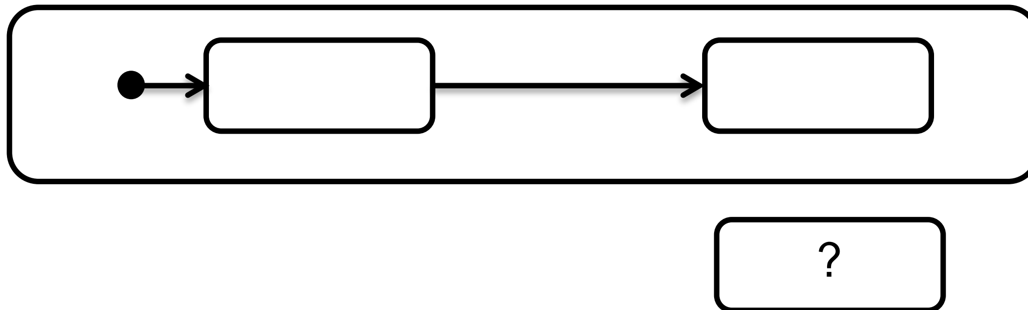
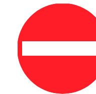
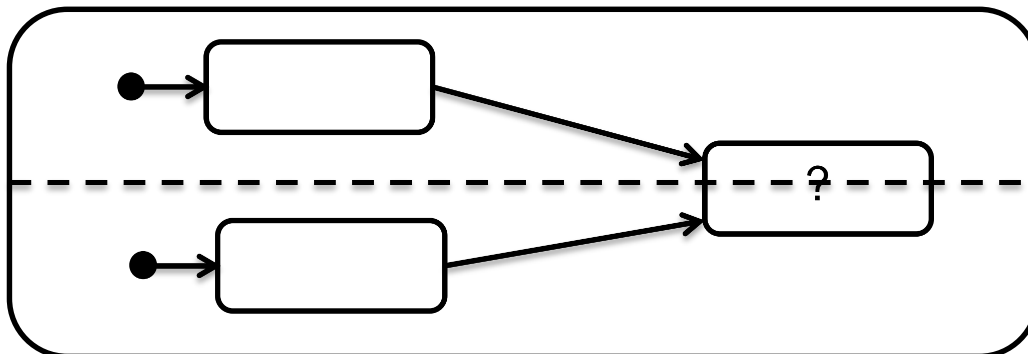


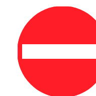


Wohlgeformtheit von Modellvarianten (2/3)

2. Zustände befinden sich in genau einem (Unter-)Automat
(Annahme eines impliziten Wurzelautomaten)





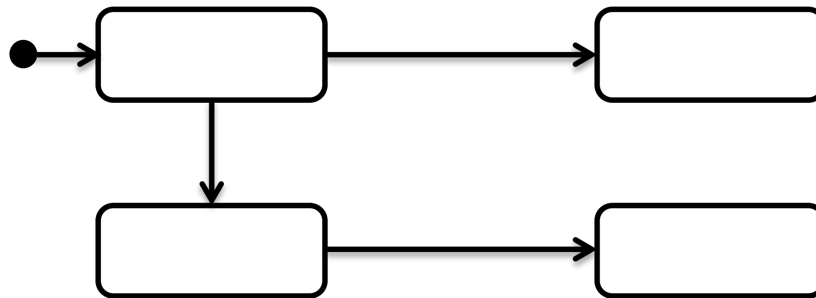
Wohlgeformtheit von Modellvarianten (3/3)

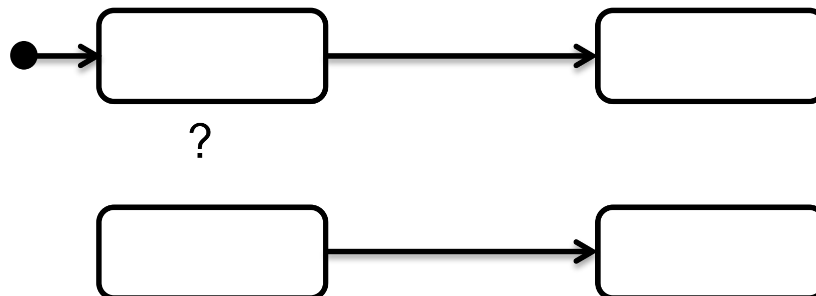
3. Analog: Unterautomaten sind in genau einen Zustand hineingeschachtelt

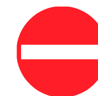
…

4. Nicht-lokale Kriterien:

- Jeder Unterautomat hat genau einen Initialzustand
- Jeder Zustand ist vom Initialzustand aus erreichbar

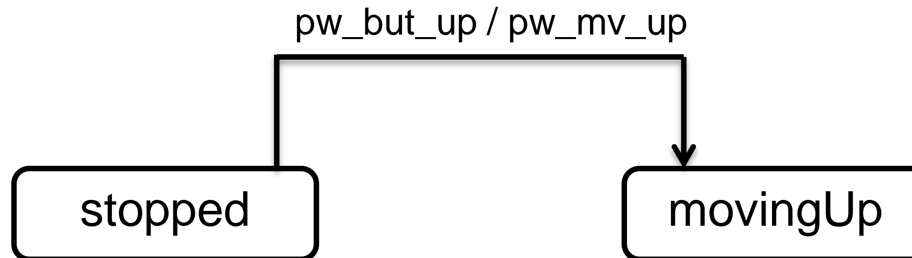




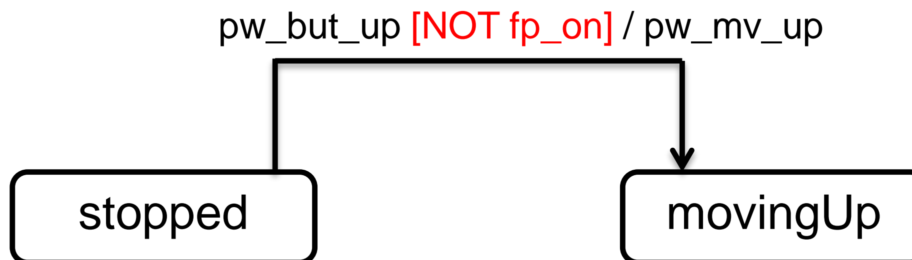


Prüfung der Wohlgeformtheit ohne Betrachtung jeder Modellvariante?

Granularität der Variabilität



Konfiguration ohne Einklemmschutz



Konfiguration mit Einklemmschutz

Die Transitionen sind identische Modell-elemente, dessen Transitionslabel je nach Konfiguration variiert.

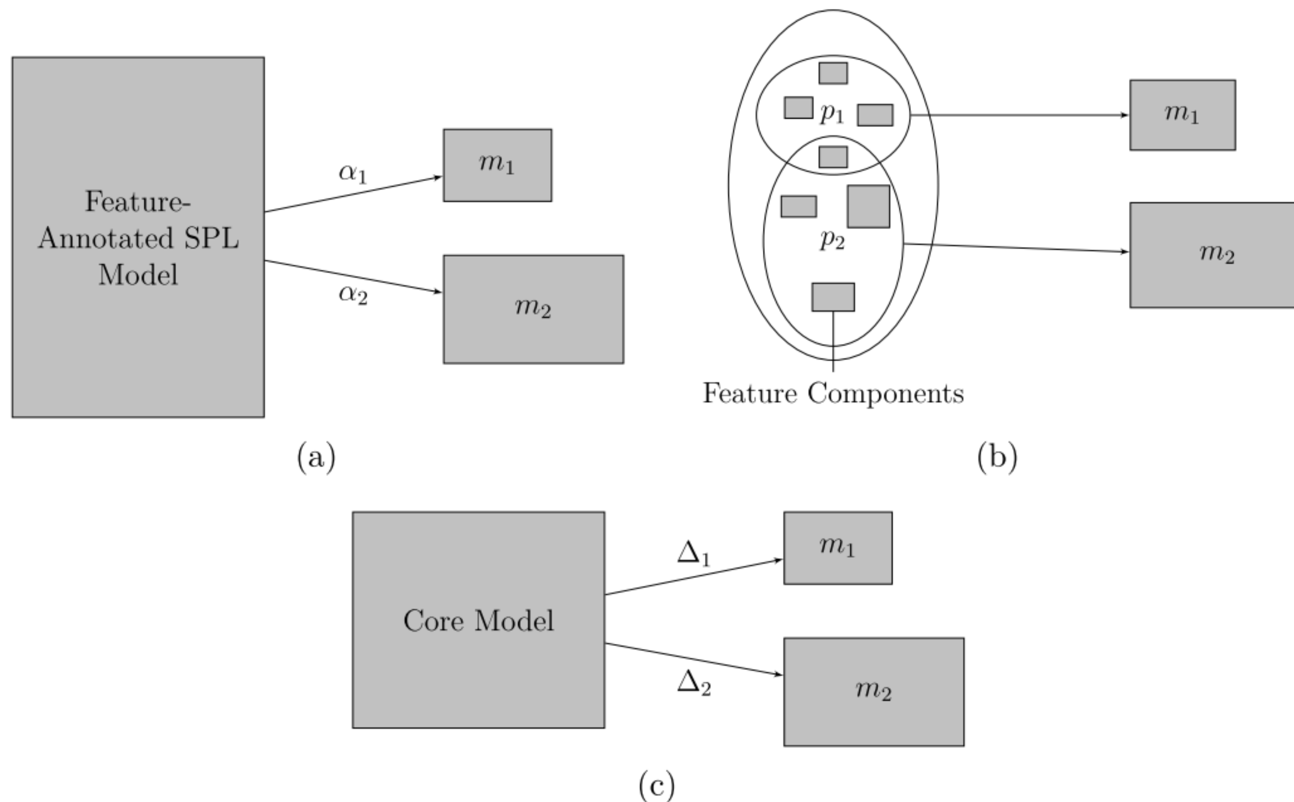
Die Transitionen sind verschie-dene Modellelemente mit gleichem Start- und Ziel-zustand, von denen je nach Konfiguration genau eine in der Modellvariante enthalten ist.

…

Beide Teil-automaten sind verschieden, da sie verschiedene Transitionen aufweisen

feingranular → grobgranular

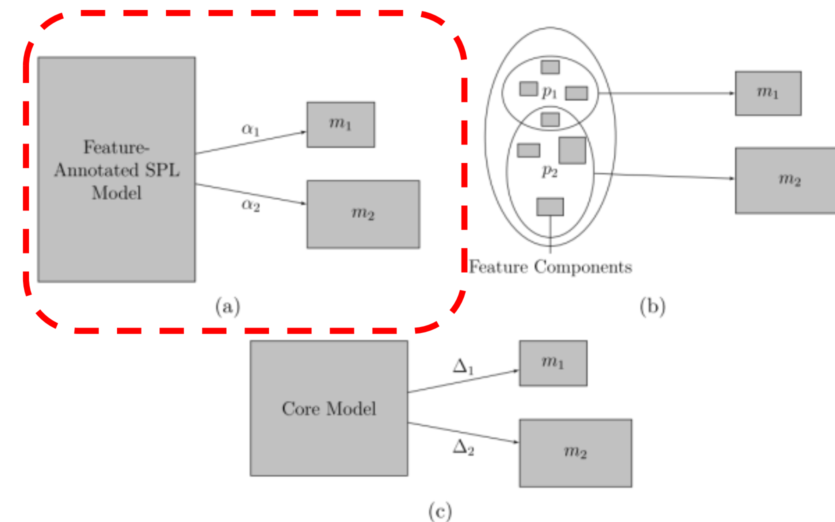
Variabilitätsmodellierung - Ansätze

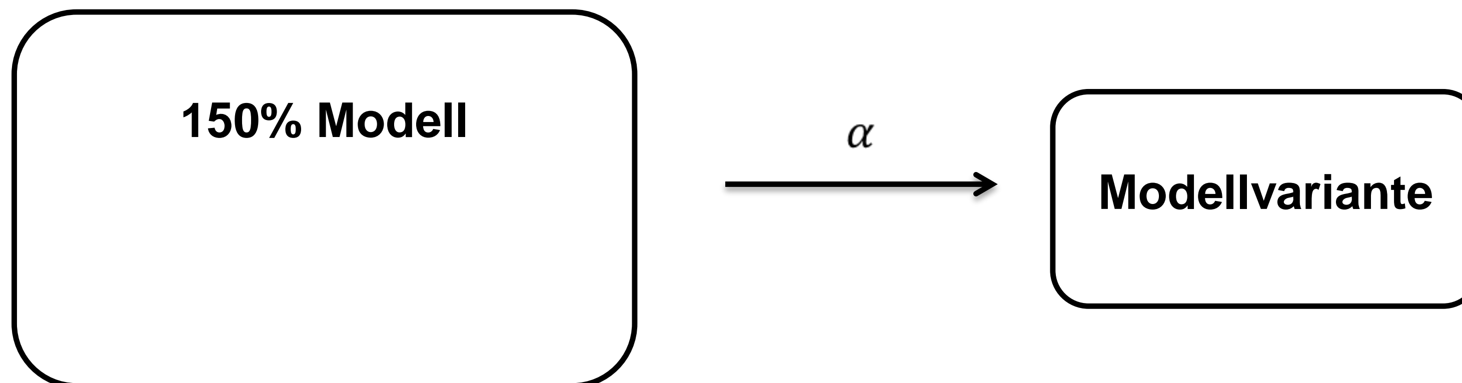


(a) Annotationsbasierte Variabilitätsmodellierung (negative Variabilität) [*Czarnecki and Antkiewicz, 2005*]
(b) Kompositionsbasierte Variabilitätsmodellierung (positive Variabilität) [*Prehofer, 1997*]
(c) Transformationsbasierte Variabilitätsmodellierung [*Schaefer, 2010*]
(d) (Intrinsische Variabilitätsmodellierung) [*Larsen et al., 2007*]

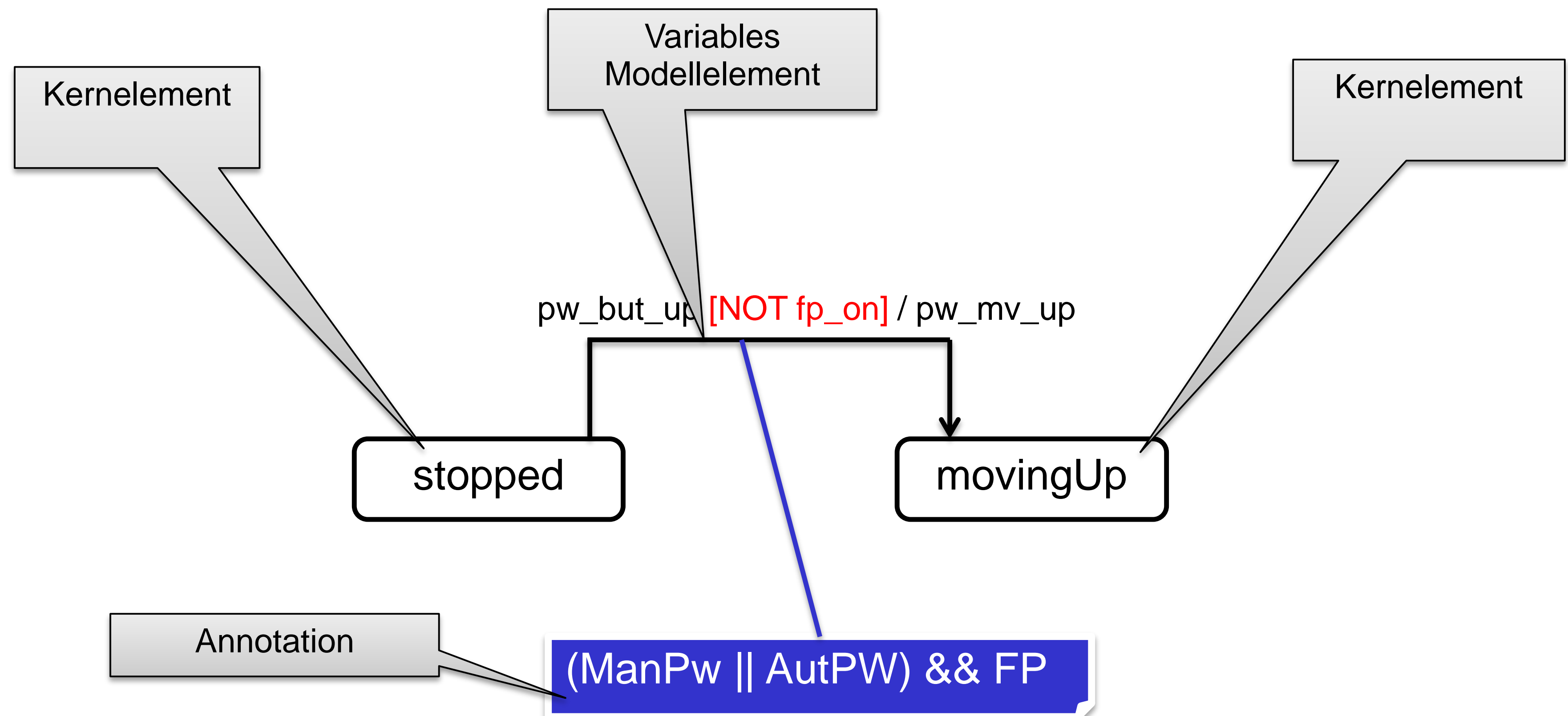
Annotative Variabilitätsmodellierung

- Integration sämtlicher Modellvarianten in ein **150% Modell**
- Variable Modellelemente werden mit **Selektionsbedingungen** über Feature-Parametern annotiert (presence conditions)
- **Negative Variabilität:** Ableitung einer Modellvariante durch Entfernen von Modellelementen, deren Annotation nicht kompatibel zur Konfiguration ist.

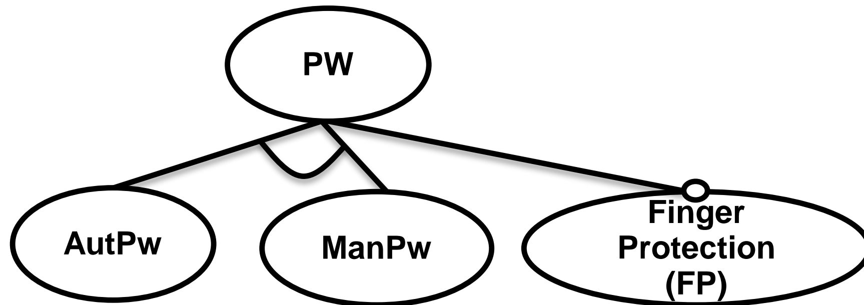




Beispiel: Annotiertes Modell



Beispiel: BCS-Small



Annahme: Variable fp_on ist konstant *false*, falls FP nicht ausgewählt

AutPw && FP

FP

AutPw

ManPw

AutPw

ManPw

AutPw

AutPw

isTop

stopped

movingDown

movingUp

isBottom

FP_off

FP_on

pw_but_dn / pw_mv_dn

pw_inBetween / pw_mv_dn

pw_but_dn / pw_mv_dn

pw_inBetween / pw_stop

pw_but_up / pw_stop

pw_isDown / pw_stop

pw_isUp / pw_stop

pw_inBetween [NOT fp_on] / pw_mv_up

pw_but_up [NOT fp_on] / pw_mv_up

pw_inBetween / pw_stop

[fp_on] / pw_stop

pw_but_dn / pw_stop

pw_but_up [NOT fp_on] / pw_mv_up

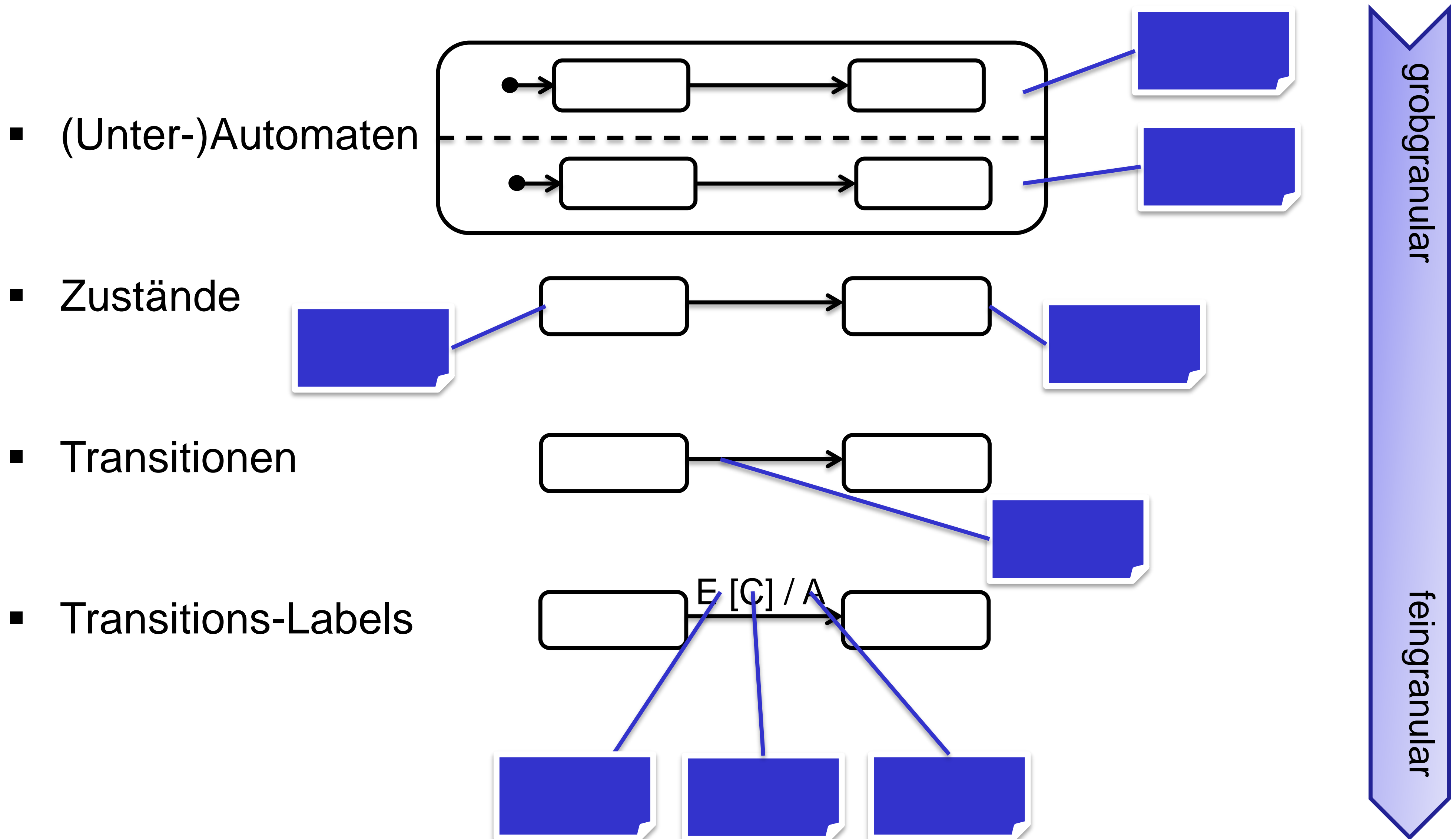
no_obstacle / fp_on :=false

obstacle / fp_on :=true

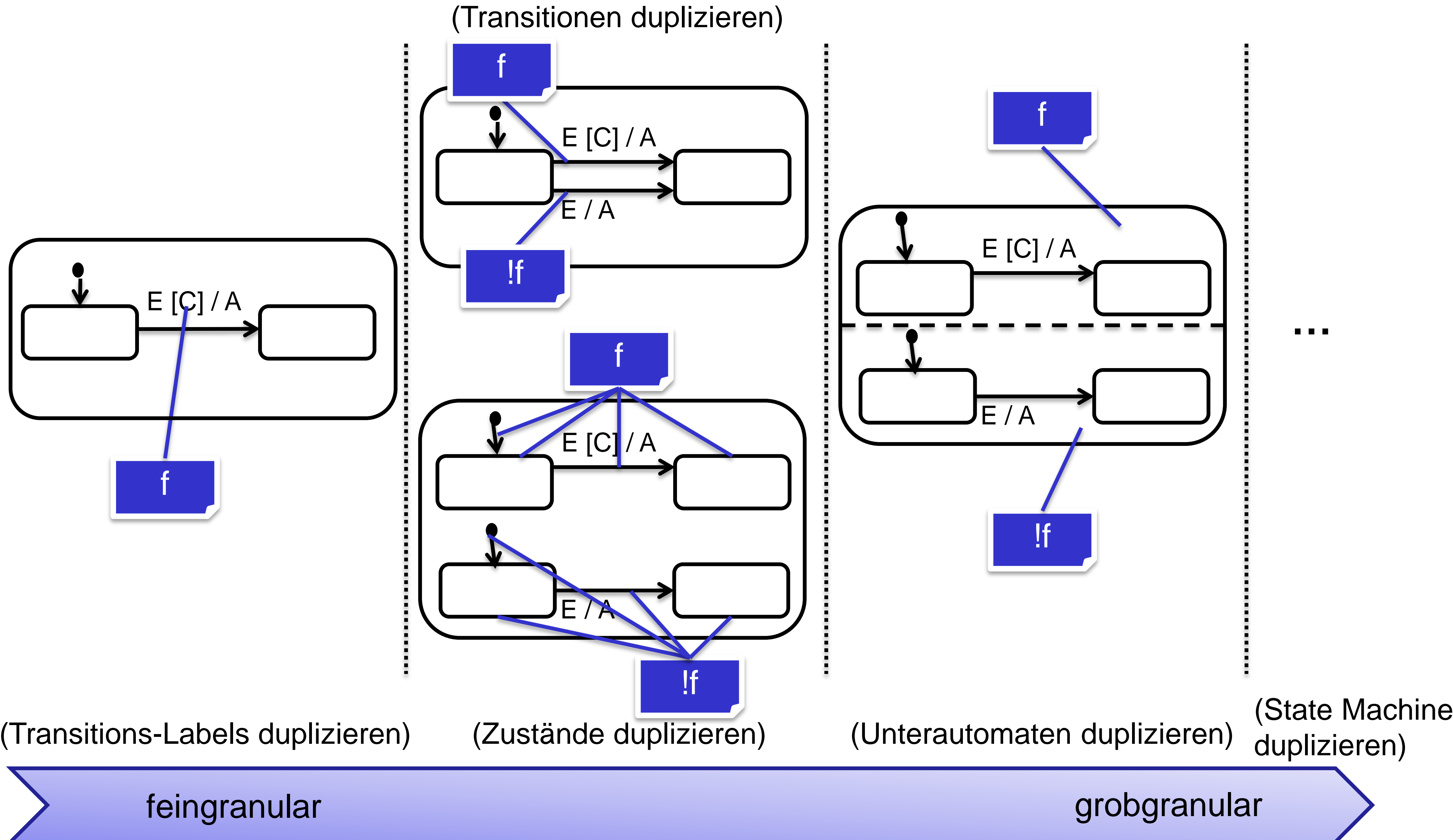
Variabilitätsgranularität in annotierten State Machines

Welche Modellelemente von State Machines können variabel sein?

- (Unter-)Automaten
- Zustände
- Transitionen
- Transitions-Labels



Granularität: Auswirkungen



Granularität: Diskussion

Für feinere Granularität spricht:

- Minimierung der Modellgröße durch Reduktion der Redundanz / Duplikation von Modellelementen
- Maximale Wiederverwendung gemeinsamer Modellelemente zwischen Varianten

Gegen feinere Granularität spricht:

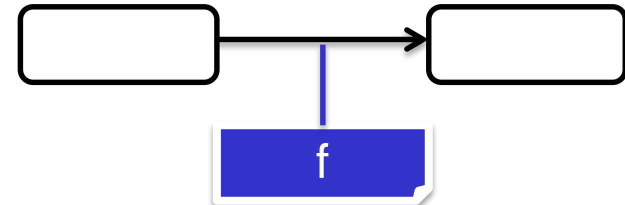
- Führt zu unstrukturierter Variabilität
- Erschwert Verständnis, Wartung und Analyse der Modelle
- Führt evtl. häufiger zu nicht wohlgeformten 150% Modellen

Wir nehmen nachfolgend an (O.B.d.A.):

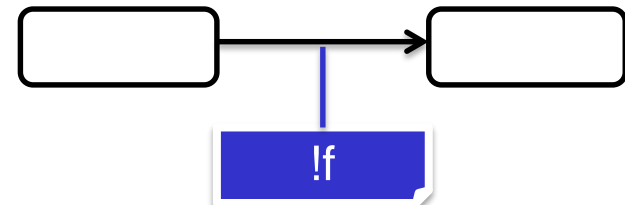
1. Das zu einem annotierten State Machine Modell gehörende Feature-Modell ist erfüllbar
2. Zustände, Transitionen und Unterautomaten können annotiert sein
3. Das 150% Modell ist wohlgeformt

Arten von Annotationen

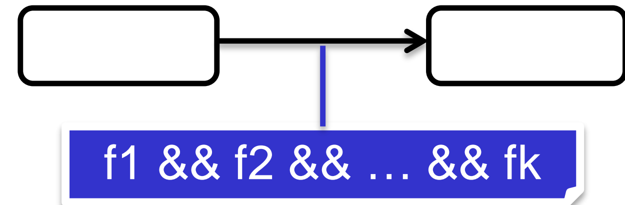
- Feature-spezifische Elemente



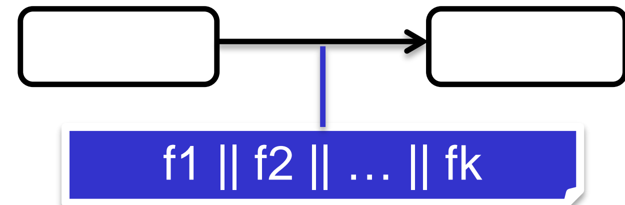
- Irrelevante Elemente



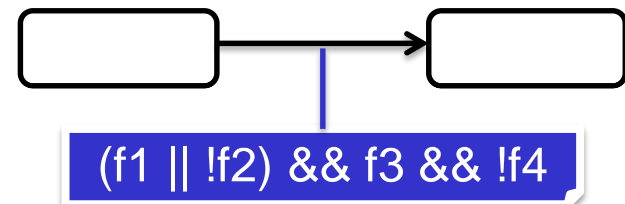
- Elemente für Feature-Kombinationen (Derivatives, Glue-Elements)



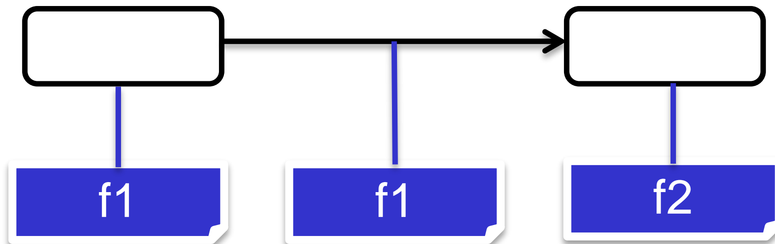
- Gemeinsame Elemente (Reuseable Elements)

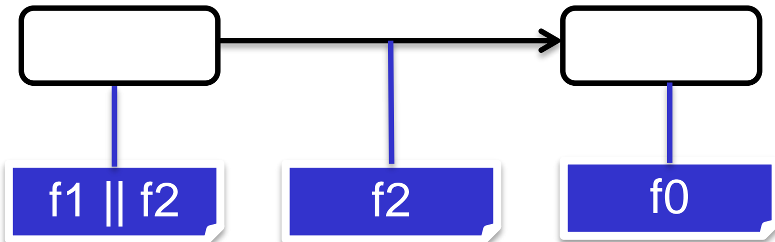


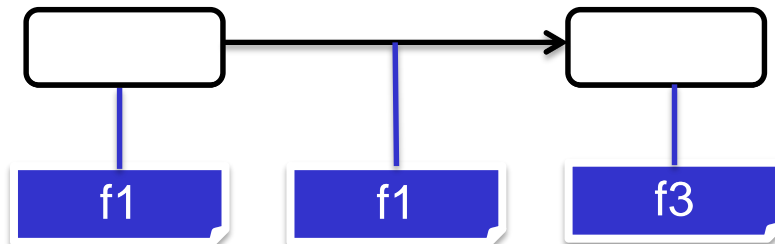
- Beliebig komplexe Annotationen

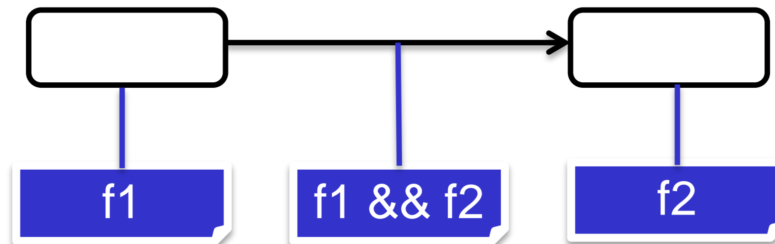


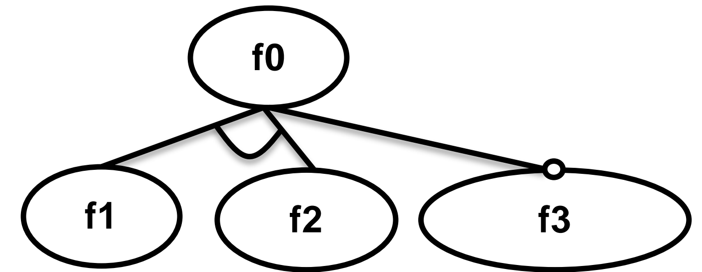
Annotierte Modelle: Beispiele

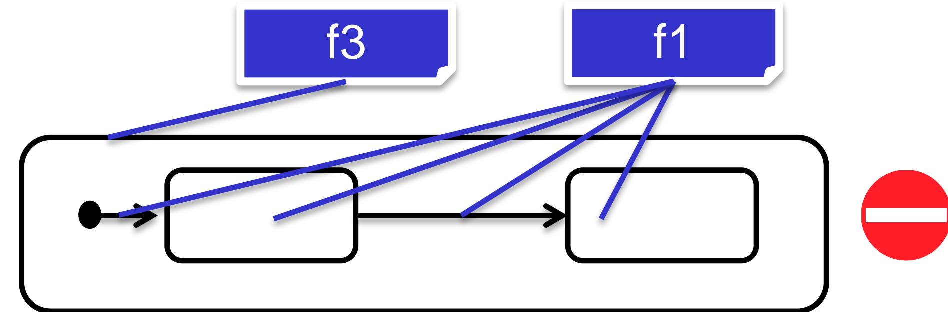


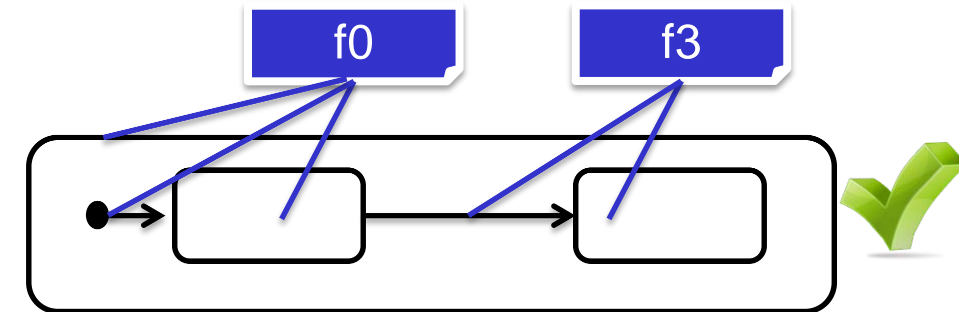


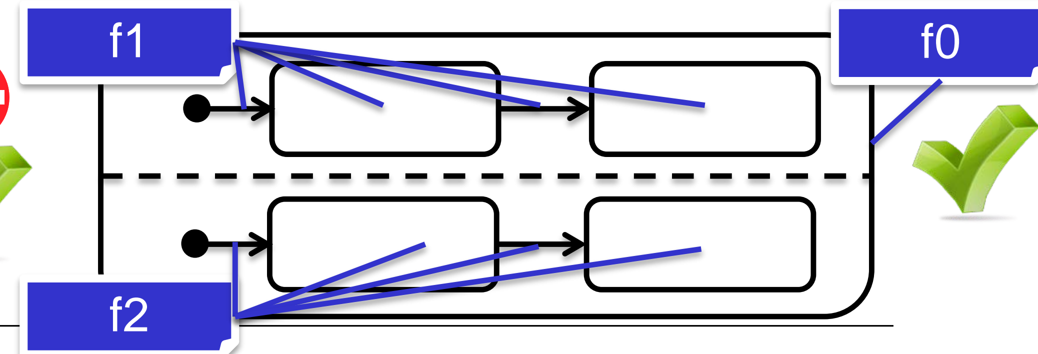












Wohlgeformtheit annotierter State Machine Modelle (1/2)

Prüfung der Wohlgeformtheit aller ableitbaren Modellvarianten eines annotierten Modells mittels SAT-Solver

Eingabe:

- Feature-Modell $\widetilde{FM}: (F \rightarrow \mathcal{B}) \rightarrow \mathcal{B}$
- State Machine Modell mit Annotationen $\varphi : (F \rightarrow \mathcal{B}) \rightarrow \mathcal{B}$ für Modellelemente $e \in (Zustände \cup Transitionen \cup Unterautomaten)$

Algorithmus: Prüfe für jedes Element e mit Annotation φ:

1. Validität der Annotation bzgl. des Feature-Modells:

$$\widetilde{FM} \land \varphi \nvdash false$$

(Jedes Modellelement kommt in mindestens einer Modellvariante vor)

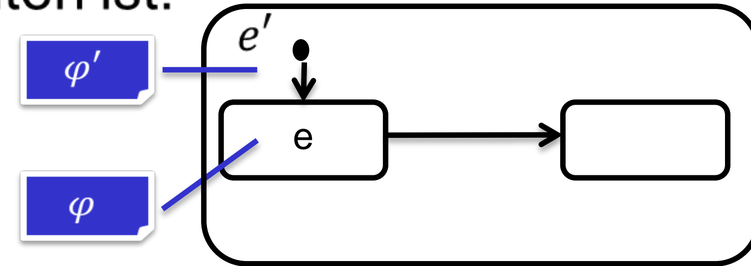
2. Falls $e \in Unterautomaten$ und e ist der äußerste Automat der Hierarchie:

$$\vdash \widetilde{FM} \rightarrow \varphi$$

(Keine leeren Modellvarianten)

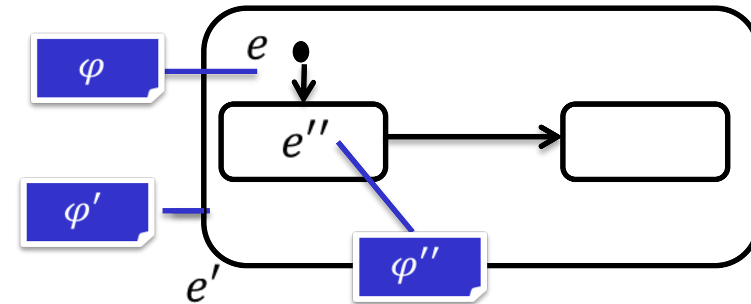
Wohlgeformtheit annotierter State Machine Modelle (2/2)

3. Falls $e \in Zustände$ und $e' \in Unterautomaten$ sei der (Unter-) Automat (annotiert mit φ'), in dem e direkt enthalten ist:



$$\vdash \varphi \rightarrow \varphi'$$

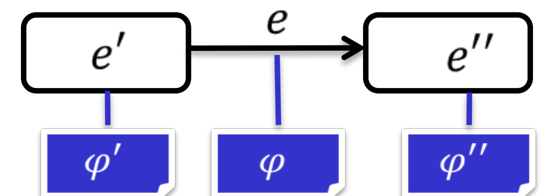
4. Falls $e \in Unterautomaten$, $e' \in Zustände$ sei der Zustand (annotiert mit φ'), in den e direkt eingeschachtelt ist und $e'' \in Zustände$ sei der Initialzustand von e (annotiert mit φ''):



$$\vdash \varphi \rightarrow (\varphi' \wedge \varphi'')$$

5. Falls $e \in Transitionen$, $e' \in Zustände$ sei der Ausgangszustand von e (annotiert mit φ') und $e'' \in Zustände$ sei der Zielzustand von e (annotiert mit φ''):

$$\vdash \varphi \rightarrow (\varphi' \wedge \varphi'')$$



Algorithmus zur Prüfung der Wohlgeformtheit: Eigenschaften

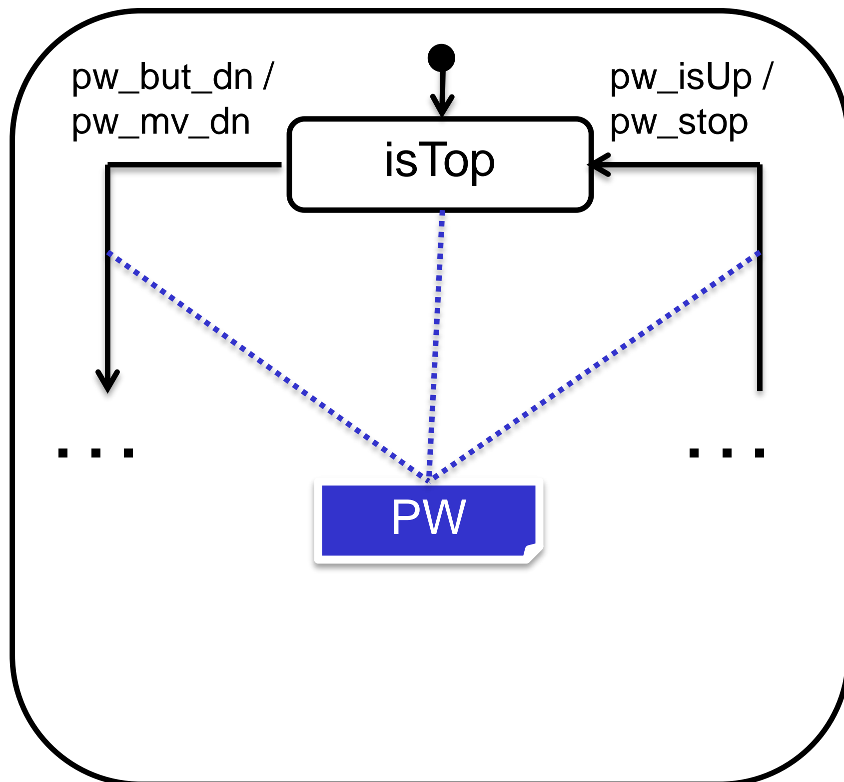
Wird ein annotiertes State Machine Modell erfolgreich durch den Algorithmus geprüft, dann ist jede aus diesem Modell ableitbare Modellvariante wohlgeformt.

Wird ein annotiertes State Machine Modell erfolgreich durch den Algorithmus geprüft, dann enthält das Model nur relevante Modellelemente, die in mindestens einer Modellvariante vorkommen.

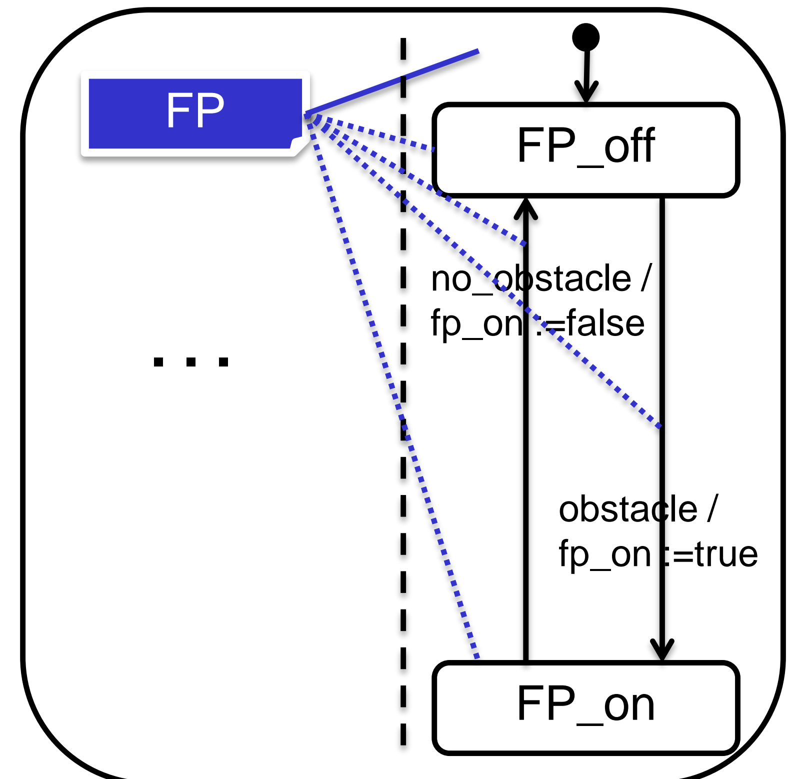
Partiell annotierte Modelle

Einschränkung: Der Algorithmus verlangt, dass jedes Modellelement annotiert ist. Häufig möchte man aber auf explizite Annotationen verzichten.

1. *Kernelemente, die Teil jeder Modellvariante sind.*



2. *Elemente, deren Präsenz sich aus ihrem Kontext ergeben.*

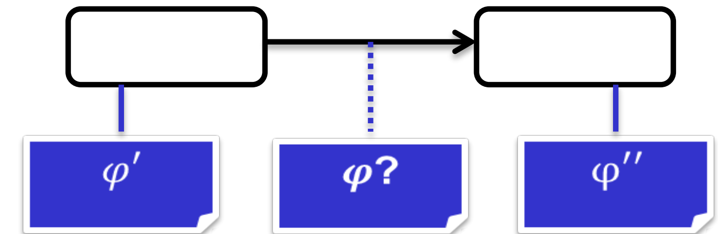


Inferenz fehlender Annotationen

Schritt 1: Ableiten fehlender Annotationen aus dem Kontext des Modellelements. Wir nehmen an, dass das Element *in so vielen Modellvarianten* wie möglich auftreten soll.

Beispiel: Eine nicht-annotierte Transition sollte in jeder Modellvariante vorhanden sein, in der Start- und Zielzustand vorhanden ist:

$$\vdash (\varphi' \land \varphi'') \rightarrow \varphi$$



Schritt 2: Prüfen auf Wohlgeformtheit (wie zuvor)

Beispiel: Eine nicht-annotierte Transition sollte nur in den Modellvarianen vorhanden sein, in denen der Start- und Zielzustand vorhanden ist.

$$\vdash \varphi \rightarrow (\varphi' \land \varphi'')$$

Komplettierung partiell annotierter Modelle (1/2)

Eingabe:

- Feature-Modell $\widetilde{FM} : (F \rightarrow \mathcal{B}) \rightarrow \mathcal{B}$
- Partiell annotiertes State Machine Modell

Algorithmus: Für jedes nicht-annotierte Element e füge eine Annotation φ ein, für die gilt:

1. Die Annotation ist valide bzgl. des Feature-Modells:

$$\widetilde{FM} \wedge \varphi \nvdash false$$

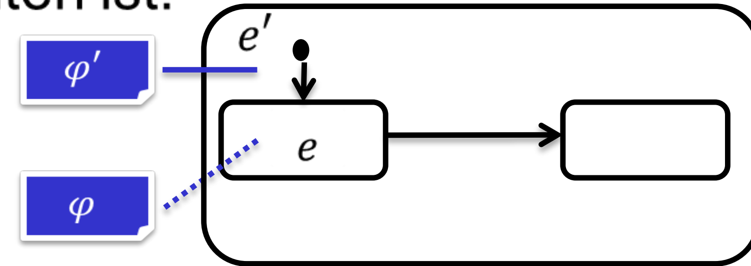
2. Falls $e \in Unterautomaten$ und e ist der äußerste Automat der Hierarchie:

$$\vdash \widetilde{FM} \rightarrow \varphi$$

Komplettierung partiell annotierter Modelle (2/2)

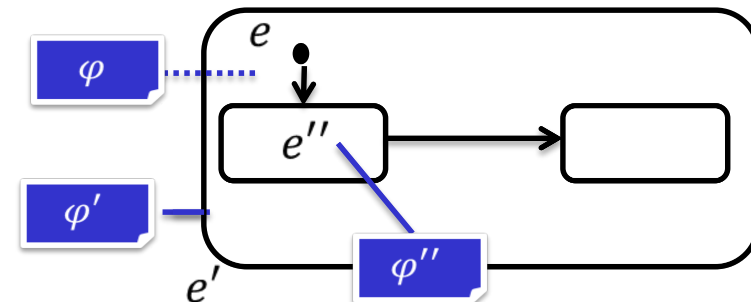
3. Falls $e \in \mathit{Zustände}$ und $e' \in \mathit{Unterautomaten}$ sei der (Unter-) Automat (annotiert mit φ'), in dem e direkt enthalten ist:

$$\vdash \varphi' \rightarrow \varphi$$



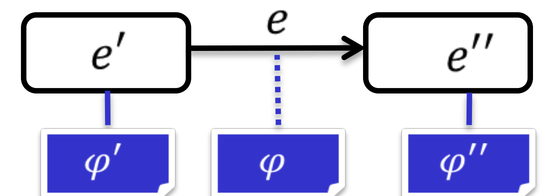
4. Falls $e \in \mathit{Unterautomaten}$, $e' \in \mathit{Zustände}$ sei der Zustand (annotiert mit φ'), in den e direkt eingeschachtelt ist und $e'' \in \mathit{Zustände}$ sei der Initialzustand von e (annotiert mit φ''):

$$\vdash (\varphi' \land \varphi'') \rightarrow \varphi$$



5. Falls $e \in \mathit{Transitionen}$, $e' \in \mathit{Zustände}$ sei der Ausgangszustand von e (annotiert mit φ') und $e'' \in \mathit{Zustände}$ sei der Zielzustand von e (annotiert mit φ''):

$$\vdash (\varphi' \land \varphi'') \rightarrow \varphi$$



Algorithmus zur Komplettierung partiell annotierter Modelle: Eigenschaften

Zwei (vollständig) annotierte State Machine Modelle sind *äquivalent annotiert*, wenn die Menge der daraus ableitbaren Modellvarianten gleich ist.

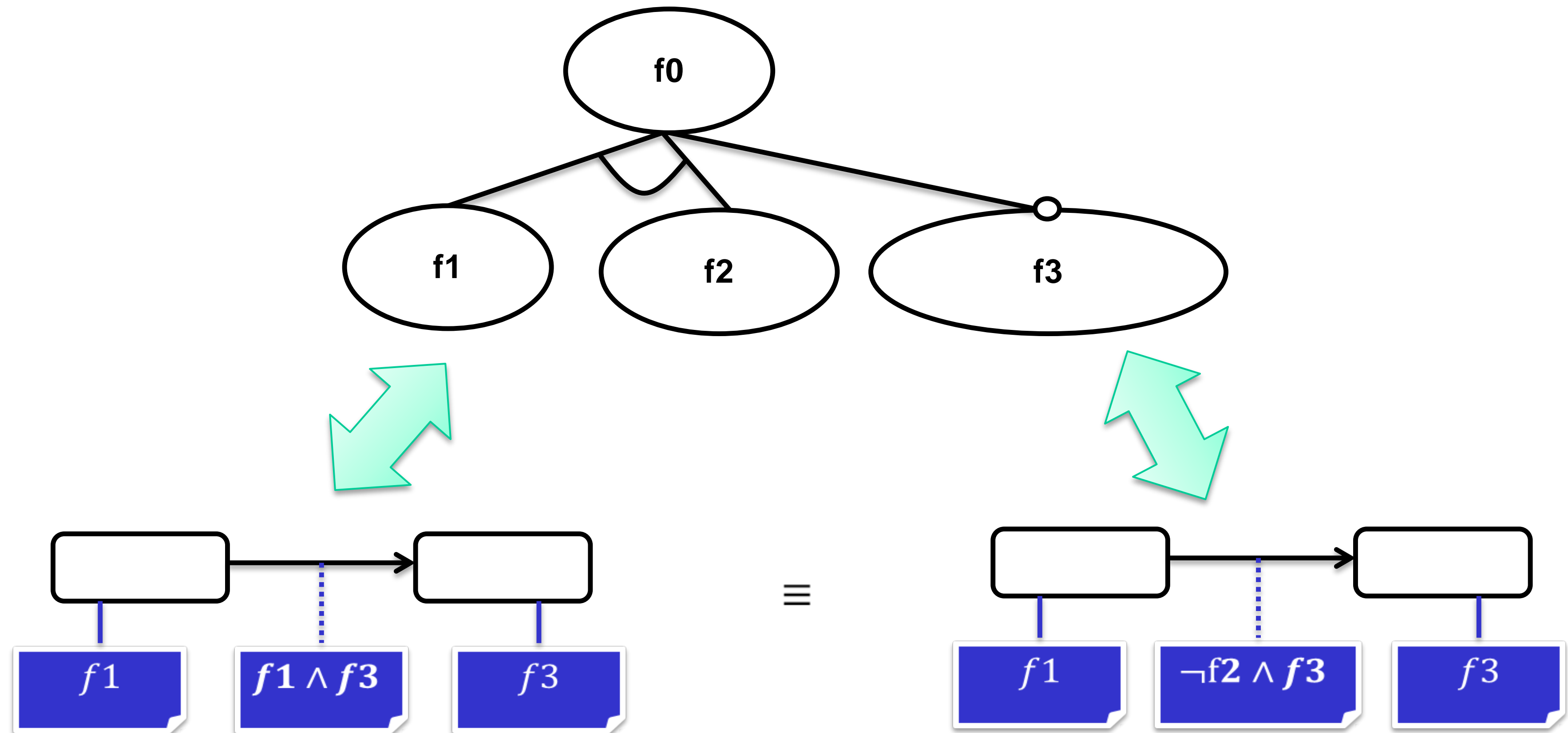
Ein partiell annotiertes State Machine Modell ist wohlgeformt, wenn es mindestens eine Komplettierung der Annotationen gibt, die wohlgeformt ist.

Satz1: Die wohlgeformte Komplettierung wohlgeformter, partiell annotierter State Machine Modelle ist im Allgemeinen nicht eindeutig.

Satz 2: Alle wohlgeformten Komplettierungen eines wohlgeformten, partiell annotierter State Machine Modells sind äquivalent annotiert.

- Häufig beschränken sich explizite State Machine Annotationen auf Transitionen
- Grundlage für *Variability Encoding*

Beispiel: Äquivalent annotierte State Machine Modelle

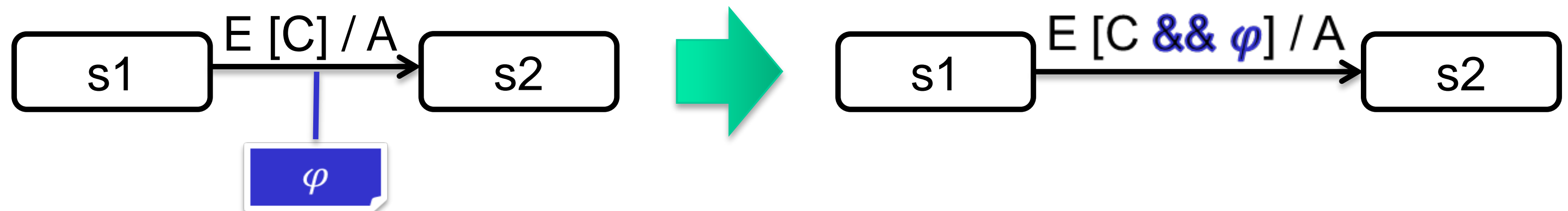


Variability Encoding (1/2)

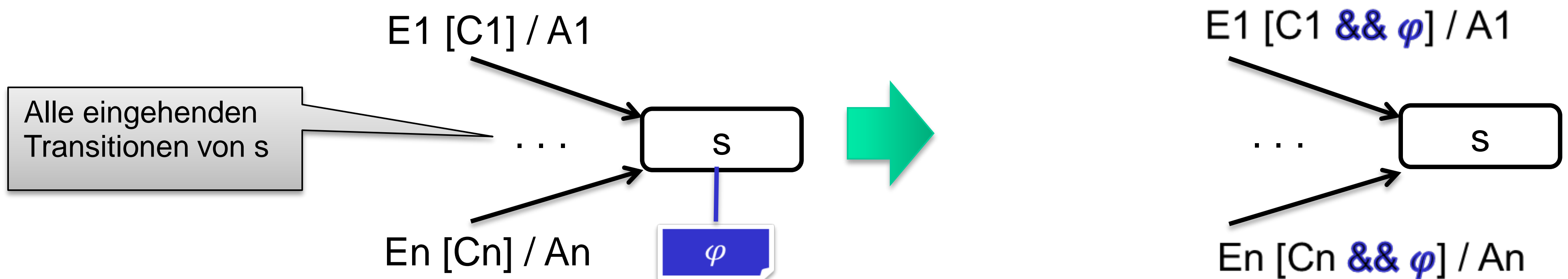
- **Problem:** Annotationen sind nicht Teil der “normalen” Syntax/Semantik der Modellierungssprache und werden
  - durch Modellierungs-Tools nicht untersützt und/oder
  - ignoriert
- **Lösung: Variability Encoding**
  - Simulation der Variabilität durch Kodierung von Annotationen über Standard-Konstrukte der Modellierungssprache
  - Nur möglich, wenn 150% Modell wohlgeformt ist
- **Beispiel:** Annotierte State Machine Modelle
  - Features als **Boole’sche Variablen** in Menge **V** hinzufügen
  - Annotationen als Teil der **Transitions-Guards** prüfen
  - Simulation von Modellvarianten durch Vorbelegung der Feature-Variablen

Variability Encoding (2/2)

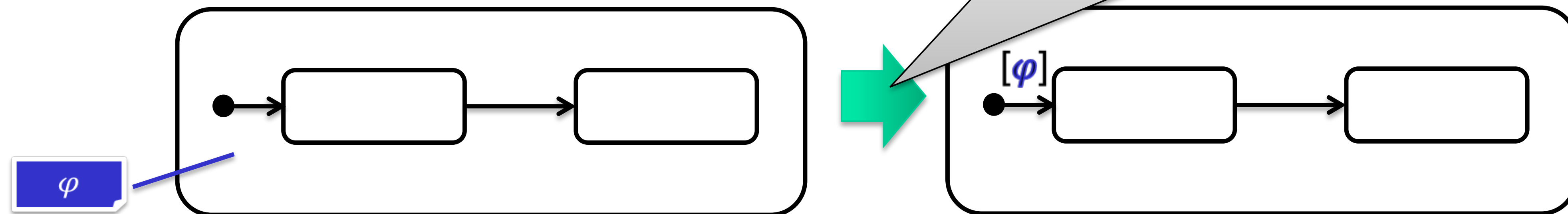
- Transitions-Annotationen



- Zustands-Annotationen

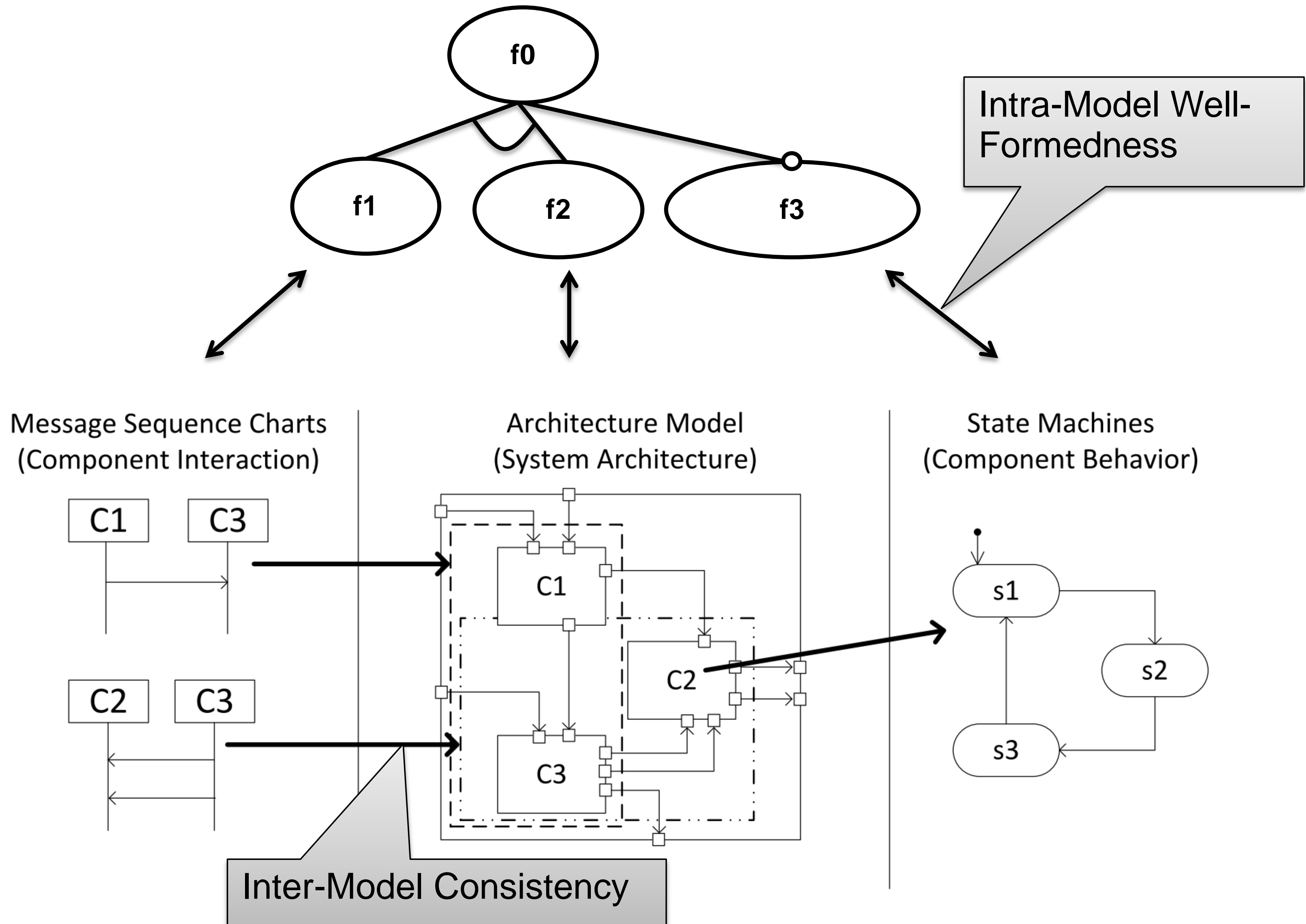


- Unterautomat-Annotationen



Annotationen im Metamodell

Tracing Features



Choice Calculus [Erwig and Walkingshaw, 2012]

- Sprachunabhängiger Kern-Kalkül zur Repräsentation von Variabilität innerhalb einer formalen Sprache
- Variabilität innerhalb eines Modells/ Programms wird durch Auflistung der zur Auswahl stehenden **Alternativen** dargestellt
- Varianten werden durch **Auswahl** von Alternativen selektiert
- Kalkül als Grundlage für Variabilitätsmodellierungstheorie
  - Korrektheitseigenschaften
  - Variantenerhaltende Transformationen
  - Normalformen

Variabilitätsdimensionen

Beispiel: Funktion zur Verdopplung einer Zahl

- Schnittstellen-Variabilität: Parameter-Name (**x** oder **y**)
- Implementierungs-Variabilität: Berechnung durch Summe (**plus**) oder Multiplikation (**times**)

	plus	times

x	`int twice(int x) {`
	`  return x+x;`
	`}`
		`int twice(int x) {`
		`  return 2*x;`
		`}`

y	`int twice(int y) {`
	`  return y+y;`
	`}`
		`int twice(int y) {`
		`  return 2*y;`
		`}`

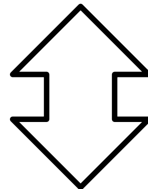
➢ Unabhängige **Variabilitätsdimensionen**
➢ Alternativen sind beliebig kombinierbar

Implementierung von Variabilität

	plus	times

x	`int twice(int x) {`
	`  return x+x;`
	`}`
		`int twice(int x) {`
		`  return 2*x;`
		`}`

y	`int twice(int y) {`
	`  return y+y;`
	`}`
		`int twice(int y) {`
		`  return 2*y;`
		`}`



```
int twice(int #ifdef X x #elif Y y #endif ) {
    return #ifdef PLUS    #ifdef X x+x;
                          #elif  Y y+y; #endif
           #elif TIMES    #ifdef X 2*x;
                          #elif  Y 2*y; #endif
           #endif
}
```

Baumrepräsentation von Modellen

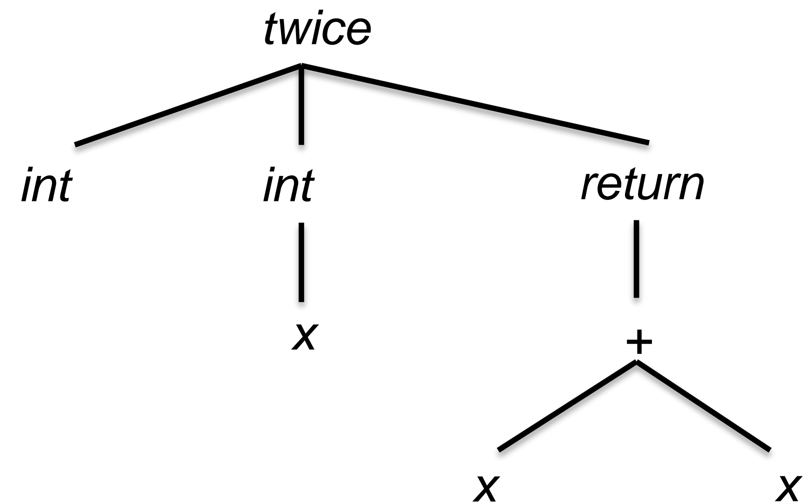
- Die Syntax einer Sprache definiert hierarchisch geschachtelte **Ausdrücke** der Form

$$\text{a}\,\langle e_1, \ldots, e_n \rangle$$

- Darstellung als Baum

$$twice\langle int, \langle int, x\rangle, \langle return\langle + \langle x, x\rangle\rangle\rangle\rangle$$

```
int twice(int x) {
  return x+x;
}
```



Variabilität als Choice

- Variabilität: Menge alternative Unterausdrücke eines Ausdrucks, zwischen denen bei der Ableitung einer Variante **ausgewählt** wird

$$\underbrace{a\,\langle \ldots, [l_1{:}e_1, l_2{:}e_2, \ldots, l_n{:}e_n], \ldots \rangle}_{\text{Choice}}$$

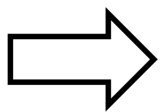
- $[\ldots]$ Dimension
- l_i Tag
- e_i Variante

- Dimension 1:

```
int twice(int [lx:x,ly:y]) {
  return 2*[lx:x,ly:y];
}
```

Dimension 2:

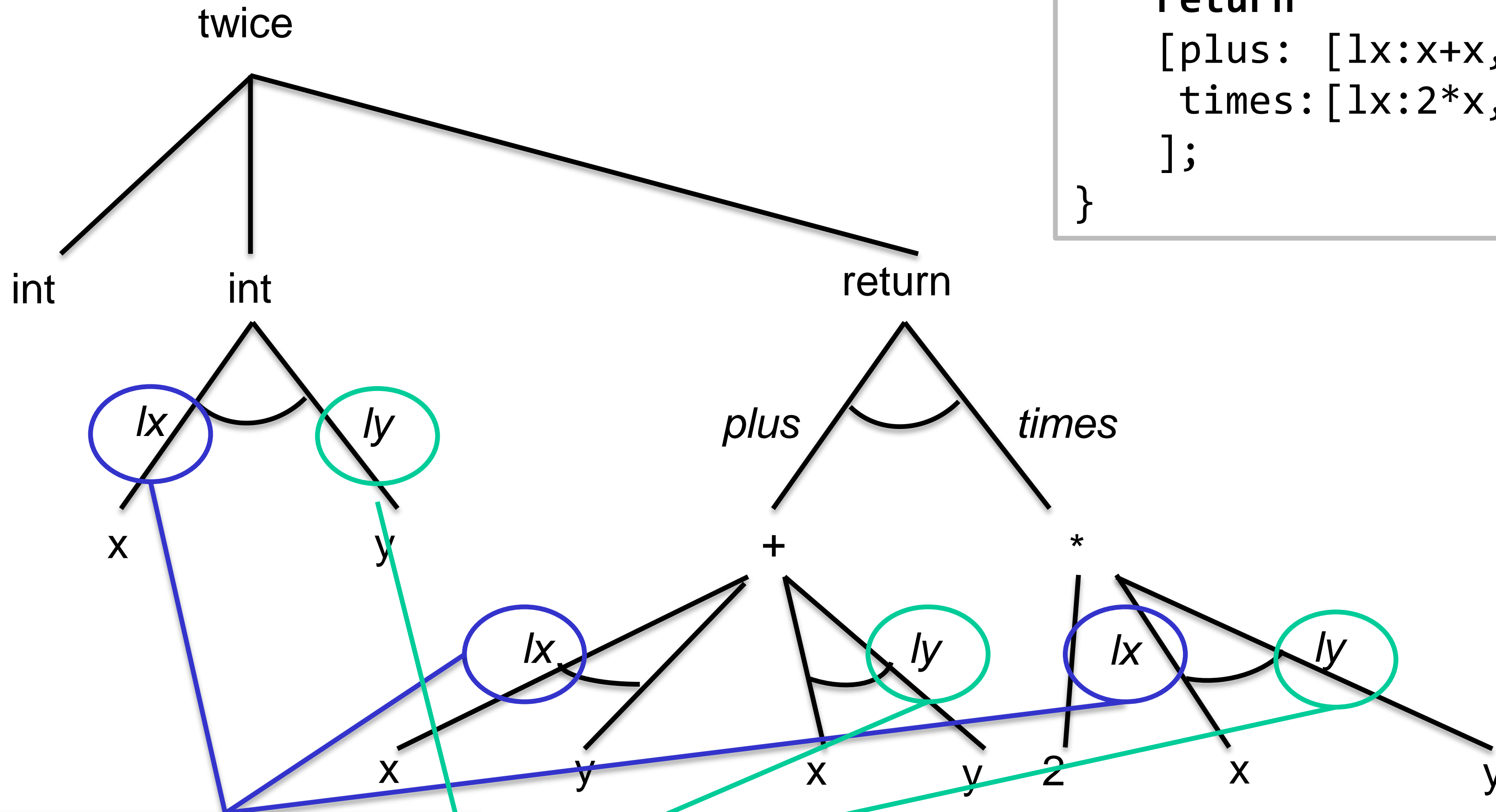
```
int twice(int x) {
  return [plus:x+x,times:2*x];
}
```



```
int twice(int [lx:x,ly:y]) {
  return [plus:[lx:x+x,ly:y+y],times:[lx:2*x,ly:2*y]];
}
```

Baumdarstellung mit Variabilität

```
int twice(int [lx:x,ly:y]) {
    return
    [plus: [lx:x+x,ly:y+y],
     times:[lx:2*x,ly:2*y]
    ];
}
```

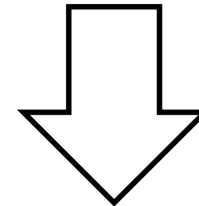


Mehrfaches Vorkommen gleich getagter Choices einer Dimension

Dimensionen und Scopes

- **Deklaration von Dimensionen,** deren Choices mehrfach vorkommen
- **Verwendung in Scoped Choices** zur Schachtelung von Dimensionen

```
int twice(int [lx:x,ly:y]) {
    return
    [plus: [lx:x+x,ly:y+y],
     times:[lx:2*x,ly:2*y]
    ];
}
```



```
dim Par[x,y] in
dim Impl[plus,times] in
int twice(int Par[x,y]) {
    return
    Impl[Par[x,y]+Par[x,y], 2*Par[x,y]];
}
```

Binden von Choices

- Beispiel: Hinzufügen einer weiteren Alternative

```
dim Par[x,y,z] in
dim Impl[plus,times] in
int twice(int Par[x,y,z]) {
    return
    Impl[Par[x,y,z]+Par[x,y,z], 2*Par[x,y,z]];
}
```

Anpassung an jeder Verwendungsstelle notwendig

- Lösung: Namensbindung für Dimensionen

```
dim Par[x,y,z] in
dim Impl[plus,times] in
let v=Par[x,y,z] in
int twice(int v) {
    return
    Impl[v+v, 2*v];
}
```

Lokale Dimensionen (1/2)

...

- Zusätzliche Funktionalität: Verdoppeln oder Verdreifachen einer Zahl (**twice** oder **thrice**)

```
dim Par[x,y,] in
dim Impl[plus,times] in
let v=Par[x,y] in
int twice(int v) {
    return Impl[v+v, 2*v];
}
int thrice(int v) {
    return Impl[v+v+v,3*v]
}
```

```
dim Impl[plus,times] in
let v=(dim Par[x,y] in Par[x,y]) in
int twice(int v) {
    return Impl[v+v, 2*v];
}
let v=(dim Par[x,y] in Par[x,y]) in
int thrice(int v) {
    return Impl[v+v+v,3*v]
}
```

- Auswahlentscheidung für den Parameternamen für beide Funktionen gleich

- Auswahlentscheidung für den Parameternamen unabhängig für beide Funktionen

Lokale Dimensionen (2/2)

...

- Zusätzliche Implementierungs-Variabilität

```
dim Impl[plus,times] in
let v=(dim Par[x,y] in Par[x,y]) in
int twice(int v) {
    return Impl[v+v, 2*v];
}
let v=(dim Par[x,y] in Par[x,y]) in
int thrice(int v) {
    return Impl[v+v+v,3*v]
}
```

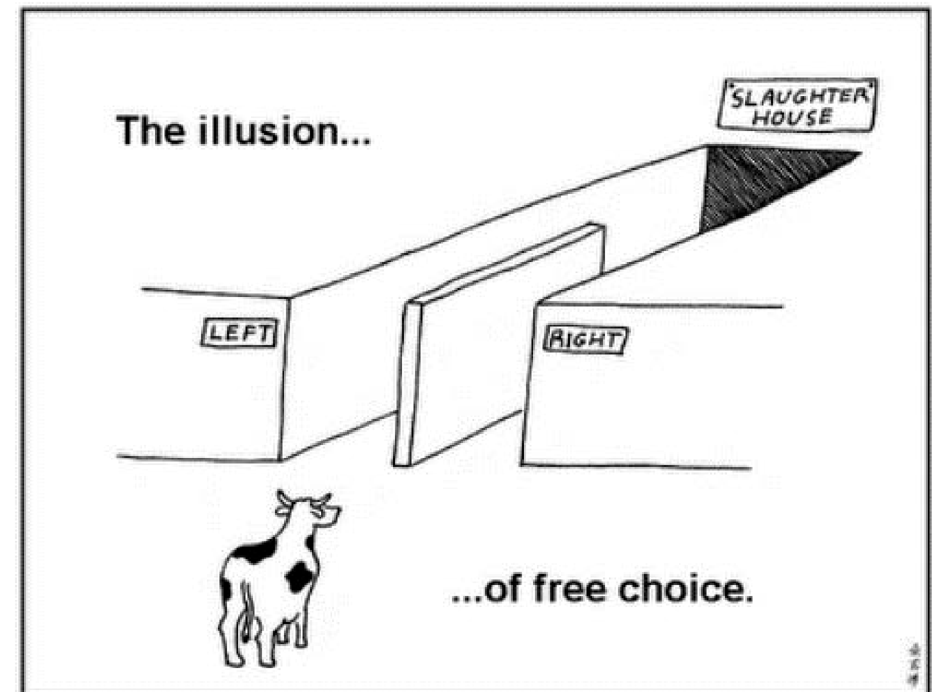
```
dim Impl[plus,times,shift] in
let v=(dim Par[x,y] in Par[x,y]) in
int twice(int v) {
    return Impl[v+v,2*v,v<<1];
}
dim Impl[plus,times,twice] in
let v=(dim Par[x,y] in Par[x,y]) in
int thrice(int v) {
    return Impl[v+v+v,3*v,v+twice(v)]
}
```

- Auswahlentscheidungen für die Implementierung für beide Funktionen gleich
- Auswahlentscheidungen für den Implementierung unabhängig für beide Funktionen

Abhängige Dimensionen

```
dim Style[pointFree,lambda] in
suc =
Style[(+1),
      dim Par[x,y] in
          Par[\x->x+1,\y->y+1]
    ]
```

- Schachtelung von Dimensionen in Choices
- Scope der Dimension beschränkt auf die umgebende Alternative



Choice Calculus – Formale Syntax

e	$::=$	$a\langle e, \ldots, e\rangle$	(Structure)
	$\mid$	$\textbf{let}\ v = e\ \textbf{in}\ e$	(Binding)
	$\mid$	v	(Reference)
	$\mid$	$\textbf{dim}\ d\,[\mathrm{t}, \ldots, \mathrm{t}]\ \textbf{in}\ \mathrm{e}$	(Dimension)
	$\mid$	$d[e, \ldots, e]$	(Choice)

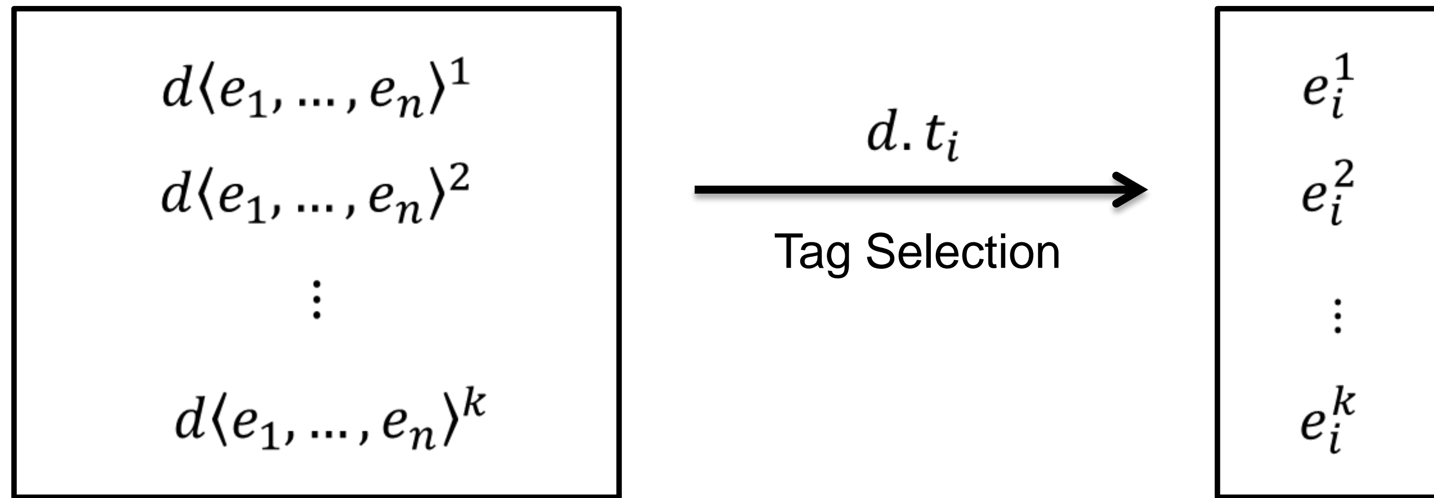
Choice Calculus – Semantics

- Was ist die exakte Bedeutung einer Spezifikation im Choice Calculus?
  1. Menge der möglichen Auswahl-Entscheidungen
  2. Menge der möglichen Varianten

- Die Semantik beschreibt Regeln für
  1. Eliminierung von Choices durch Auswahlentscheidungen
  2. Systematische Einschränkung und Wiederholung von Entscheidungen als Folge vorheriger Entscheidungen

Choice Elimination

$\mathbf{dim}\ d[t_1, \dots, t_n]\ \mathbf{in}$



Auswahl für Dimension d

- Entfernen von d innerhalb des Scopes von d
- Entfernen aller nicht gewählten Choices
- Tags können z.B. an Features gebunden sein

Repeated Tag Selection

```
dim Par[x,y] in
dim Impl[plus,times] in
int twice(int Par[x,y]) {
    return
    Impl[Par[x,y]+Par[x,y], 2*Par[x,y]];
}
```

Par.x

Par.y

```
dim Impl[plus,times] in
int twice(int x) {
    return
    Impl[x+x,2*x];
}
```

```
dim Impl[plus,times] in
int twice(int y) {
    return
    Impl[y+y,2*y];
}
```

Impl.plus

Impl.times

Impl.plus

Impl.times

```
int twice(int x) {
    return  x+x;
}
```

```
int twice(int x) {
    return  2*x;
}
```

```
int twice(int y) {
    return  y+y;
}
```

```
int twice(int y) {
    return  2*y;
}
```

Choice Calculus Semantik: Beispiel (1/2)

- Choice Elimination Semantics

$$[\![_]\!] : CC \rightarrow (Tag^* \rightarrow Expr)$$

- Beispiel:

```
e = dim Par[x,y] in
    dim Impl[plus,times] in
    int twice(int Par[x,y]) {
        return
        Impl[Par[x,y]+Par[x,y], 2*Par[x,y]];
    }
```

```
⟦e⟧ = { ([Par.x,Impl.plus],   int twice(int x) {return x+x}),
        ([Par.x,Impl.times],  int twice(int x) {return 2*x}),
        ([Par.y,Impl.plus],   int twice(int y) {return y+y}),
        ([Par.y,Impl.times],  int twice(int y) {return 2*y})   }
```

Choice Calculus Semantik: Beispiel (2/2)

- Auflösen abhängiger Dimensionen:

```
e = dim Style[pointFree,lambda] in
    suc = Style[(+1),
       dim Par[x,y] in
           Par[\x->x+1,\y->y+1]
     ]
```

```
⟦e⟧ = {    ([Style.pointFree],         suc = (+1)     ),
           ([Style.lambda,Par.x],      suc = \x->x+1  ),
           ([Style.lambda,Par.y],      suc = \y->y+1  )   }
```

Dimension im Scope von Choice Style.lambda

Transformationen im Choice Calculus

- Semantik-erhaltende Term-Umformungen:

$$e \equiv e' \quad :\Leftrightarrow$$

$$[\![e]\!] = [\![e']\!]$$

Choice Factorization/Distribution

distribute

$$a\langle e_1, \ldots, d[e'_1, \ldots, e'_k], \ldots, e_n\rangle \quad \equiv \quad \begin{matrix} d[a\langle e_1, \ldots, e'_1, \ldots, e_n\rangle, \\ \vdots \\ a\langle e_1, \ldots, e'_k, \ldots, e_n\rangle] \end{matrix}$$

factor

- Distribution: Choices “nach außen ziehen”
- Factorization: Choices “nach innen schieben”

Beispiel: Factorization/Distribution

```
int twice(int Par[x,y]) {
    return 2*Par[x,y];
]
```

≡

```
int twice(Par[int x, int y]) {
    return 2*Par[x,y];
]
```

≡

≡

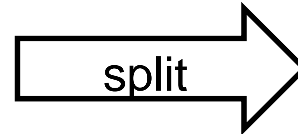
```
int twice(int Par[x,y]) {
    return Par[2*x,2*y];
]
```

≡

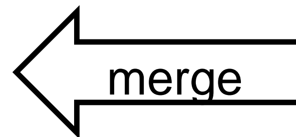
```
int twice(Par[int x, int y]) {
    return Par[2*x,2*y];
]
```

Beobachtung: Die Anwendung der Distributionsregel entspricht der Verringerung der Granularität und erhöht somit Repetition von Ausdrücken, die von verschiedenen Varianten geteilt werden

Choice Splitting/Merging



$$d[e_1, \dots, e_i', \dots, e_n] \equiv d[e_1, \dots, d[e_1', \dots, e_n'], \dots, e_n]$$



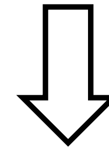
- Split: Choice von außen nach innen duplizieren
- Merge: Choices von innen nach außen zusammenfassen

Beispiel

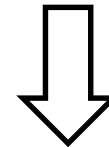
```
int twice(Par[int x,int y]) {
    return Par[2*x,2*y];
}
```

≡

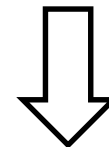
```
int twice(Par[int x,int y]) {
    Par[return 2*x, return 2*y];
}
```




```
Par[int twice(int x) {
    Par[return 2*x, return 2*y];
},  int twice(int y) {
    Par[return 2*x, return 2*y];
}]
```




```
Par[int twice(int x) {
    return 2*x;
},  int twice(int y) {
    return 2*y;
}]
```



Normalformen

1. Choice Normal Form (CNF)

- CC Term enthält ausschließlich maximal faktorisierte Choices
- Representation mit minimaler Redundanz

2. Dimension Normal Form (DNF)

- Alle nicht-abhängigen Dimensionen befinden sind ganz außen
- Macht Varianten explizit sichtbar und ermöglicht Merging

3. Dimension Choice Normal Form (DNF & CNF)

- Zu jedem CC Term existiert ein äquivalenter CC Term in CNF
- Für DNF gilt das im Allgemeinen nicht

Variation Design Theory

Choice Simplification

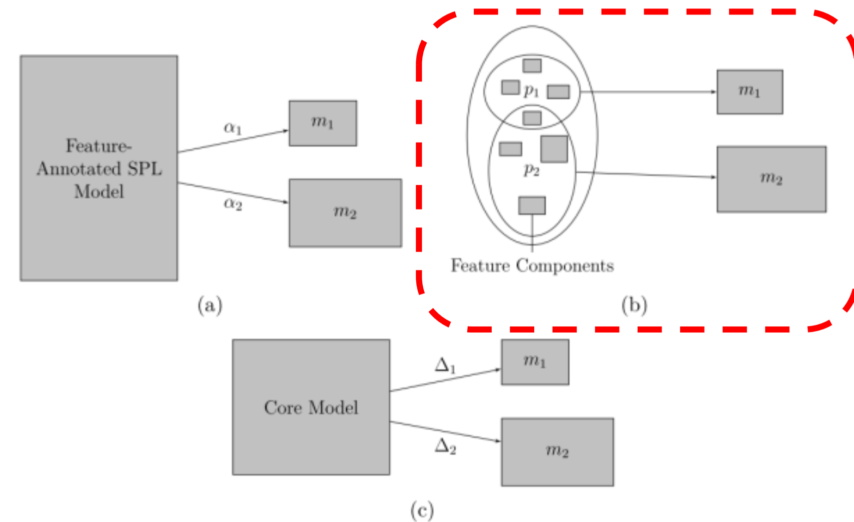
- Redundante Tags: Choice ohne diese Alternative ist äquivalent zum Choice mit dieser Alternative
  => Tag und zugehörige Alternativen entfernen
- Pseudo Choices: alle Alternativen eines Choices sind äquivalent
  => Choice durch konstante Alternative ersetzen

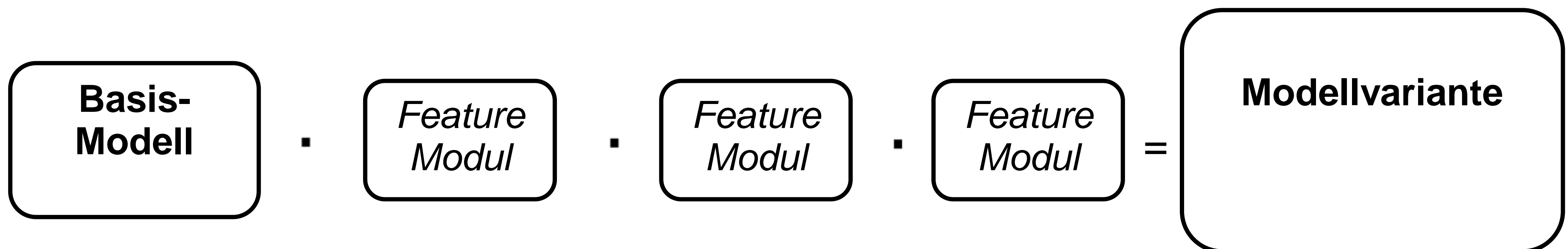
Choice Erasure

- Pseudo Dimension: Alle Tags sind paarweise redundant
  => Dimension entfernen und jeden zugehörigen Choice im Scope durch eine der Alternativen ersetzen
- …

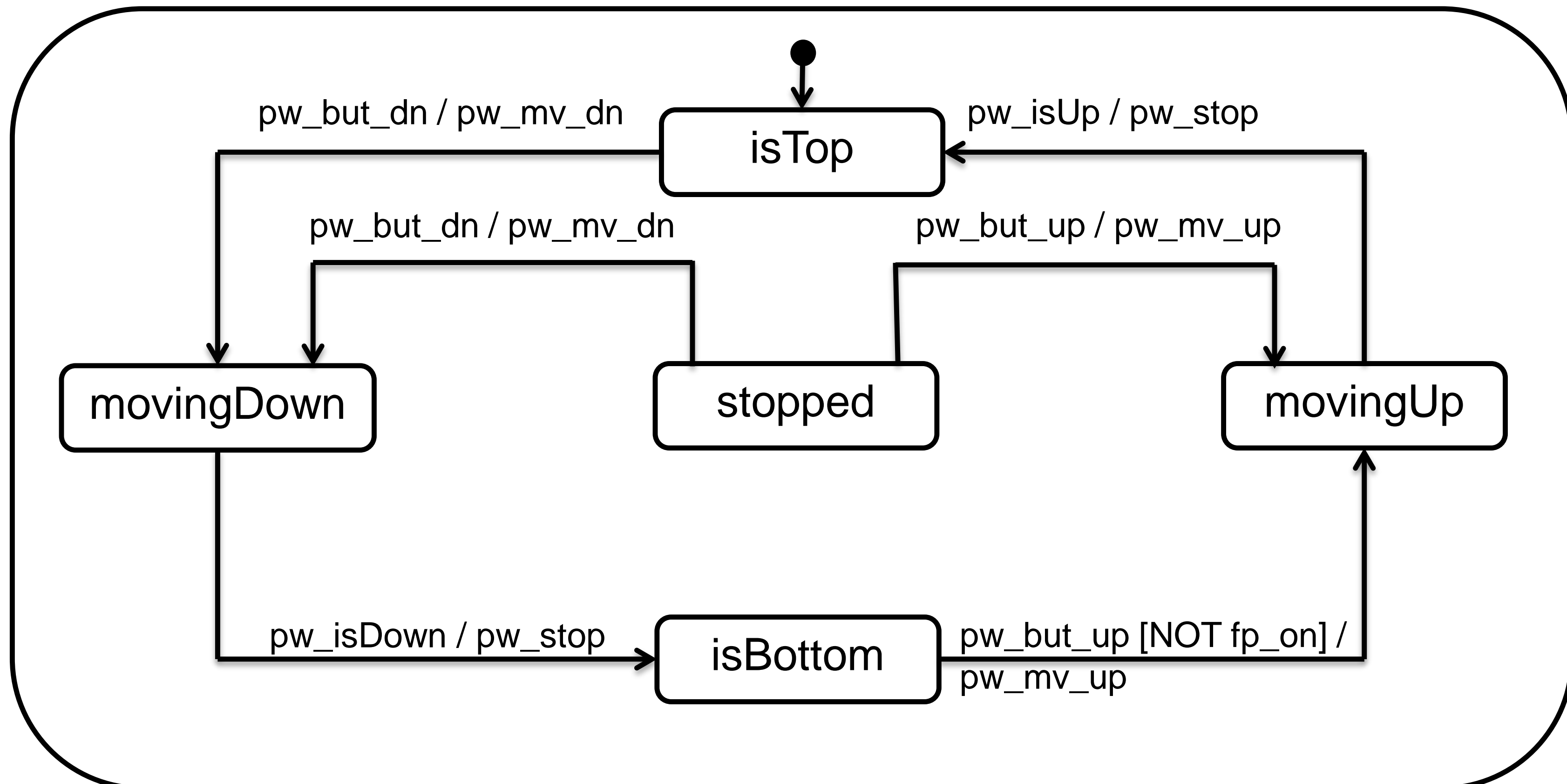
Komposition von Modellvarianten

- Zerlegen von Design-Modellen in **Feature-Module**
- Modellvariante ergibt sich durch **Komposition** der Feature-Module, deren Features in der Variante enthalten sind
- **Positive Variabilität:** Komposition fügt Elemente hinzu
- Ausgangspunkt ist häufig ein **Basis-Modell**





Basis-Modell

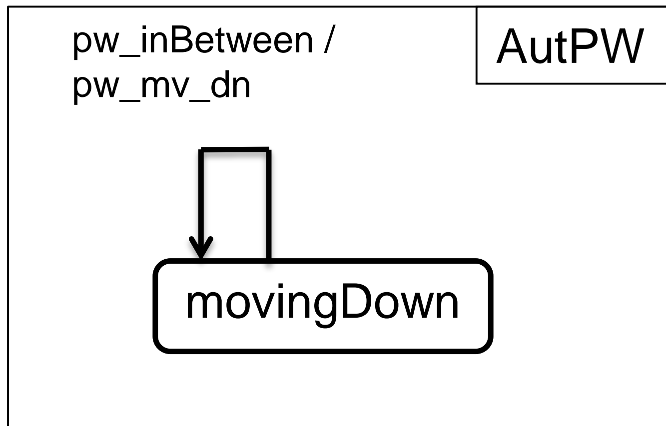


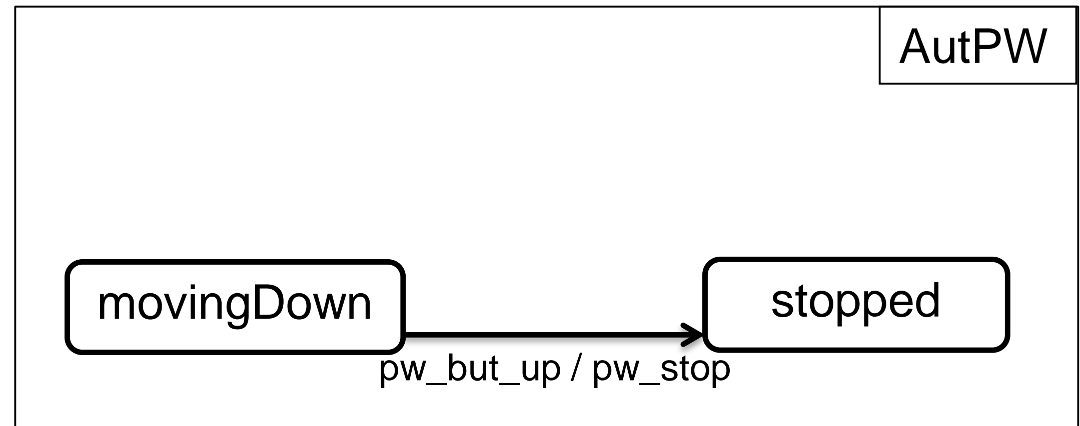
Wahl des Basis-Modells:

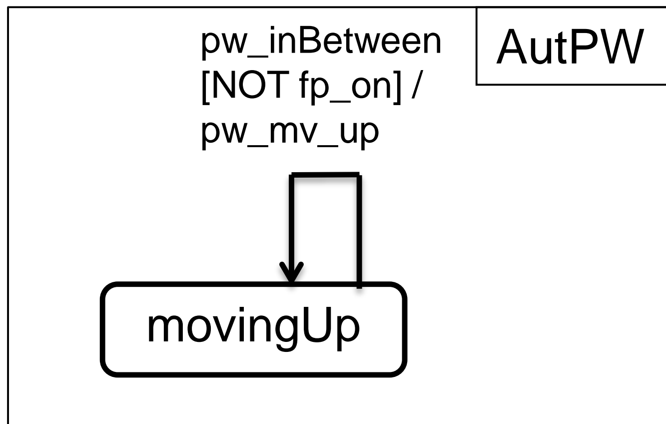
- (Maximales) Modell, das Gemeinsamkeiten aller Modellvarianten enthält
- Alternativ: leeres Modell als Basis-Modell

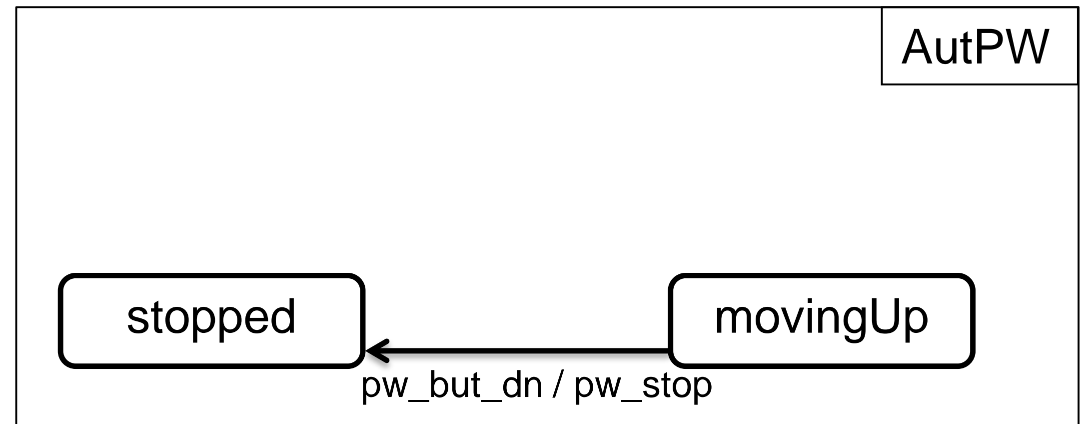
Definition von Feature-Modulen

Beispiel: Module für Feature *AutPW*

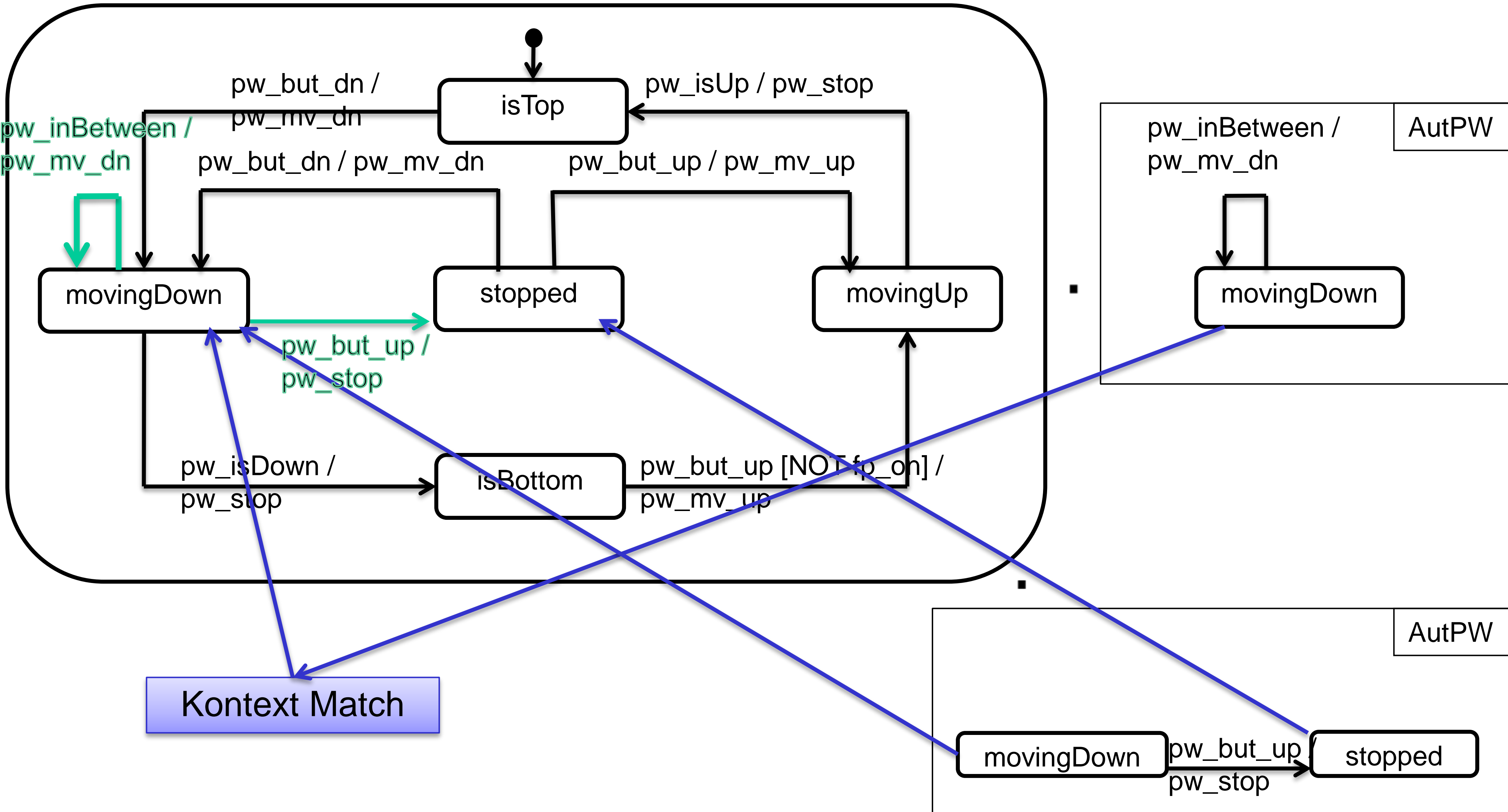






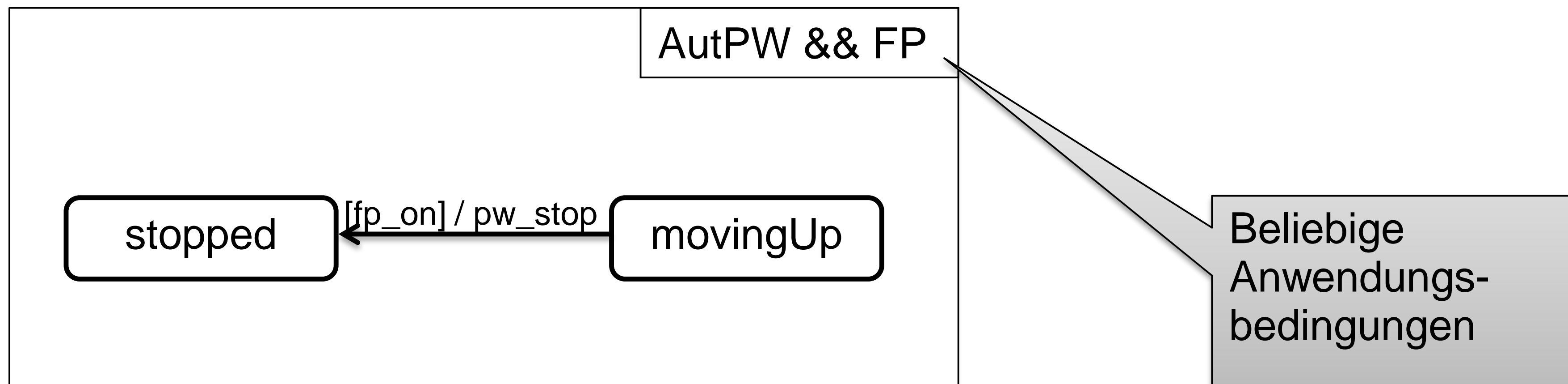


Komposition von Feature-Modulen



Derivative-Module

Beispiel: Modul zur Koordination der Features *AutPW* und FP



- Im worst case benötigt man für jede n-stellige Feature-Kombination Derivative-Module
- Siehe Optional-Feature-Problem

Diskussion

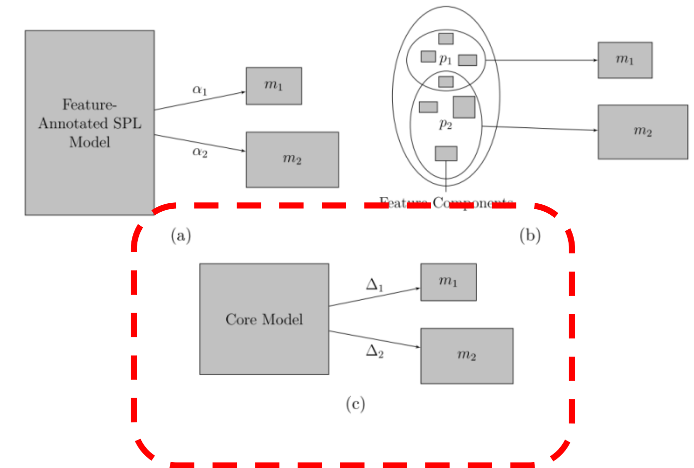
- Tool-Support: AHEAD **[Batory et al. 2003]**
- Kompositionsoperator basiert auf *Superimposition*
- Kalkül für typsichere Modulkomposition: gDeep **[Apel and Hutchins 2010]**
- Grundlage für Feature-Oriented Programming (FOP)

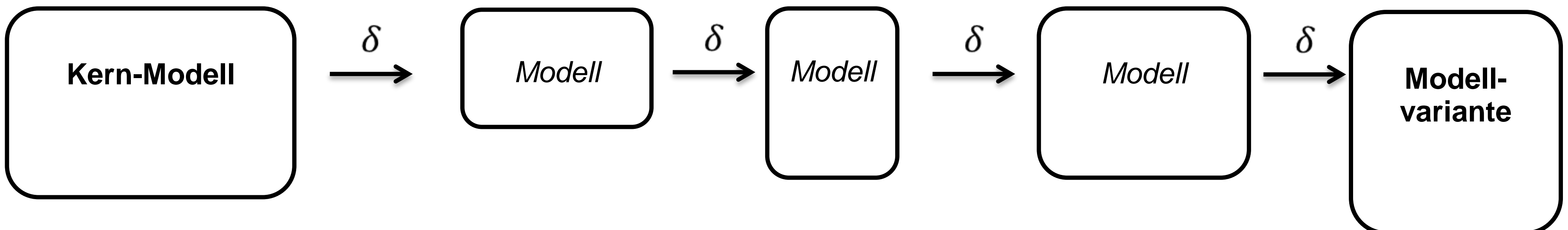
⇒ **Mehr dazu in Kapitel II.3**

- **Einschränkung des kompositionsbasierten Ansatzes:** Basis-Modell nicht frei wählbar
  - Basis-Modell wohlgeformt?
  - Entspricht das Basis-Modell einer gültigen Modellvariante?

Transformative Variabilitätsmodellierung

- Auswahl einer beliebigen Variante als **Kern-Modell**
- Modellvarianten ergeben sich durch **Transformation** des Kern-Modells
- Die anzuwendenden Transformationsregeln ergeben sich aus der Feature-Auswahl der Variante
- Transformationsregeln können Modellelemente **hinzufügen, löschen und ändern**





Delta-Modellierung

Ein **Delta Modell** besteht aus:

- Feature-Modell $\widetilde{FM} : (F \to \mathcal{B}) \to \mathcal{B}$
- Kern-Modell $c \in \mathcal{L}$ (core) in einer Modellierungssprache $\mathcal{L}$
- Delta Menge $\Delta \subseteq \left(Op \times \mathcal{E} \times \left((F \to \mathcal{B}) \to \mathcal{B}\right)\right)$ mit
  - $Op = \{\,\mathbf{add}, \mathbf{delete}, \mathbf{modify}\,\}$ (Delta Operation)
  - Universum $\mathcal{E}$ (Menge aller Modellelemente der Sprache $\mathcal{L}$)
  - Anwendungsbedingungen über Features (Application Condition)

Anmerkungen:

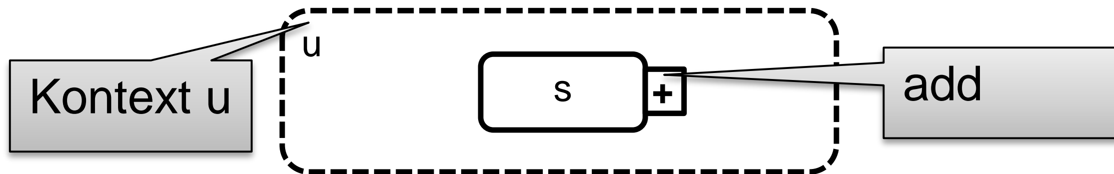
- Häufig enthält ein Delta zusätzliche Kontext-Informationen
- **modify** lässt sich durch Kombination von **delete** und **add** simulieren und wird deshalb nachfolgend weggelassen
- Notation häufig: $\delta = (\langle op\ e \rangle, \varphi)$ bzw. $\delta = \langle op\ e \rangle$

Beispiel: Delta State Machines

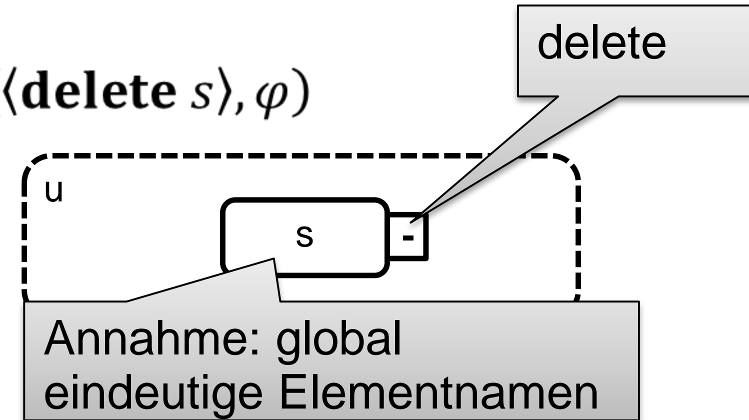
Modellelemente $\mathcal{E} = (Zustände \cup Transitionen \cup Unterautomaten)$

- Zustände

$\delta = (\langle \textbf{add}\ s\ \textbf{to}\ u\rangle, \varphi)$

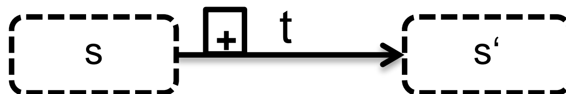


$\delta = (\langle \textbf{delete}\ s\rangle, \varphi)$

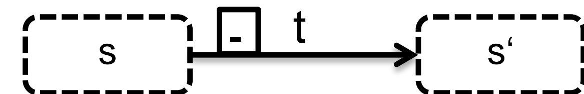


- Transitionen

$\delta = (\langle \textbf{add}\ t\ \textbf{from}\ s\ \textbf{to}\ s'\rangle, \varphi)$



$\delta = (\langle \textbf{delete}\ t\rangle, \varphi)$



- Unterautomaten

$\delta = (\langle \textbf{add}\ u\ \textbf{into}\ s\rangle, \varphi)$

$\delta = (\langle \textbf{delete}\ u\rangle, \varphi)$

Einfache Delta Anwendung

- Anwendung eines Deltas $\delta \in \Delta$ auf ein Modell $m \in \mathcal{L}$ ist definiert durch eine Funktion

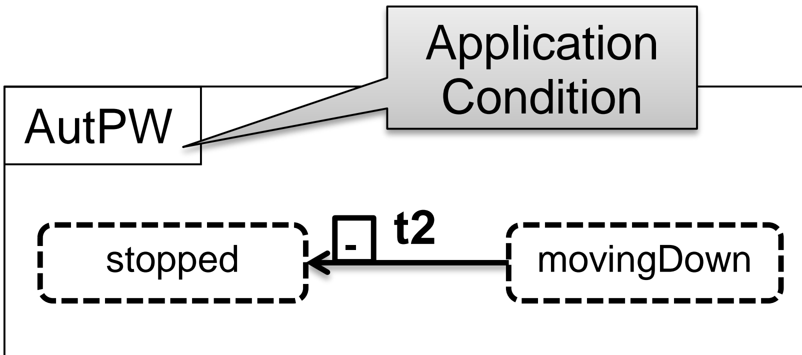
$$apply : \Delta \times \mathcal{L} \rightarrow \mathcal{L}$$

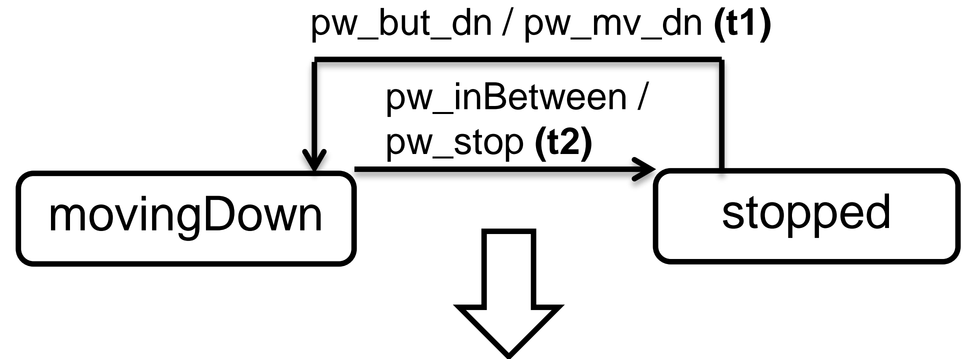
- Modell $m' = apply(\delta, m)$ entsteht durch Anwendung der Delta-Operation von δ auf m

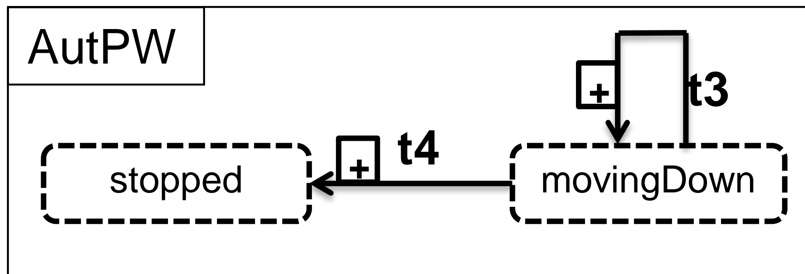
Anmerkungen:

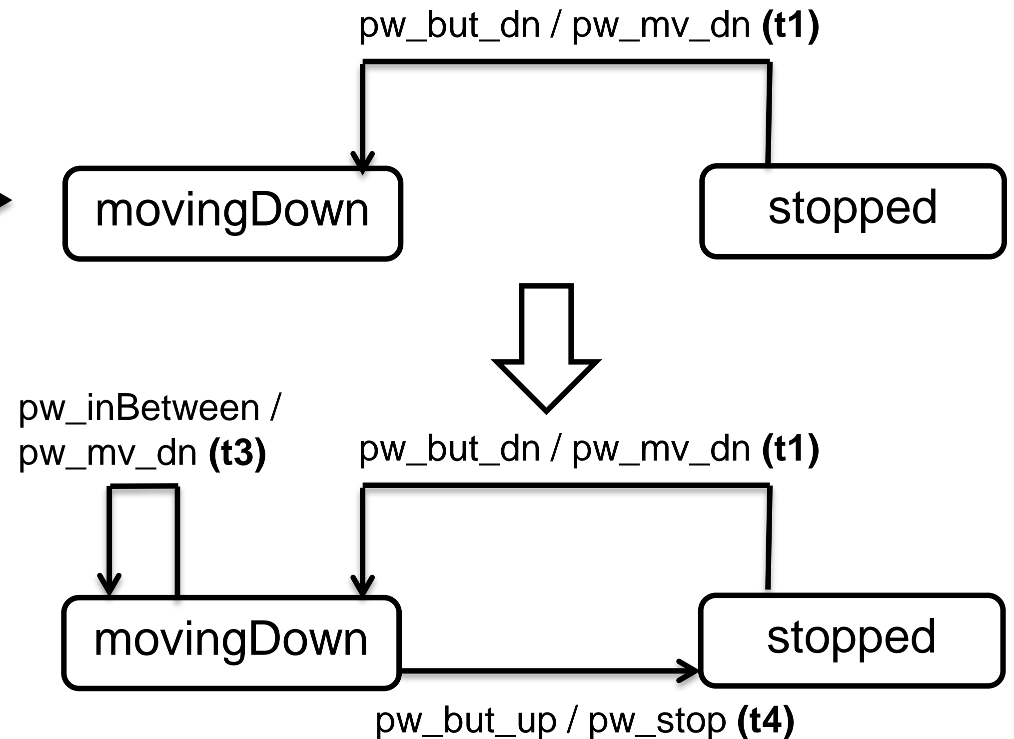
- Eine einfache Delta-Anwendung wird häufig notiert als $\delta(m) = m'$
- Die apply-Funktion wird total definiert, indem $\delta(m) = m$ bei nicht Anwendbarkeit (fehlender Kontext) von δ auf Modell m angenommen wird.

Beispiel: Einfache Anwendung von State Machine Deltas









Variantenbildung durch Delta Anwendung

- Die Ableitung einer Modellvariante für eine Produktkonfiguration p erfolgt durch Anwendung der Delta Menge auf das Kern-Modell c

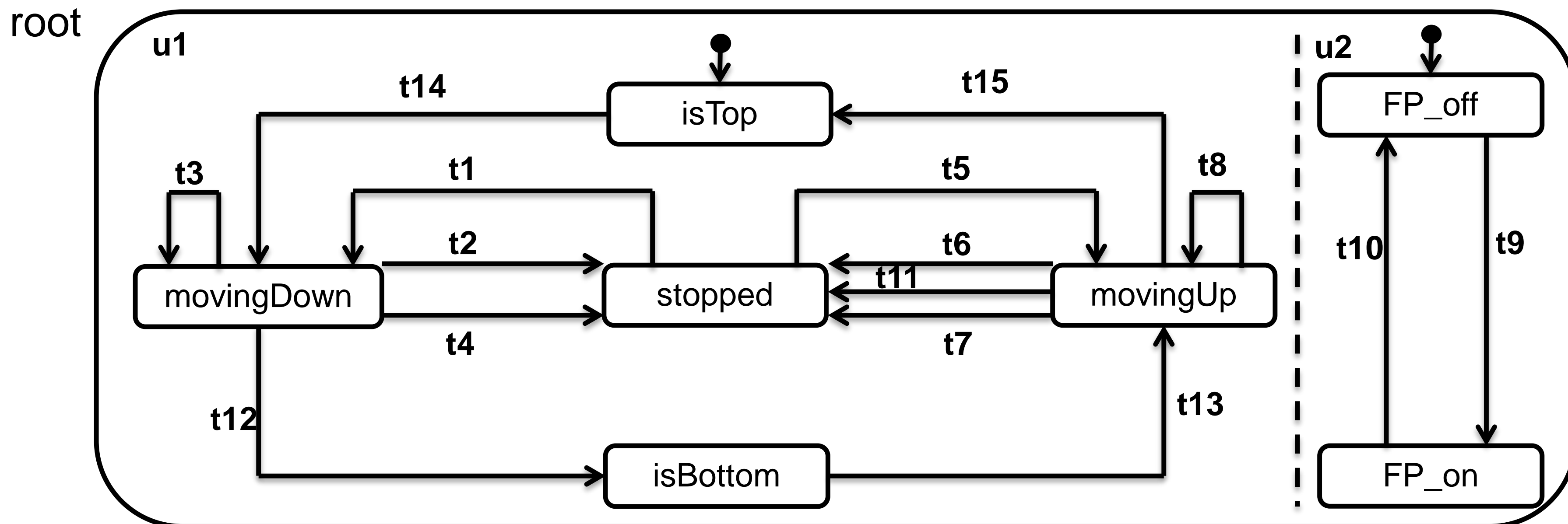
$$\Delta_p = \{ \delta = (\langle op\ e \rangle, \varphi) \in \Delta \mid p \vdash \varphi \}$$

- Die Anwendung einer Delta Menge Δ_p auf ein Modell m ist definiert durch die Funktion

$$apply : 2^{\Delta} \times \mathcal{L} \rightarrow \mathcal{L}$$

mit $apply(\Delta', m) = \begin{cases} apply(\Delta'', \delta(m)) & \text{falls } \Delta'' = \Delta' \setminus \{\delta\} \\ m & \text{falls } \Delta' = \emptyset \end{cases}$

Beispiel: Variantenbildung aus Delta State Machines



ManPw

$\Delta^{+}_{ManPw} = \{$
add *t2*
add *t6* }

!ManPw

$\Delta^{-}_{ManPw} = \{$
delete *t2*
delete *t6*
}

AutPw

$\Delta^{+}_{AutPw} = \{$
add *t3*,
add *t4*,
add *t7*,
add *t8* }

!AutPw

$\Delta^{-}_{AutPw} = \{$
delete *t3*,
delete *t4*,
delete *t7*,
delete *t8* }

FP

$\Delta^{+}_{FP} = \{$
add *u2*,
add *FP_off*,
add *FP_on*,
add *t9*,
add *t10* }

!FP

$\Delta^{-}_{FP} = \{$
delete *u2*,
delete *FP_off*,
delete *FP_on*,
delete *t9*,
delete *t10* }

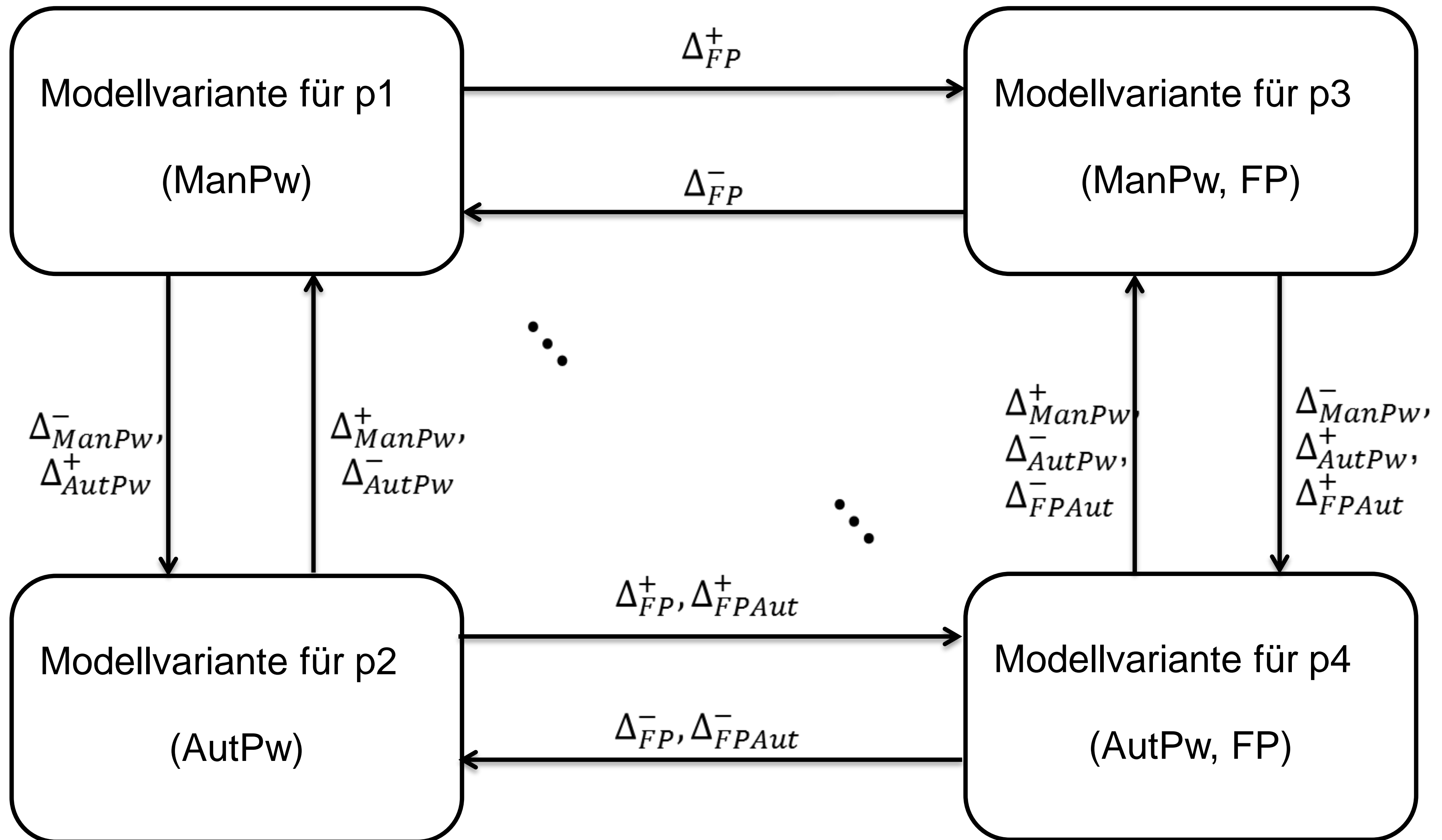
FP && AutPW

$\Delta^{+}_{FPAut} = \{$
add *t11* }

!(FP && AutPW)

$\Delta^{-}_{FPAut} = \{$
delete *t11* }

Beispiel: Variantenbildung aus Delta State Machines



Delta Konflikte

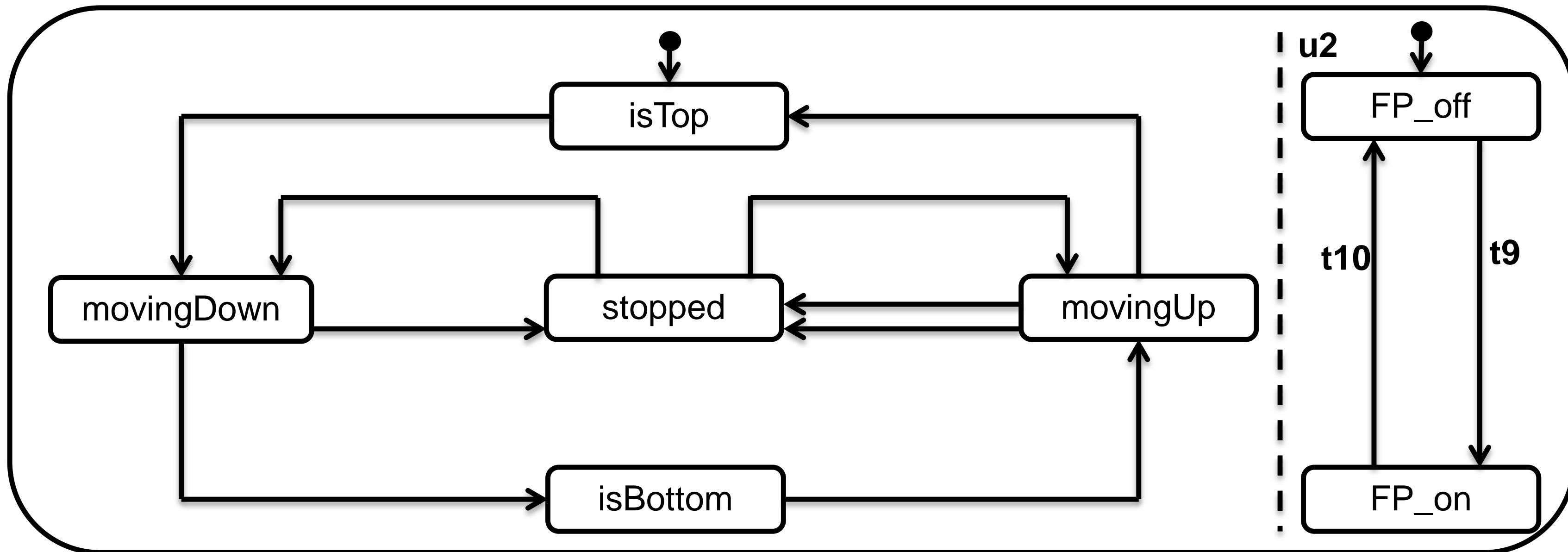
- Das Ergebnis der Anwendung einer Delta Menge kann sich je nach Anwendungsreihenfolge unterscheiden

- Zwei Deltas δ_1 und δ_2 stehen in **Konflikt**, wenn das Ergebnis ihrer gemeinsamen Anwendung auf ein Modell m je nach Reihenfolge unterschiedlich ist:

$$\delta_1\ (\delta_2\ (m)) \neq \delta_2\ (\delta_1\ (m))$$

- Das passiert häufig, wenn Modellelemente, die von Deltas transformiert werden, Teil des Kontextes des anderen Deltas sind

Beispiel: Konflikte zwischen State Machine Deltas



1. Ableitung: $\textbf{add}\ u2,\quad \textbf{add}\ FP_on,\quad \textbf{add}\ FP_off,\quad \textbf{add}\ t9,\quad \textbf{add}\ t10$

2. Ableitung: $\textbf{add}\ FP_on,\quad \textbf{add}\ u2,\quad \textbf{add}\ FP_off,\quad \textbf{add}\ t9,\quad \textbf{add}\ t10$

- Modellelemente können erst eingefügt werden, nachdem der umgebene Unterautomat eingefügt wurde (umgekehrt beim Löschen…).
- Transitionen können erst eingefügt werden, nachdem der Start- und Zielzustand eingefügt wurde (umgekehrt beim Löschen…).

Delta Ordnung

- Lösung von Konflikten durch Erweiterung eines Delta Modells um eine **partielle Ordnung**

$$\prec \subseteq \Delta \times \Delta$$

- Wenn zwei Deltas δ_1 und δ_2 in Konflikt stehen, dann gilt entweder $\delta_1 \prec \delta_2$ oder $\delta_2 \prec \delta_1$

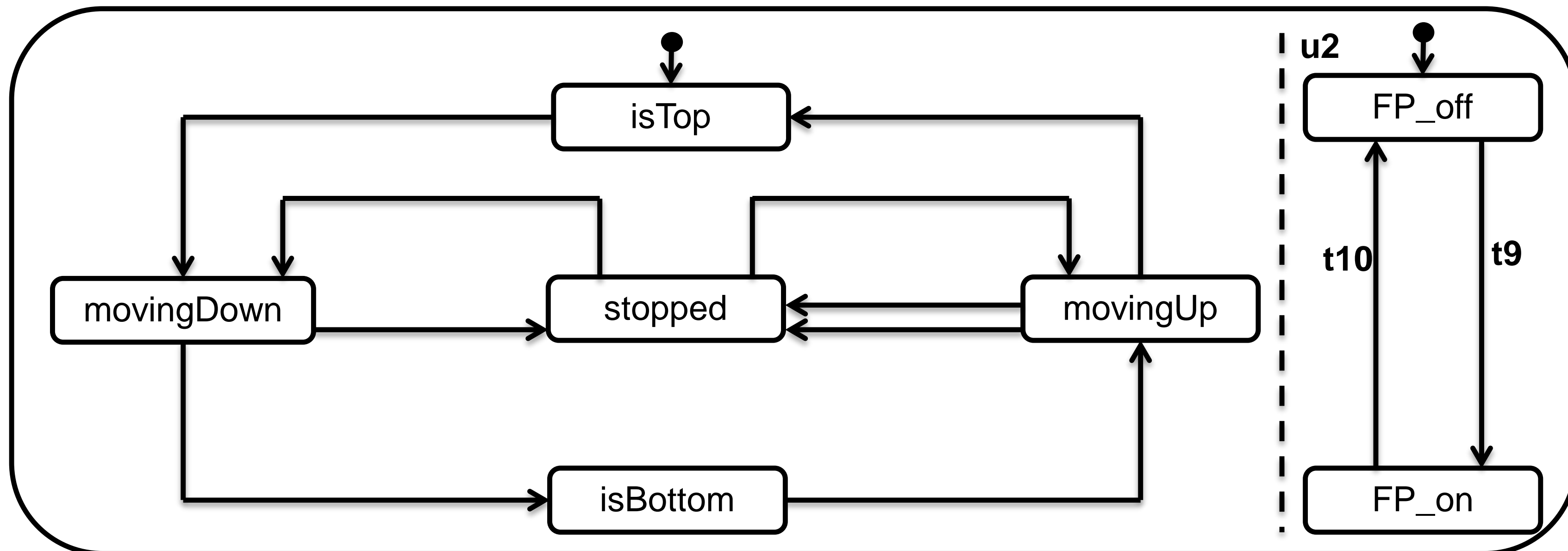
- Anpassung der Funktion zur Anwendung einer Delta Menge Δ_p auf ein Modell m

$$apply : 2^{\Delta} \times \mathcal{L} \rightarrow \mathcal{L}$$

$$\text{mit } apply(\Delta', m) = \begin{cases} apply(\Delta'', \delta(m)) & \text{falls } \Delta'' = \Delta' \setminus \{\delta\} \wedge \neg\exists \delta' \in \Delta'' : \delta' \prec \delta \\ m & \text{falls } \Delta' = \emptyset \end{cases}$$

Reihenfolge der Delta-Anwendung gemäß der Delta Ordnung

Beispiel: Ordnung auf State Machine Deltas



- Unterautomat zuerst: $\textbf{add}\; u2 \prec \textbf{add}\; FP_on$, $\textbf{add}\; u2 \prec \textbf{add}\; FP_off$, $\textbf{add}\; u2 \prec \textbf{add}\; t9$, $\textbf{add}\; u2 \prec \textbf{add}\; t10$
- Start- und Zielzustand vor der Transition: $\textbf{add}\; FP_on \prec \textbf{add}\; t9$, $\textbf{add}\; FP_off \prec \textbf{add}\; t9$, $\textbf{add}\; FP_on \prec \textbf{add}\; t10$, $\textbf{add}\; FP_off \prec \textbf{add}\; t10$

Beobachtung: Delta Ordnung aus der Modellstruktur ableitbar

Abstract Delta Modeling [Clarke et al., 2012]

- Beobachtung: **Konfliktbegriff** ist wesentlich für die Korrektheit von Delta-Modellen
- Algebraische Charakterisierung von Deltas und Konflikten in Delta-Modellen unabhängig von der konkreten Modellierungssprache

- Produktlinie: Globale Menge von Deltas zur Ableitung von Modellvarianten aus einem Kernmodell
- Delta-Modell: partiell geordnete Teilmenge von Deltas zur Ableitung einer Modellvariante
- Delta-Anwendung: Anwendung von Deltas auf ein Kernmodell in einer Reihenfolge entsprechend der partiellen Ordnung
- Widerspruchsfreies Delta-Modell: jede mögliche Delta-Anwendung ergibt eine eindeutige Modellvariante

Abstract Delta Modeling: Notationen

- Kernmodell*:* c
- Deltas*:* x, y, z *bzw.* $x_1, x_2, x_3, \ldots$
- Modellvarianten: $p, p', p'', \ldots$
- Delta-Anwendung: $p' = x(p)$
- Delta-Komposition: $x_n(x_{n-1}(\cdots(x_1\,(\,c\,))\cdots)) = (x_n \cdot x_{n-1} \cdot \cdots \cdot x_1)(c)$

Pure Delta Modeling

- Leere Modellvariante: $\mathbf{0}$
- Kern-Delta: $x_c(\mathbf{0}) = c$

Satz: $(x_n \cdots x_1) = (x_n \cdots x_1 x_c)(\mathbf{0})$

➢ O.B.d.A. gehen wir im Folgenden von Pure Delta Modellen aus.

Deltoid

Definition: Ein Deltoid ist ein *Monoid* $(\mathcal{D}, \cdot, \epsilon)$ bestehend aus

- Delta-Menge $\mathcal{D}$
- Delta-Komposition: $\cdot\, \mathcal{D} \times \mathcal{D} \rightarrow \mathcal{D}$
- Neutralem Delta-Element: $\epsilon \in \mathcal{D}$

In einem Monoid gilt

1. $x \cdot (y \cdot z) = (x \cdot y) \cdot z$ (Assoziativität der Komposition)
2. $\epsilon \cdot x = x \cdot \epsilon = x$ (Existenz eines neutralen Elements)

In einem Monoid gilt hingegen im Allgemeinen nicht die Kommutativität der Komposition

3. $x \cdot y \neq y \cdot x$ (x und y sind inkompatibel)

Delta Model

- Basierend auf einem Deltoid können wir nun ein Delta Model zur Ableitung einer konkreten Modellvariante aus dem (leeren) Kern definieren.

Definition: Ein Delta Model $(D, \prec)$ besteht aus

- einer endlichen Teilmenge $D \subseteq \mathcal{D}$
- einer strikten partiellen Ordnung $\prec \subseteq D \times D$

Für eine strikte partielle Ordnung $\prec$ gilt

1. $\neg(x \prec x)$ (Irreflexivität)
2. $x \prec y \Rightarrow \neg(y \prec x)$ (Asymmetrie)
3. $x \prec y \wedge y \prec z \Rightarrow x \prec z$ (Transititvität)

Ableitung von Modellvarianten

- Ableitung einer Produktvariante aus einem Delta Modell erfolgt durch Anwendung aller Deltas der Produktvariante auf das (leere) Kernmodell in einer Reihenfolge gemäß der strikten partiellen Ordnung.

Definition: Die Menge möglicher Ableitungen von Modellvarianten aus einem Delta Modell $\mathrm{M} = (D, \prec)$ ist

$$derv(M) \coloneqq \{\, x_n \cdot x_{n-1} \cdot \cdots \cdot x_1 \mid x_1, x_2, \ldots, x_n \; ist\; eine\; Linearisierung\; von \prec \}$$

Die Linearisierung einer partiellen Ordnung $\prec$ auf einer Menge D ist eine Menge von Sequenzen $w = x_1, x_2, \ldots, x_n \in D^*$ mit

1. $\forall w', w'', w''' \in D^*: w = w', x_i, w'', x_{i+1}, w''' \Rightarrow \neg(x_{i+1} \prec x_i)$
2. $\forall x \in D: \exists!\, w', w'' \in D^*: w', x, w'' = w$

Eindeutigkeit von Modellvarianten

- Die Ableitung einer Modellvariante aus einem Delta Modell kann auf unterschiedlichen Wegen (Reihenfolgen von Delta Anwendungen) erfolgen.
- Trotzdem soll das Ergebnis der Ableitung einer Modellvariante aus einem Delta Modell eindeutig sein.

Definition: Die Menge möglicher Ableitungen von Modellvarianten aus einem Delta Modell $\mathrm{M} = (D, \prec)$ ist **eindeutig**, wenn $|derv(M)| = 1$.

Satz: Die Menge möglicher Ableitungen von Modellvarianten aus einem Delta Modell $\mathrm{M} = (D, \prec)$ ist **eindeutig**, wenn für alle Linearisierungen $x_1, x_2, \cdots, x_n$ und $x'_1, x'_2, \cdots x'_n$ von $\prec$ gilt

$$(x'_n \cdot x'_{n-1} \cdot \cdots \cdot x'_1) = (x_n \cdot x_{n-1} \cdot \cdots \cdot x_1)$$

Satz: Die Menge möglicher Ableitungen von Modellvarianten aus einem Delta Modell $\mathrm{M} = (D, \prec)$ ist **eindeutig**, wenn $\prec$ eine strikte Totalordnung ist.

Delta Konflikte

- Für die explizite Prüfung der Eindeutigkeit möglicher Modellvarianten eines Delta Modells müssen alle Ableitungen gebildet und miteinander verglichen werden.
- Dafür müssen im worst case $n!$ Ableitungen betrachtet werden.
- Eine alternative Charakterisierung auf dem Begriff des Delta Konflikts.

Definition: Die Deltas $x, y \in D$ eines Delta Modells $\mathrm{M} = (D, \prec)$ sind in **Konflikt**, wenn

$$x \cdot \mathrm{y} \neq y \cdot x \wedge \neg(x \prec y) \wedge \neg(y \prec x)$$

- Zwei Deltas eines Delta Modells stehen demnach in Konflikt, wenn
  1. das Ergebnis von der Reihenfolge ihrer Anwendung abhängt und
  2. die Reihenfolge ihrer Anwendung nicht durch die Ordnung $\prec$ vorgegeben wird

Eindeutigkeit durch Konfliktfreiheit

- Beobachtung: Die Menge möglicher Ableitungen von Modellvarianten aus einem Delta Modell ist eindeutig, wenn die Delta Menge konfliktfrei ist.

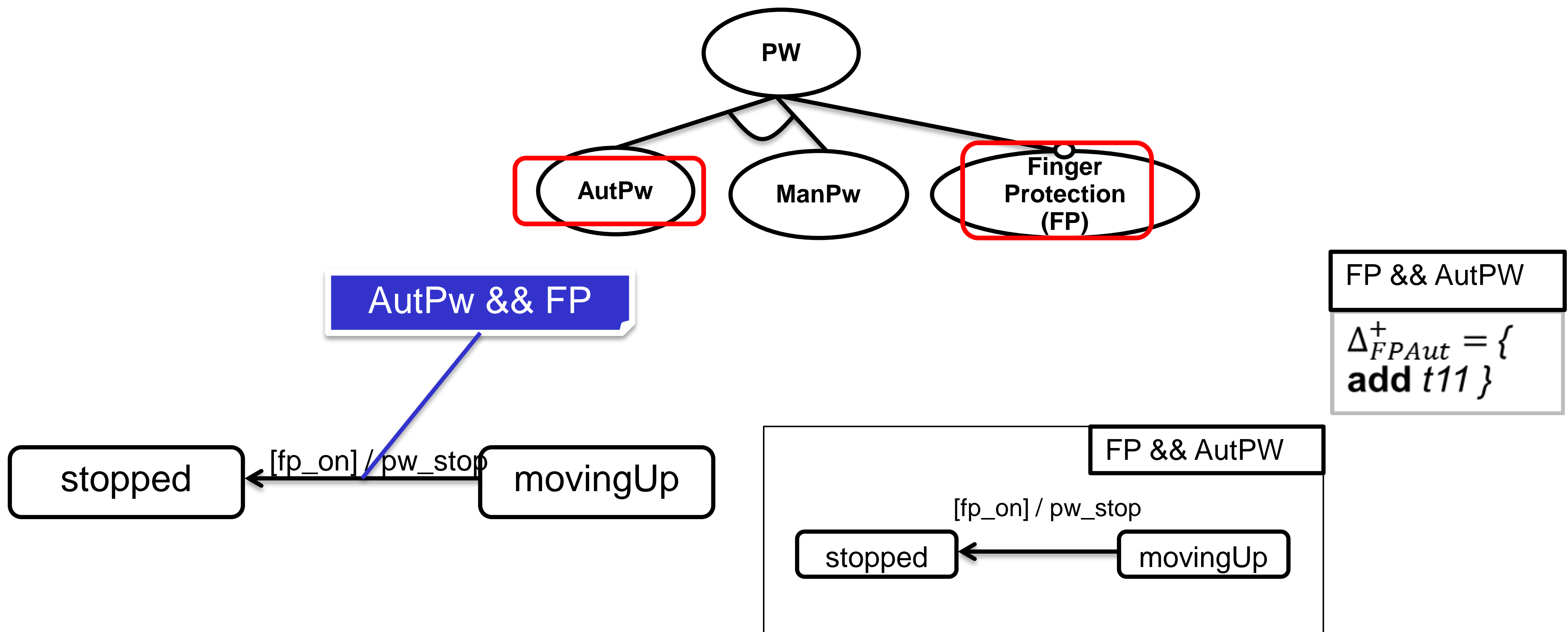
- Die Eindeutigkeit von Modellvarianten kann somit sichergestellt werden, indem zwischen jedem Paar von Deltas, die potentiell in Konflikt stehen können, eine Anwendungsordnung erzwungen wird.

- Problem: Auf diese Weise wird häufig auch eine (implizite) Abhängigkeit zwischen ursprünglich konzeptionell unabhängigen Modell-Teilen bzw. Features erzwungen

  => Optional-Feature-Problem [Kästner et al., 2009]

Das Optional-Feature-Problem

Das Optional-Feature-Problem tritt auf, wenn zwei oder mehr Features unabhängig in der Problemdomäne sind, aber Abhängigkeiten zwischen ihren Implementierungsartefakten im Lösungsraum bestehen.



Konfliktlösende Deltas

- Mögliche Lösung: uneindeutige Zwischenzustände bei der Ableitung von Modellvarianten zulassen und sicherstellen, dass der Konflikt später während der Modellvariantenableitung **garantiert** durch ein weiteres Delta aufgelöst wird.

Definition: Die Deltas $x, y \in D$ eines Delta Modells $\mathrm{M} = (D, \prec)$ seien in Konflikt. Ein Delta $\mathrm{z} \in D$ ist konfliktlösend für x, y falls

$$x \prec z \land y \prec z \land \forall d \in D^*: z \cdot d \cdot y \cdot x = z \cdot d \cdot x \cdot y$$

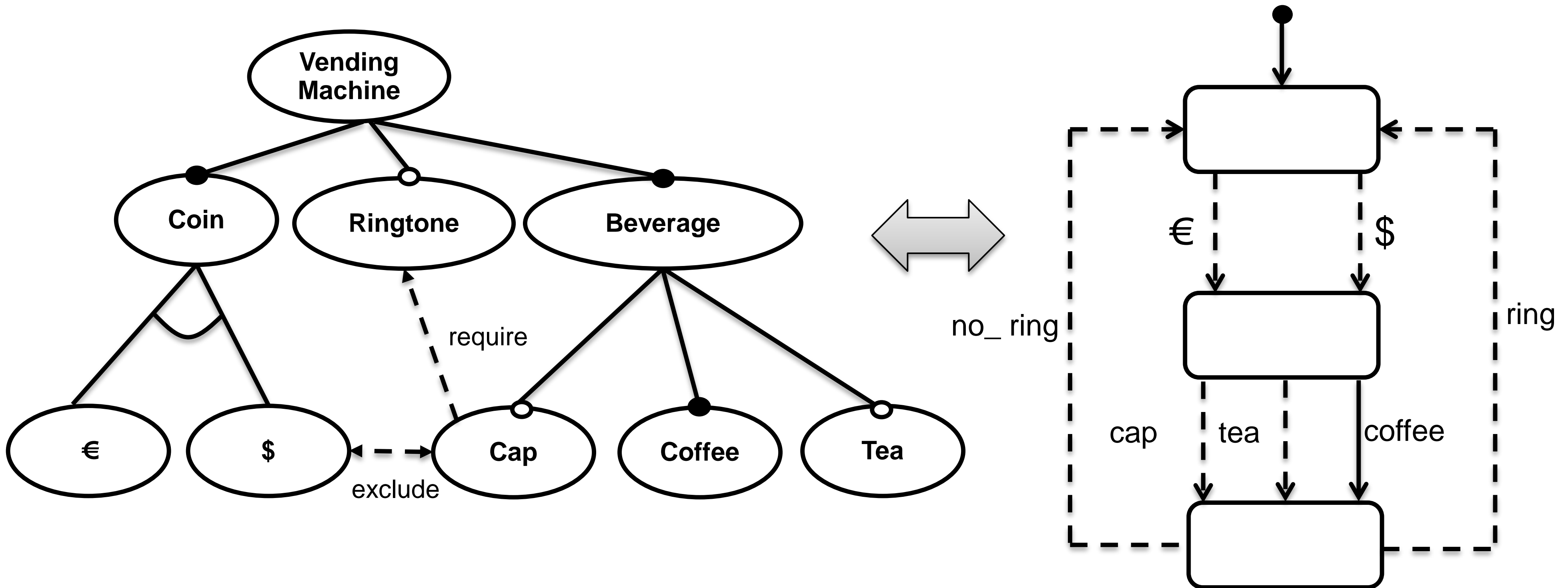
Theorem: Die Menge möglicher Ableitungen eines Delta Modells $\mathrm{M} = (D, \prec)$ ist eindeutig, wenn es für jeden Delta Konflikt ein konfliktlösendes Delta gibt.

Verallgemeinerung: Eine Delta-orientierte Produktlinie besteht aus einer Menge von Delta Modellen $\mathrm{M} = (D, \prec)$. Die Selektion der Delta Menge $D \subseteq \mathcal{D}$ für eine Produktkonfiguration erfolgt auf Basis der Feature-Auswahl.

Intrinsische Variabilitätsmodellierung

- Bisher: Variabilitätsmodellierung durch Mechanismen zur **strukturellen/syntaktischen** Änderungen eines Modells
- Problem: Zusammenhang zwischen strukturellen Unterschieden und Verhaltensunterschieden zwischen Modellvarianten unklar
- Bei intrinsischer Variabilitätsmodellierung ist Variabiltität Teil der **Semantik** der Modellierungssprache
  (ähnlich wie Variability Encoding)
- Beispiel: Modal-Automaten [Larsen et al., 2007], [Asirelli et al., 2011]

Beispiel: Modal-Automaten



Constraints zur Einschränkung gültiger Abläufe:

- € **ALT** $
- $ **EXC** cap
- cap **REQ** ring
- ring **ALT** no_ring

Transitionsmodalitäten:

mandatory

optional

Diskussion: Variabilitätsmodellierung

Derivatives

Passst zur reaktiven SPL Entwicklung

Auswahl eines beliebigen Basis-Produktes

Konflikte

Passt zur extraktiven SPL Entwicklung

Feature-orientierte Trennung gemeinsamer und variabler Teile

Variabilität durch Projektion einer Auswahl

Integrierte Darstellung und Analyse der gesamten SPL

Große und unstrukturierte Modelle

Beliebige Trennung gemeinsamer und variabler Teile

Variabilität durch Kombination von Teilen

Variabilität durch Änderung

Passt zur proaktiven SPL Entwicklung

Common Variability Language (CVL)

Object Management Group

First Needham Place
250 First Avenue, Suite 100
Needham, MA 02494

Telephone: +1-781-444-0404
Facsimile: +1-781-444-0320

Common Variability Language (CVL)

Request For Proposal

OMG Document: ad/2009-12-03

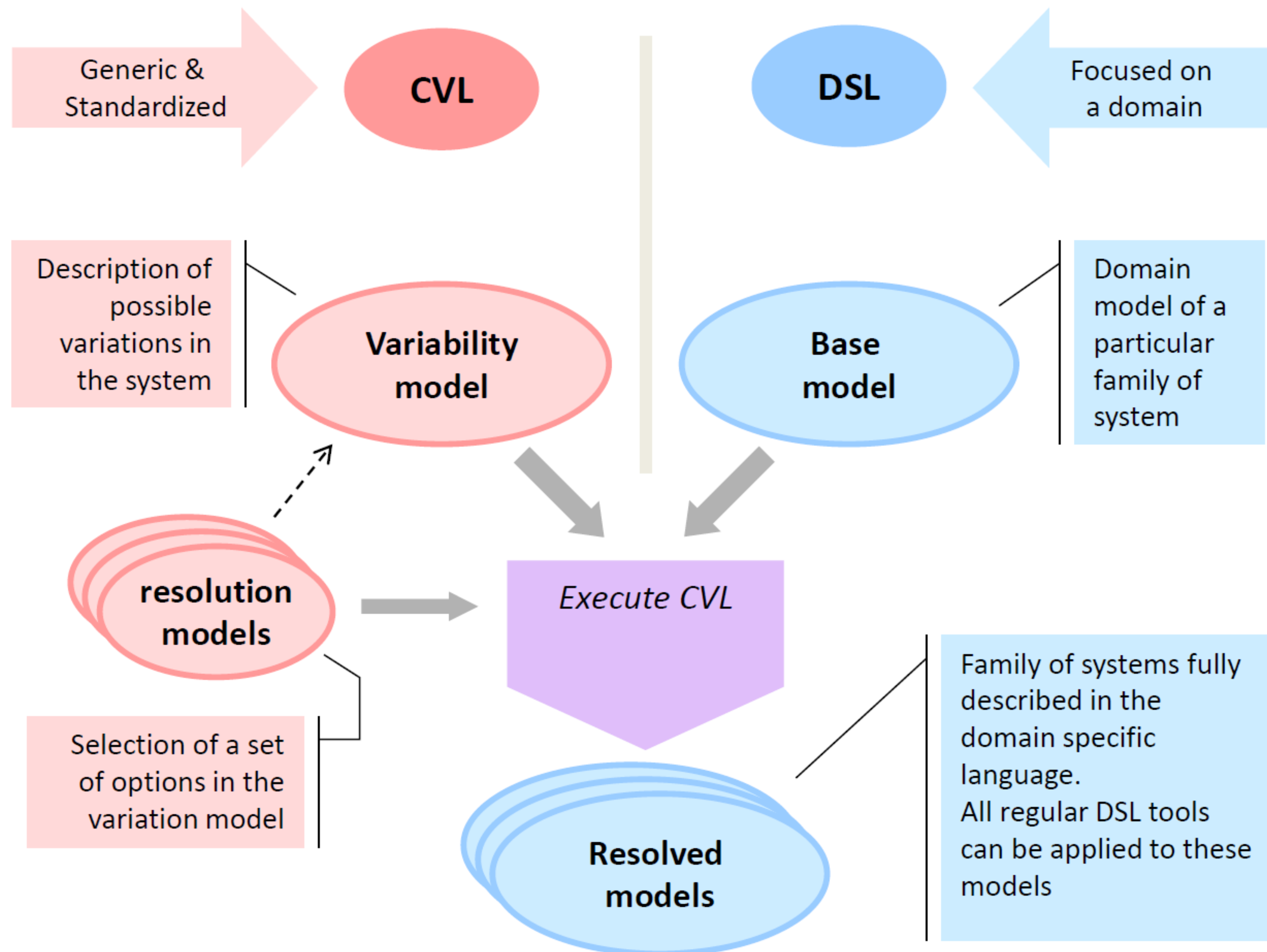
Letters of Intent due: May, 2010
Submissions due: August, 2010

Objective of this RFP

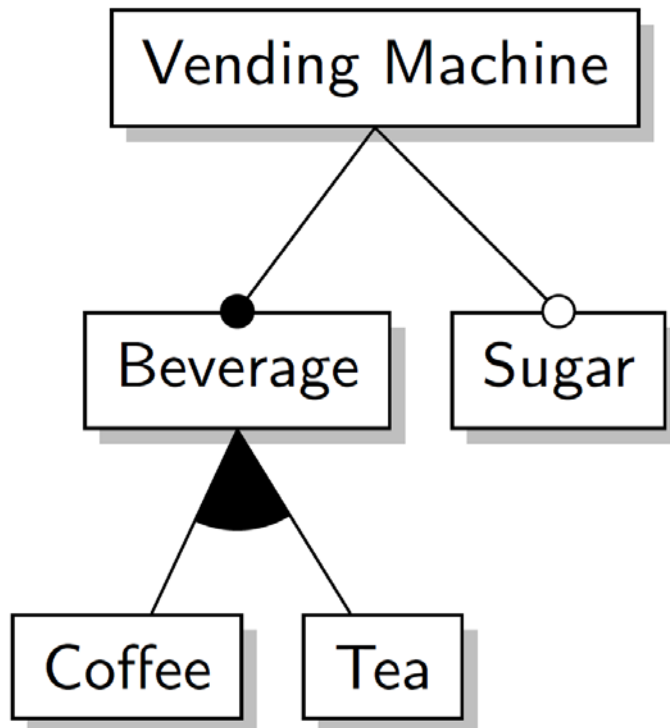
The objective of this RFP is to enable the specification of the variability in product line models in order to support seamless product line modeling across the whole product line engineering process. This CVL RFP requests a specification language including a metamodel, semantics and concrete syntax for

- Domain-independent language for specifying and resolving variability
- Intuitive ways to describe the product line variability
- Automatic means to produce products from product line
- Generic ways to describe variability
- Techniques for making generic tools

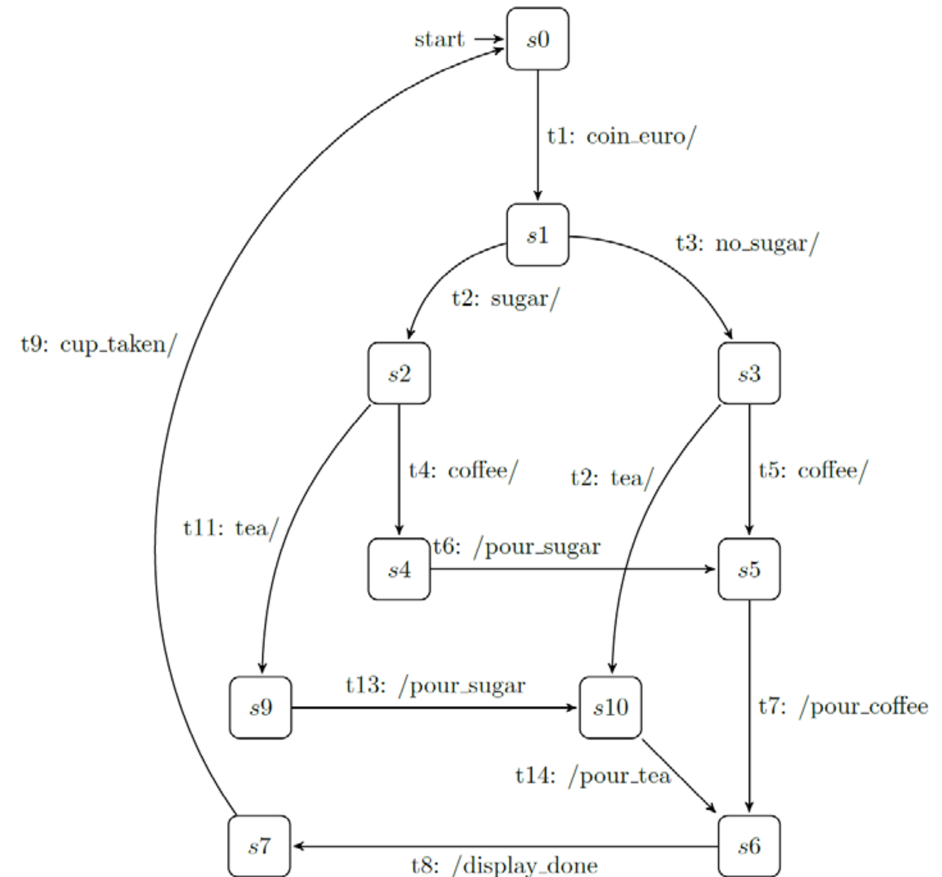
The CVL Process



Example: Vending Machine SPL

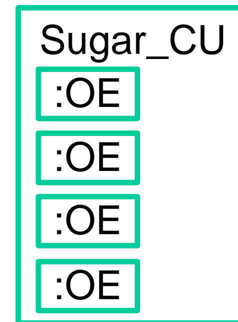


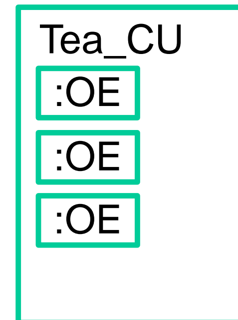
Variability Model
(z.B. Feature Model)

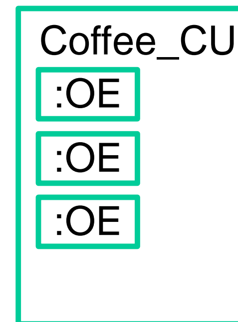


Base Model
(z.B. Zustandsautomat)

Vending Machine – Variation Points

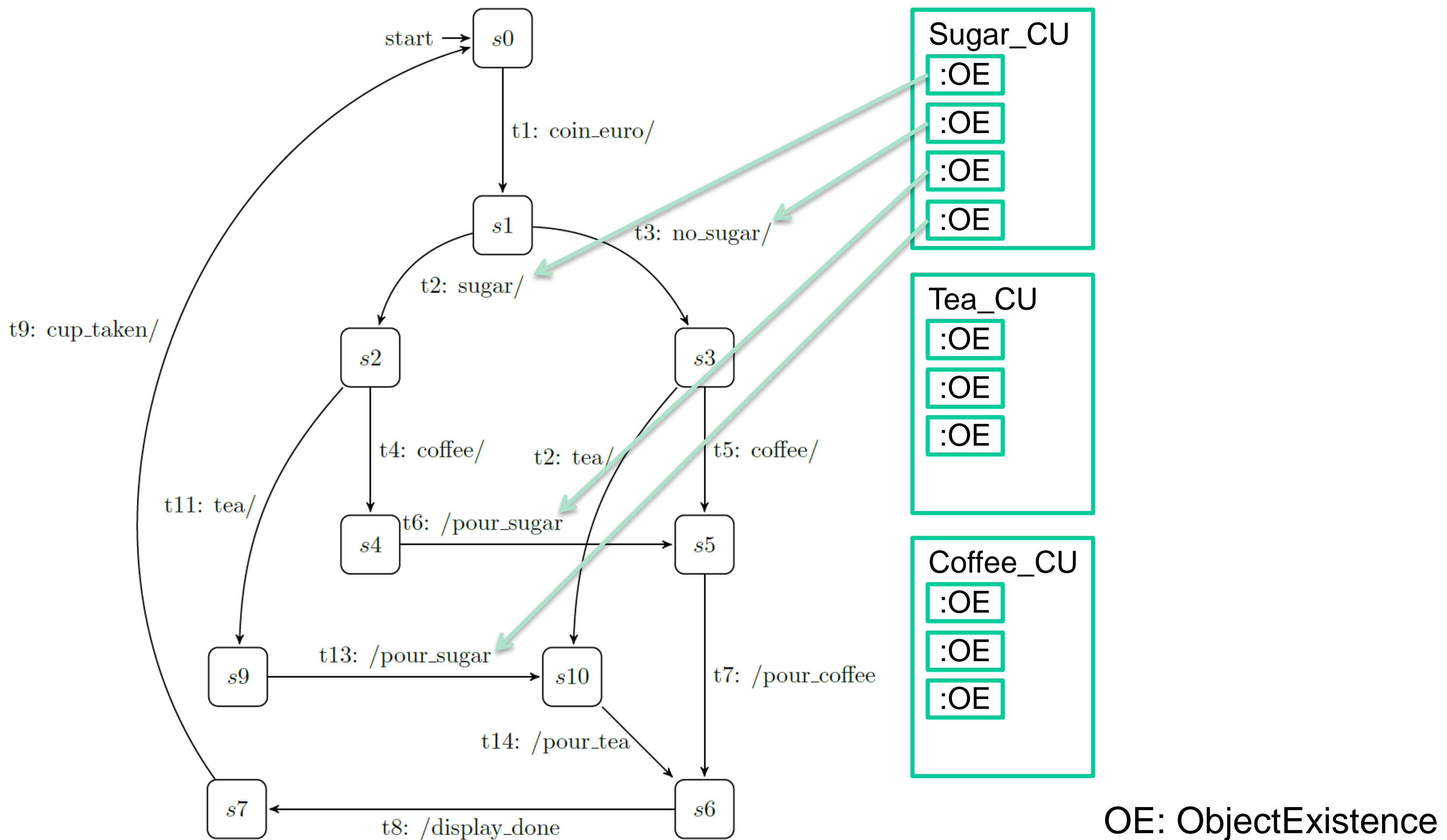






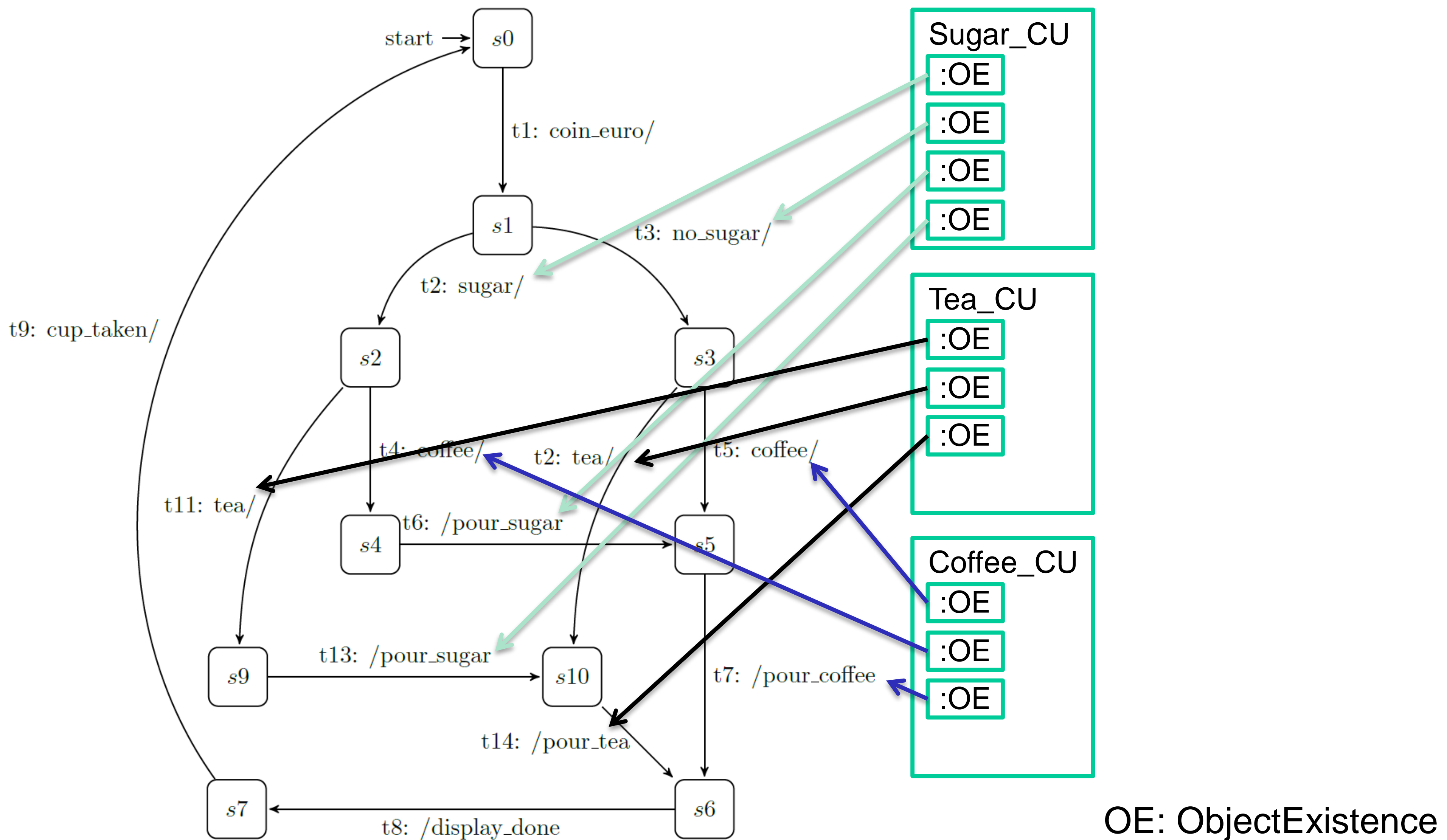
OE: ObjectExistence

Vending Machine – Binding



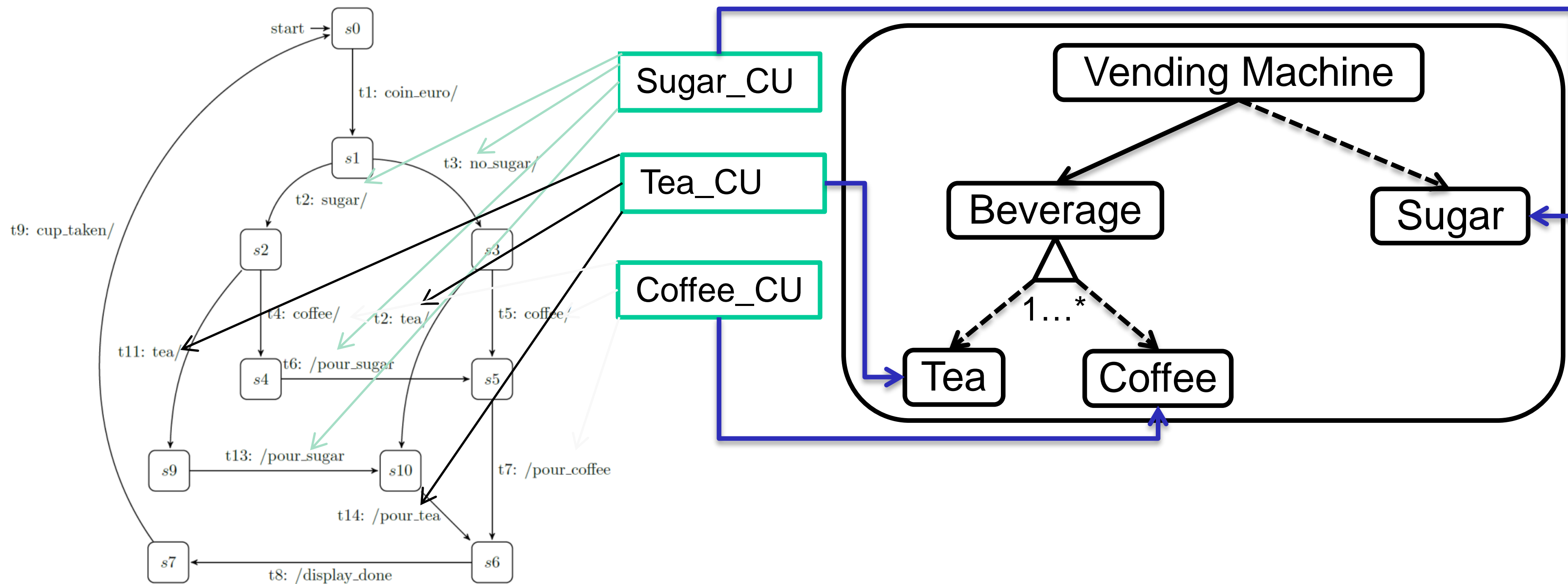
OE: ObjectExistence

Vending Machine – Binding

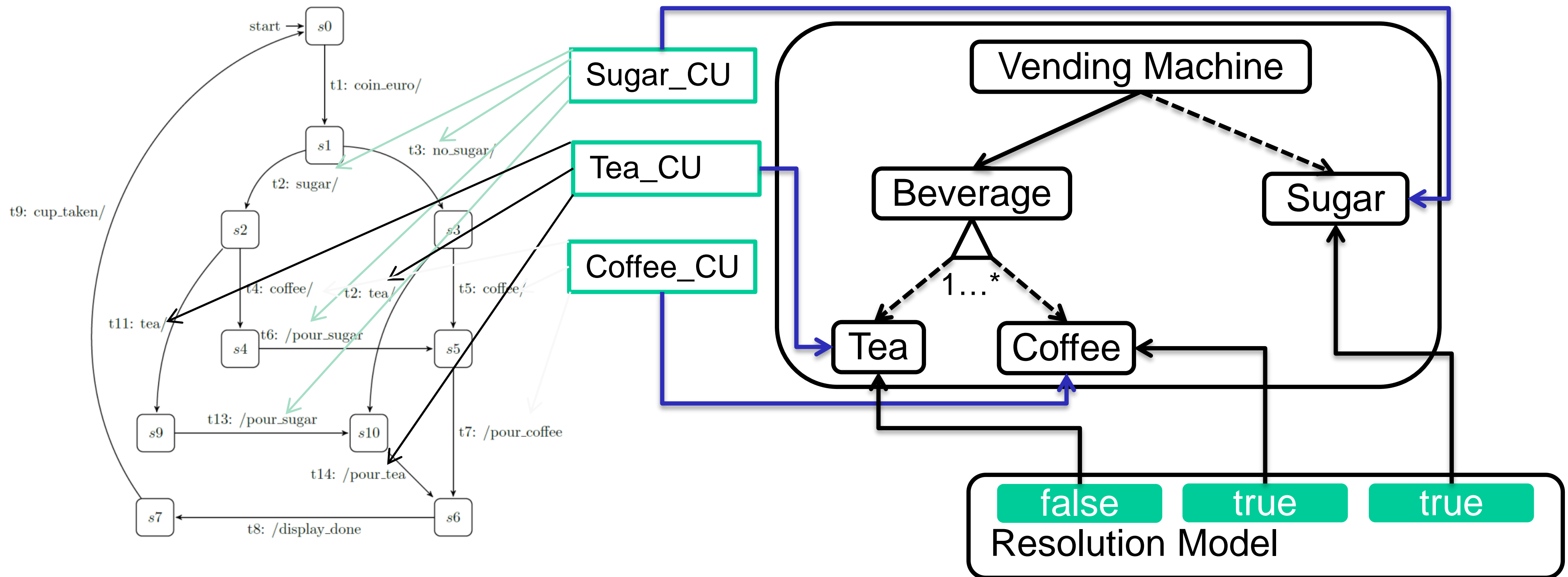


OE: ObjectExistence

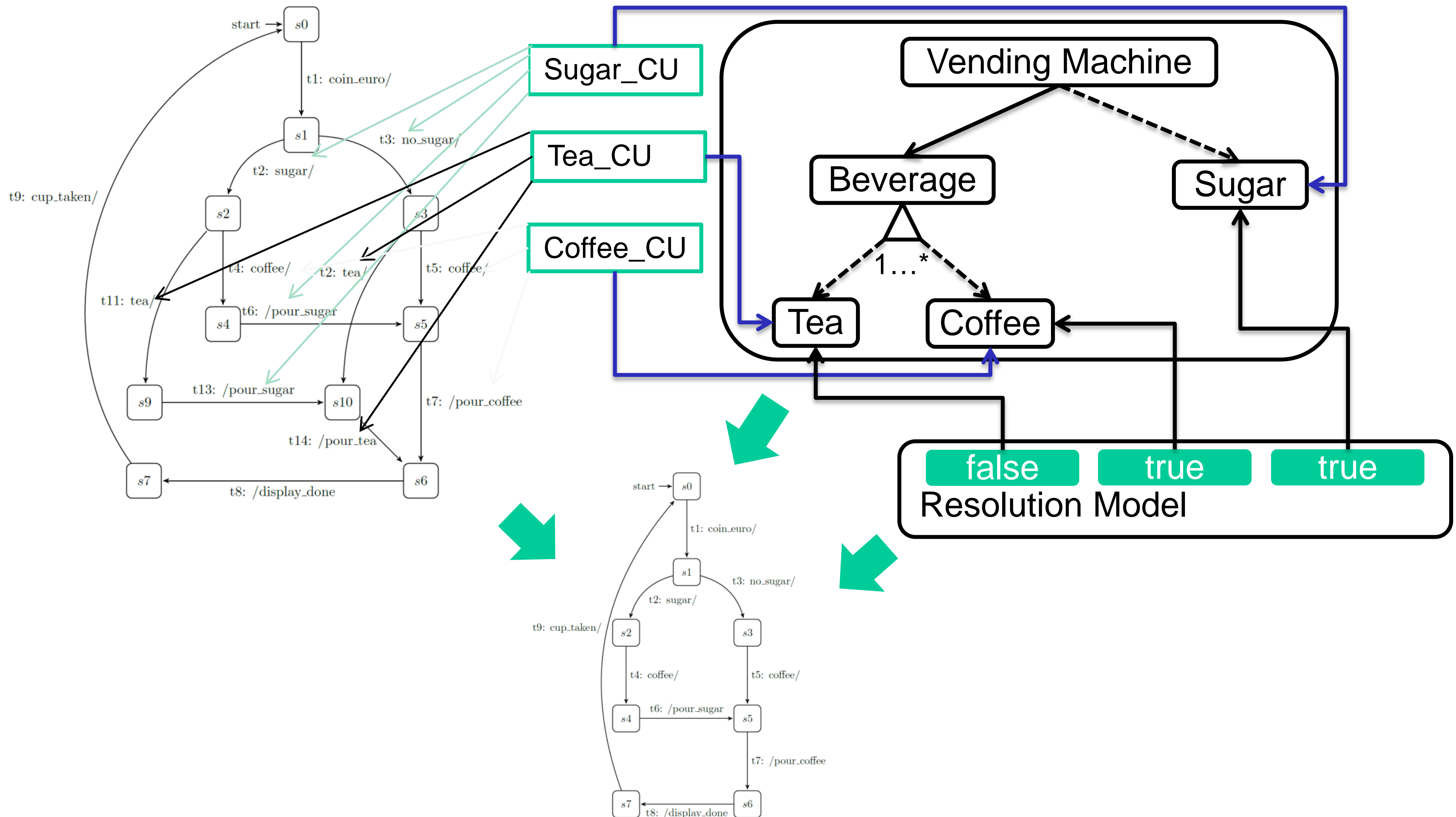
Vending Machine – Variability Model



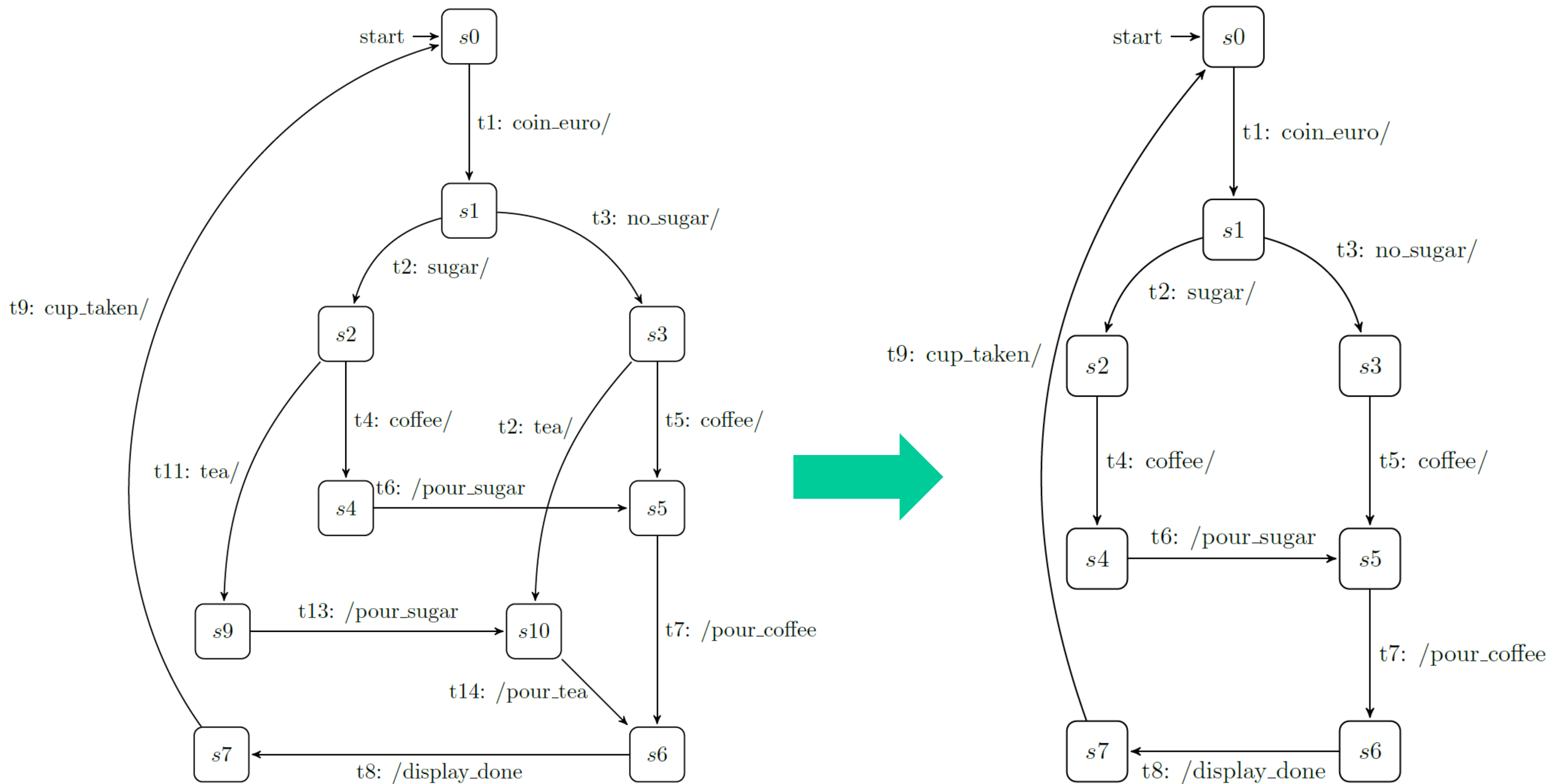
Vending Machine – Resolution Model



Vending Machine – Materialization

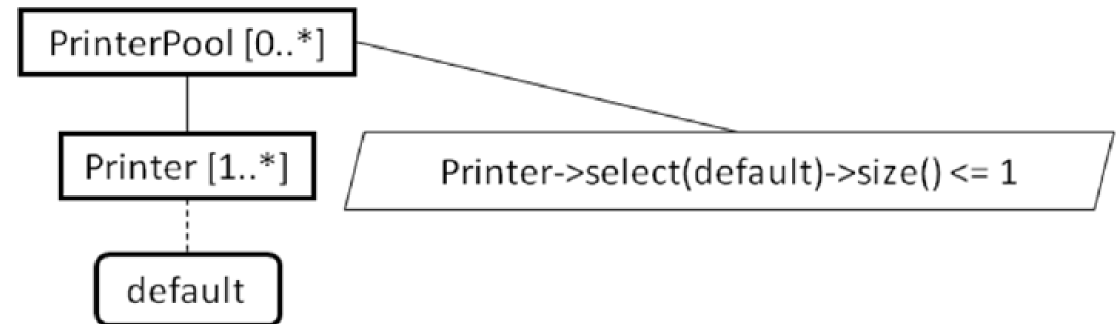


Vending Machine – Materialization



CVL Constraints

- Propositional constraints:
  fax implies printer



- Arithmetic Constraints:
  Speed = minSpeed + 300
- Quantification with ∃ and ∀:
  colorCapable = Printer->exists(color)
- Constraints over sets
- …

Referenzen (1/2)

- *Krzysztof Czarnecki and Michal Antkiewicz.* **Mapping Features to Models: A Template Approach Based on Superimposed Variants.** Generative Programming and Component Engineering, volume 3676 of Lecture Notes in Computer Science, pages 422–437, 2005.
- *Christian Kästner, Sven Apel, and Martin Kuhlemann.* **Granularity in Software Product Lines.** In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08. pages 311-320, 2008.
- *Christian Prehofer.* **Plug-and-Play Composition of Features and Feature Interactions with Statechart Diagrams.** In FIW, pages 43–58, 2003.
- *Dave Clarke, Michael Helvensteijn, and Ina Schaefer.* **Abstract Delta Modeling.** Mathematical Structures in Computer Science, 2011.
- *Ina Schaefer:* **Variability Modelling for Model-Driven Development of Software Product Lines.** VaMoS 2010: 85-92
- *Kim G. Larsen, Ulrik Nyman, and Andrzej Wąsowski.* **Modal I/O Automata for Interface and Product Line Theories.** In Proceedings of the 16th European Conference on Programming, ESOP'07, pages 64–79, Berlin, Heidelberg, 2007.

Referenzen (2/2)

- *Christian Kästner, Sven Apel, Syed Saif Ur Rahman, Marko Rosenmüller, Don Batory, and Gunter Saake.* **On the Impact of the Optional Feature Problem: Analysis and Case Studies,** *2009.*

- *Martin Erwig and Eric Walkingshaw.* **The Choice Calculus: A Representation for Software Variation.** *ACM Transactions on Software Engineering and Methodology, 2011.*

- *Don Batory.* **Feature-Oriented Programming and the AHEAD Tool Suite.** In *Proceedings of the 26th International Conference on Software Engineering* (ICSE '04), 2004.

- *Sven Apel and DeLesley Hutchins.* **A Calculus for Uniform Feature Composition.** ACM Transactions on Programming Languages and Systems (TOPLAS), 2008.

- *Øystein Haugen:* **Common Variability Language (CVL),** 2012, URL: http://www.omgwiki.org/variability/lib/exe/fetch.php?id=start&cache=cache&media=cvl-revised-submission.pdf