Software Product Lines

Concepts, Analysis and Implementation

Testansätze für Produktlinien

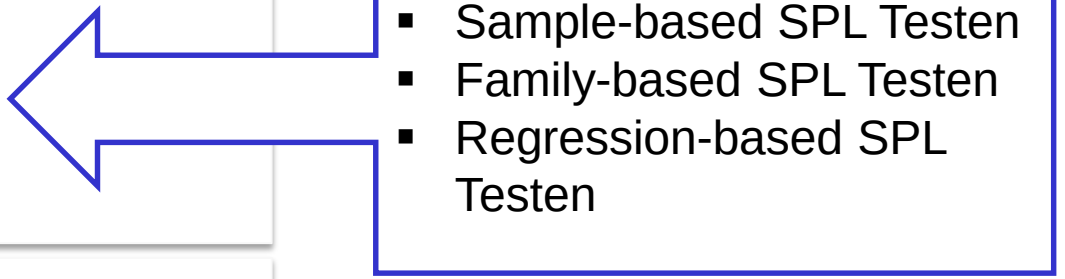
Dr. Malte Lochau
Malte.Lochau@es.tu-darmstadt.de

Inhalt

I. Einführung
- Motivation und Grundlagen
- Feature-orientierte Produktlinien

II. Produktlinien-Engineering
- Feature-Modelle und Produktkonfiguration
- Variabilitätsmodellierung im Lösungsraum
- Programmierparadigmen für Produktlinien

III. Produktlinien-Analyse
- Feature-Interaktion
- Testen von Produktlinien
- Verifikation von Produktlinien



IV. Fallbeispiele und aktuelle Forschungsthemen

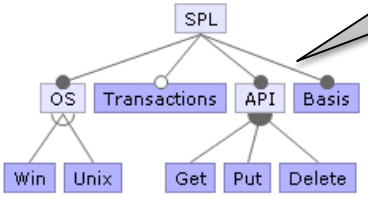
Produktlinien-Analyse

Domain Eng.

Application Eng.

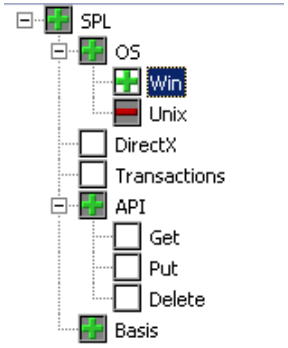
Valider Produktraum korrekt spezifiziert?

Feature-Modell

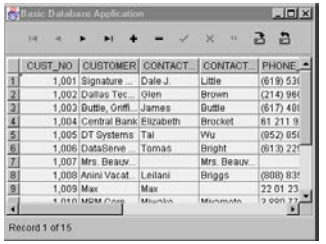


Feature-Artefakte korrekt spezifiziert?

Feature-Artefakte korrekt gemappt und komponiert?



Feature-Auswahl



Korrekt?

Produktkonfiguration korrekt abgeleitet und gemappt?

Produkt korrekt generiert?

Korrektheit der SPL

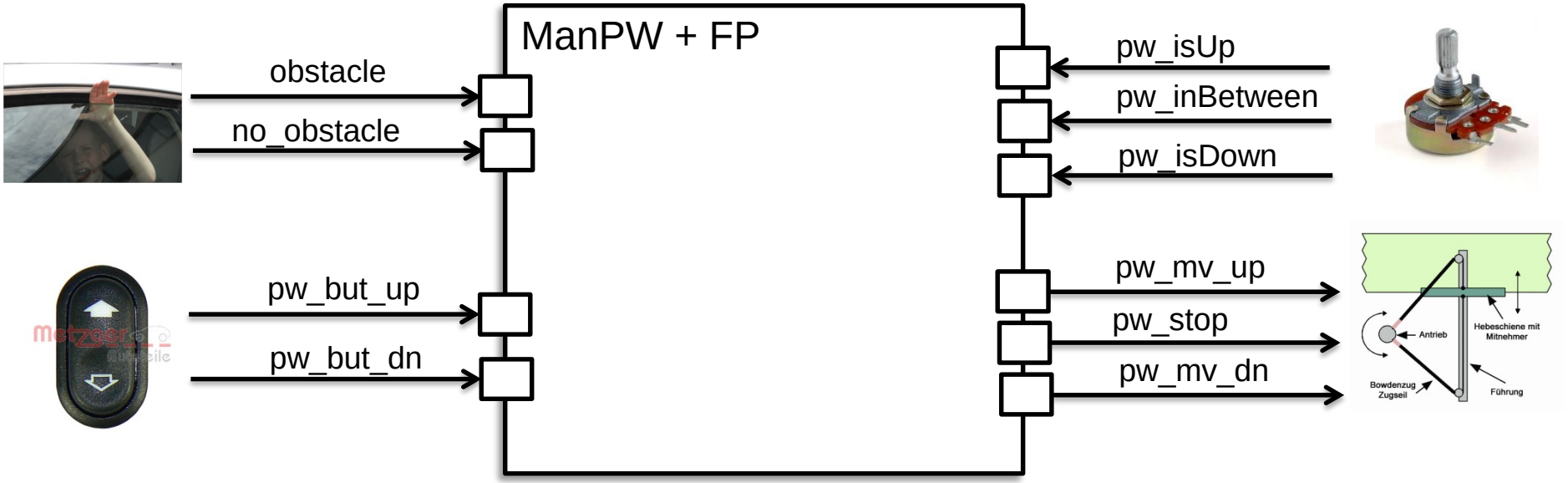
Korrektheit der SPL Werkzeuge

Qualitätssicherung von Software

Techniken
- Simulation
- **Testen**
- **Model Checking**
- Theorem Proving
- …

Prüfen der **Korrektheit**
- Die Implementierung erfüllt die Anforderungen
- Die Implementierung verhält sich wie erwartet
- …

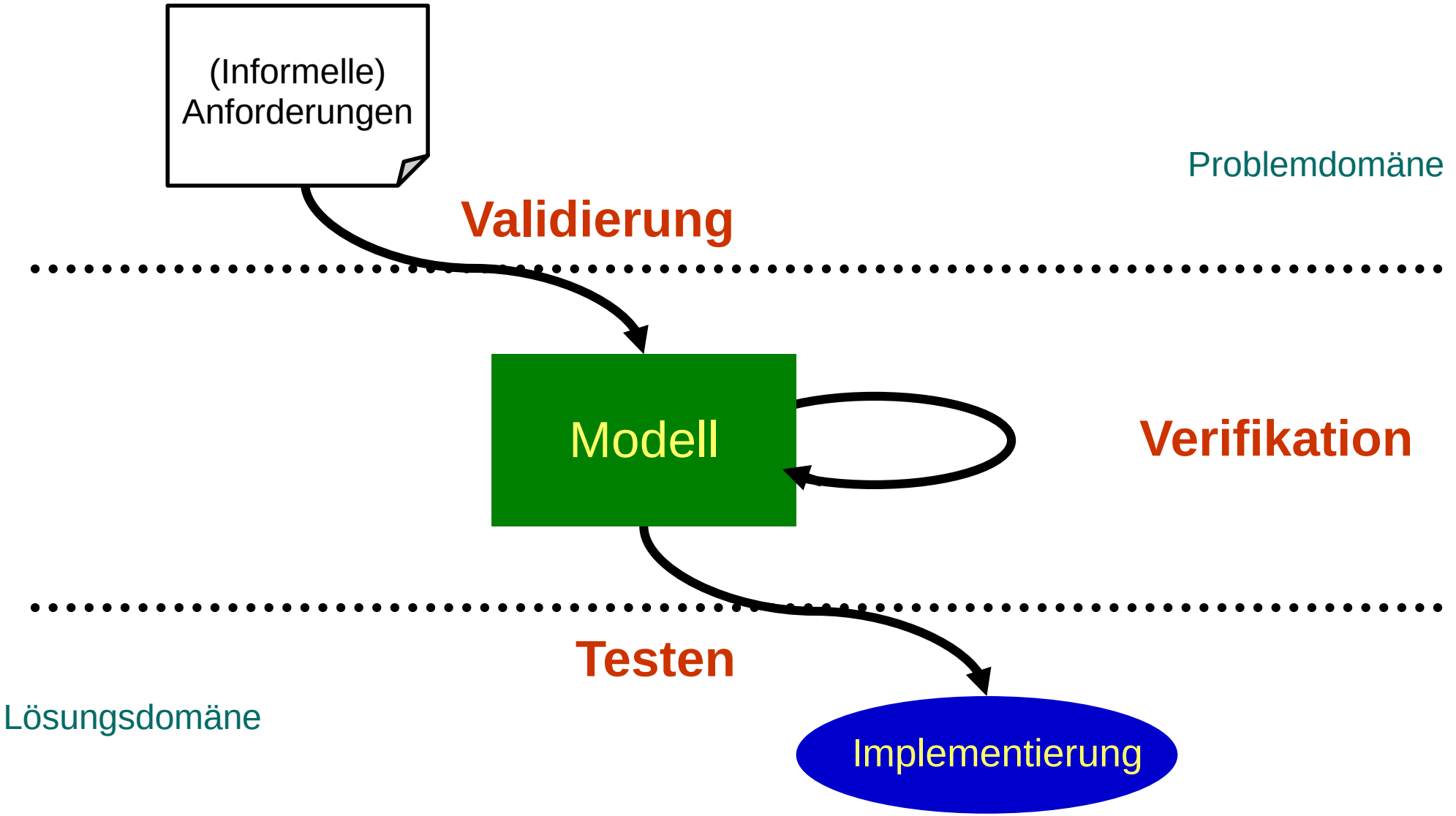
Beispiel: Anforderungen an BCS



- [...] Bei gedrücktem Hoch/Runter-Taster soll sich das Fenster hoch/runter bewegen
- […] Wenn das Fenster die höchste/tiefste Position erreicht, muss der Lauf stoppen
- […] Beim Loslassen des Tasters soll der Lauf stoppen
- […] Wenn ein Hindernis während des Hochlaufs erkannt wird, soll der Hochlauf stoppen
- […] Nachdem ein Hindernis entfernt wurde, soll der Fensterlauf fortgesetzt werden können
- […] Bei Auftreten eines Hindernisses muss der Hochlauf innerhalb von 10 ms stoppen
- …

- Funktionale Sicherheits-anforderungen
- Funktionale Lebendig-keitsanforderungen
- Nichtfunktionale Anforderungen

Validierung, Verifikation, und Testen



Modellbasierte Verifikation vs. Testen

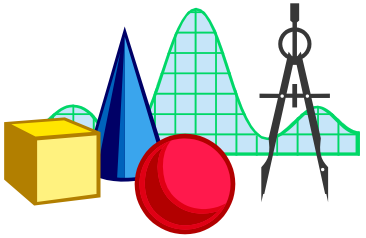
Modellbasierte Verifikation:

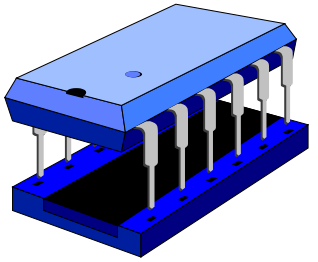
- Formaler Beweis von Korrektheitseigenschaften
- Mathematische Manipulation eines Systemmodells

Modellbasierter Test:

- Stichprobenartiges Suchen nach Fehlern
- Experimentelle Manipulation einer konkreten Systemimplementierung

Abstraktion des Systems



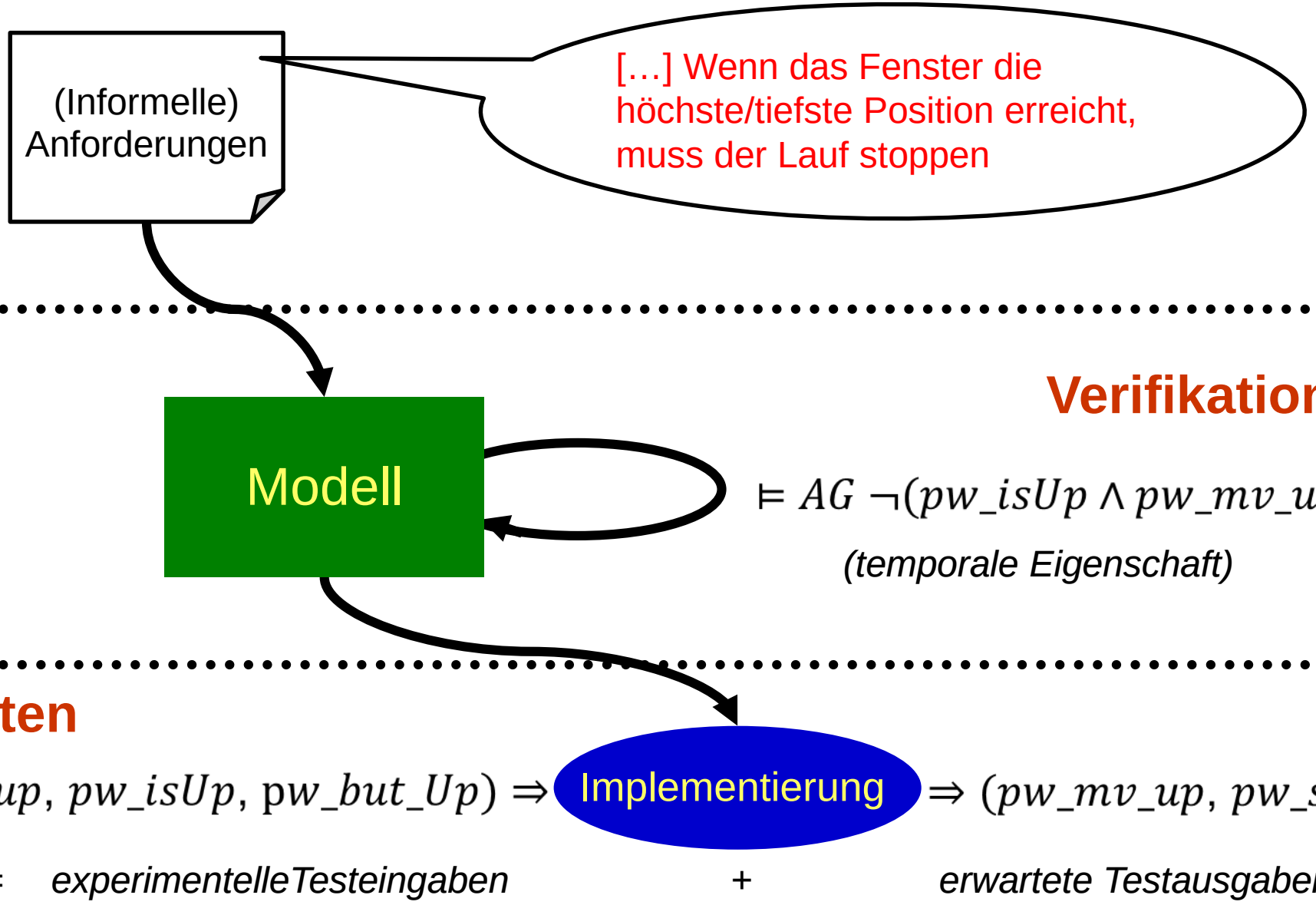


Konkrete Implementierung des Systems

Qualität der Verifikationsergebnisse ist abhängig von der Validität des Systemmodells.

Durch Testen kann nur das Vorhandensein von Fehlern gezeigt werden, nicht deren Abwesenheit.

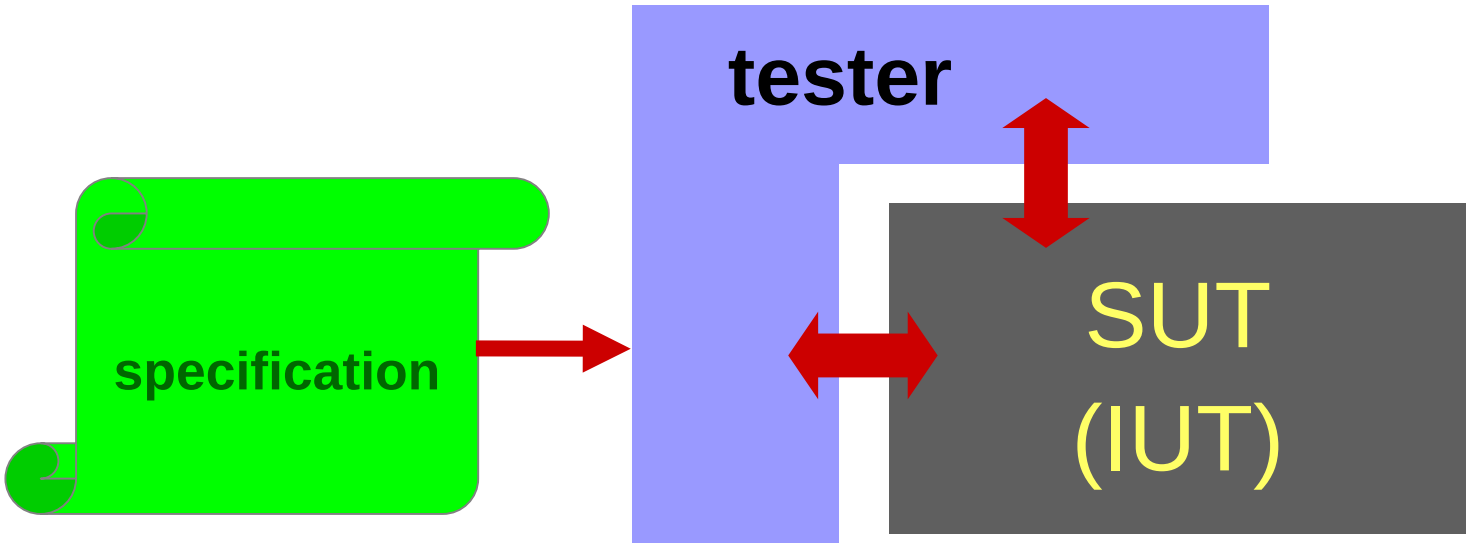
Beispiel: BCS



(Software-)Testen ist…

... an activity performed for evaluating product quality, and for improving it, by identifying defects and problems. **[IEEE 1990]**

… an activity for checking or measuring some quality characteristics of an executing object by performing experiments in a controlled way w.r.t. a specification **[Tretmans 1999]**



Testansätze - Taxonomien

Testtechnik

- Statisch – dynamisch
- Passiv – aktiv ...
- Positiv – negative

Testart

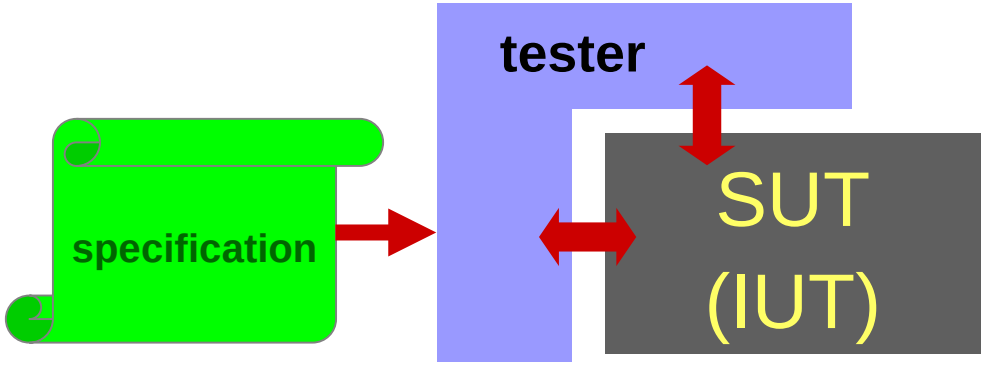
- Funktional (conformance)
- Performanz
- Robustheit
- Stresstest
- Zuverlässigkeit …

Testlevel

- Unit
- Components
- Integration
- System

Testsichten

- Black-Box
- White-Box
- Grey-Box



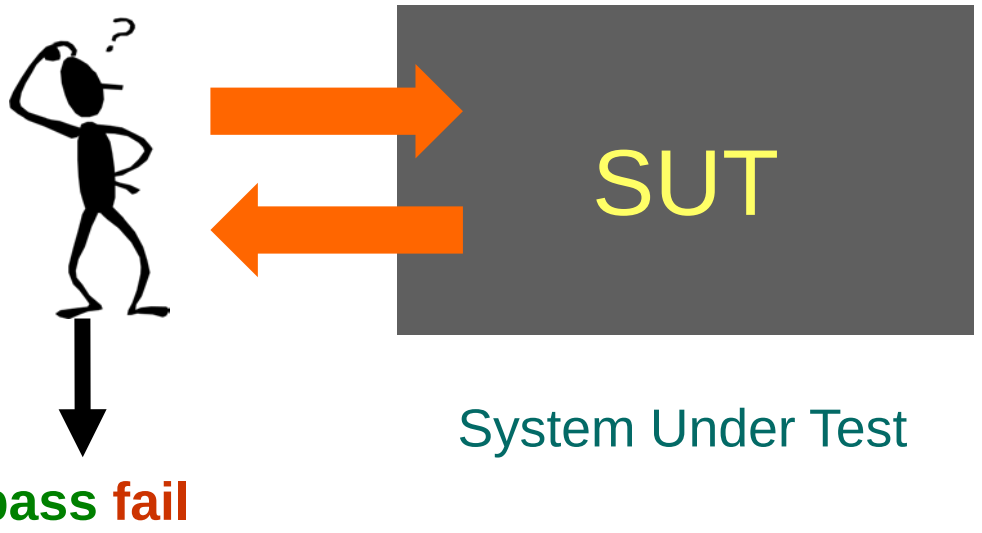
Dynamischer Software-Test

> […] Software testing consists of the **dynamic validation/verification** of the behavior of a program on a **finite set of test cases** suitably selected from the usually infinite input/execution domain against the expected behavior. **[Utting 2007]**

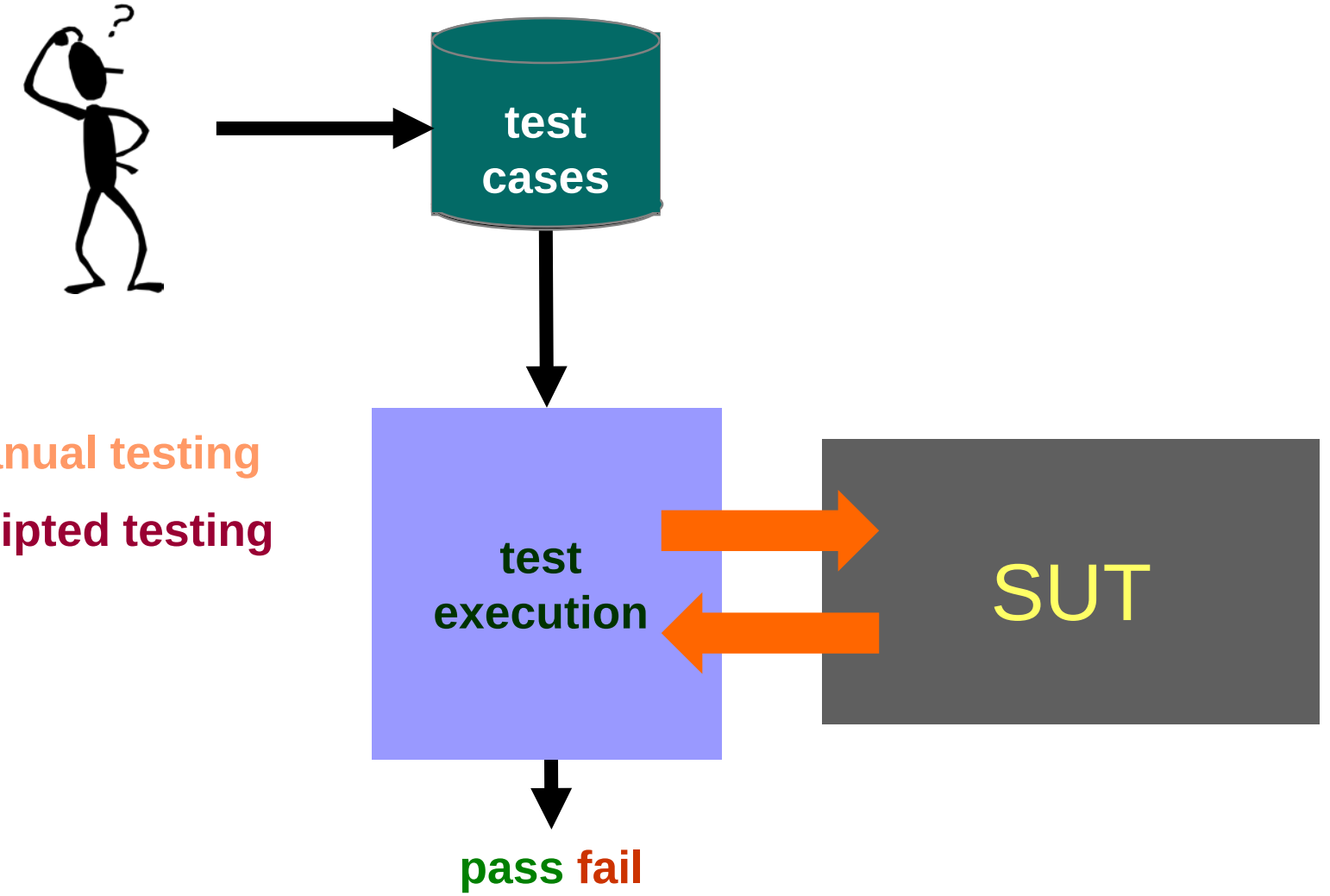
- Gezieltes Stimulieren und Beobachten des Verhaltens einer Software-Implementierung durch Testfälle
- Experimentelle Ausführung des Systems durch Kontrolle der Umgebung
- Testfallerstellung auf Grundlage einer Systemspezifikation

1. Manueller Test

1. Manual testing

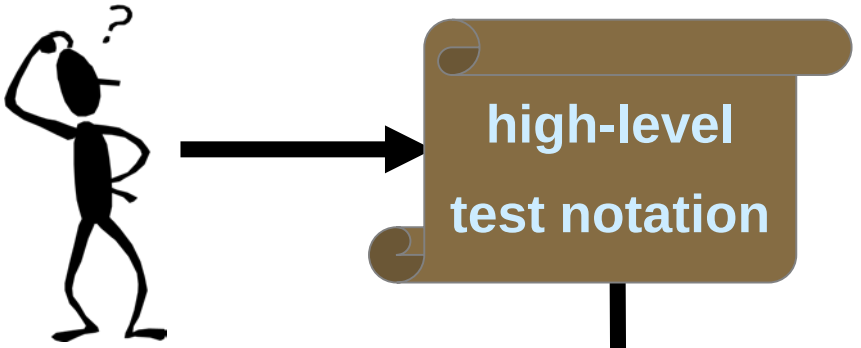


2. Scripted Testing



1. Manual testing
2. **Scripted testing**

3. High-Level Scripted Testing



high-level test notation

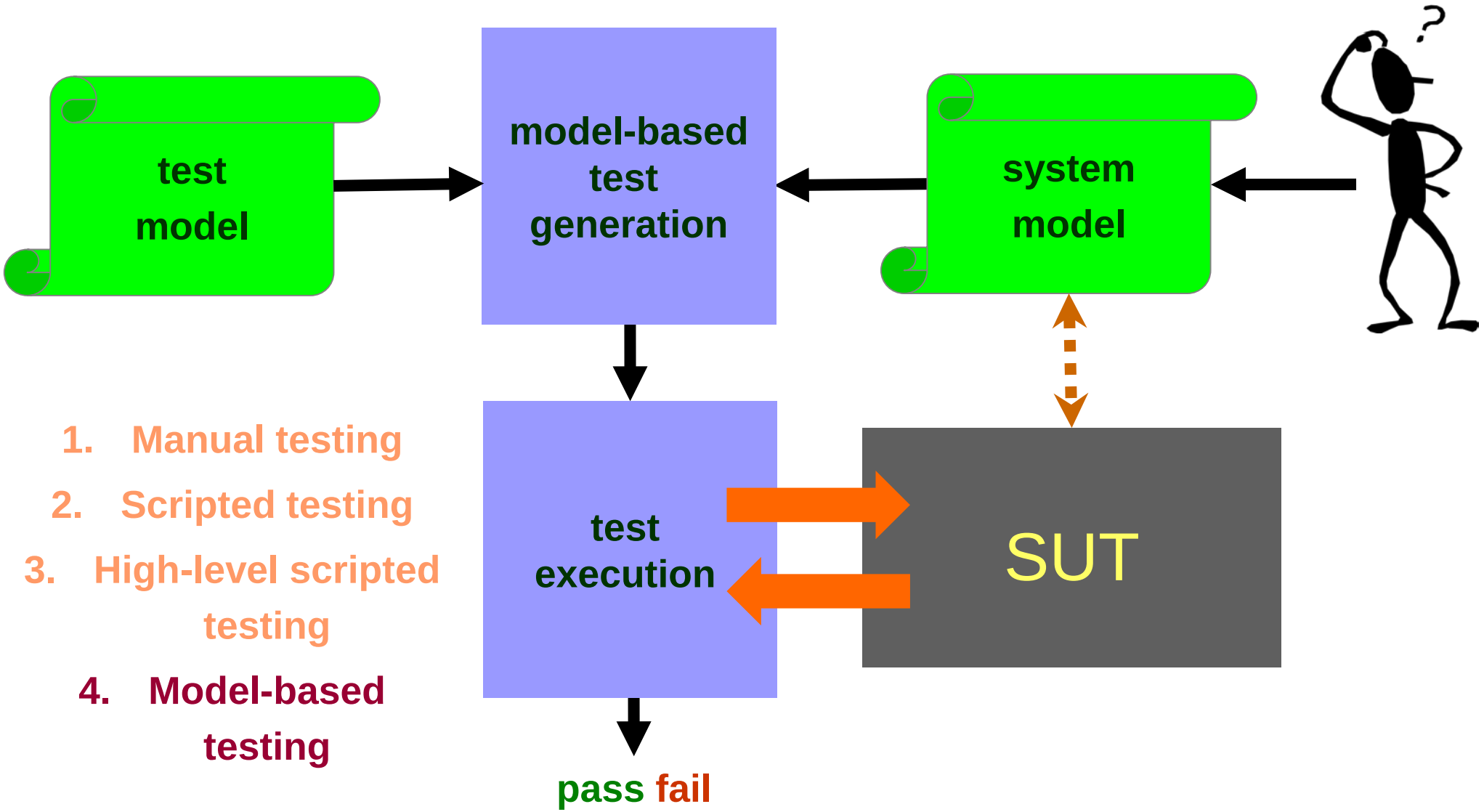
1. Manual testing
2. Scripted testing
3. **High-level scripted testing (keyword testing)**

test execution

SUT

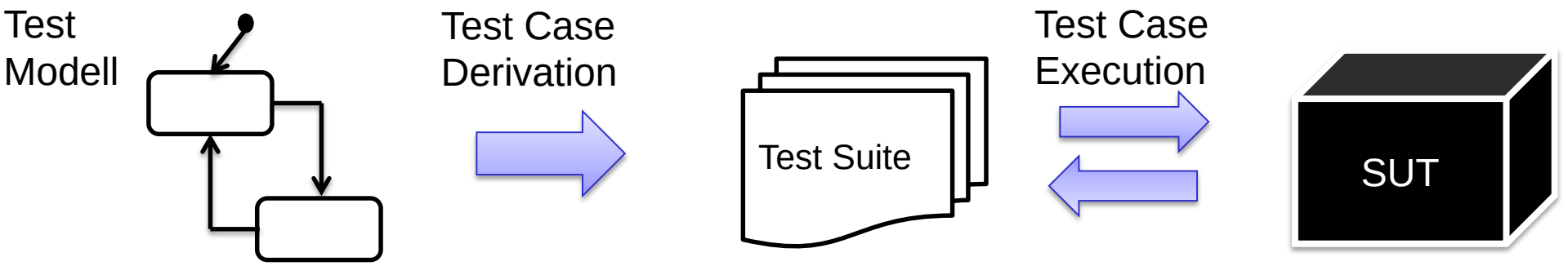
pass fail

4. Model-Based Testing

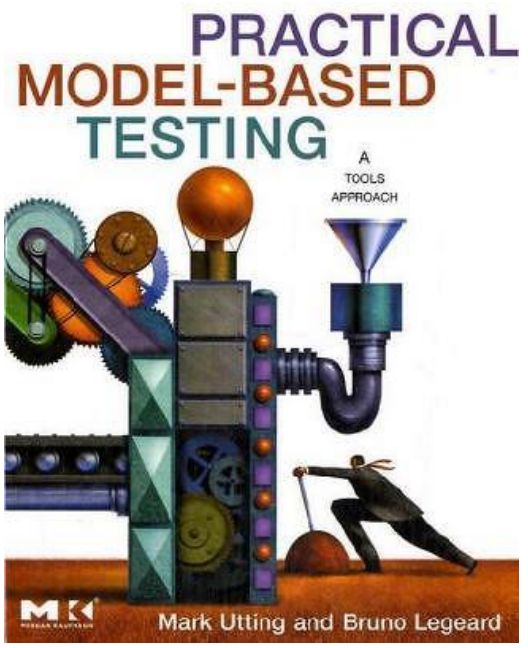


1. Manual testing
2. Scripted testing
3. High-level scripted testing
4. **Model-based testing**

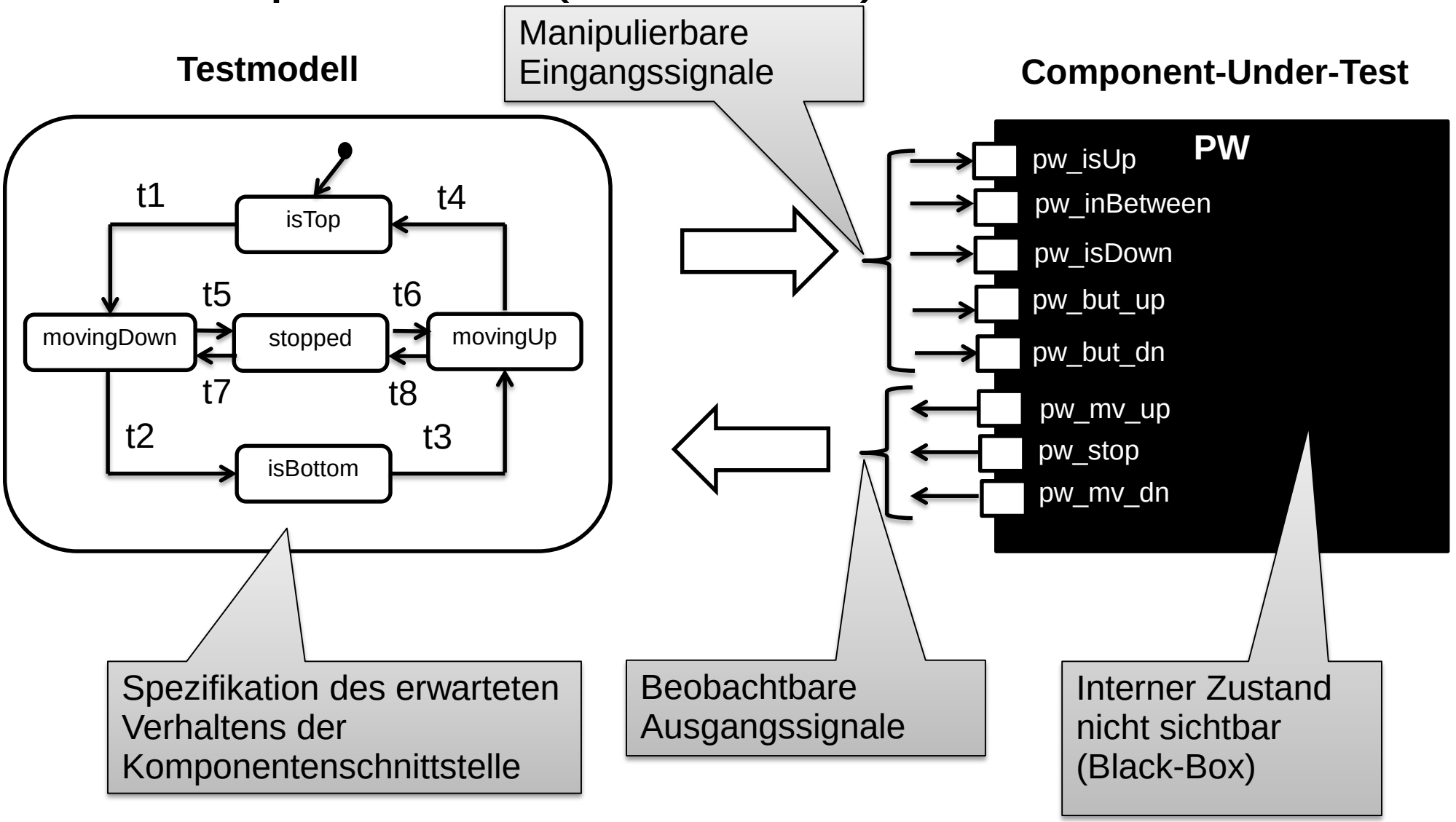
Modellbasierter Test: Grundlagen



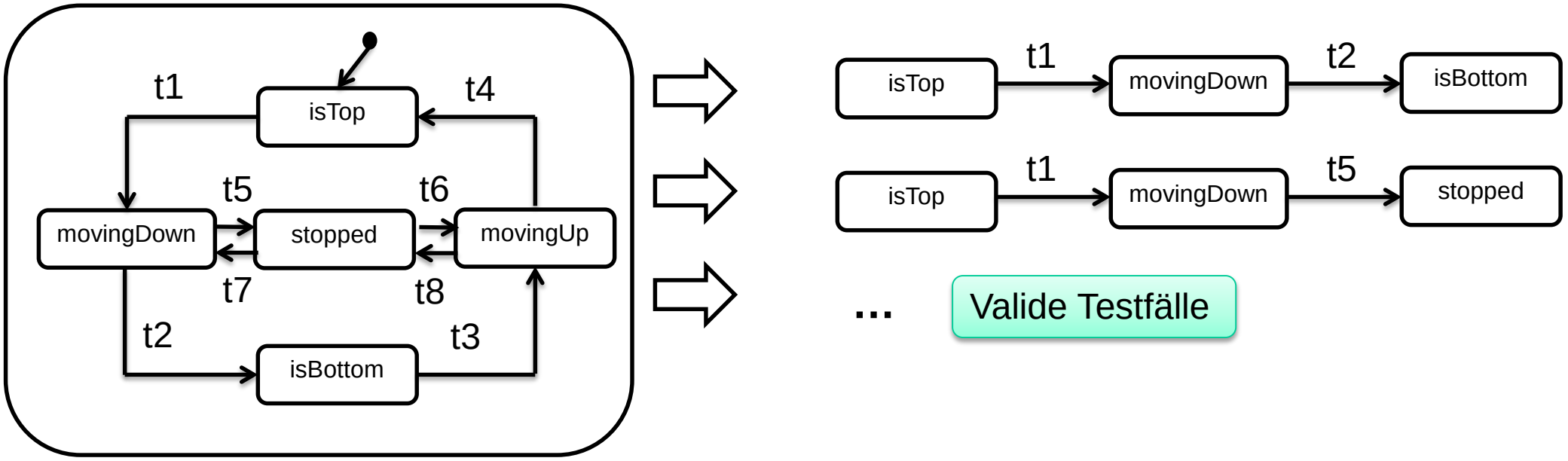
- Black-Box-Annahme für SUT
- Testfallerzeugung, Testfallselektion, Testauswertung, Abdeckungs-messung etc. auf Grundlage des **Testmodells**
- Testfall: Sequenz von **kontrollierbaren Eingaben** und **beobachten Ausgaben** des IUT



Beispiel: Modellbasierter Komponenten-(Funktions-)Test

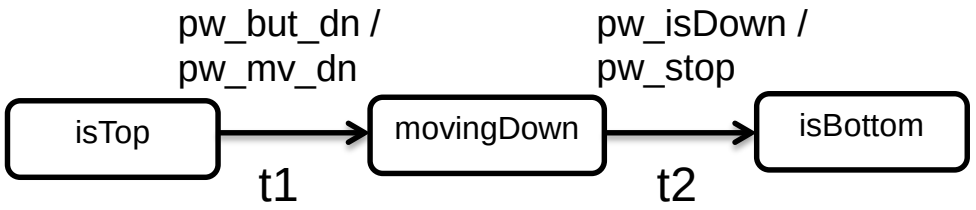


Testfallerzeugung aus State Machines



- Ein **Testfall** entspricht einer endlichen Sequenz von Transitionen des Testmodells
- Ein **valider Testfall** entspricht einem validen Transitionspfad im Testmodell
- Ein **Testschritt** entspricht einer Transition im Transitionspfad

Testfallausführung



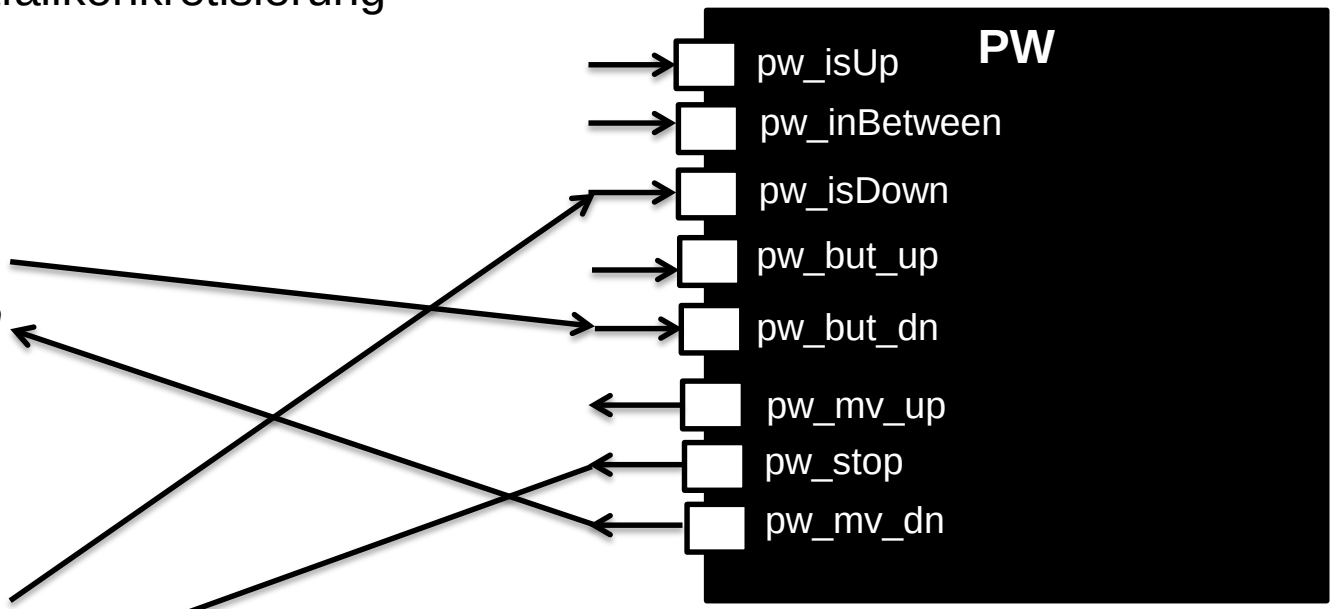
Testfallkonkretisierung

1. Testschritt: Transition t1

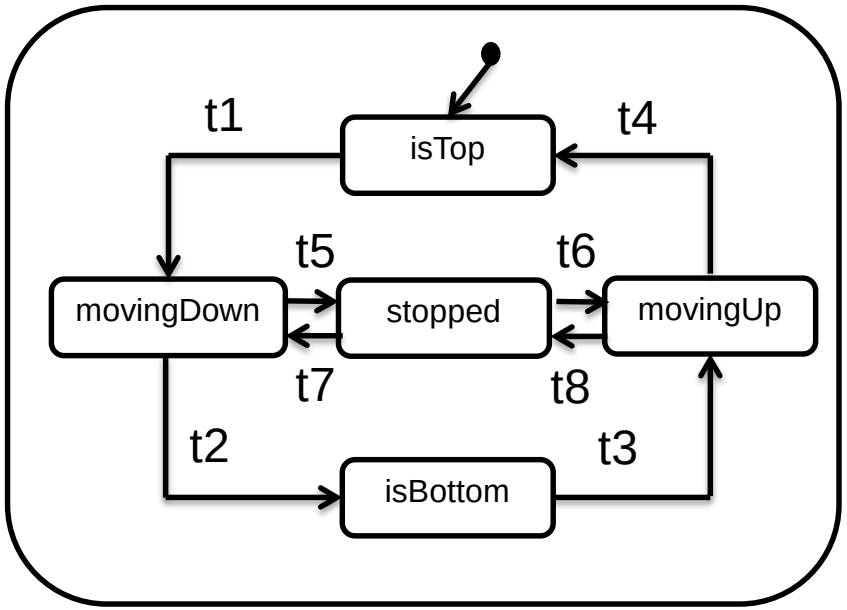
- Eingabesignal *!pw_but_dn*
- Ausgabesignal *?pw_mv_dn*

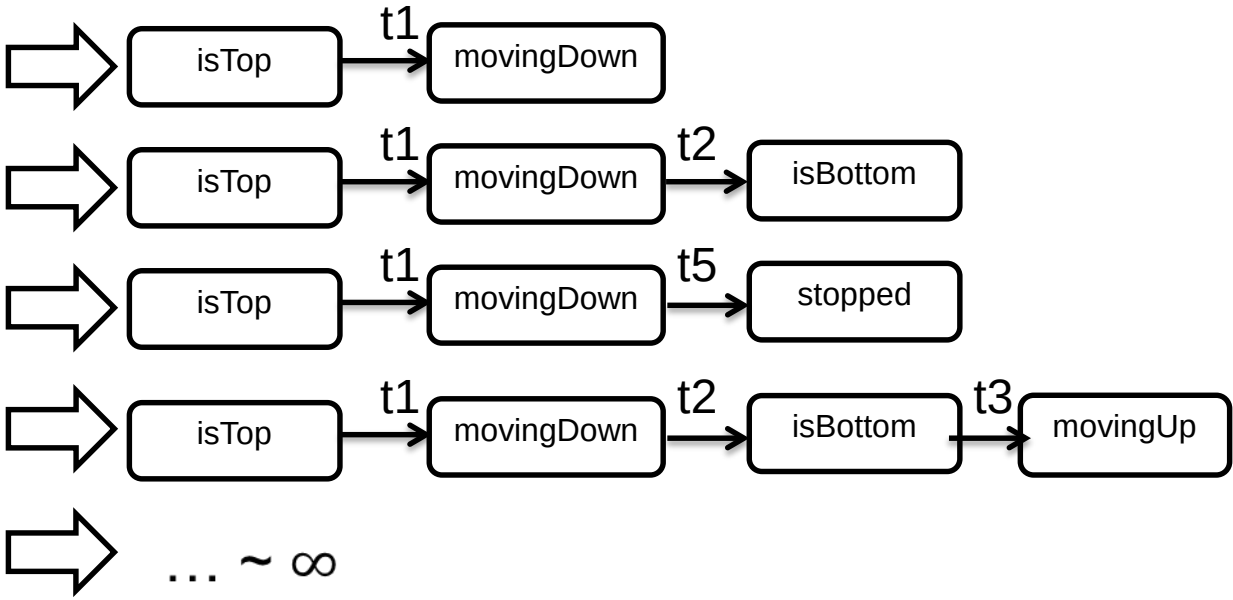
2. Testschritt: Transition t2

- Eingabesignal *!pw_isDown*
- Ausgabesignal *?pw_stop*



Test Suite Design



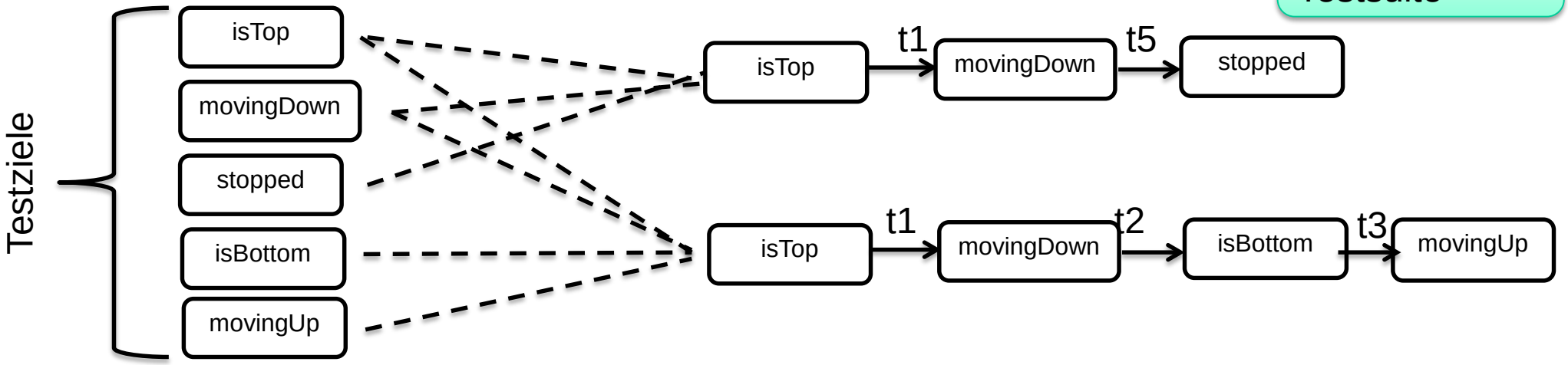


Abdeckungskriterium
(z.B. Zustandsabdeckung)

Testfallselektion

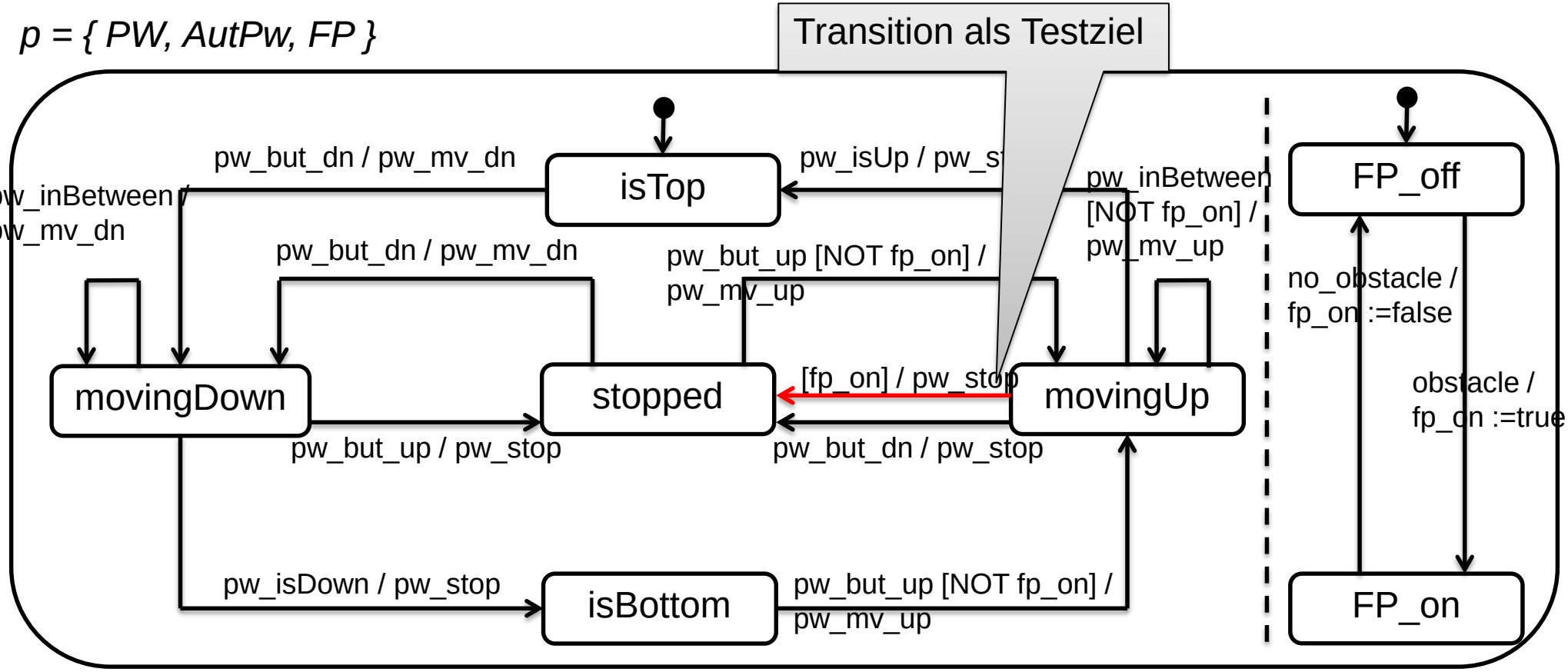
Vollständige Testsuite

Testziele

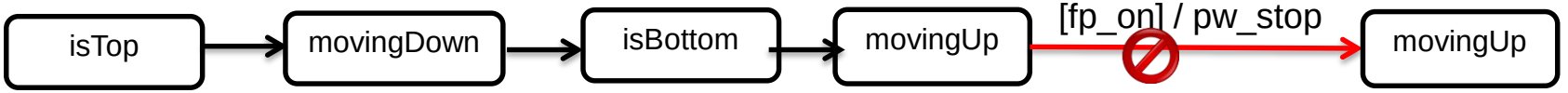


Ableitung valider Testfälle

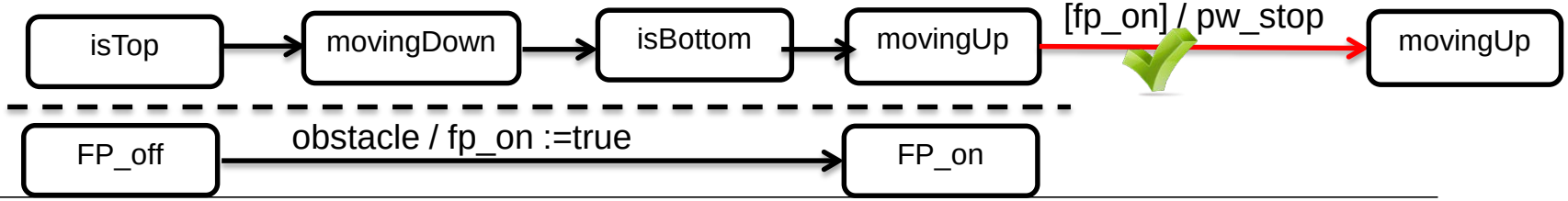
p = { PW, AutPw, FP }



- Invalider Pfad:



- Valider Pfad:



Test Suite Design - Ziele

Für das Design einer **adäquaten** Test Suite gibt es folgende Zielsetzungen

- **Soundness:** Die Test Suite enthält nur valide Testfälle
- **Completeness:** Die Test Suite enthält für jedes Testziel mindestens einen Testfall
- **Efficiency:** Die Test Suite ist möglichst minimal
- *Effectiveness: Die Test Suite deckt möglichst viele Fehler ab*

Effectiveness ist eher eine Eigenschaften der gewählten Testmethodik

- Ist das Testmodell valide, präzise und vollständig?
- Ist das Abdeckungskriterium ausreichend?
- Wie ist die statistische Fehlerverteilung in der Problemdomäne?

Effectiveness und Efficiency sind gegenläufige Optimierungsziele beim Test Suite Design

Soundness/Completeness: Automatisierte Test Suite Generierung

Eingabe:
- Testmodell *tm*
- Testzielmenge *G* in *tm*

Ausgabe:
- Vollständige Test Suite *TS* für *G*

Algorithmus:

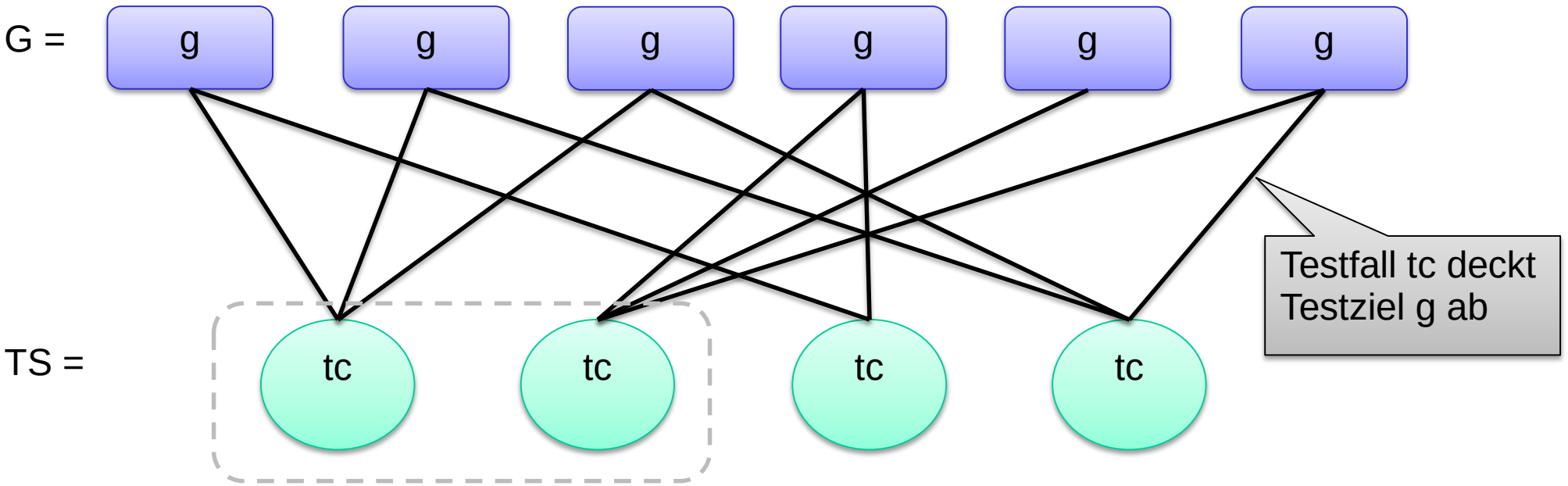
```
TS := { };
foreach g ∈ G
   tc := generateTestCase(tm,g);
   TS  := TS ∪ {tc};
   G  := G - { g∈G | g covered by tc }
endfor
```

Verwendung eines Model-Checkers:

$$tm \models AG(\neg g)$$

Falls *g* erreichbar über einen validen Pfade, liefert der Model-Checker ein Gegenbeispiel als Testfall *tc*

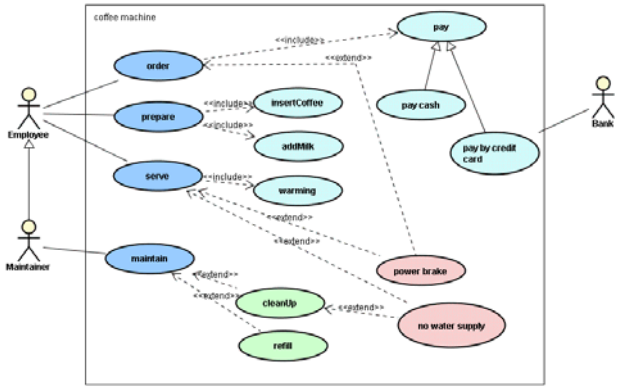
Efficiency: Test Suite Optimierung

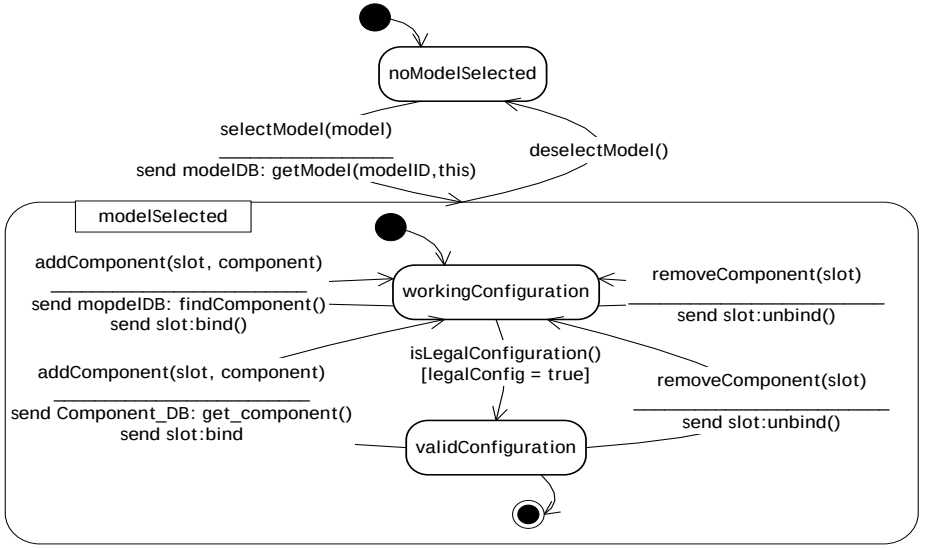


- Beliebige n:m-Beziehung zwischen Testzielen und Testfällen möglich.
- Minimierung einer Test Suite *TS* durch Selektion einer minimalen Teilmenge *TS'*, sodass jedes Testziel abgedeckt bleibt.
- NP-vollständig (Minimum Set Cover Problem)

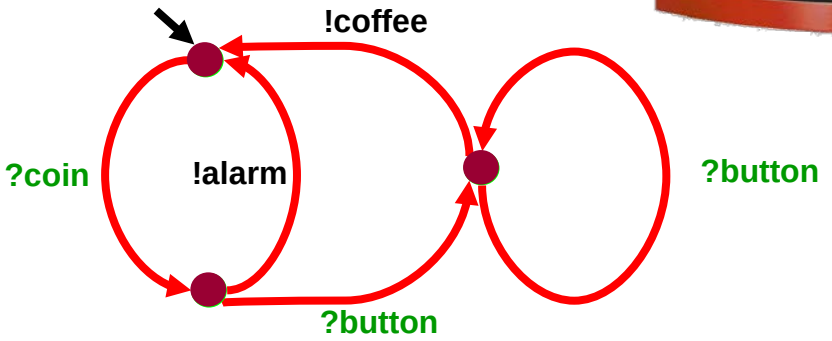
[Harrold et al., 1993]

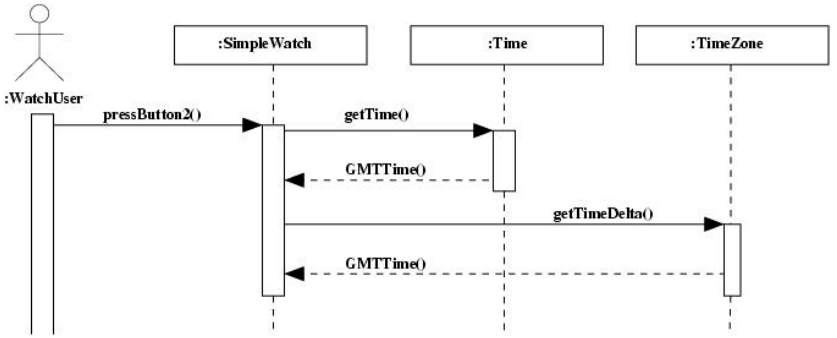
Testmodelle für verschiedene Testlevel



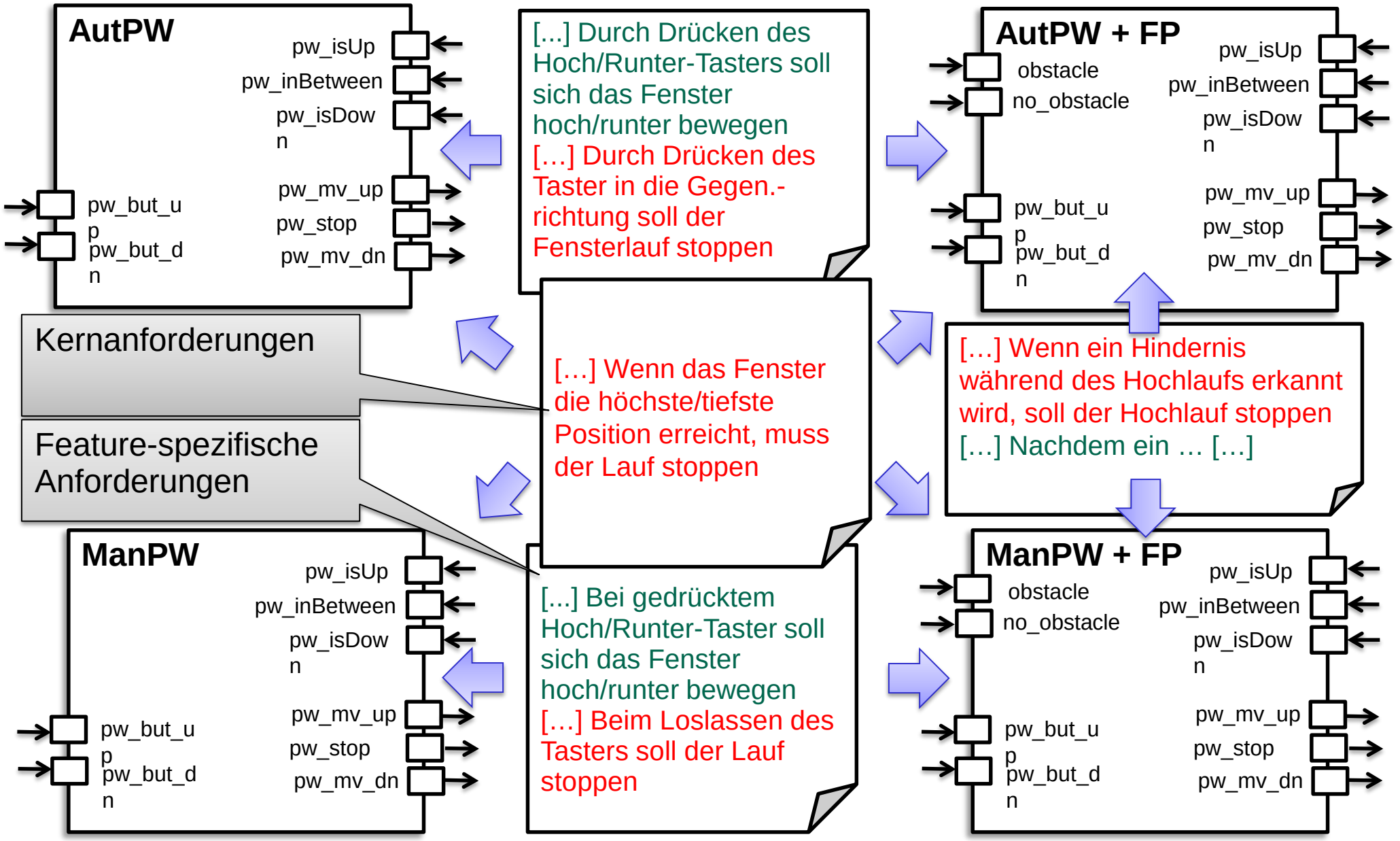


etc.





SPL Testing: Variable Spezifikationen



SPL Analyse

- **Kernanforderungen**: Anforderungen, die durch alle Produktvarianten der Produktlinie erfüllt werden müssen

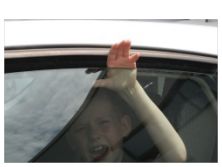
- **Feature-spezifische Anforderungen:** Anforderungen, die durch Produktvarianten erfüllt werden müssen, in denen ein bestimmtes Feature implementiert ist

- **Gewollte Feature-Interaktionen:** Anforderungen, die durch Produktvarianten erfüllt werden müssen, in denen eine bestimmte Feature-Kombination enthalten ist

➢ **Positive** Feature-orientierte SPL -Analyse: Prüfen, dass Konfigurations-spezifische Anforderungen in Produktvarianten korrekt implementiert sind.

➢ **Negative** Feature-orientierte SPL-Analyse: Prüfen, dass Produktvarianten keine Anforderungen implementieren, die nicht den Konfigurations-spezifischen Anforderungen entsprechen.

Feature-orientierte SPL-Analyse

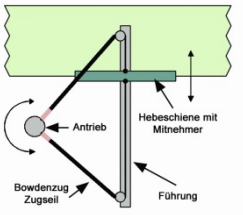

$\models \varphi$


$\models \varphi$


$\models \varphi$

$|F| * O(\varphi)$


$\models \varphi$



$\models \varphi$

Annahme:

1. Verifikation/Testen der einzelnen Feature-Artefakte möglich
2. Verifizierte Eigenschaften bleiben nach der Komposition erhalten

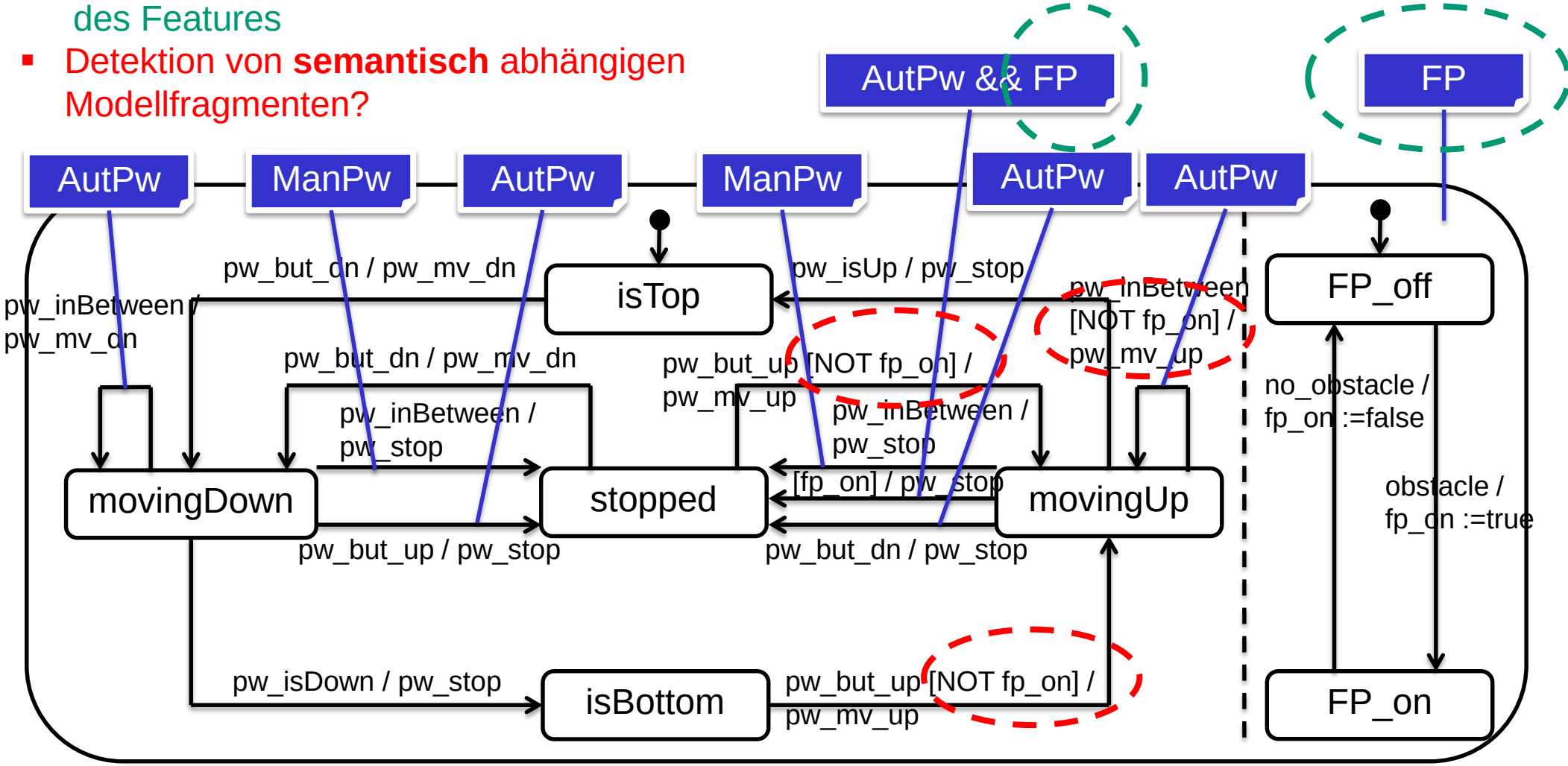
Aber: In der Praxis nicht möglich

- Feature-Artefakte sind meistens **nicht separierbar** vom Basisprodukt
- Aufgrund von Querschnittsbeziehungen und Interaktionen zwischen Features ist Feature-Verifikation **nicht kompositional**

Beispiel: Feature-based Analysis

Testen des Features „Finger Protection“

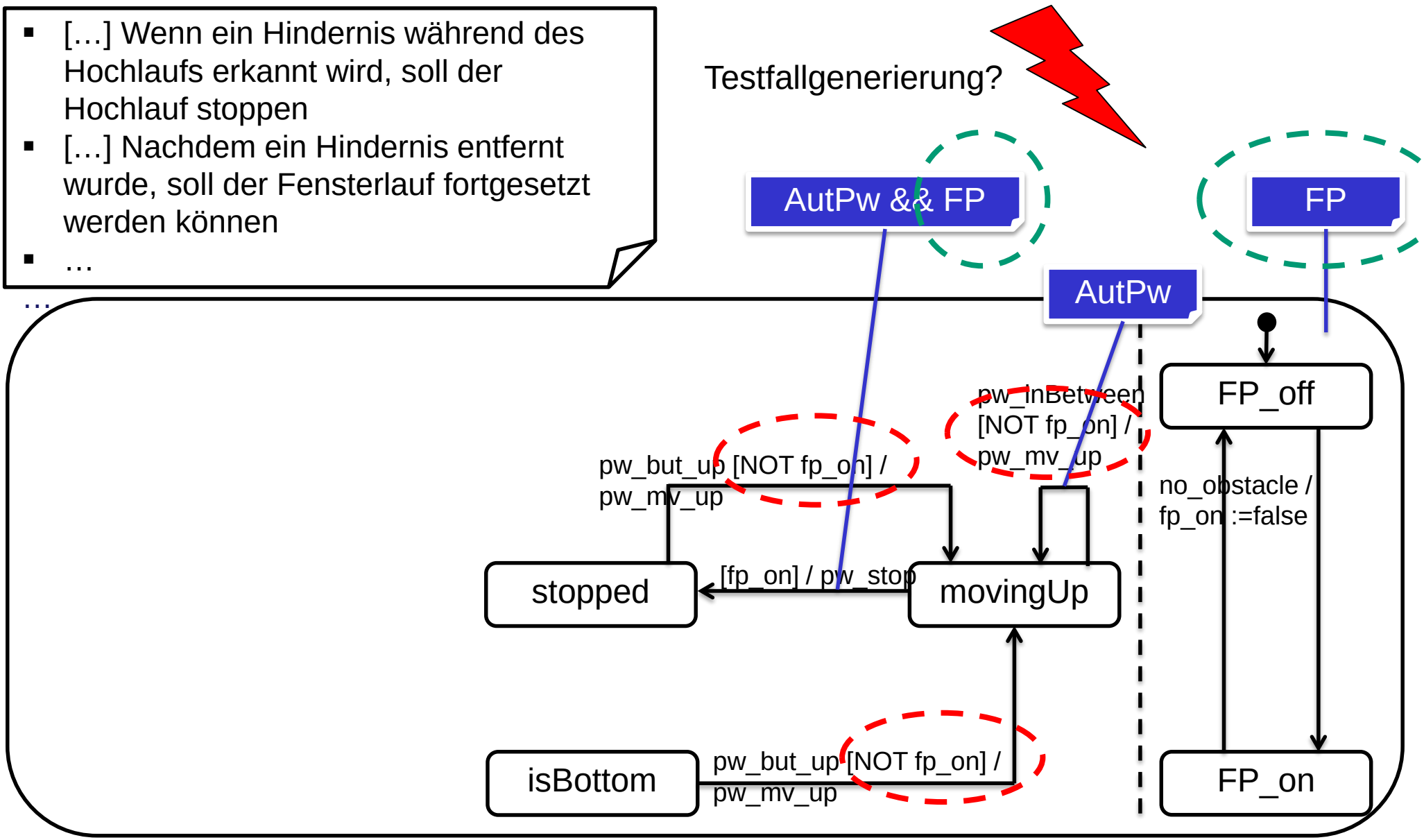
- Separierung der **syntaktisch** zum Features gehörenden Modellfragmente des Features
- Detektion von **semantisch** abhängigen Modellfragmenten?



Beispiel: Feature-based Analysis

- […] Wenn ein Hindernis während des Hochlaufs erkannt wird, soll der Hochlauf stoppen
- […] Nachdem ein Hindernis entfernt wurde, soll der Fensterlauf fortgesetzt werden können
- …

…



Produktlinien-Analyse

Annahmen

- Um die korrekte Implementierung von Features zu testen, müssen diese **in konkreten Produktvarianten** betrachtet werden (Products-Under-Test)
- Um sämtliche (gewollten oder ungewollten) Interaktionen mit anderen Features zu prüfen, müssen Features in **sämtlichen Kontexten** (Produktvarianten), in denen sie vorkommen, getestet werden

Product-by-Product SPL Analysis

 $\models \varphi$

 $\models \varphi$

 $\models \varphi$

 $\models \varphi$

$2^{|F|} * O(\varphi)$

 $\models \varphi$

 $\models \varphi$

 $\models \varphi$

 $\models \varphi$

Sample-based SPL Analysis

 $\models \varphi$

 $\models \varphi$

 $\models \varphi$

 $\models \varphi$

$2^{|F|} * O(\varphi)$

 $\models \varphi$

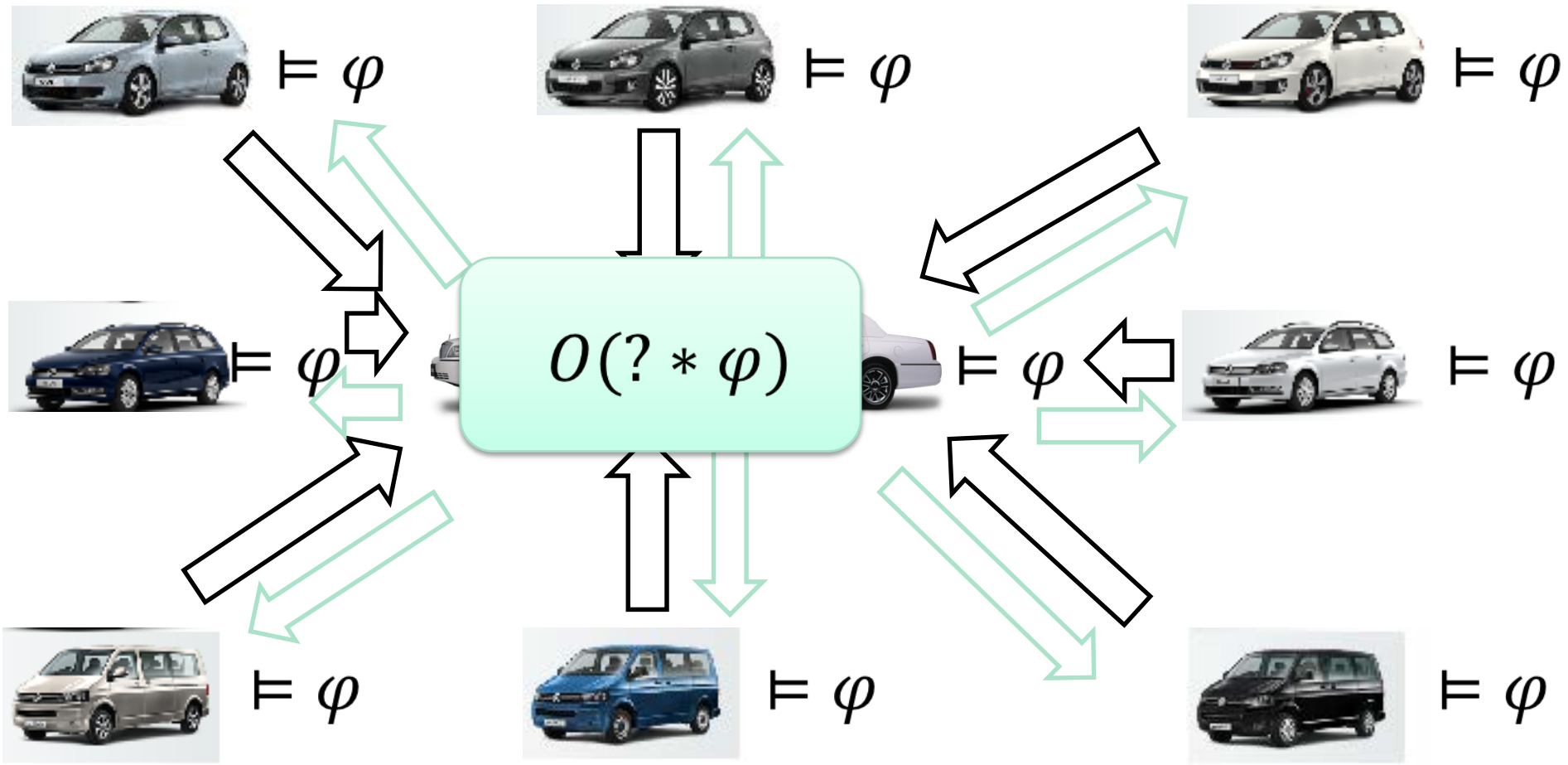
 $\models \varphi$

 $\models \varphi$

 $\models \varphi$

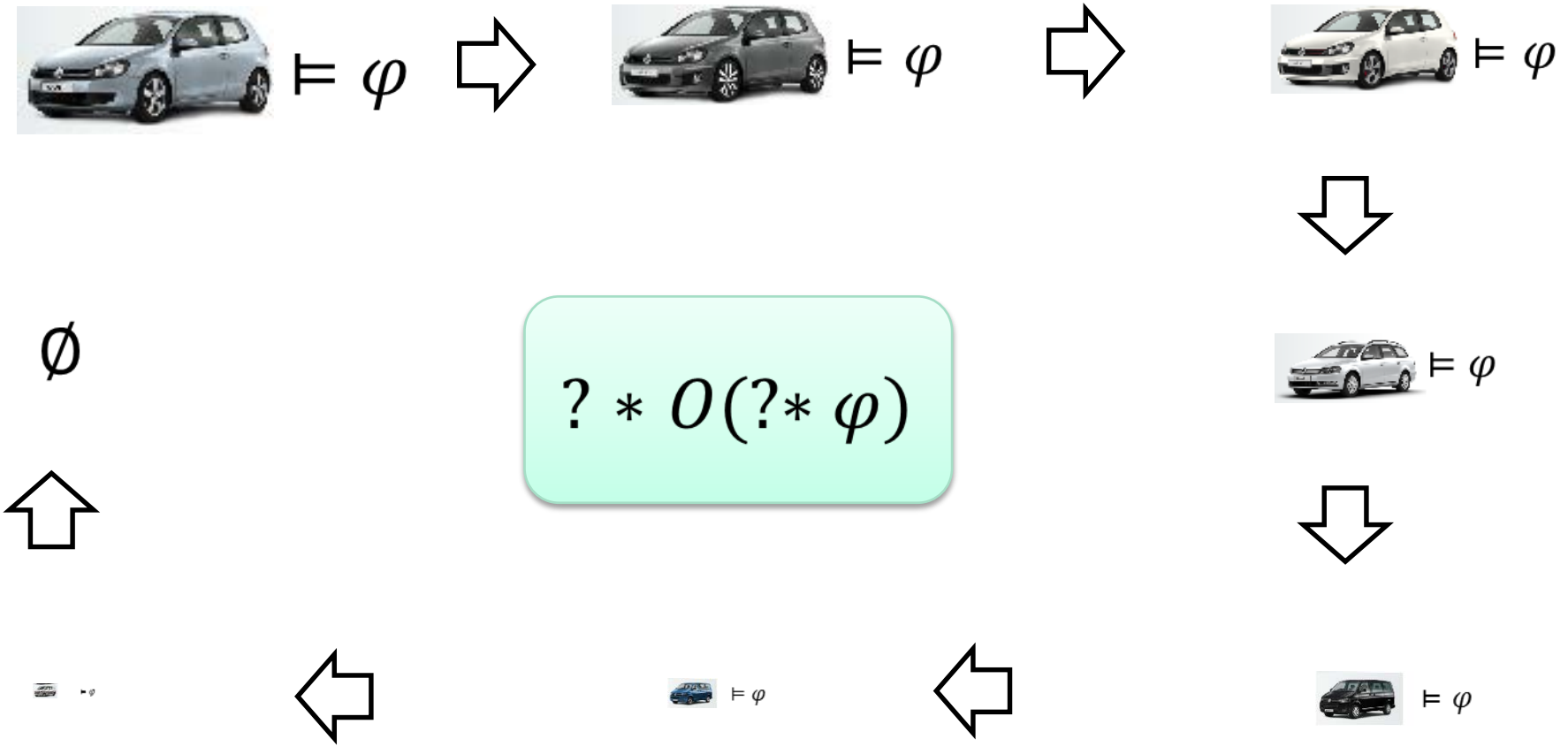
[Cohen et al. 2006], [Gustafsson 2007], [Oster et al., 2010], [Perrouin et al, 2010], [Kim et al. 2011], [Johansen et al., 2012], [Henard et al., 2013], [Haslinger et al., 2013], ….

Family-based SPL Analysis



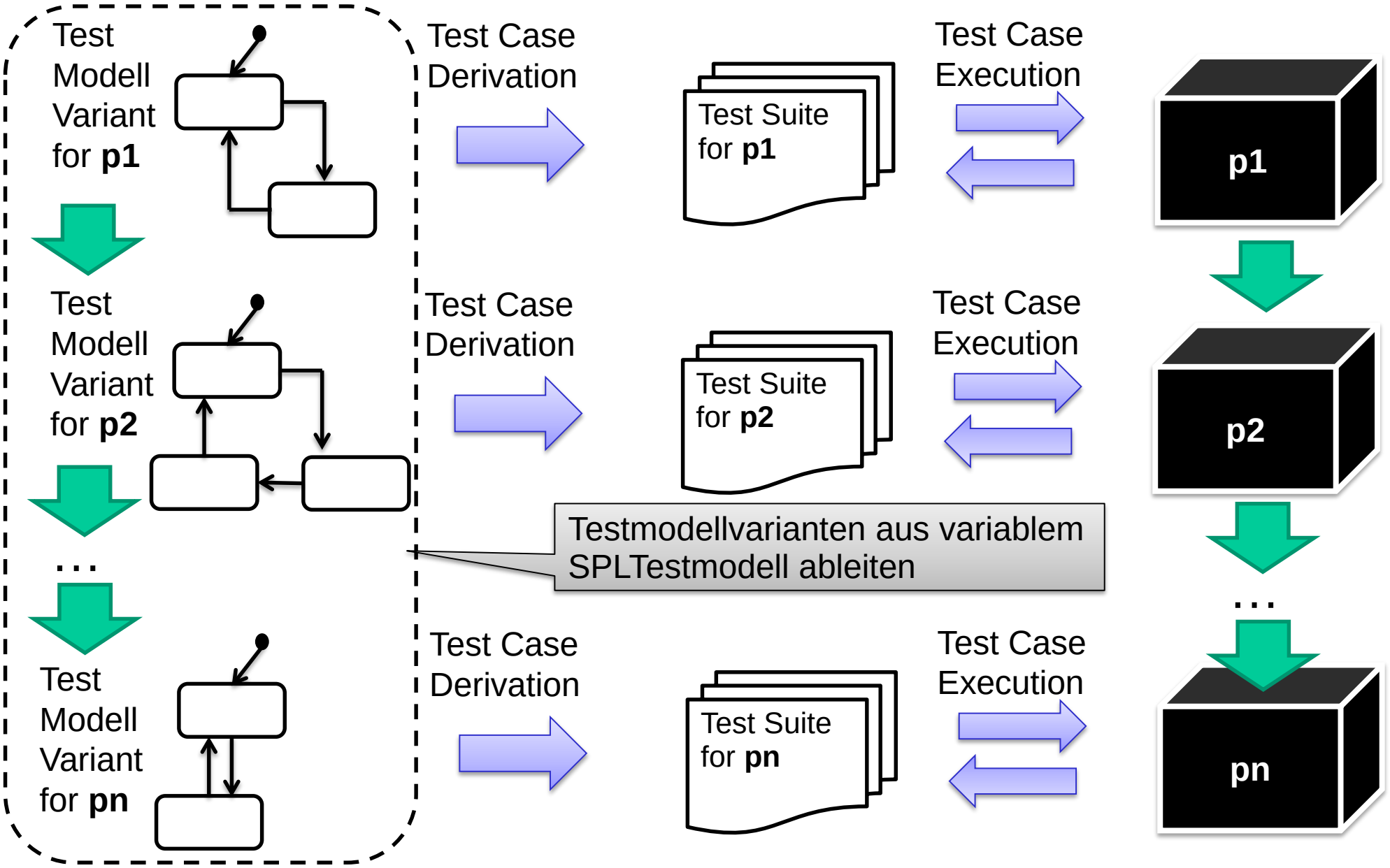
[Krishnamurti et al., 2006], [Gruler et al., 2010], [Classen et al. 2010], [Gnesi et al. 2011], [Cordy et al., 2013], …

Incremental SPL Analysis

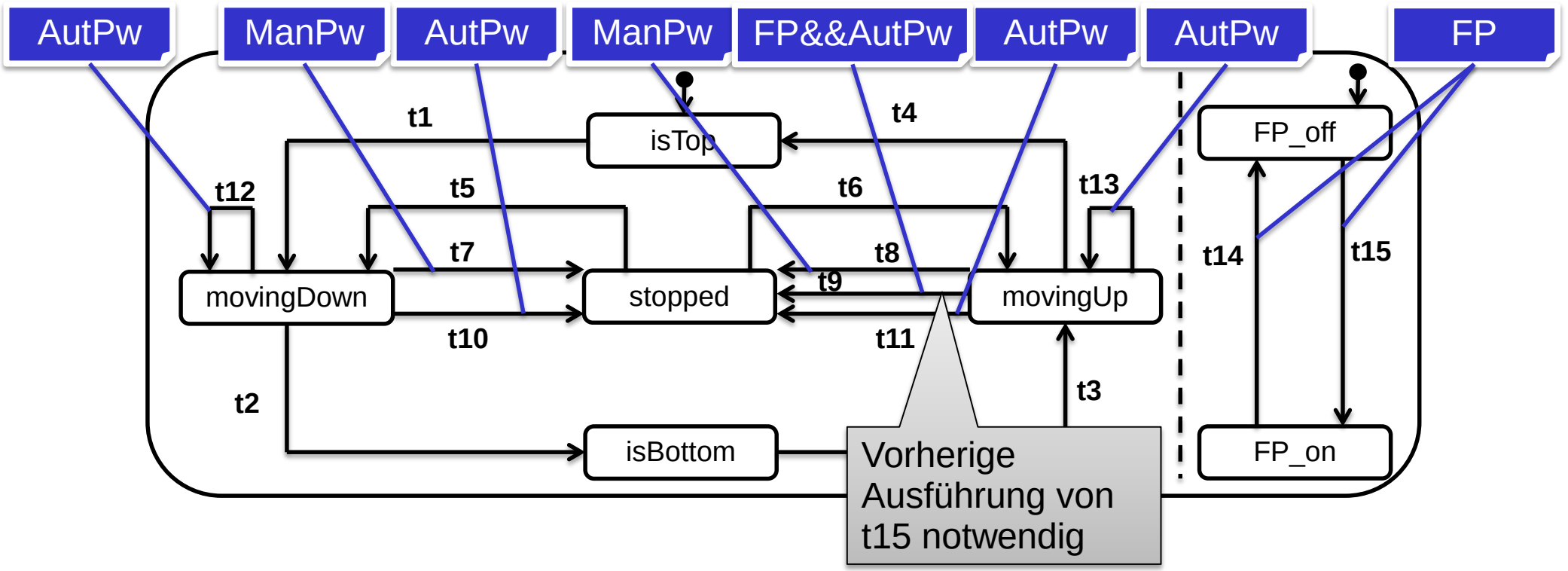


[Baller et al. 2014, Al-Hajjaji et al. 2015], …

Product-by-Product SPL Testing



Beispiel: BCS SPL Test Model



- Ableitung der Testmodellvarianten tm_p für Produktkonfigurationen p
  ⇒ Modellvarianten enthalten nur die produktspezifischen Testziele G_p

- Ableitung vollständiger Test Suiten TS_p für Produktkonfigurationen p aus den Testmodellvarianten tm_p
  ⇒ Jeder abgeleitete Testfall ist valide für die Produktvariante

Automatisierte Product-by-Product Test Suite Generierung

Eingabe:

- SPL Testmodell *stm*
- Testzielmenge *G* in *stm*
- Produktmenge *P*

Ausgabe:

- Vollständige Test Suiten TS_p für G_p für alle Produkte $p \in P$

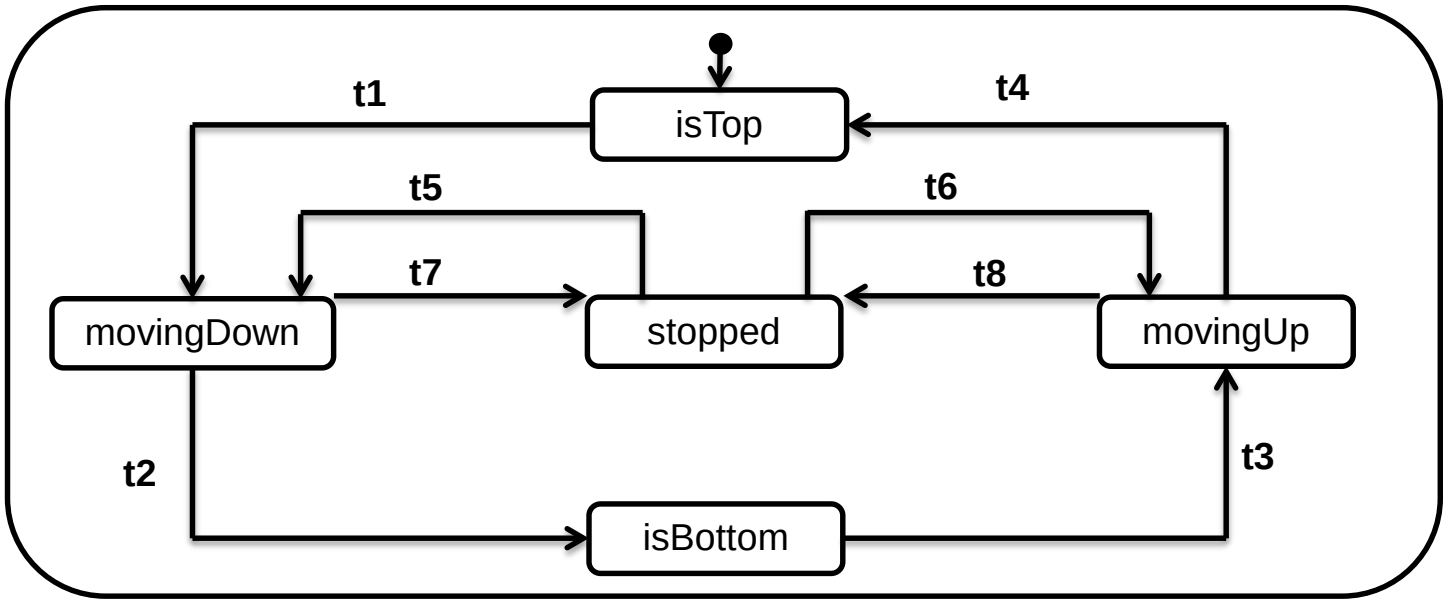
Algorithmus:

```
foreach p ∈ P
    TS_p := { };
    tm_p := stm(p);
    foreach g ∈ G_p
        tc := generateTestCase(tm_p,g);
        TS_p := TS ∪ {tc};
        G_p := G_p – { g ∈ G_p | g covered by tc }
    endfor
endfor
```

Ableiten der Testmodellvariante für Konfiguration *p* aus *stm*

Test Suite für BCS Varianten (1/4)

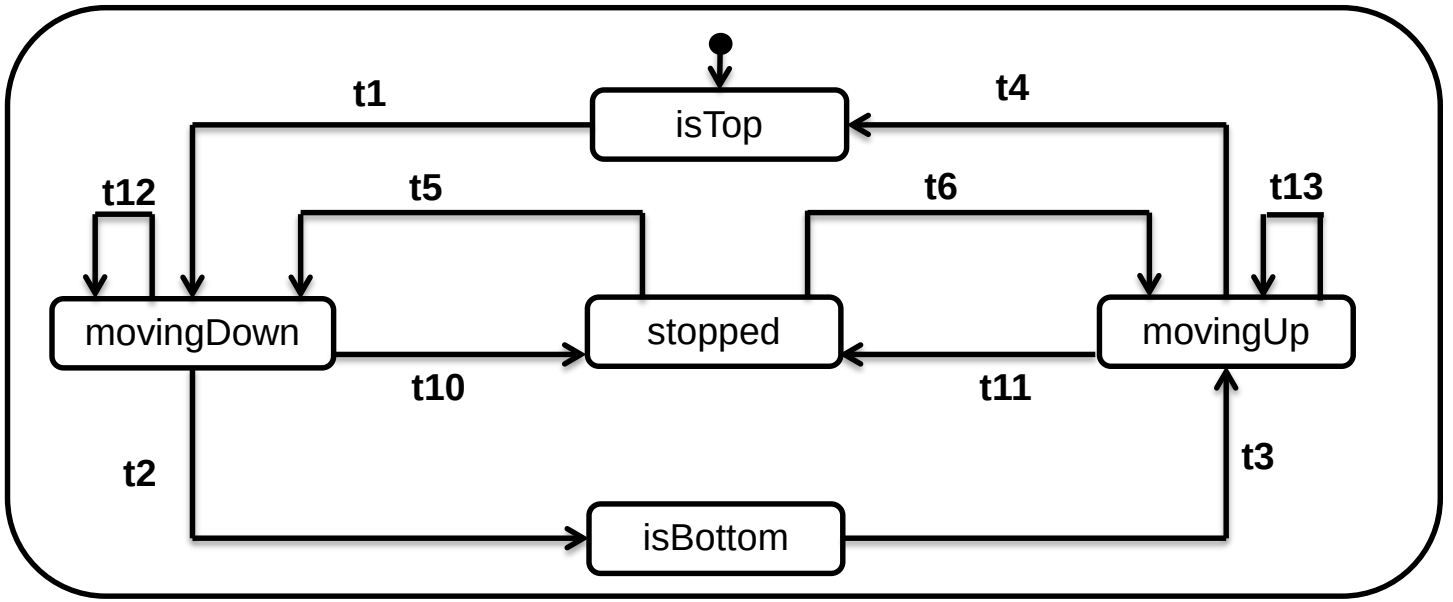
- Transitionsabdeckung für p1 = { PW, ManPw }
- G_{p1} = { t1, t2, t3, t4, t5, t6, t7, t8 }



Testziel	Testfall	Verbleibende Testziele
t4	tc1 = t1-t2-t3-t4	t5, t6 ,t7, t8
t5	tc2 = t1-t7-t5	t6,t8
t6	tc3 = t1-t2-t3-t8-t6	-

Test Suite für BCS Varianten (2/4)

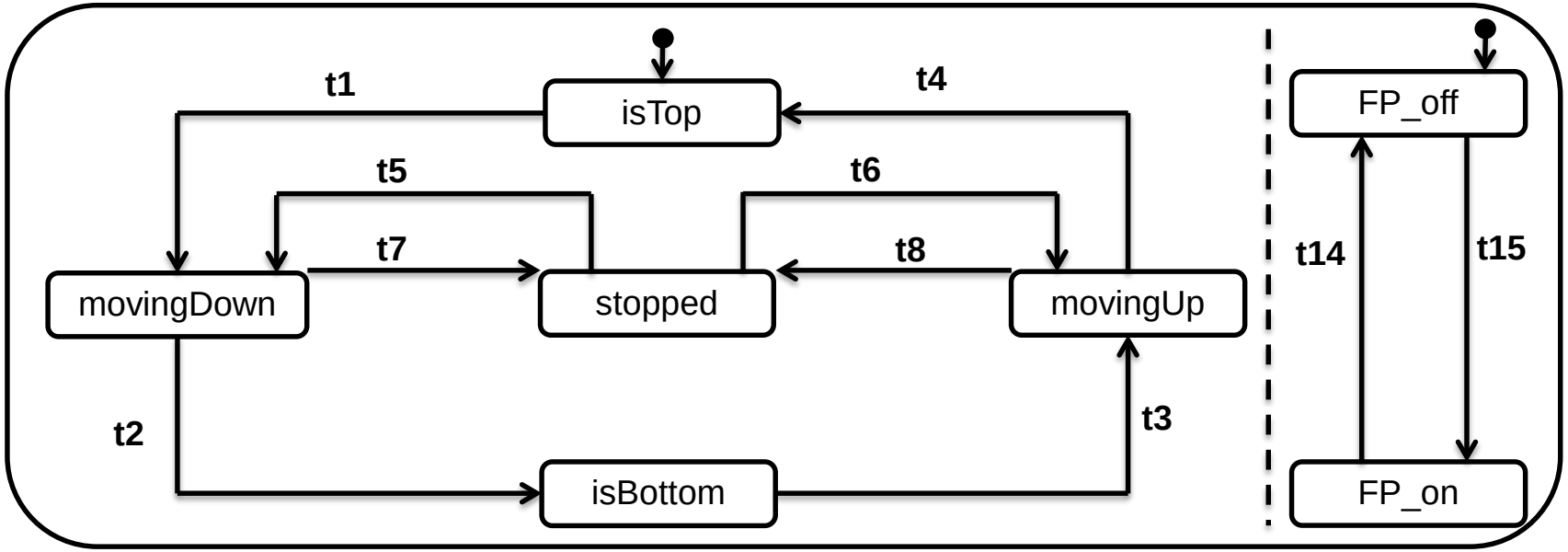
- Transitionsabdeckung für p2 = { PW, AutPw }
- G_{p1} = { t1, t2, t3, t4, t5, t6, t10, t11, t12, t13 }



Testziel	Testfall	Verbleibende Testziele
t4	tc1 = t1-t2-t3-t4	t5, t6, t10, t11, t12, t13
t5	tc3 = t1-t12-t10-t5	t6, t11, t13
t6	tc2 = t1-t2-t3-t13-t11-t6	-

Test Suite für BCS Varianten (3/4)

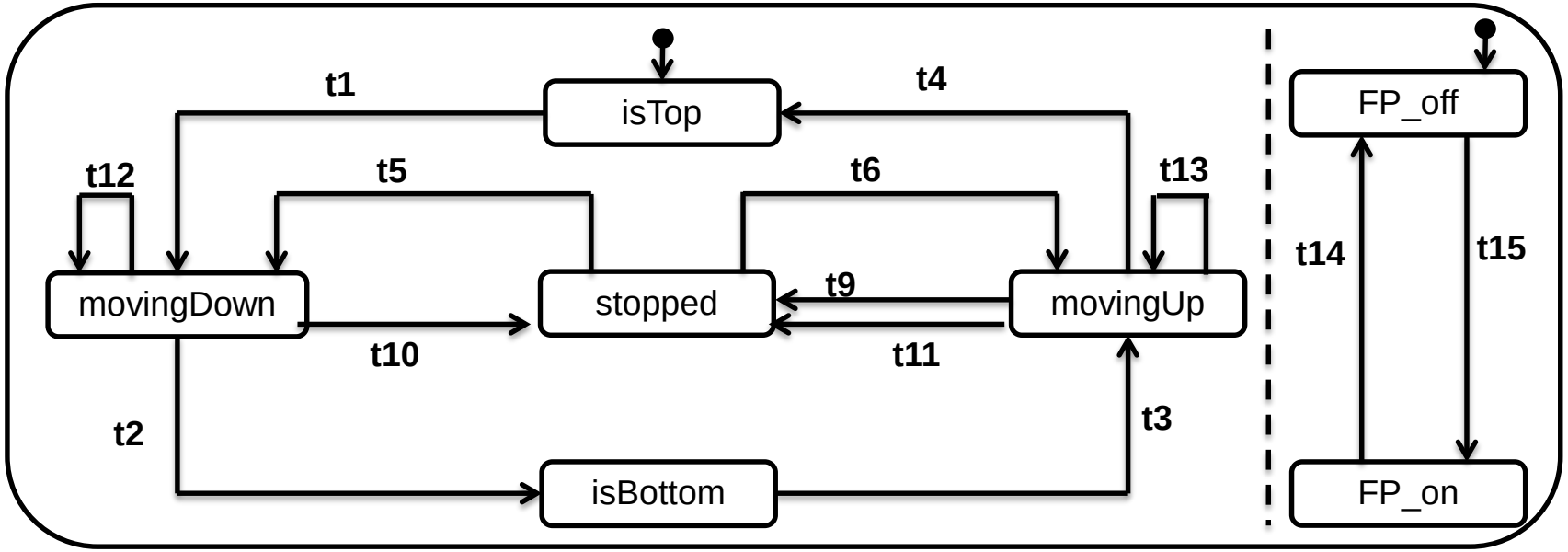
- Transitionsabdeckung für p3 = { PW, ManPw, FP }
- G_{p1} = { t1, t2, t3, t4, t5, t6, t7, t8, t14, t15 }



Testziel	Testfall	Verbleibende Testziele
t4	tc1 = t1-t2-t3-t4	t5, t6, t7, t8, t14, t15
t5	tc2 = t1-t7-t5	t6, t8, t14, t15
t6	tc3 = t1-t2-t3-t8-t6	t14, t15
t14	tc4 = t1-t2-t3-t15-t14	-

Test Suite für BCS Varianten (4/4)

- Transitionsabdeckung für p4 = { PW, AutPw, FP }
- G_{p1} = { t1, t2, t3, t4, t5, t6, t9, t10, t11, t12, t13 }



Testziel	Testfall	Verbleibende Testziele
t4	tc1 = t1-t2-t3-t4	t5, t6, t7, t8, t9, t14, t15
t5	tc2 = t1-t7-t5	t6, t8, t9, t14, t15
t6	tc3 = t1-t2-t3-t8-t6	t9, t14, t15
t9	tc4 = t1-t2-t3-t15-t9-t14	-

Product-by-Product SPL Testing: Diskussion

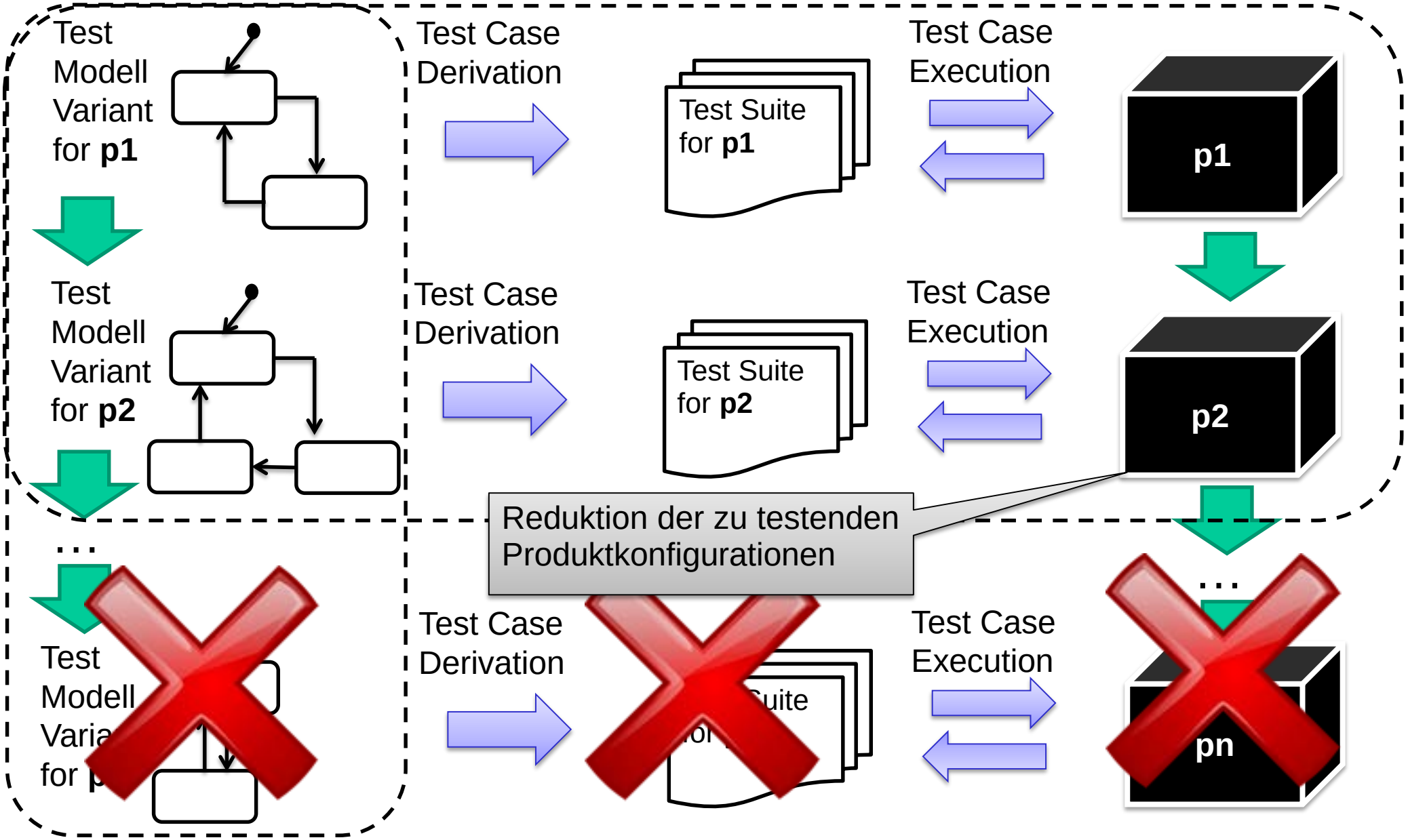
Vorteile:

- Einfache und garantiert vollständige Testabdeckung aller Produktvarianten
- Vorhandene Werkzeuge zur Testfallgenerierung können wiederverwendet werden

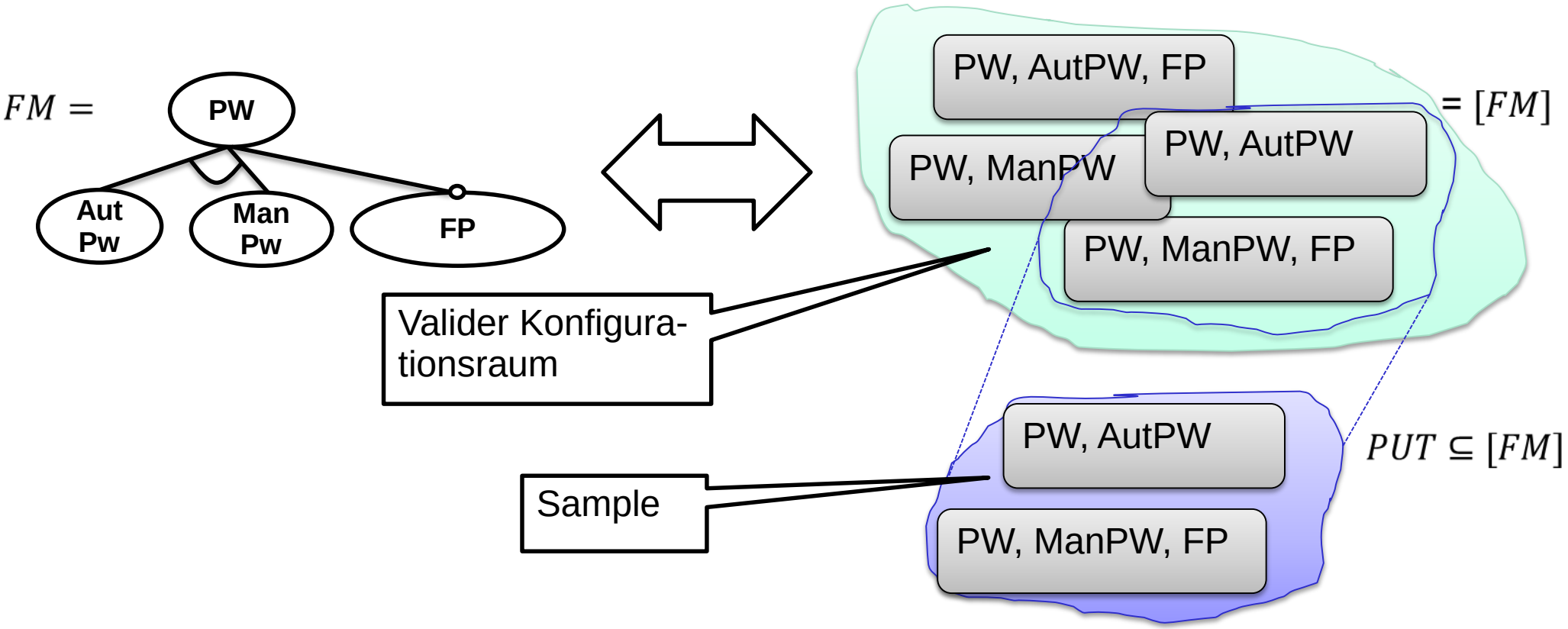
Nachteile:

- Hoher Aufwand bei Produktlinien mit hoher Anzahl Produktkonfigurationen ⇒ Sampling
- In der Regel viele redundante Generierungsschritte durch mehrfache Generierung gleicher Testfälle für verschiedene Varianten ⇒ Family-based / Incremental SPL Testing

Product Sampling

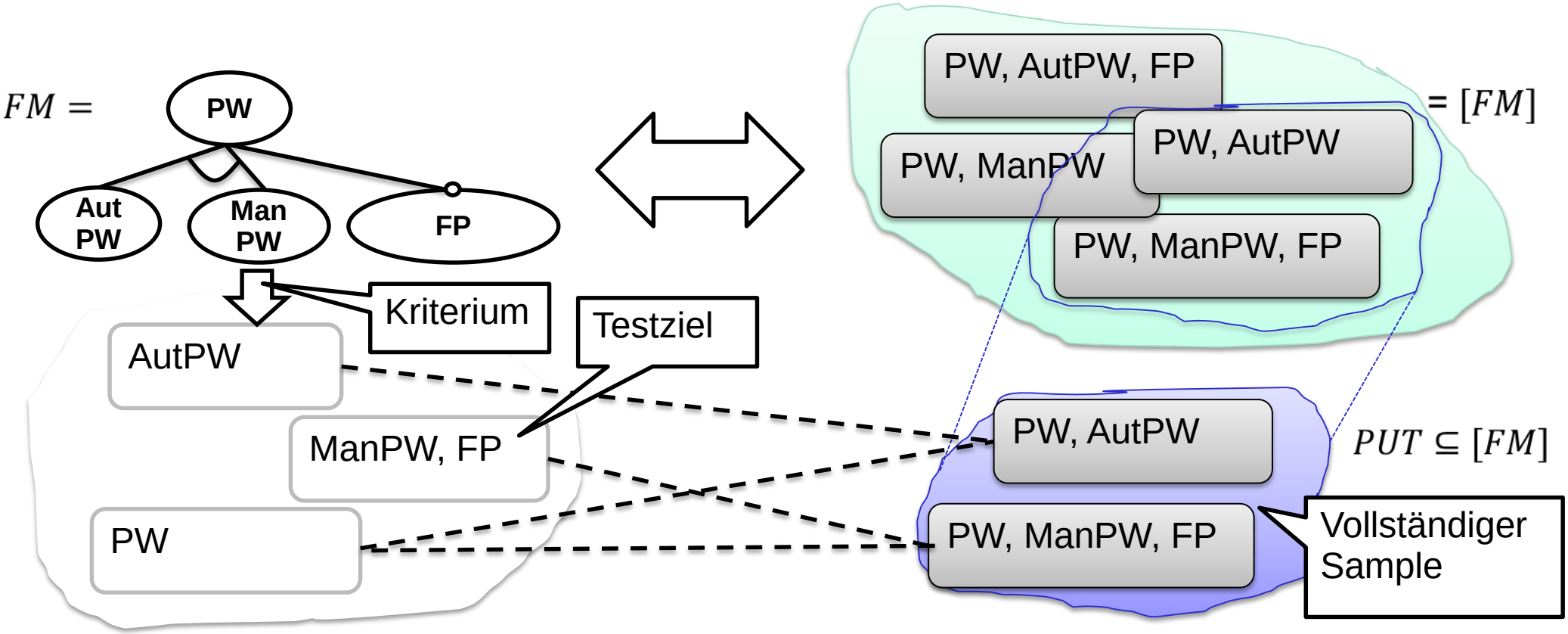


Sample-based SPL Testing



- Selektion einer **repräsentativen Teilmenge (Sample)** von Products-under-Test (*PUT*) aus dem validen Konfigurationsraum
- Product-by-Product Testing der Teilmenge
- Definition geeigneter **Selektionskriterien?**

Feature-Orientierte Sample-Selektion



- Definition von Feature-Orientierten *Abdeckungskriterien* zur automatischen Extraktion von Testzielen auf Basis des Feature-Modells
- **Soundness:** Die Testziele sollen *erfüllbar* sein
- **Efficiency:** Das Kriterium soll die Produktmenge *deutlich reduzieren* und die Berechnung der Menge der Testziele sollte *möglichst effizient* sein.
- **Effectiveness:** Die Artefakte im Lösungsraum sollten *hinreichend abgedeckt* sein

Feature-Orientiertes Abdeckungskriterium

Abdeckungskriterium CC auf Feature-Modellen FM über Feature-Mengen F:

$$CC(FM) \subseteq \wp^{+}(F), \text{ s.t. } \forall F' \in CC(FM): \exists p \in [FM]: F' \subseteq p$$

Satz (Soundness): Sei CC ein Abdeckungskriterium auf Feature-Modellen FM. Dann gilt:

$$\exists PUT \subseteq [FM]: \forall F' \in CC(FM): \exists p \in PUT: F' \subseteq p$$

Sample

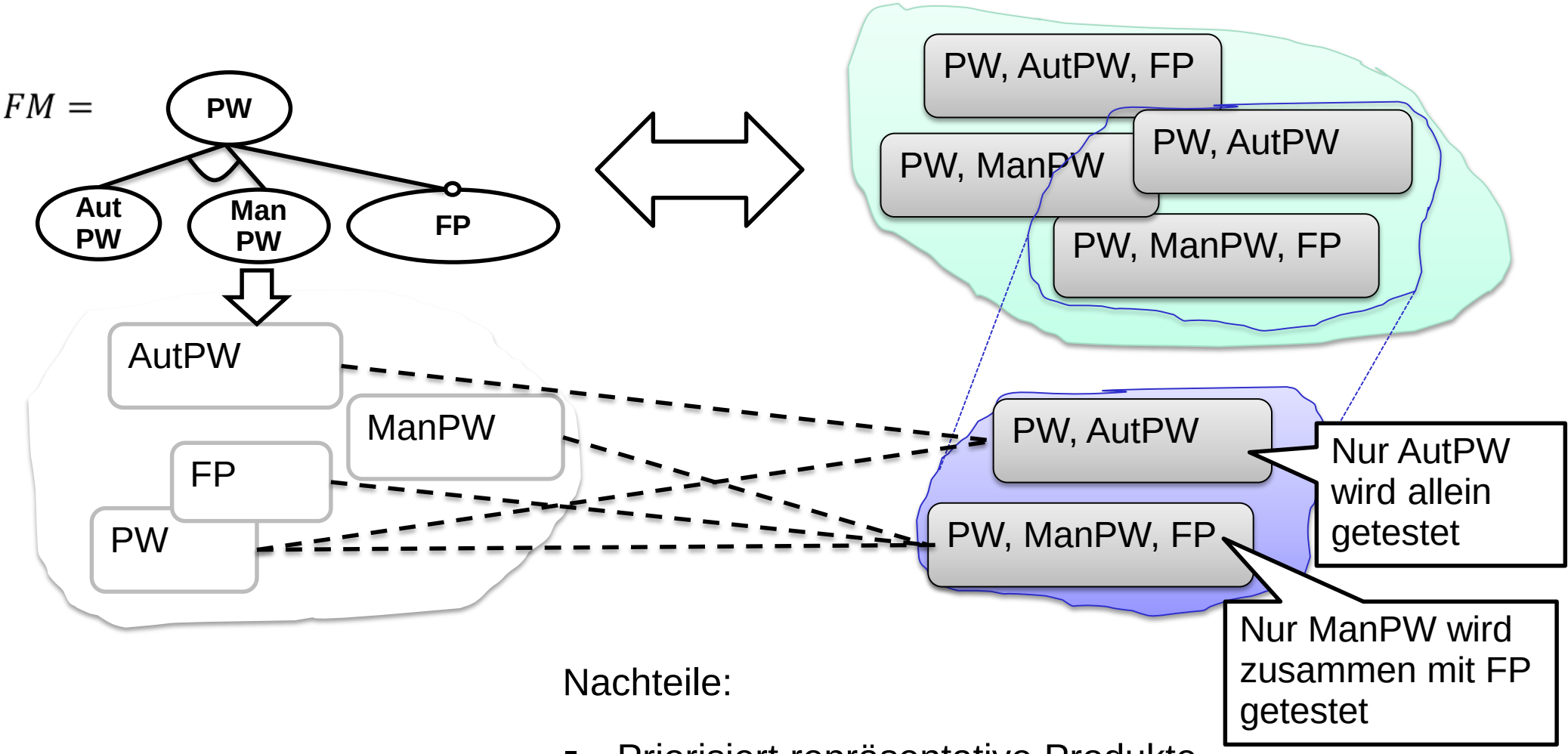
Ein (vollständiger) **Sample** $PUT \subseteq [FM]$ für ein Abdeckungskriterium CC:

$$\forall F' \in CC(FM): \exists p \in PUT: F' \subseteq p$$

Ein Sample $PUT \subseteq [FM]$ für ein Abdeckungskriterium CC ist **minimal**, wenn für alle weiteren vollständigen Samples $PUT' \subseteq [FM]$ gilt:

$$|PUT| < |PUT'|$$

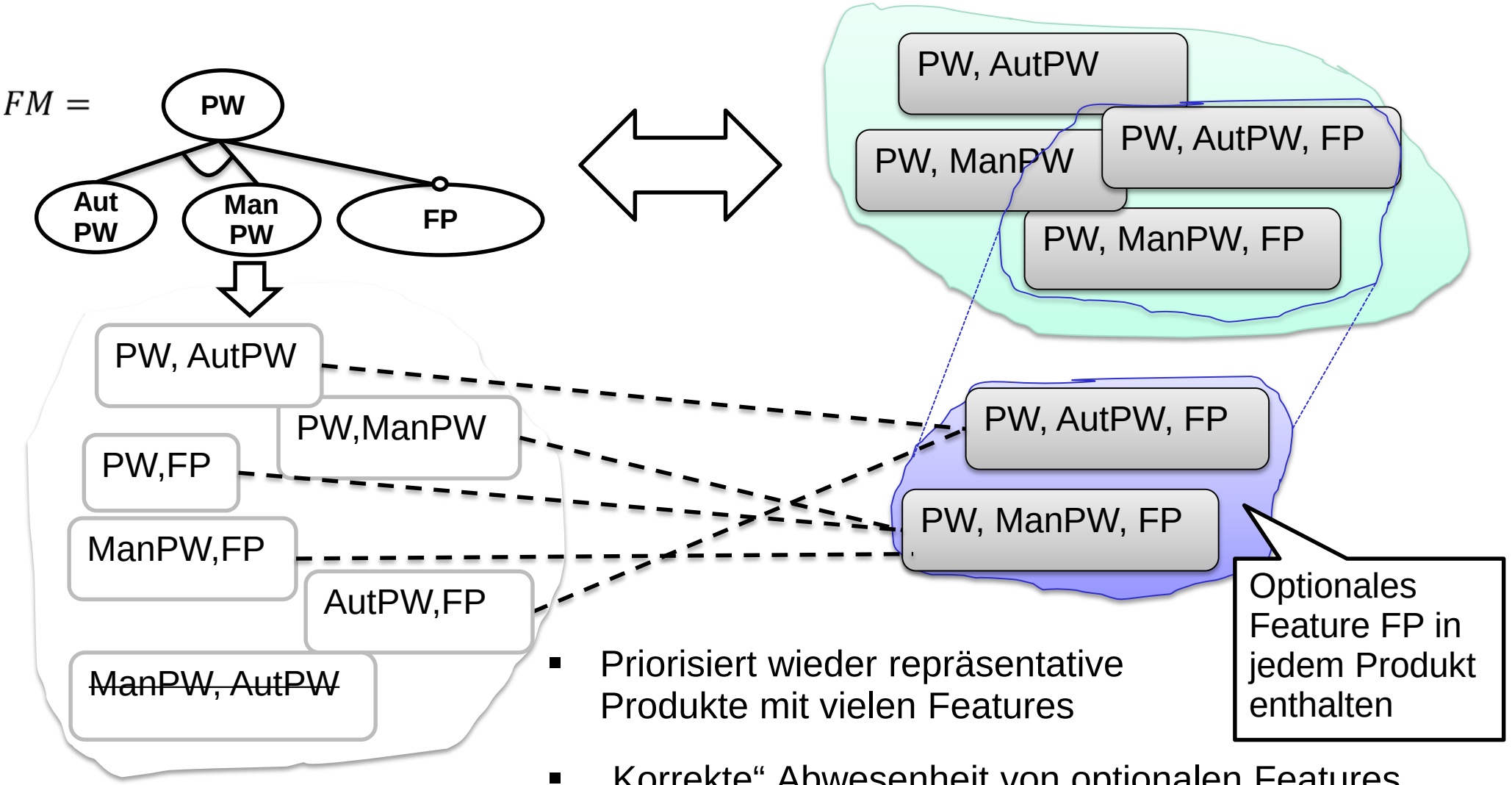
Beispiel: Feature-Coverage



Nachteile:

- Priorisiert repräsentative Produkte mit vielen Features
- Optionale Features werden nur in wenigen Kombinationen getestet

Beispiel: Feature-Pair-Coverage



- Priorisiert wieder repräsentative Produkte mit vielen Features
- „Korrekte“ Abwesenheit von optionalen Features wird evtl. nicht getestet
- Interaktionen zwischen mehr als zwei Features?

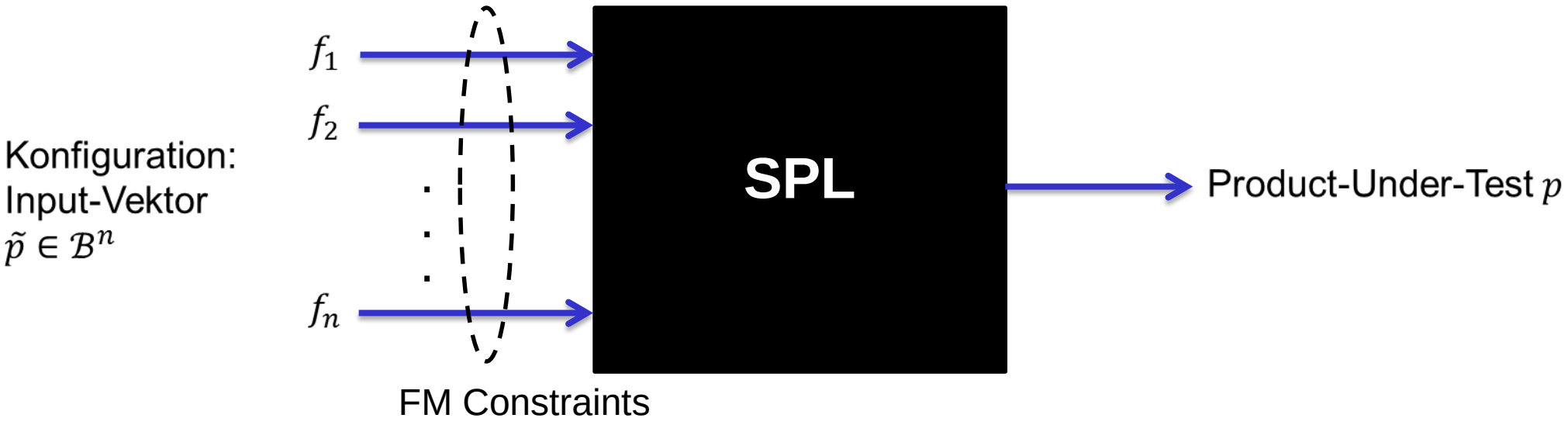
Verbesserte Sample-Kriterien

Ziel: Abdeckung aller T-wise Feature-Interaktionen

1. Für jede detektierte Feature-Interkation in Feature-Menge $F' \subseteq F$ mit $1 \leq |F'| \leq T \leq |F|$ füge **Feature-Menge** $\mathrm{F}' \in CC(FM)$ hinzu
   - Positiver Test: Die korrekte Implementierung jeder gewollten Feature-Interaktion kann in einem repräsentativen Produkt getestet werden
   - Detektion aller Interkationen a apriori möglich?
   - Wahl von T?

2. Für jede T-wise Feature-Menge $F' \subseteq F, |F'| = T \leq |F|$,verlange Abdeckung jeder **Feature-Kombination** durch PUT
   - Negativer Test: Die Abwesenheit ungewollten Feature-Interaktion kann in einem repräsentativen Produkt getestet werden

Kombinatorische Feature-Abdeckung



- Übertragung von Prinzipien des (Constraint) Black-Box Combinatorial Testing auf Sample-based SPL Testing
- Kombinatorische Feature-Abdeckung: T-wise Feature Combinations

T-wise Combinatorial Feature-Sets

Ein ***T*-wise kombinatorisches Testziel** über einer Feature-Menge $F' \subseteq F, |F'| = \mathrm{T}$ ist ein Vektor $v \in \mathcal{B}^T$

Wir nehmen an, dass jedes Feature $f \in F'$ stets eine feste Position in jedem Vektor $v \in \mathcal{B}^T$ über F' hat und verwenden folgende Notationen:

- $v(f) \in \{false, true\}$ für die Belegung von f in v
- $v = (f, \neg f', \ldots)$falls $v(f) = true$, $v(f') = false$, etc.

Ein ***T*-wise kombinatorisches Testziel** $v \in \mathcal{B}^T$ über einer Feature-Menge $F' \subseteq F, |F'| = \mathrm{T}$ ist **valide** für ein Feature-Modell FM über Feature-Menge F, falls

$$\exists p \in [FM]: \forall f \in F': (v(f) = true \Rightarrow f \in p) \wedge (v(f) = false \Rightarrow f \notin p)$$

T-wise Combinatioral Feature-Coverage

T-wise Combinatorial Feature-Coverage $TCC(FM)$ auf Feature-Modellen FM über Feature-Menge F:

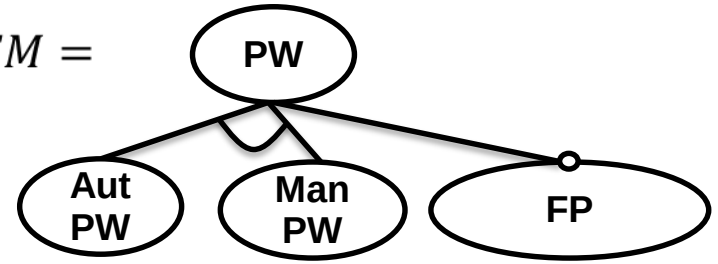
$$TCC(FM) = \{v_1\,,\ v_2\,, \ldots\}$$

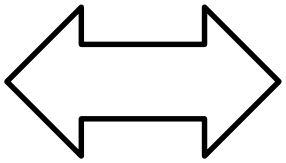
sodass für alle Teilmengen $F' \subseteq F, |F'| = \mathrm{T}$ und jedes valide Testziel $v \in \mathcal{B}^T$ über Teilmenge F' gilt, dass $v \in TCC(FM)$.

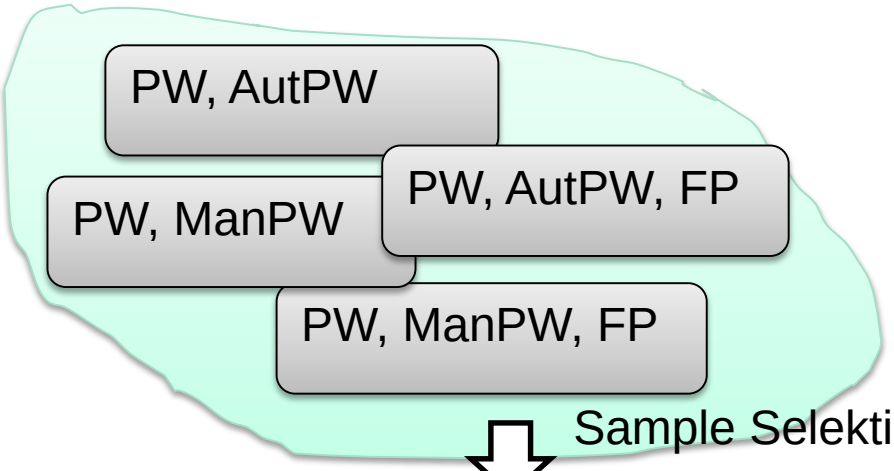
- Vollständiger und minimaler **Sample** $PUT \subseteq [FM]$ für Abdeckungskriterium $\mathrm{T}CC$ analog definiert.

Beispiel: Pairwise Combinatorial Feature Coverage

$FM =$







valide kombinatorische Paare

Sample Selektion

PW	ManPW	
~~0~~	~~0~~	
~~0~~	~~1~~	
1	0	(v1)
1	1	(v2)

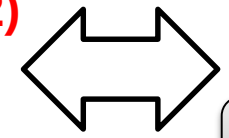
PW	FP	
~~0~~	~~0~~	
~~0~~	~~1~~	
1	0	(v5)
1	1	(v6)

FP	ManPW	
0	0	(v9)
0	1	(v10)
1	0	(v11)
1	1	(v12)

PW	AutPW	
~~0~~	~~0~~	
~~0~~	~~1~~	
1	0	(v3)
1	1	(v4)

ManPW	AutPW	
~~0~~	~~0~~	
0	1	(v7)
1	0	(v8)
~~1~~	~~1~~	

FP	AutPW	
0	0	(v13)
0	1	(v14)
1	0	(v15)
1	1	(v16)



Abgedeckte Paare:

PW, AutPW — **v1,v4,v5,v7,v9,v14**

PW, ManPW — **v2,v3,v5,v8,v10,v13**

PW, AutPW, FP — **v1,v4,v6,v7,v11,v16**

PW, ManPW, FP — **v2,v3,v6,v8,v12,v15**

Sample muss alle Konfigurationen enthalten => Efficiency?

Sample Selektion: Implementierung

Pairwise Combinatorial Feature-Coverage

- **Eingabe:** Feature-Modell FM über Feature-Menge F
- **Ausgabe:** PUT

1. Ermitteln alle validen kombinatorischen Feature-Paare für jedes Paar $f, f' \in F$

f	f‘
0	0
0	1
1	0
1	1

$(\neg f \wedge \neg f') \wedge \widetilde{FM} \not\vdash false$

$(\neg f \wedge f') \wedge \widetilde{FM} \not\vdash false$

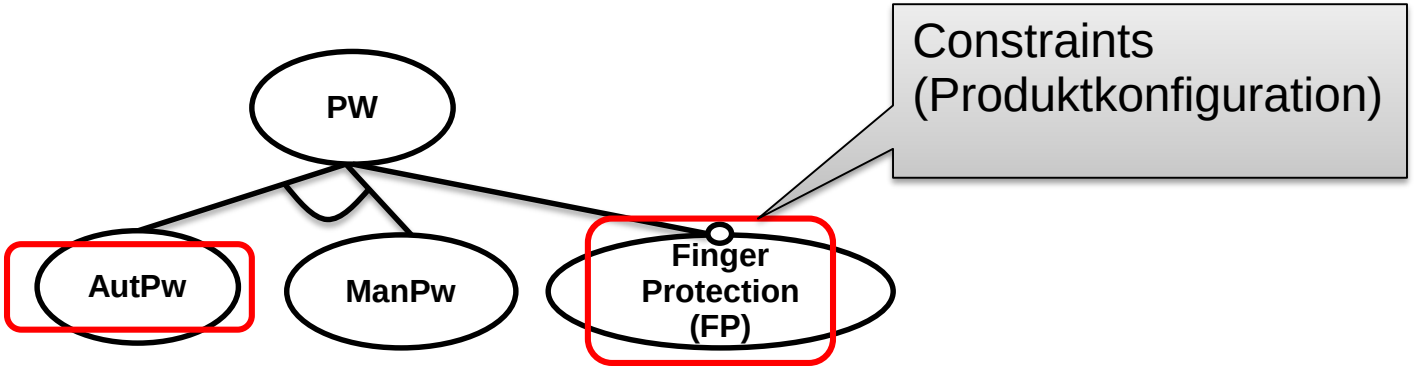
$(f \wedge \neg f') \wedge \widetilde{FM} \not\vdash false$

$(f \wedge f') \wedge \widetilde{FM} \not\vdash false$

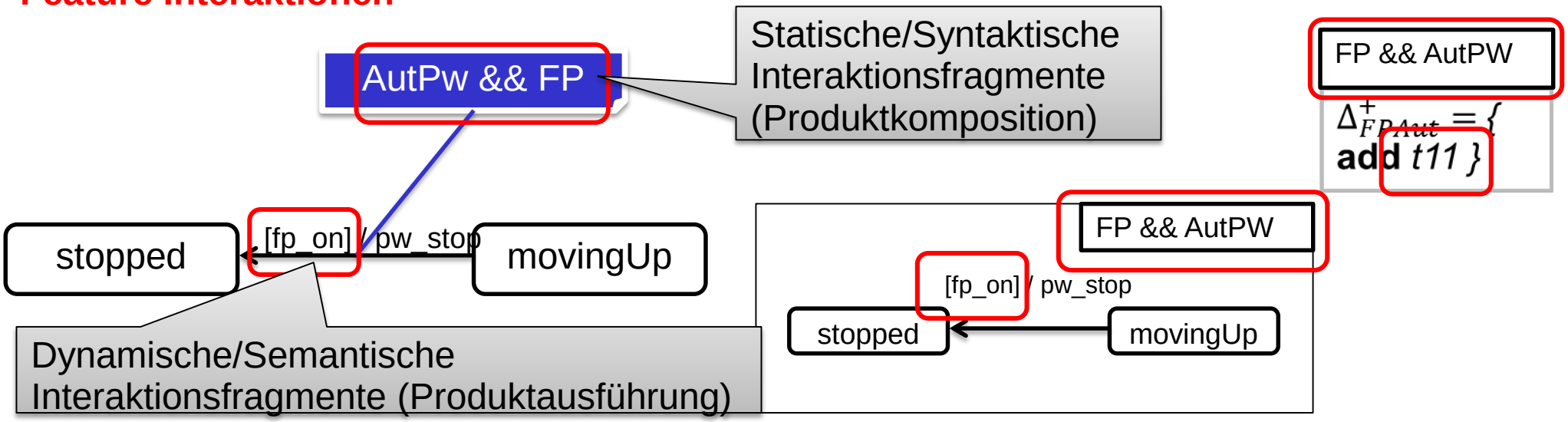
2. Starte mit $PUT = \{\}$ und füge solange weitere Konfigurationen p hinzu, bis alle validen Paare abgedeckt sind (=> Optimierungen?).

Beispiel: BCS

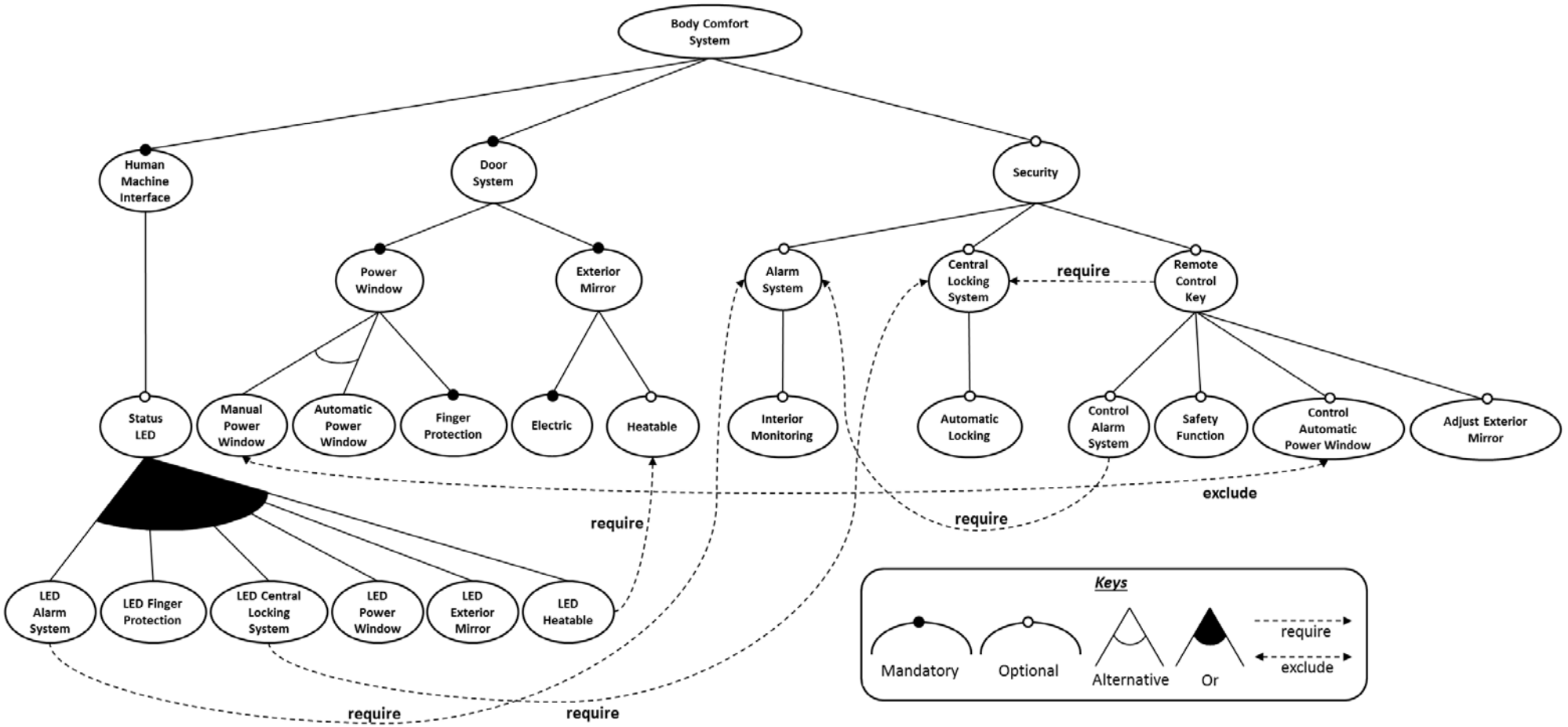
Problem Space:
Feature Abhängigkeiten



Solution Space:
Feature Interaktionen

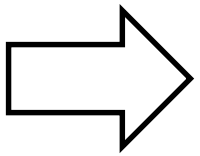


Beispiel: Pairwise BCS Coverage



- 27 Features
- 11.767 valide Konfigurationen

Beispiel: Pairwise BCS Coverage



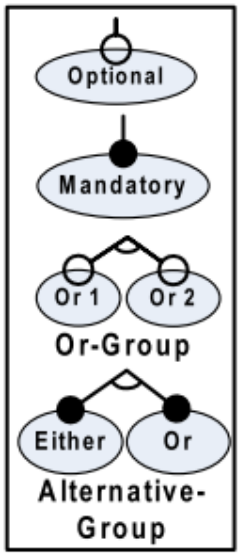
Pairwise Sample

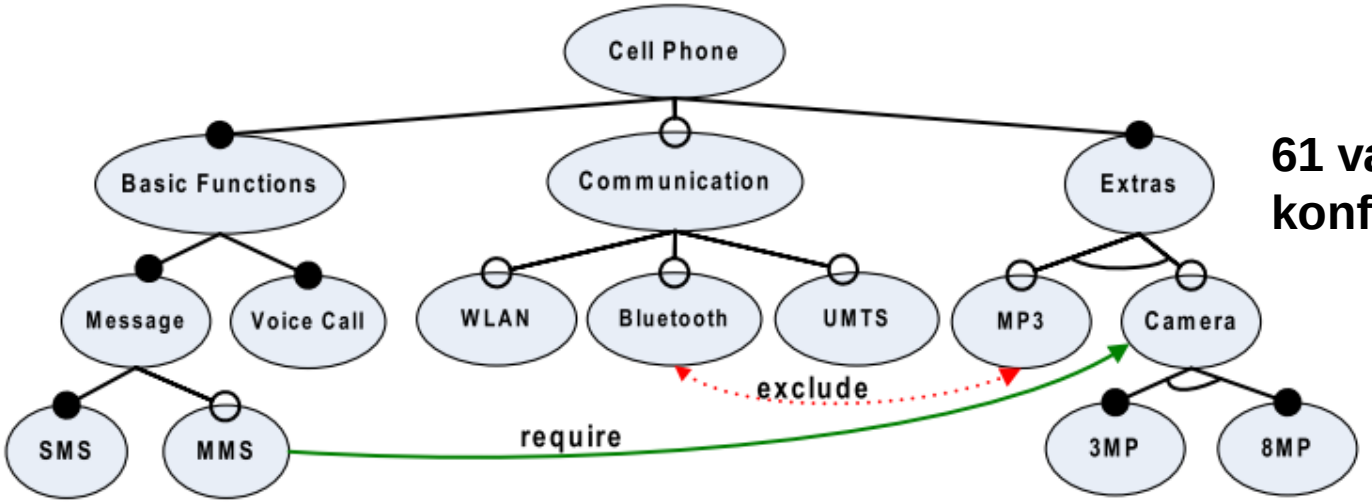
	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17
Security		x	x	x	x	x	x			x		x	x	x	x	x	x	
LED		x		x		x	x		x	x	x	x		x			x	
LED CLS				x			x			x		x		x			x	
LED PW				x		x	x		x	x		x						
LED Heat				x		x	x		x	x				x			x	
LED EM				x		x	x		x	x	x						x	
LED AS				x		x	x			x		x					x	
LED FP		x				x	x		x	x	x	x		x			x	
ManPW	x			x		x	x				x	x		x			x	x
AutoPW		x	x		x			x	x	x			x		x	x		
Heatable		x		x		x	x		x	x			x	x	x	x	x	x
CLS		x	x	x	x		x			x		x	x	x	x	x	x	
RCK		x	x	x	x					x		x	x	x	x	x	x	
AS		x	x	x	x	x	x			x		x			x	x	x	
AL			x		x		x					x	x	x	x	x	x	
CAS		x	x	x						x		x			x	x	x	
SF			x		x					x		x	x	x	x		x	
CAutoPW			x		x					x			x		x	x		
IM		x			x	x	x			x		x			x	x	x	
#Features	1	10	9	13	9	11	14	1	7	17	4	14	8	11	11	10	16	2

	k=1	k=2	k=3	k>3	∑
#syntakt. FI	12	37	17	-	66
#covered FI	12	37	14	-	63
%covered FI	100	100	82.5	-	95.5

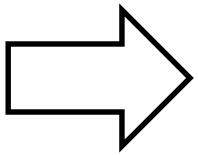
	1<k<8
#sem. FI	26
#covered	10
%covered	38.5

Beispiel: Cell Phone SPL





61 valide Produkt-konfigurationen



Pairwise Sample

	B,M,S,V	Com	Extras	MMS	WLAN	BT	UMTS	MP3	Camera	ALF_F
P 1	B,M,S,V	Com	Extras	MMS	WLAN	BT	UMTS	¬MP3	Camera	8MP
P 2	B,M,S,V	¬Com	Extras	¬MMS	¬WLAN	¬BT	¬UMTS	MP3	Camera	8MP
P 3	B,M,S,V	Com	Extras	¬MMS	¬WLAN	BT	¬UMTS	¬MP3	Camera	3MP
P 4	B,M,S,V	¬Com	Extras	MMS	¬WLAN	¬BT	¬UMTS	¬MP3	Camera	3MP
P 5	B,M,S,V	Com	Extras	¬MMS	WLAN	¬BT	UMTS	MP3	Camera	3MP
P 6	B,M,S,V	Com	Extras	¬MMS	¬WLAN	¬BT	UMTS	MP3	¬Camera	¬Camera
P 7	B,M,S,V	¬Com	Extras	¬MMS	¬WLAN	¬BT	¬UMTS	MP3	¬Camera	¬Camera
P 8	B,M,S,V	Com	Extras	MMS	WLAN	¬BT	¬UMTS	MP3	Camera	8MP
P 9	B,M,S,V	Com	Extras	¬MMS	WLAN	¬BT	¬UMTS	MP3	¬Camera	¬Camera

Efficiency

- Samples für Pairwise Combinatorial Coverage ermöglichen starke Reduktion insbesondere großer valider Konfigurationsräume.

Feature Model	Features	Possible Products	Pairwise Products	Runtime [ms]
Smart Home	35	1,048,576	11	0
Inventory	37	2,028,096	12	16
Sienna	35	2,520	24	16
Web Portal	38	2,120,800	26	15
Doc_Generation	44	$5.57 \cdot 10^{7}$	18	0
Arcade Game	61	$3.3 \cdot 10^{9}$	25	32
Model_Transformation	88	$1.65 \cdot 10^{13}$	40	46
Coche ecologico	94	$2.32 \cdot 10^{7}$	114	47
Electronic Shopping	287	$2.26 \cdot 10^{49}$	62	797

Aber:

- Die Selektion **valider Paare** erfordert SAT-Solving (NP-complete)
- Die Selektion eines **mininalen** Samples ist NP-hard

Effectiveness

Satz: Sei PUT ein Sample, der $TCC(FM)$ erfüllt. Dann erfüllt PUT auch $T'CC(FM)$ mit $T' < T$.

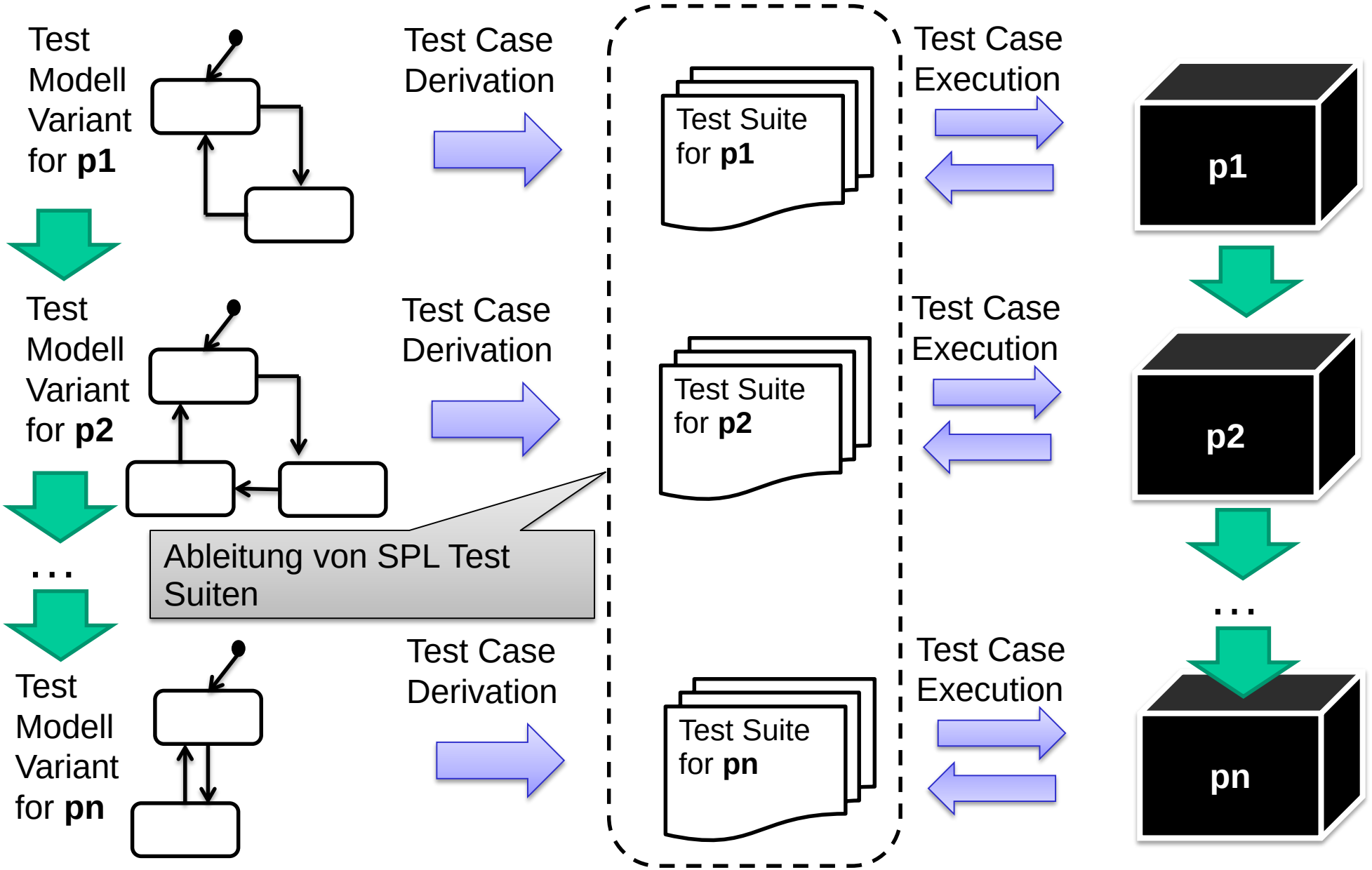
- Anmerkung: Für $T'CC(\mathrm{FM})$ ist PUT allerdings im Allgemeinen kein **minimaler** Sample, auch wenn er es für $TCC(FM)$ ist.

Zur Effectiveness von Pairwise Combinatioral Testing:

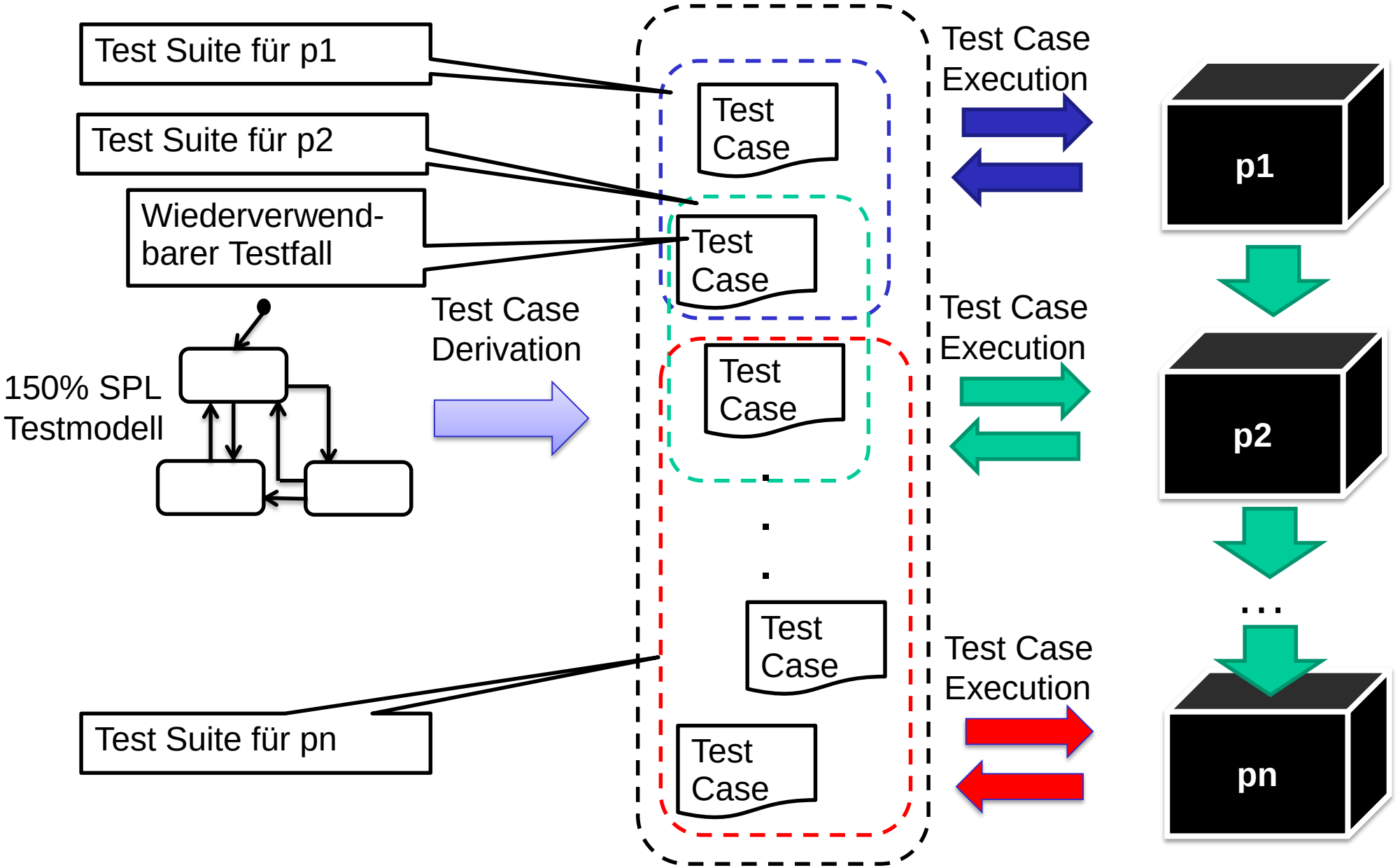
[…] The majority of faults originate from a single parameter value [feature] or are caused by the interaction of two parameter values [features].

[Stevens and Mendelsohn, 1998]

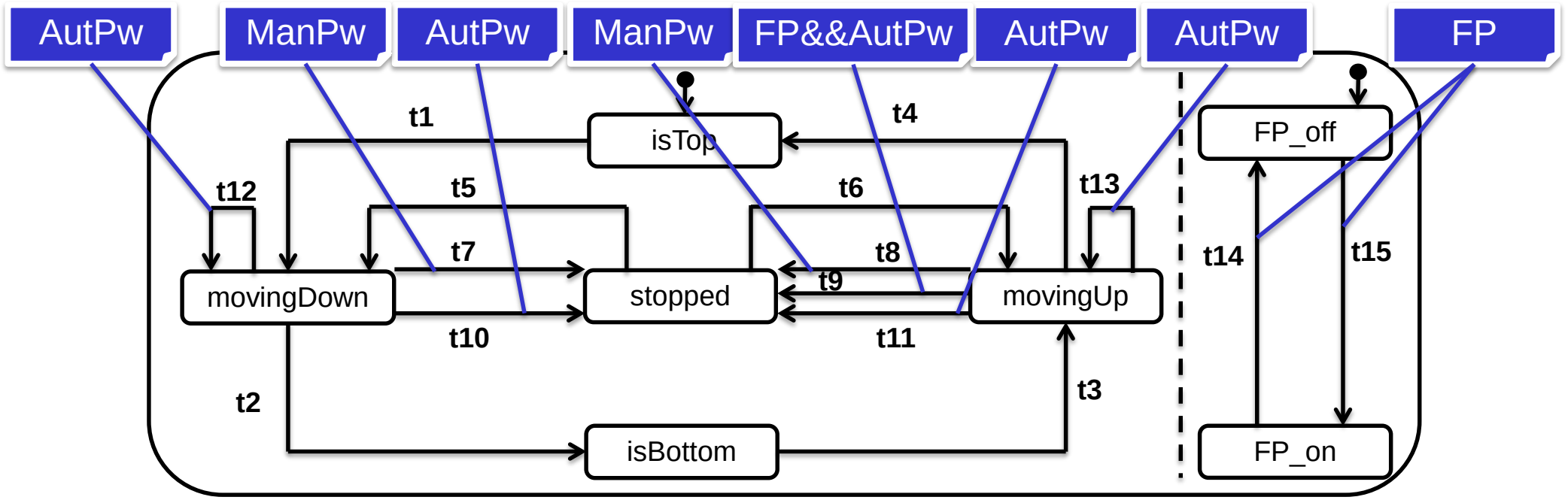
Family-based SPL Testing

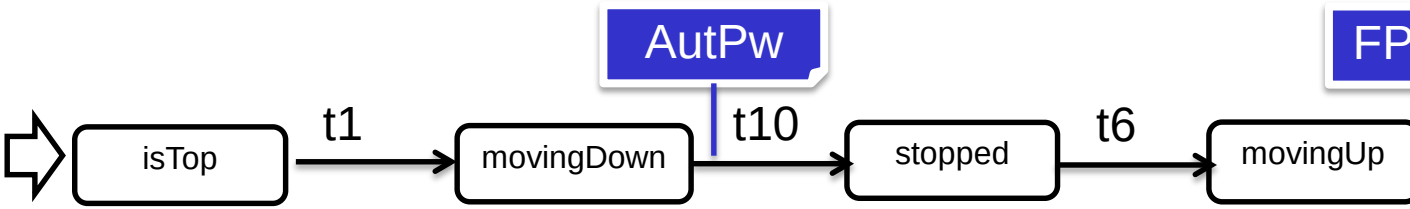


Wiederverwendbare Testfälle



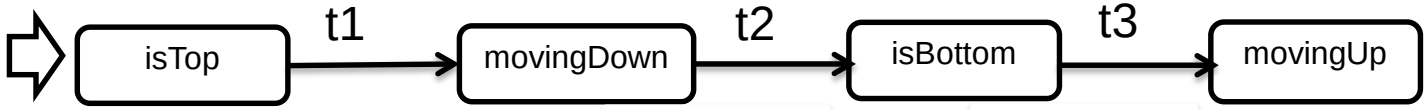
Ableitung wiederverwendbarer Testfälle





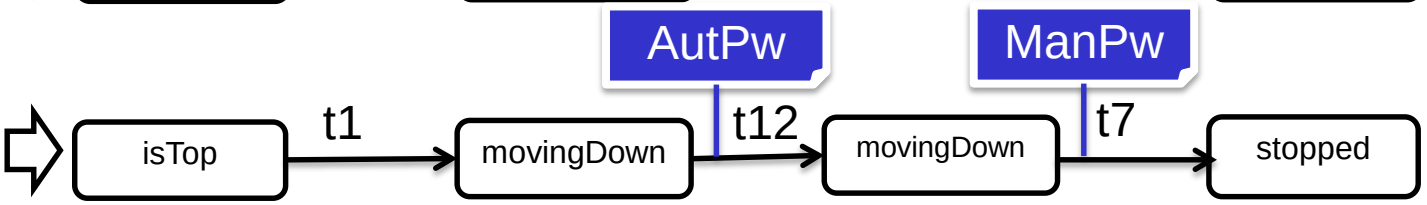
Presence Condition:

AutPW && FP



Kerntestfall:

true



Invalider Testfall:

AutPw && ManPW

Automatisierte Valide SPL Test Suite Generierung

Eingabe:
- SPL Testmodell *stm*
- Testzielmenge G in *stm*
- Feature-Modell *fm*

Ausgabe:
- Valide SPL Test Suite *STS* für *G* mit wiederverwendbaren Testfällen (tc,φ)

Algorithmus:

```
STS = { };
foreach g ∈ G
    (tc,φ) := generateTestCase(stm,g);
    STS    := STS ∪ {(tc,φ)};
    G      := G - { g ∈ G | g covered by tc}
endfor
```

Presence Condition φ für Testfall *tc* ergibt sich durch Konjunktion der Feature-Annotationen der Transitionen auf dem Pfad von *tc* in *stm.* Dabei muss gelten:

$$\varphi \wedge fm \nvdash false$$

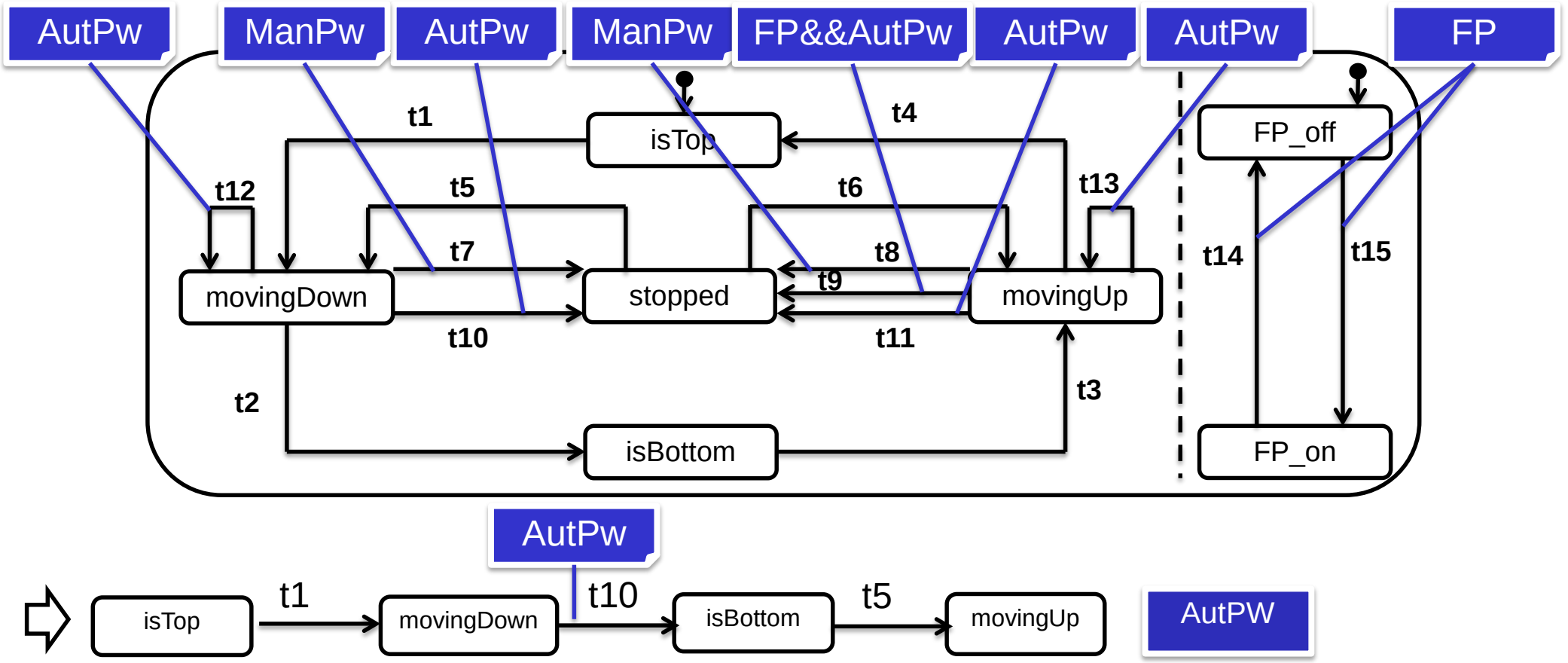
ansonsten Generierung wiederholen

Valide SPL Test Suite

Eine **valide SPL Test Suite** *STS* für ein SPL Testmodell *stm* und eine Menge von Testzielen *G* in *stm* enthält für jedes Testziel $g \in G$ einen Testfall (tc, φ) mit $\varphi \wedge fm \nvdash false$.

Beispiel: SPL Abdeckung

Beispiel: Testziel t5



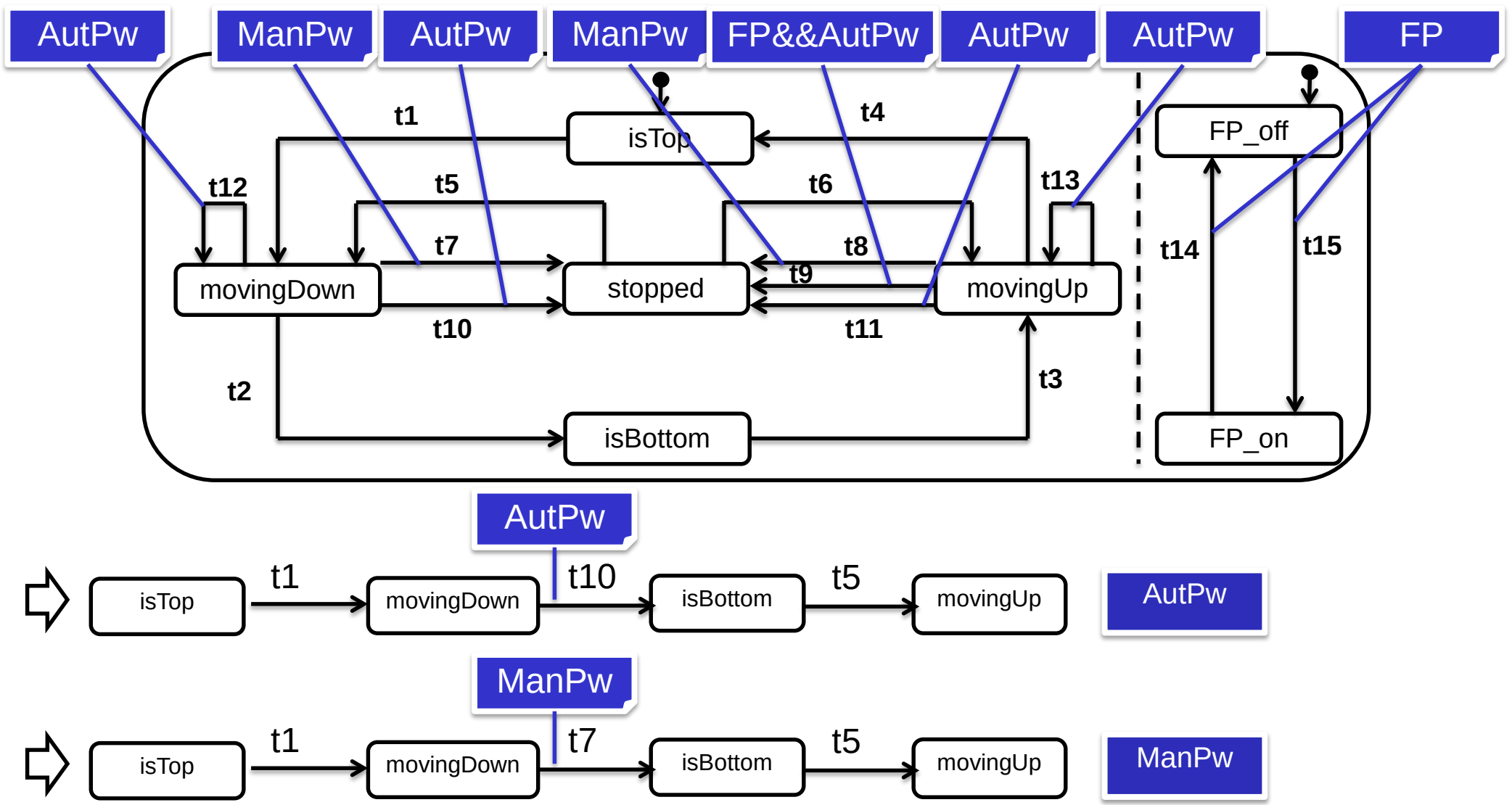
- Testziel t5 in Produktvarianten durch validen Testfall für die Varianten { AutPw } und { AutPw, FP } abgedeckt
- Aber: Testziel t5 **nicht** in Produktvarianten { ManPw } und { ManPw, FP } abgedeckt

Vollständige SPL Test Suite

Eine **vollständige SPL Test Suite** *STS* für ein SPL Testmodell *stm* und eine Menge *G* von Testzielen in *stm* enthält für jedes Testziel $g \in G$ und jede Produktkonfiguration *p* mit $g \in G_p$ einen Testfall (tc, φ) mit $p \vdash \varphi$.

Beispiel: Vollstände SPL-Abdeckung

Beispiel: Testziel t5



Automatisierte Vollständige SPL Test Suite Generierung

Eingabe:
- SPL Testmodell *stm*
- Testzielmenge G in *stm*
- Feature-Modell *fm*, Produktmenge *P*

Ausgabe:
- Vollständige SPL Test Suite *STS* für *G* mit wiederverwendbaren Testfällen (tc,φ)

Algorithmus:

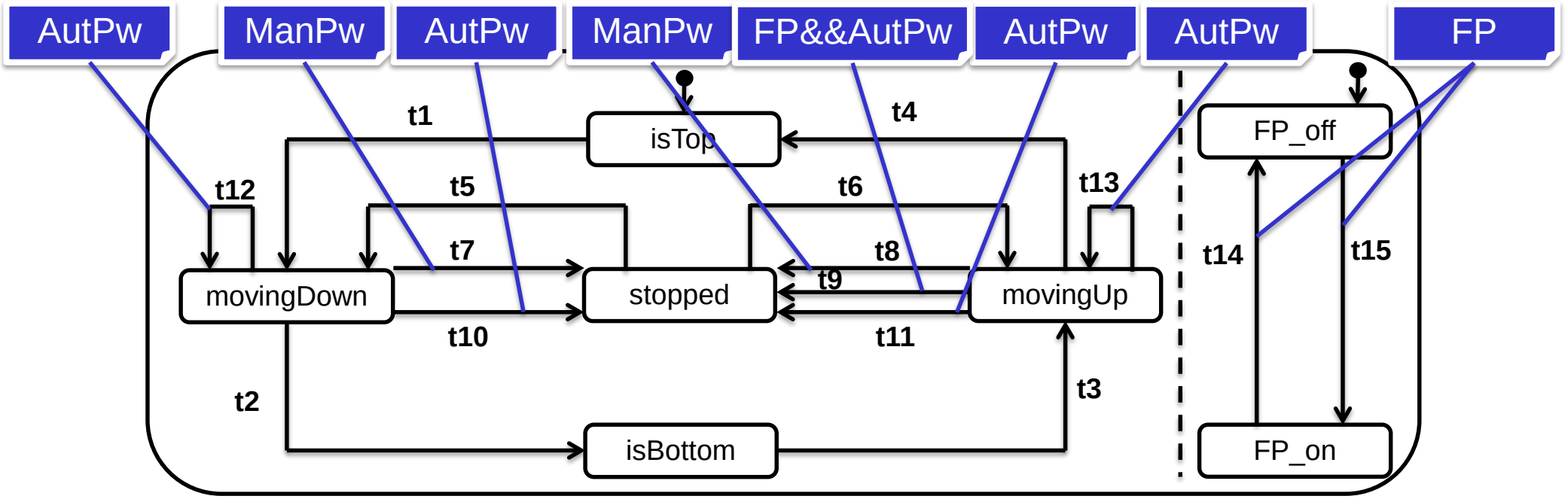
```
STS = { };
foreach g ∈ G
    P_g := { p ∈ P | g ∈ G_p }
    foreach p ∈ P_g
        (tc,φ) := generateTestCase(stm,g);
        STS    := STS ∪ {(tc,φ)};
        P_g    := P_g − { p' ∈ P_g | p ⊢ p'}
    endfor
endfor
```

Dabei muss gelten:

$$p \vdash \varphi$$

ansonsten Generierung wiederholen.

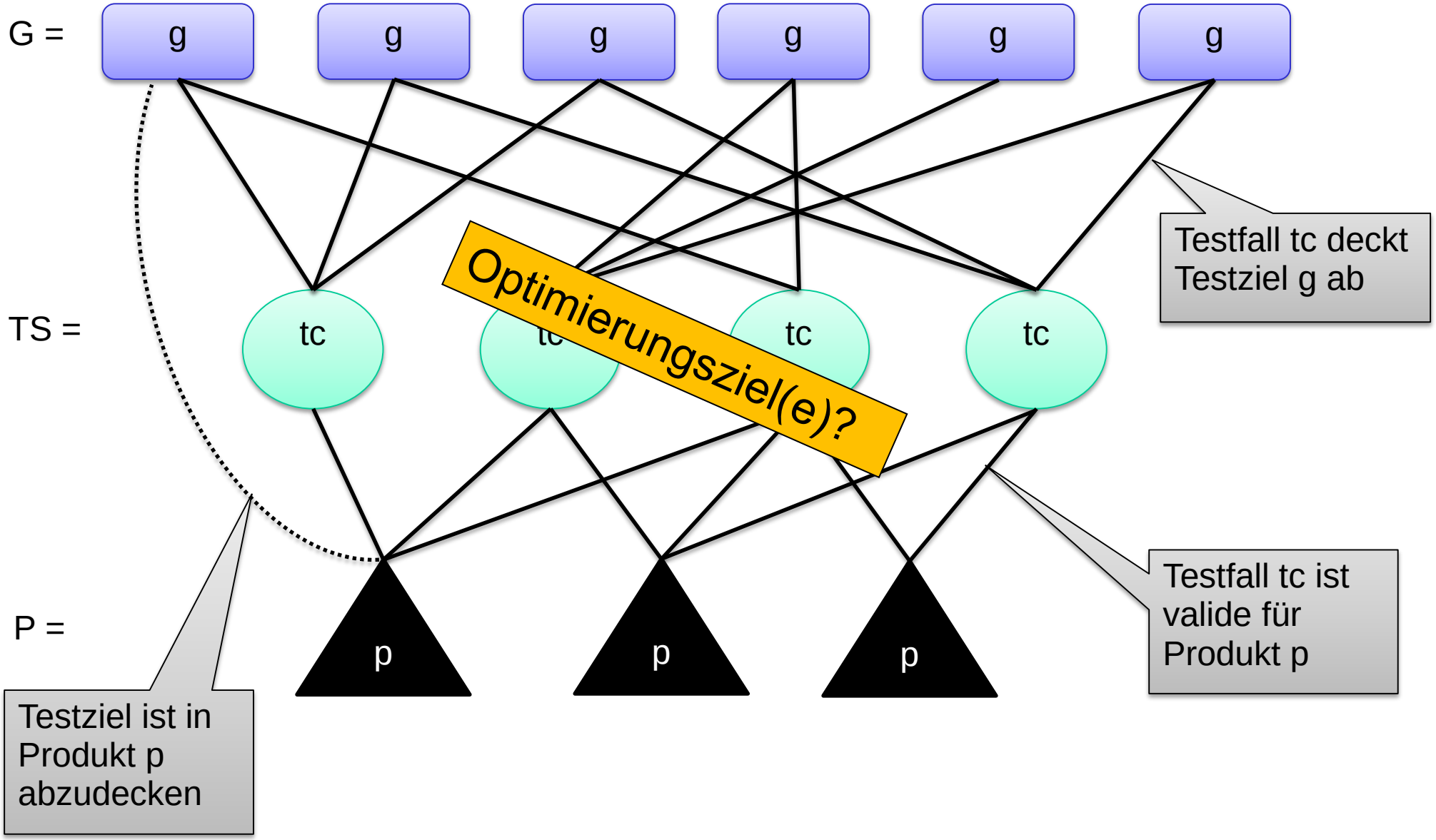
Beispiel: Vollständige SPL Test Suite Generierung



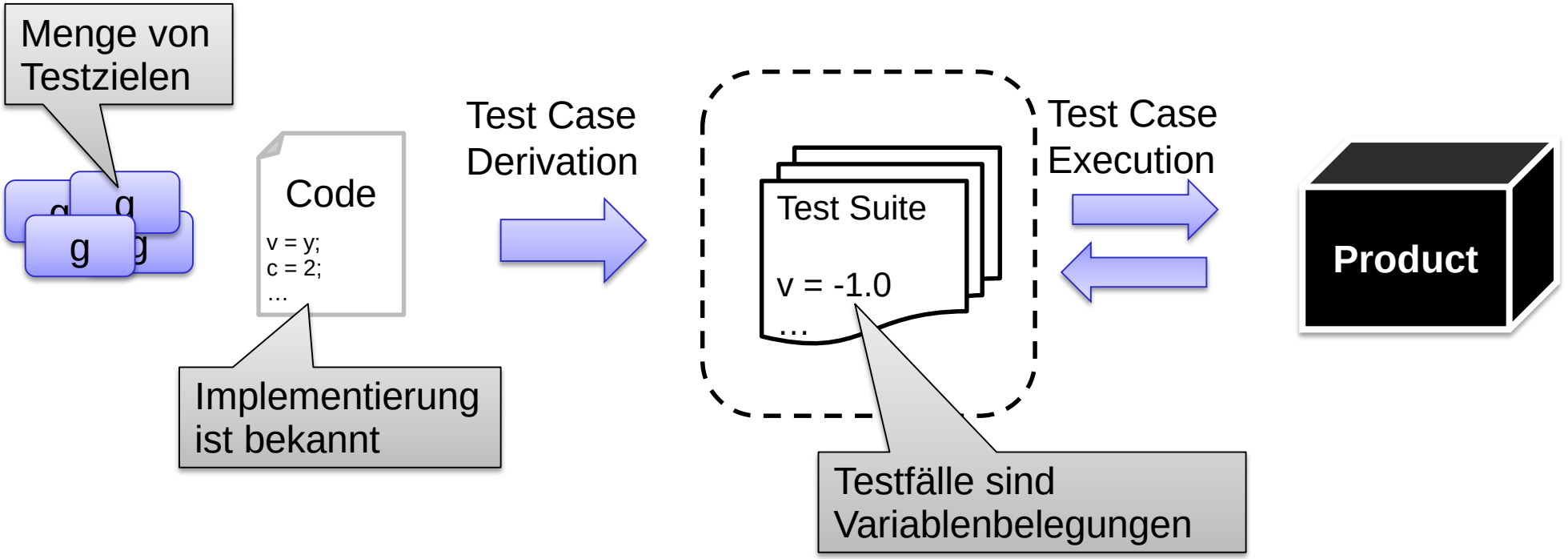
Testfall	Pfad	Valide für Konfigrationen
tc1	t1-t2-t3-t4	p1, p2, p3, p4
tc2	t1-t7-t5	p1, p3
tc3	t1-t12-t10-t5	p2, p4
tc4	t1-t2-t3-t13-t11-t6	p2, p4
tc5	t1-t2-t3-t8-t6	p1, p3
tc6	t1-t2-t3-t15-t4	p3, p4
tc7	t1-t2-t3-t15-t9-t14	p4

p1 = { PW, ManPw }
G_{p1} = { t1,t2,t3,t4,t5,t6,t7,t8 }
p2 = { PW, AutPw }
G_{p2} = { t1,t2,t3,t4,t5,t6,t9,t10,t11,t12,t13 }
p3 = { PW, ManPw, FP }
G_{p3} = { t1,t2,t3,t4,t5,t6,t7,t8,t14,t15 }
P4 = { PW, AutPw, FP }
G_{p4} = { t1,t2,t3,t4,t5,t6,t9,t10,t11,t12,t13,t14,t15 }

SPL Test Suite Optimierung



White Box Testing



- Testfälle sind initiale Programmeingaben (Variablenbelegungen)
- Testfälle leiten sich aus der Struktur des Programmes ab

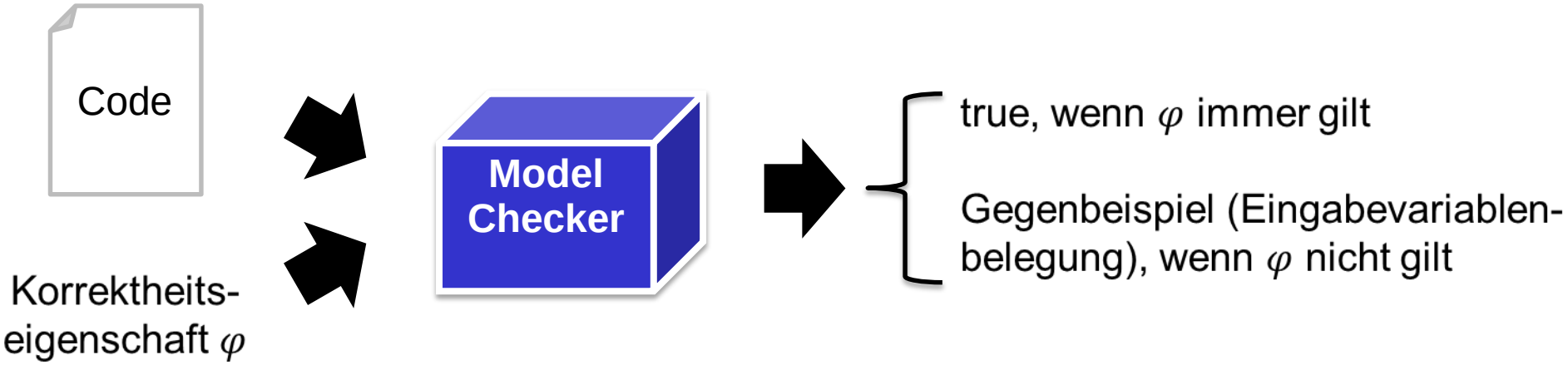
Model Checking

- Model Checking dient dem formalem Beweis von Korrektheitseigenschaften

$$Spec \vDash \varphi$$

- Model Checking
  - Es wird überprüft, ob die Eigenschaft φ immer gilt
  - Model Checker such einen Zustand, in dem φ nicht gilt
  - Gibt es diesen Zustand wird ein Pfad dahin ausgegeben (Gegenbeispiel)
  - Gilt φ in jedem Zustand, ist das Programm korrekt

- Software Model Checking
  - Die Spezifikation ist der Code



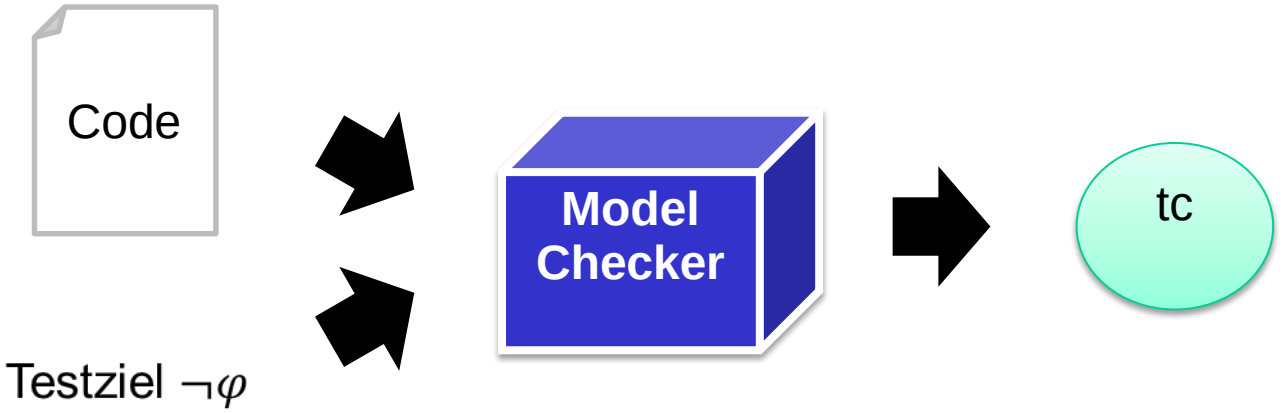
Testfallgenerierung mit symbolischem Model Checking

- Das Tesziel φ wird als Korrektheitseigenschaft codiert und negiert

$$Prog \vDash \neg\varphi$$

- Testziele sind meistens Programmlabel oder Prädikate über Variablen

- Es wird überprüft, ob die Eigenschaft $\neg\varphi$ immer gilt
  - Gilt φ ist das Testziel erreicht und die Korrekheitseigenschaft verletzt
  - Der Model Checker gibt ein Gegenbeispiel aus
  - Das Gegenbeispiel wird als Testfall interpretiert



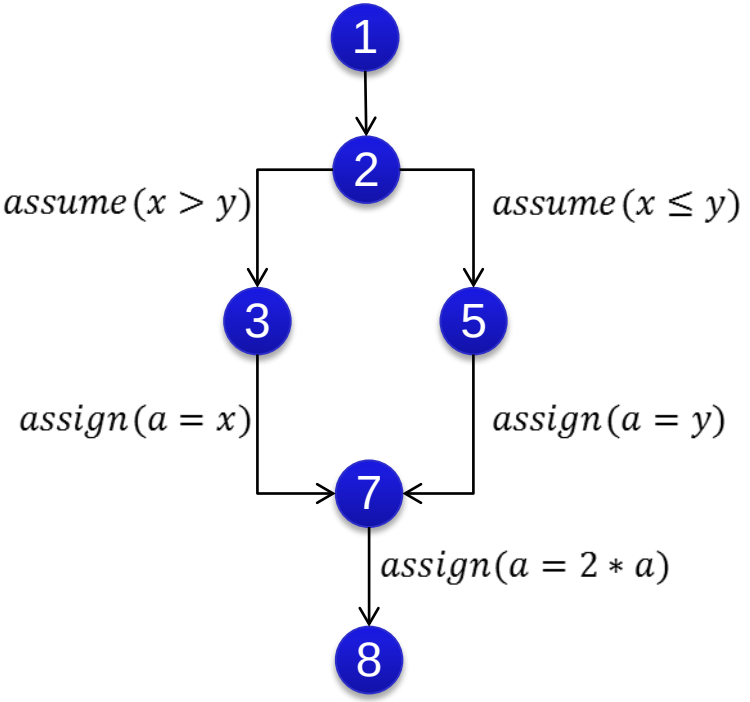
Beispiel: Testfallgenerierung mit symbolischem Model Checking (1/3)

Ein **Kontrollflussautomat** $A = (L, G)$ besteht aus
- einer Menge von Programm Locations L und
- einer Menge von Kontrollflusskanten $G \subseteq L \times Ops \times L$, welche die auszuführenden Operationen auf dem Kontrollfluss zwischen den Programm Locations beschreiben.

Dabei ist Ops eine Menge von Operationen und Prädikaten über Variablen.

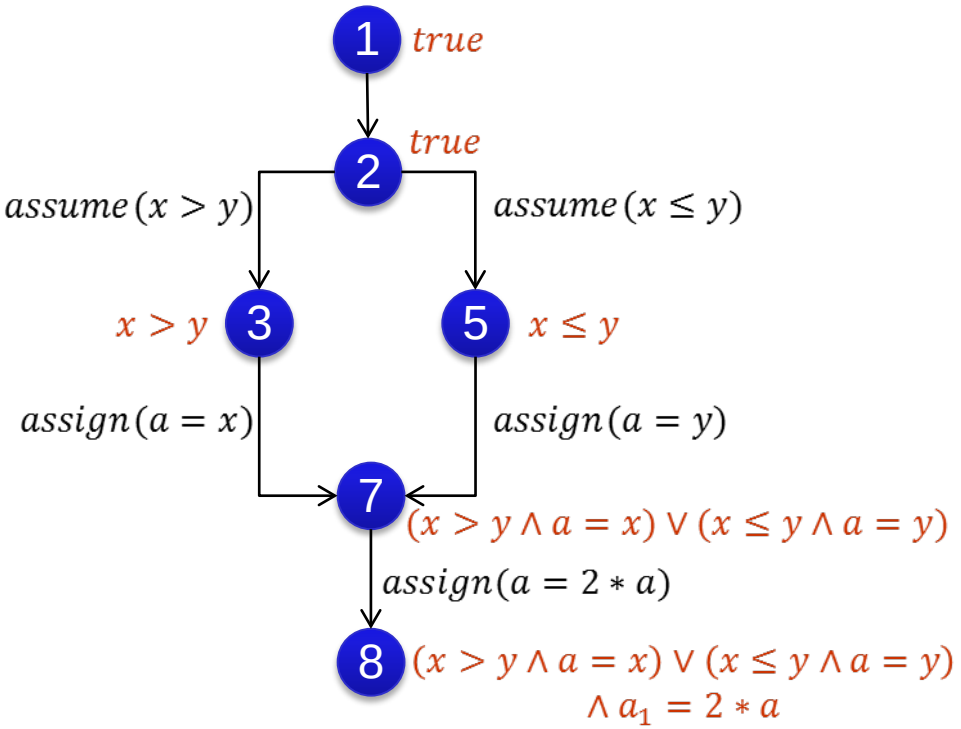
Schritt 1: Kontrollflussgraphen aus dem Code generieren

```
  int calc(int x, int y) {
1   int a;
2   if(x > y) {
3     a = x;
4   } else {
5     a = y;
6   }
7   a = 2*a;
8   return a;
  }
```



Beispiel: Testfallgenerierung mit symbolischem Model Checking (2/3)

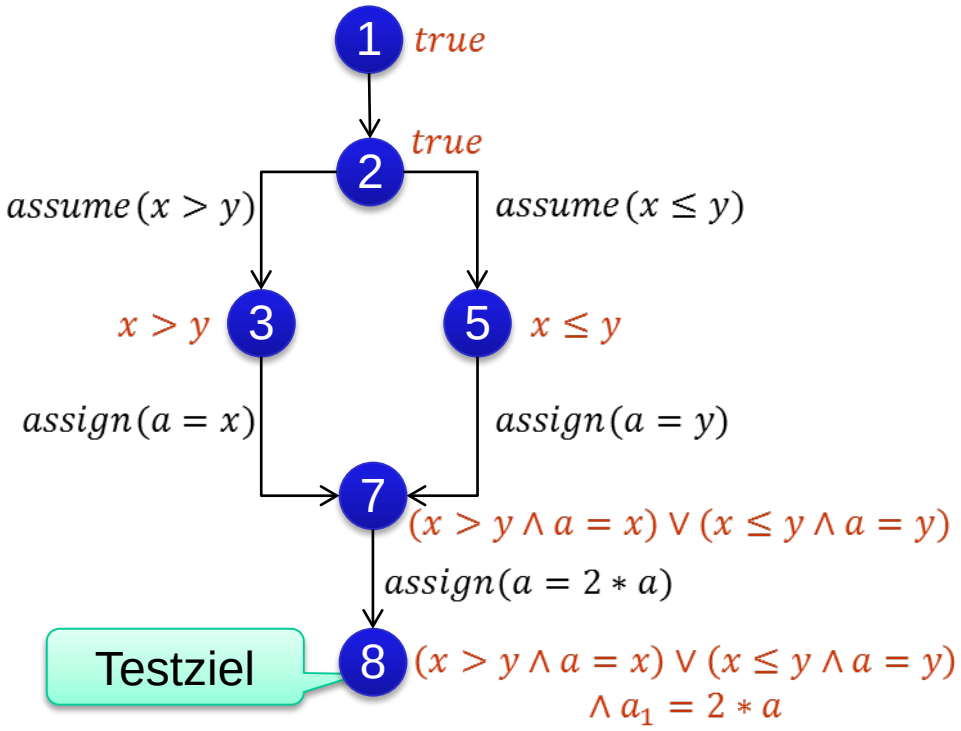
Schritt 2: Pfadbedingungen für die Programm Locations erzeugen



Die Pfadbedingung ist eine Formel über die Programmvariablen, welche angibt unter welchen Bedingungen eine Programm Location erreichbar ist.

Beispiel: Testfallgenerierung mit symbolischem Model Checking (3/3)

Schritt 3: Testfälle aus Pfadbedingungen mit Hilfe eines SMT-Solvers berechnen

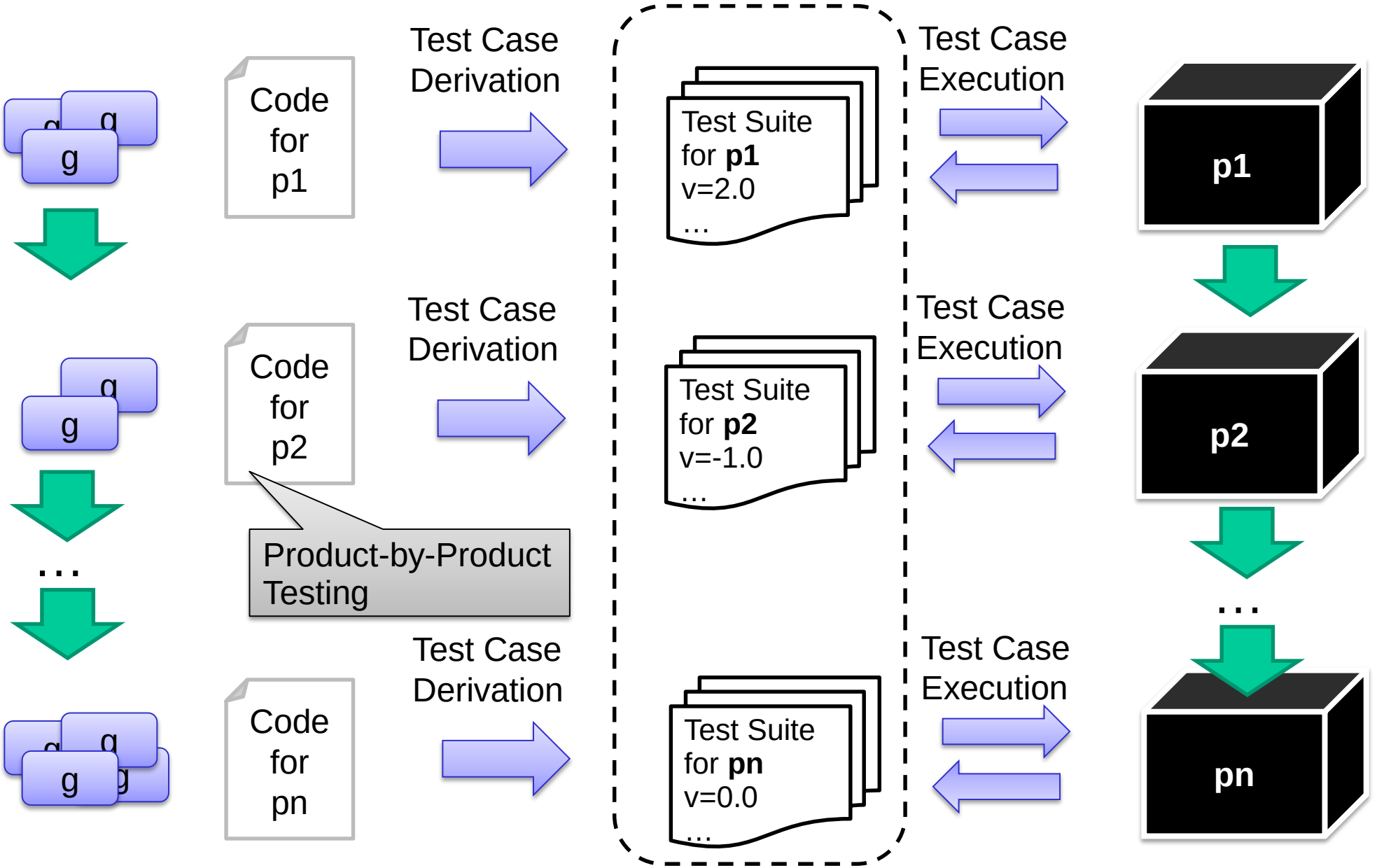


Testfall	Pfad	x	y	a	a1
tc1	1-2-3-7-8	2	1	2	4
tc2	1-2-5-7-8	1	3	3	6

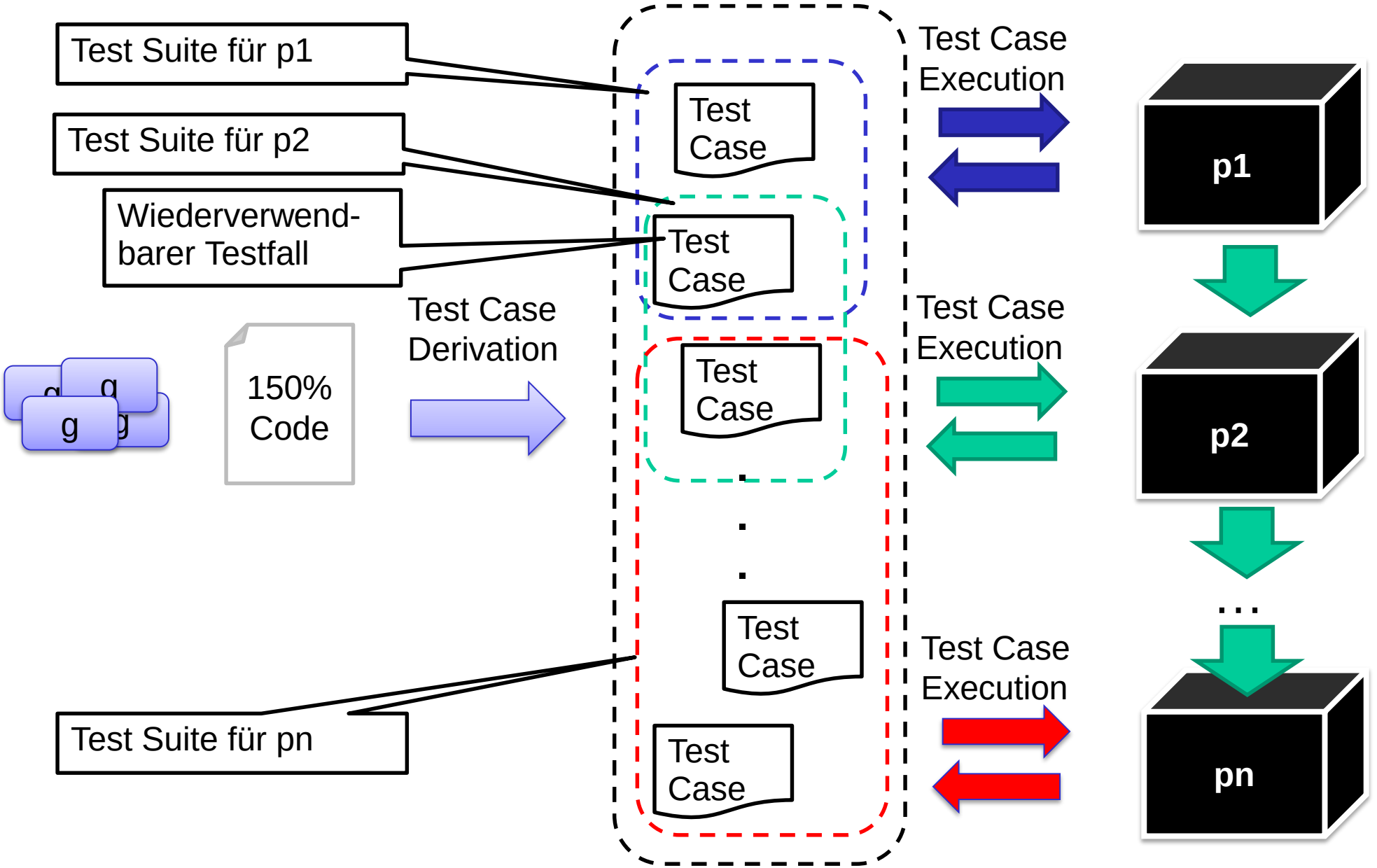
Testfälle

Eine Variablenbelegung zum Erreichen einer Programm Location kann mit Hilfe eines SMT-Solvers berechnet werden.

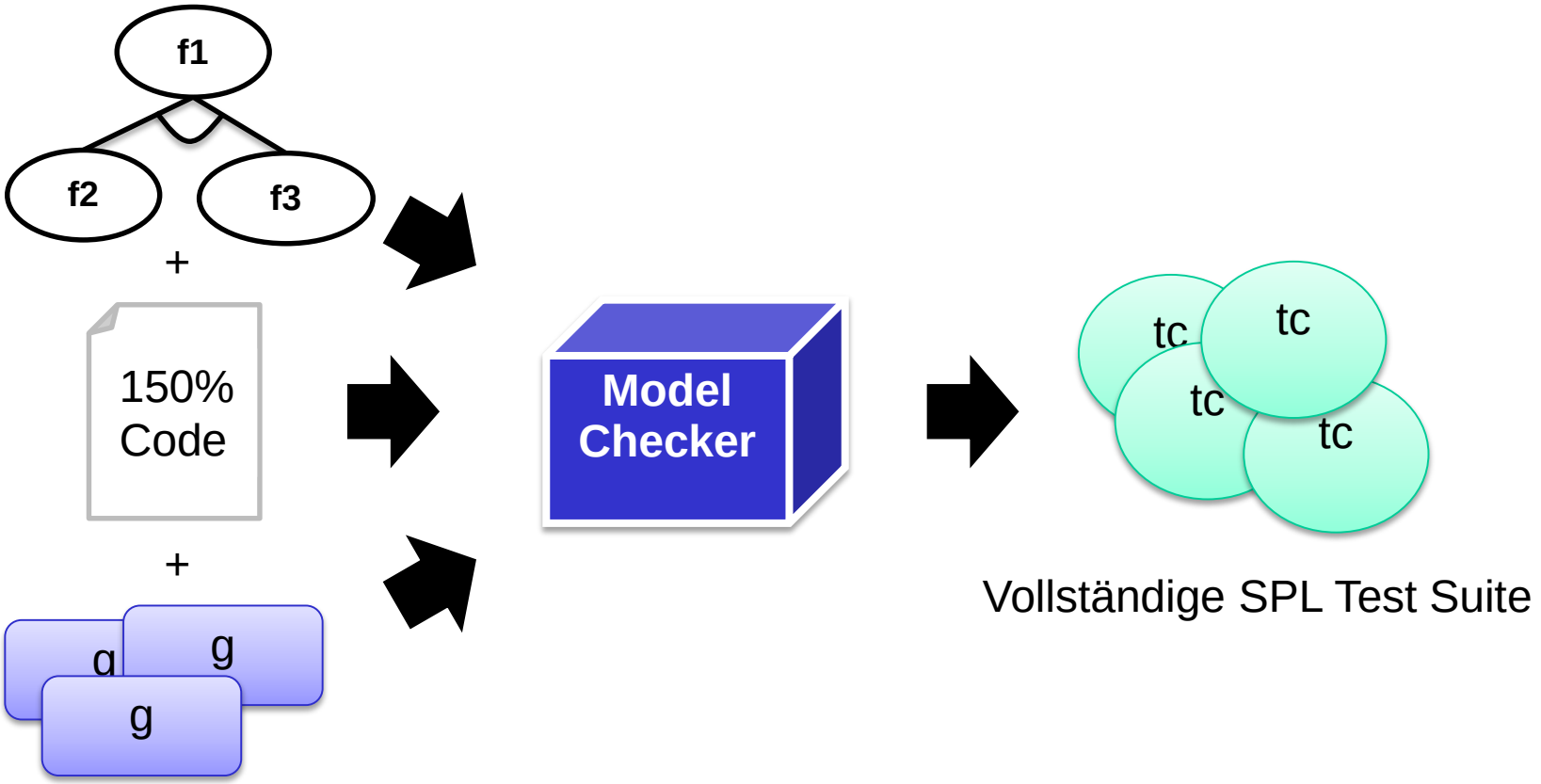
White Box SPL Testing



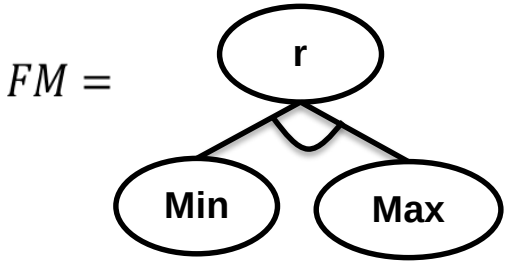
Family-based White Box Testing



Family-based Test Case Generierung mit symbolischem Model Checking



Beispiel: Vollständige SPL Test Suite Generierung mit Model Checking (1/4)




```
  int calc(int x, int y) {
1   int a;
2   if(x > y) {
3     a = x;
4   } else {
5     a = y;
6   }
7   a = 2*a;
8   return a;
  }
```

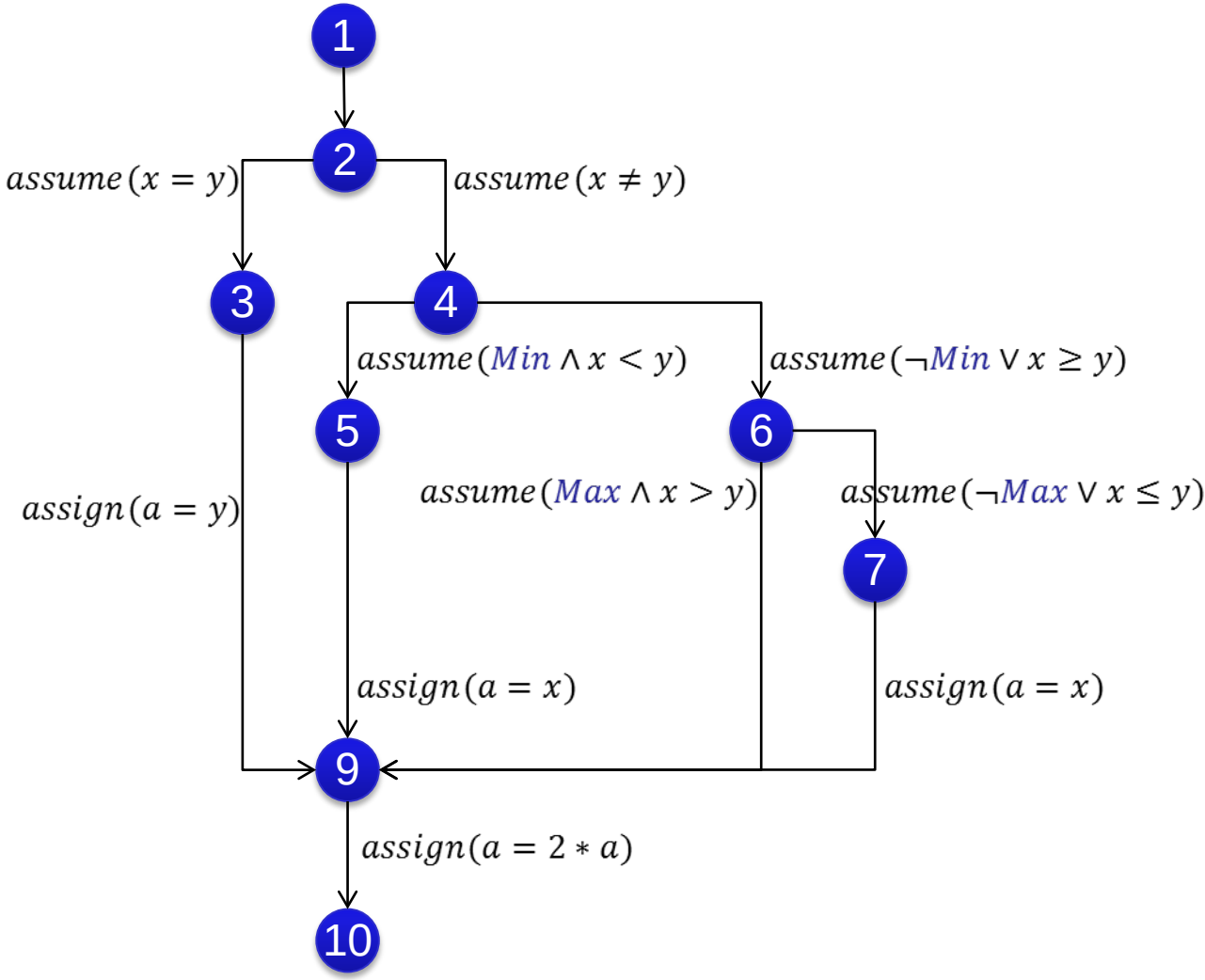
```
  int calc(int x, int y) {
1   int a;
2   if (x == y) {
3     a = y;
4   else if (Min && x < y) {
5     a = x;
6   } else if (Max && x > y) {
7     a = x;
8   }
9   a = 2*a;
10  return a;
  }
```

- Features Min und Max zur Unterscheidung von < und > hinzugefügt.
- Features werden als Variablen codiert.

Beispiel: Vollständige SPL Test Suite Generierung mit Model Checking (2/4)

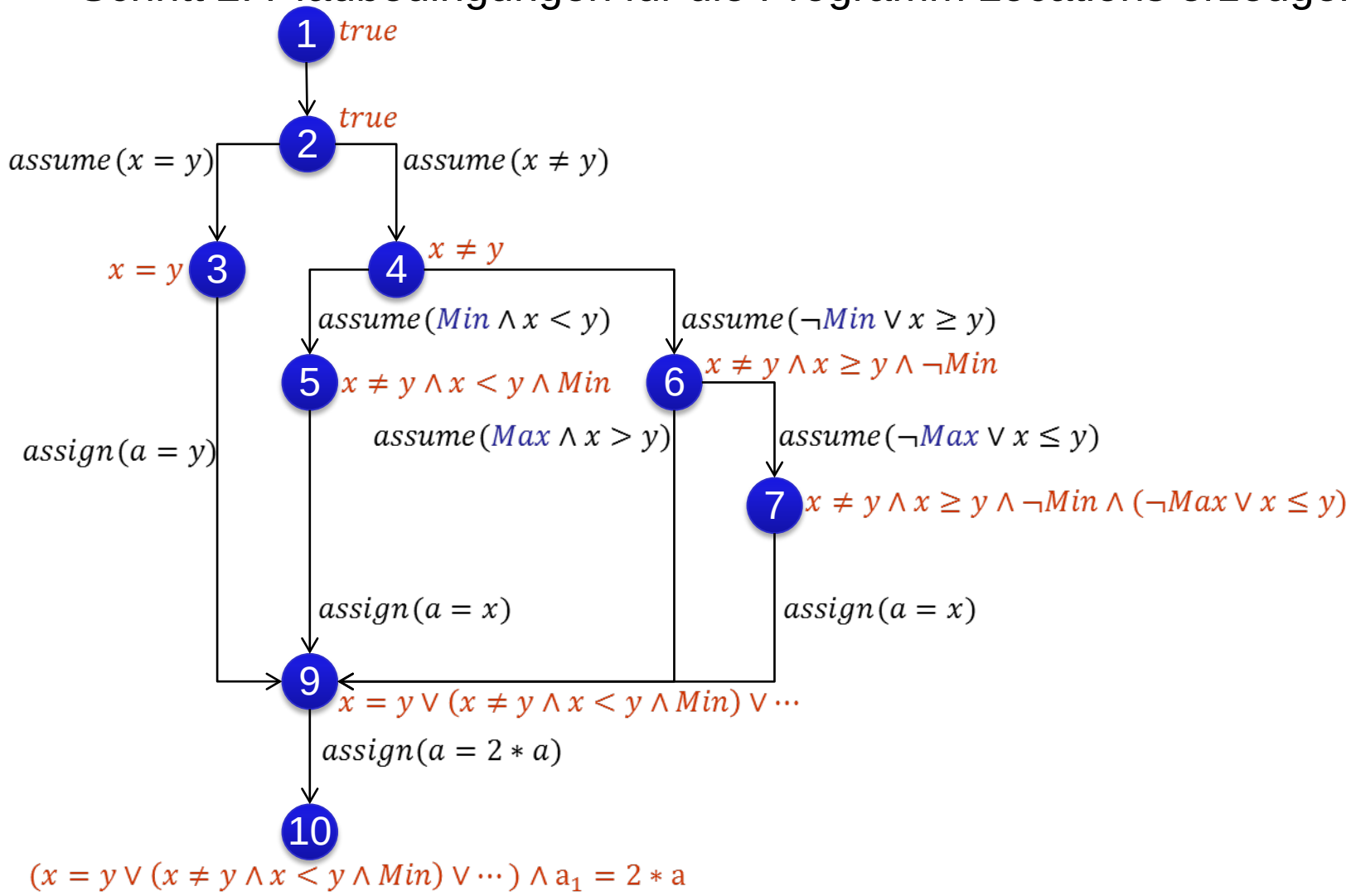
Schritt 1: Kontrollflussgraphen aus dem Code generieren

```
  int calc(int x, int y) {
1   int a;
2   if (x == y) {
3     a = y;
4   else if (Min && x < y) {
5     a = x;
6   } else if (Max && x > y) {
7     a = x;
8   }
9   a = 2*a;
10  return a;
  }
```



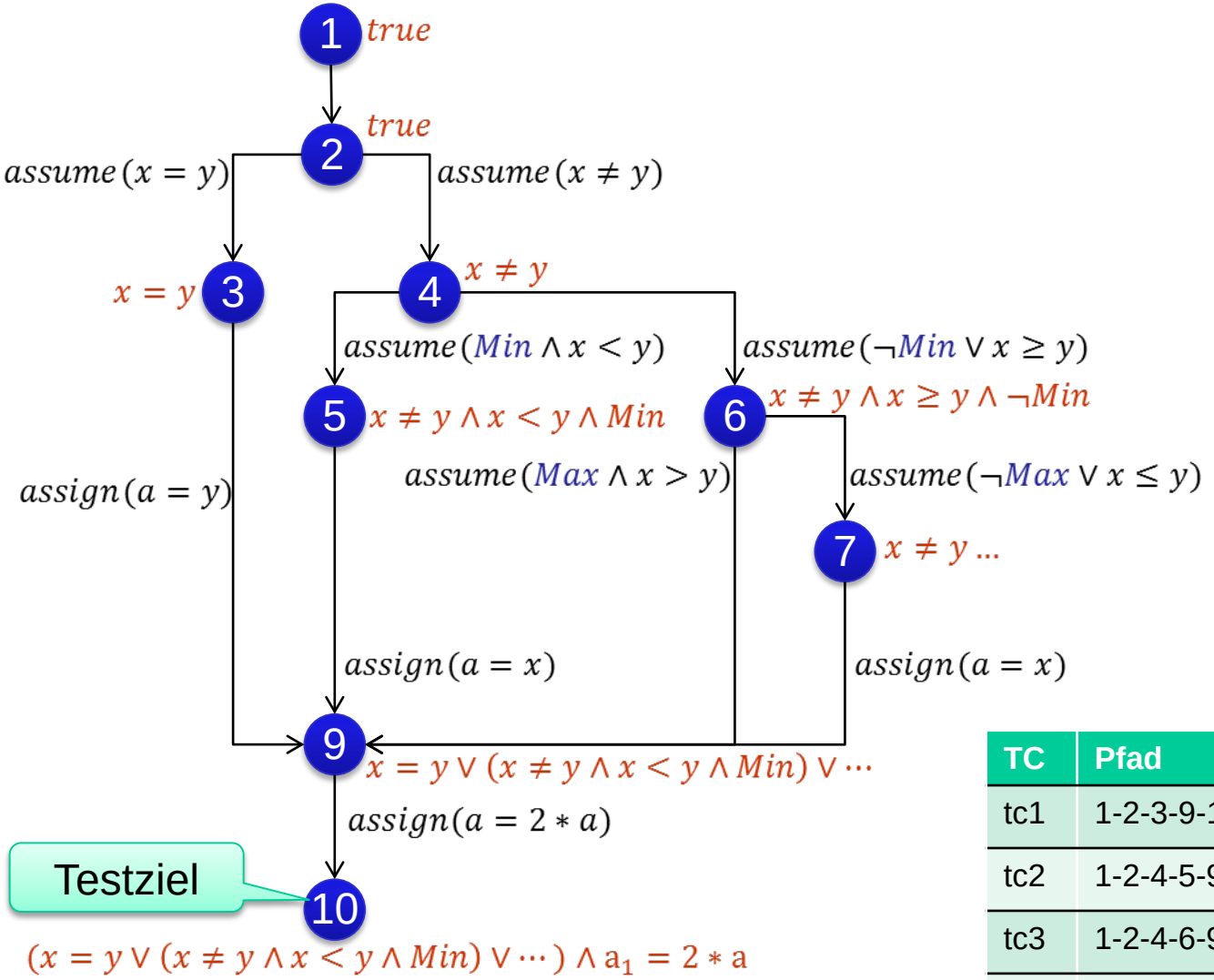
Beispiel: Vollständige SPL Test Suite Generierung mit Model Checking (3/4)

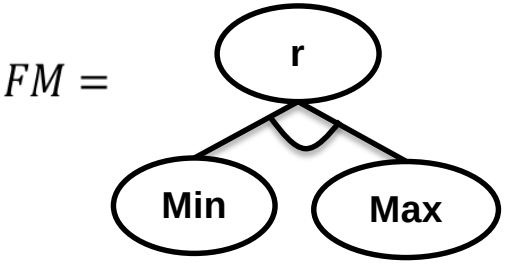
Schritt 2: Pfadbedingungen für die Programm Locations erzeugen



Beispiel: Vollständige SPL Test Suite Generierung mit Model Checking (4/4)

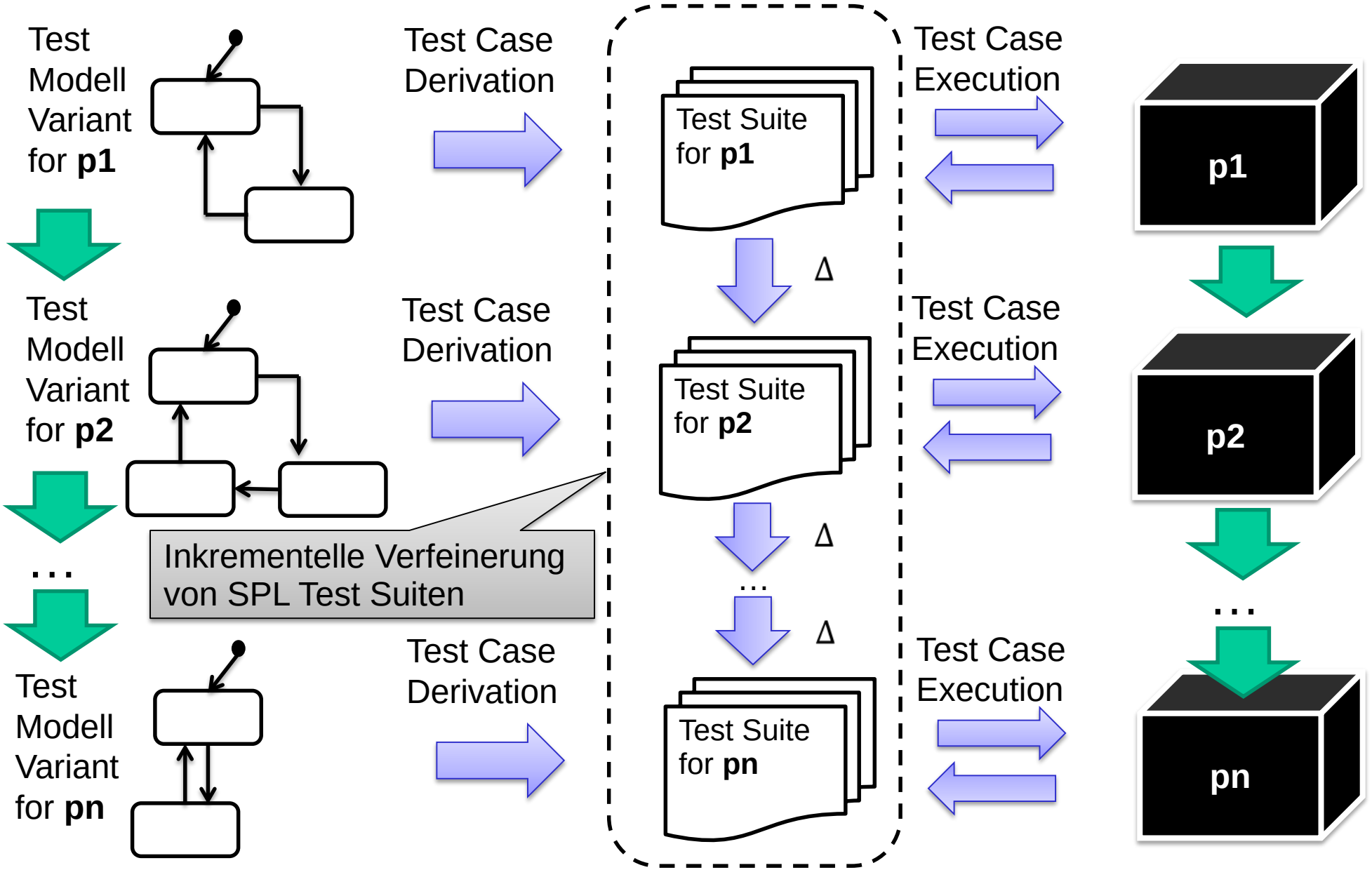
Schritt 3: Testfälle für Testziele mit Hilfe eines SMT-Solvers berechnen



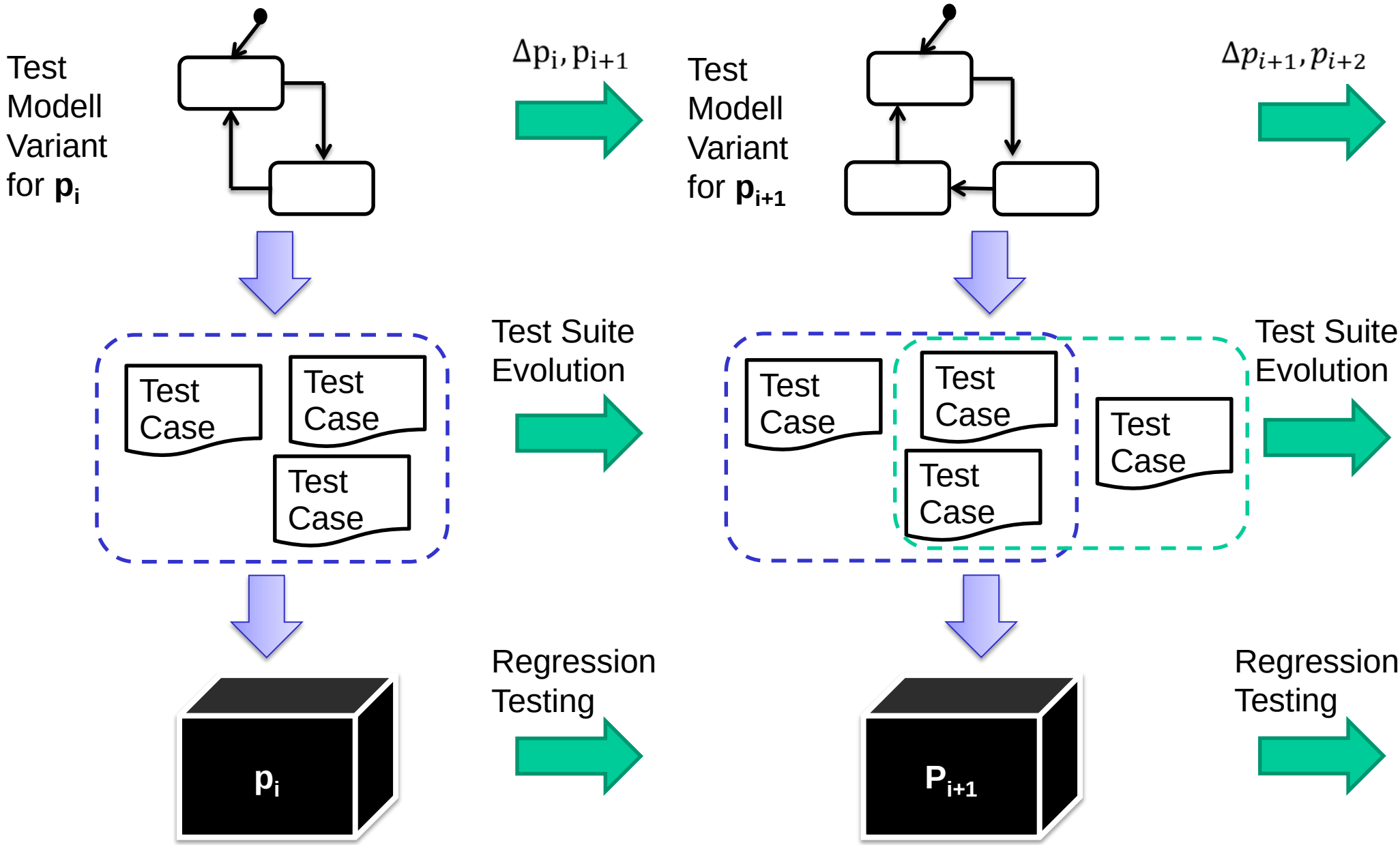


TC	Pfad	x	y	a	a1	PC
tc1	1-2-3-9-10	1	1	1	2	*true*
tc2	1-2-4-5-9-10	1	2	1	2	*Min*
tc3	1-2-4-6-9-10	2	1	2	4	$\neg Min \wedge Max$
~~tc4~~	~~1-2-4-6-7-9-10~~	Dead Code				~~$\neg Min \wedge \neg Max$~~

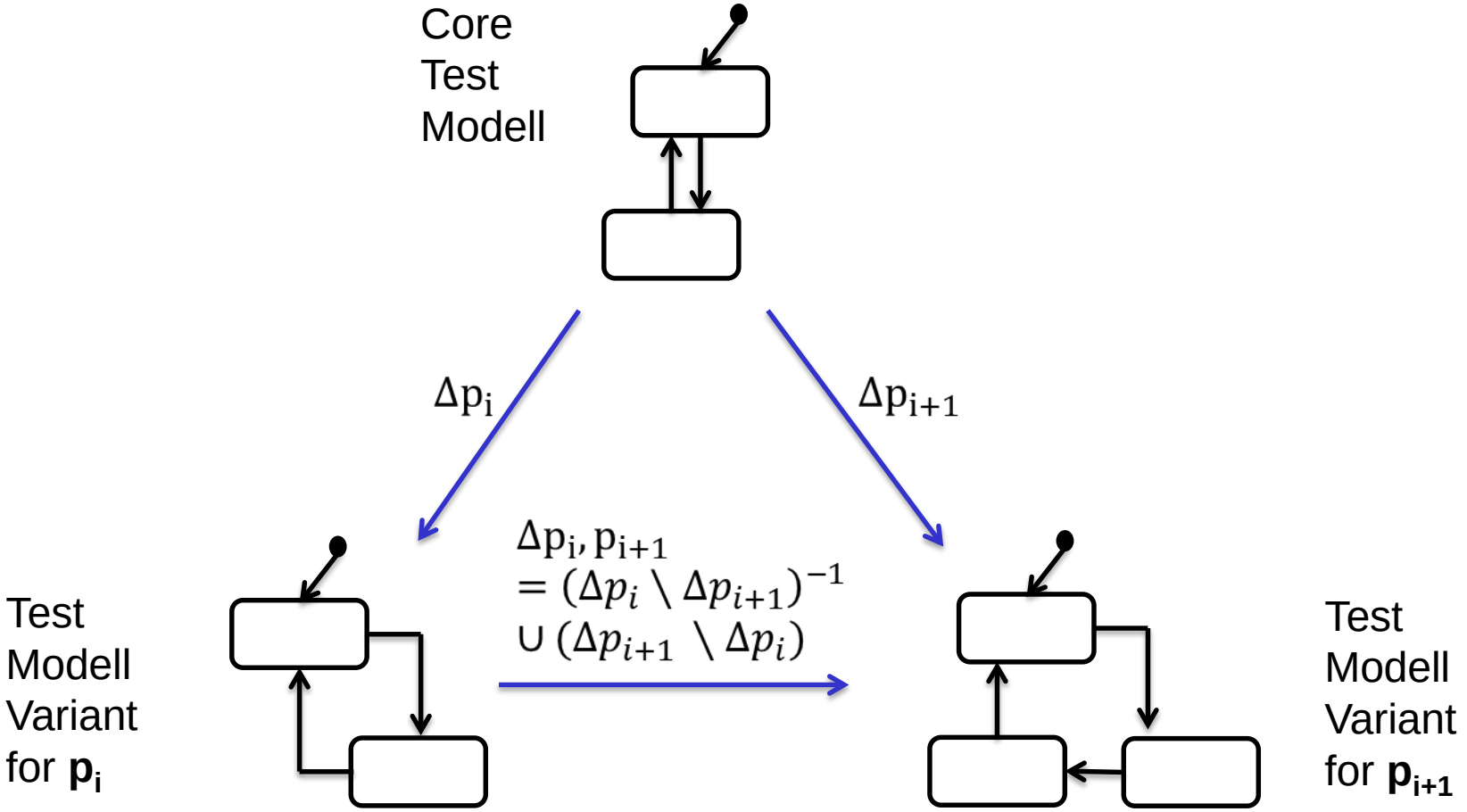
Incremental SPL Testing



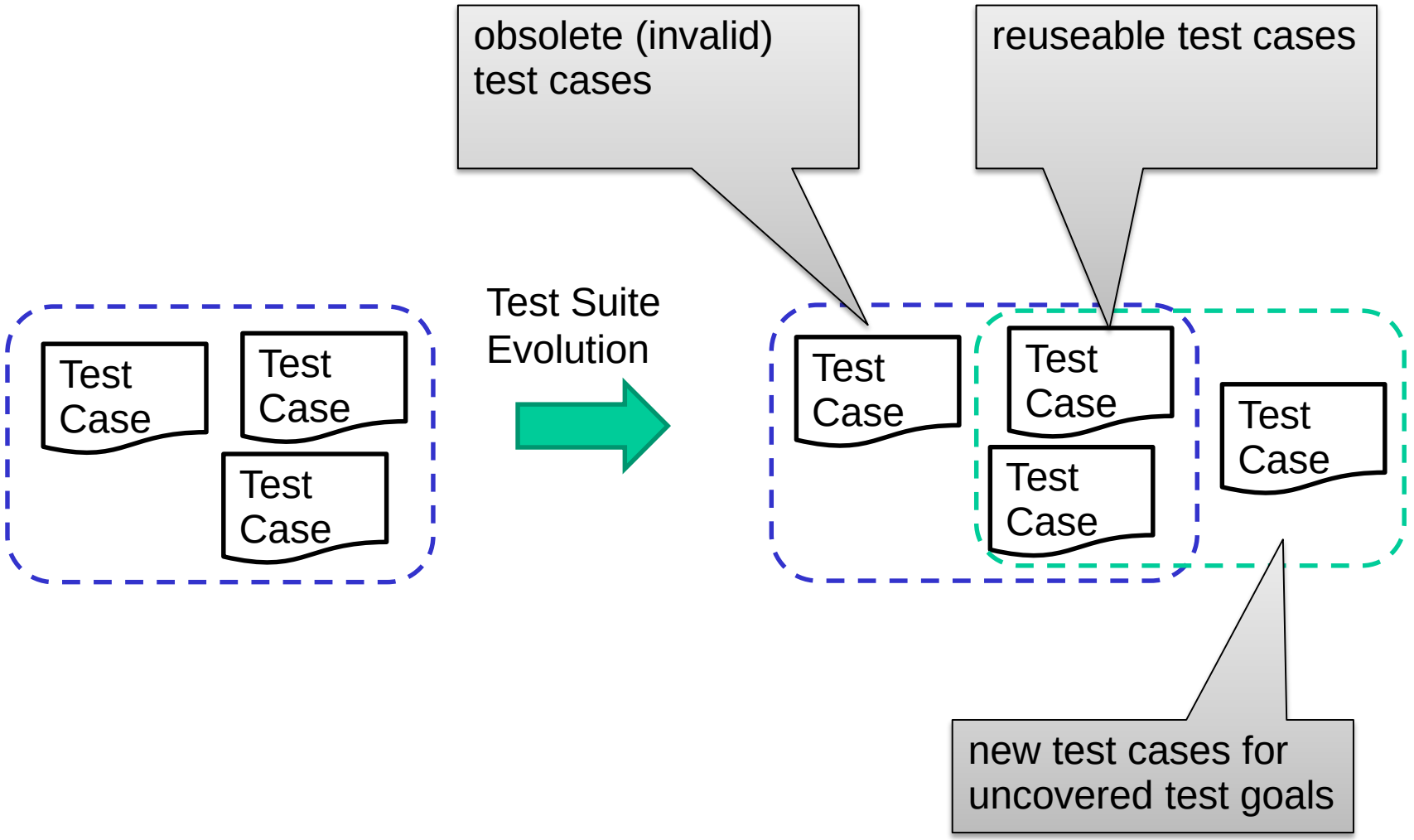
Incremental SPL Test Suite Generation



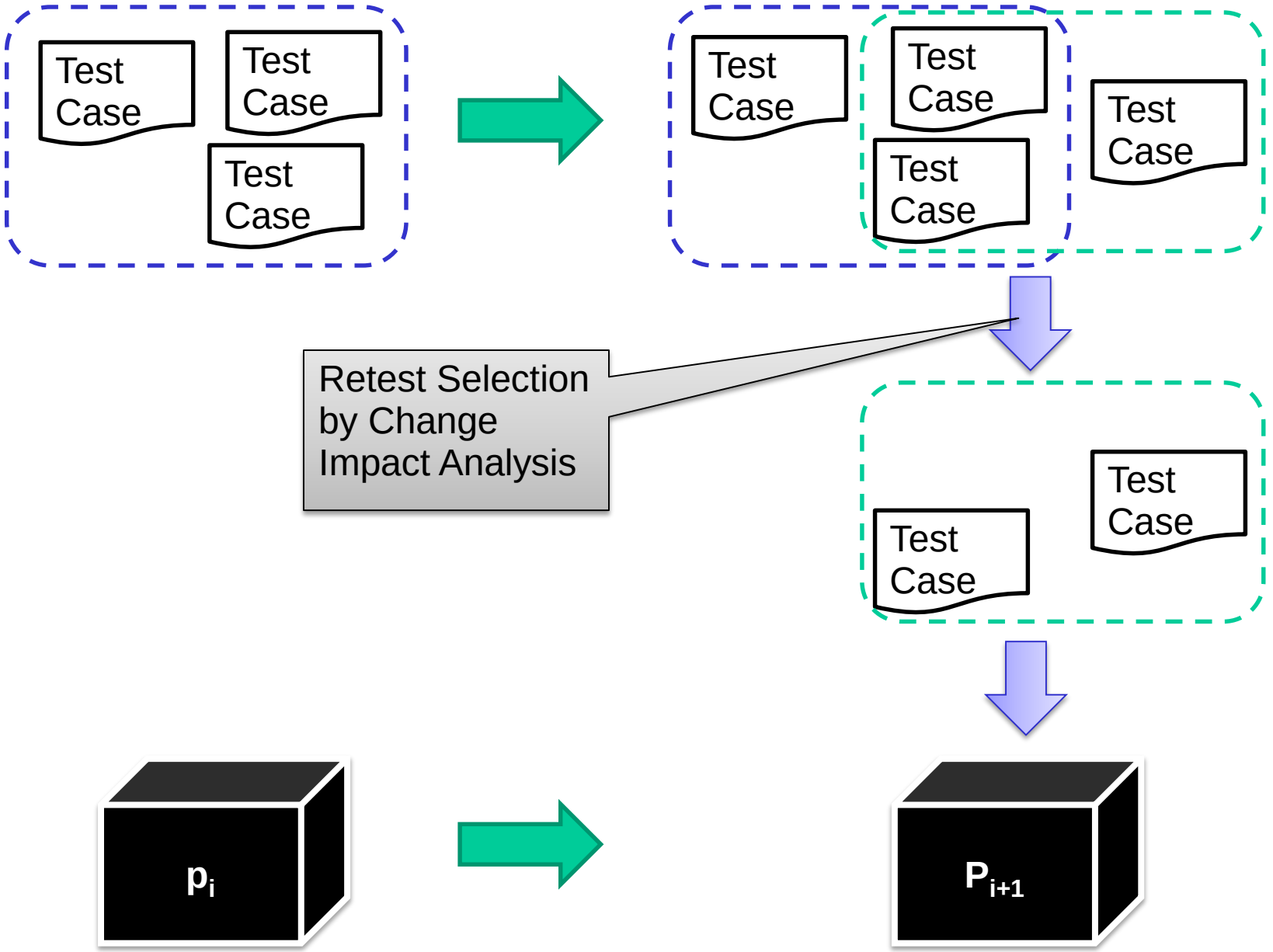
Regression Delta



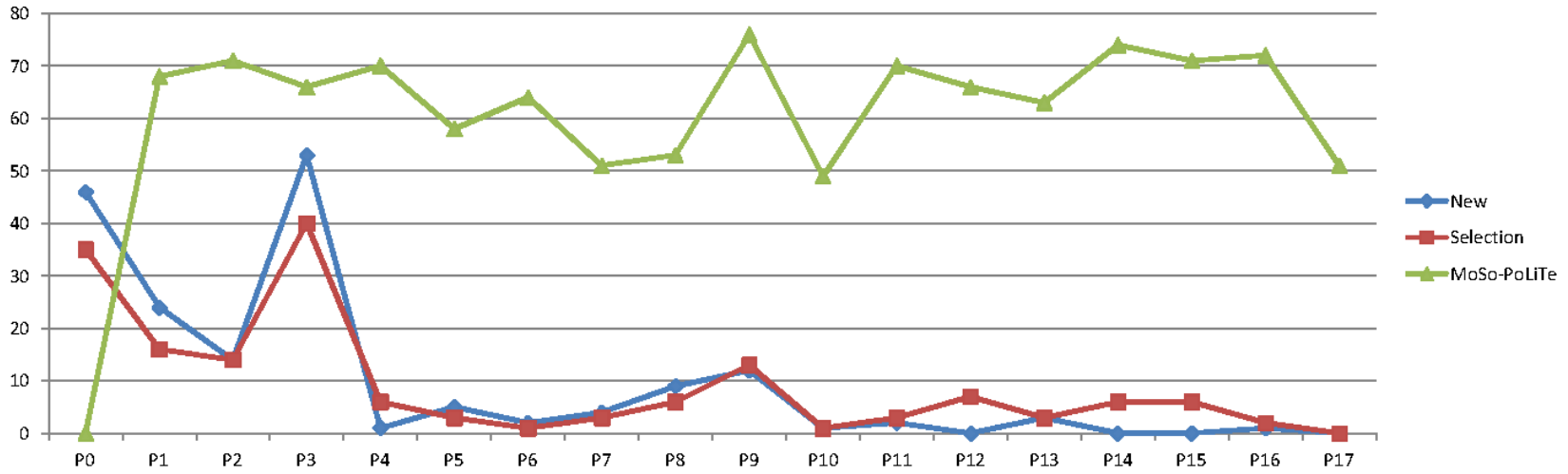
SPL Test Suite Evolution



SPL Regression Testing

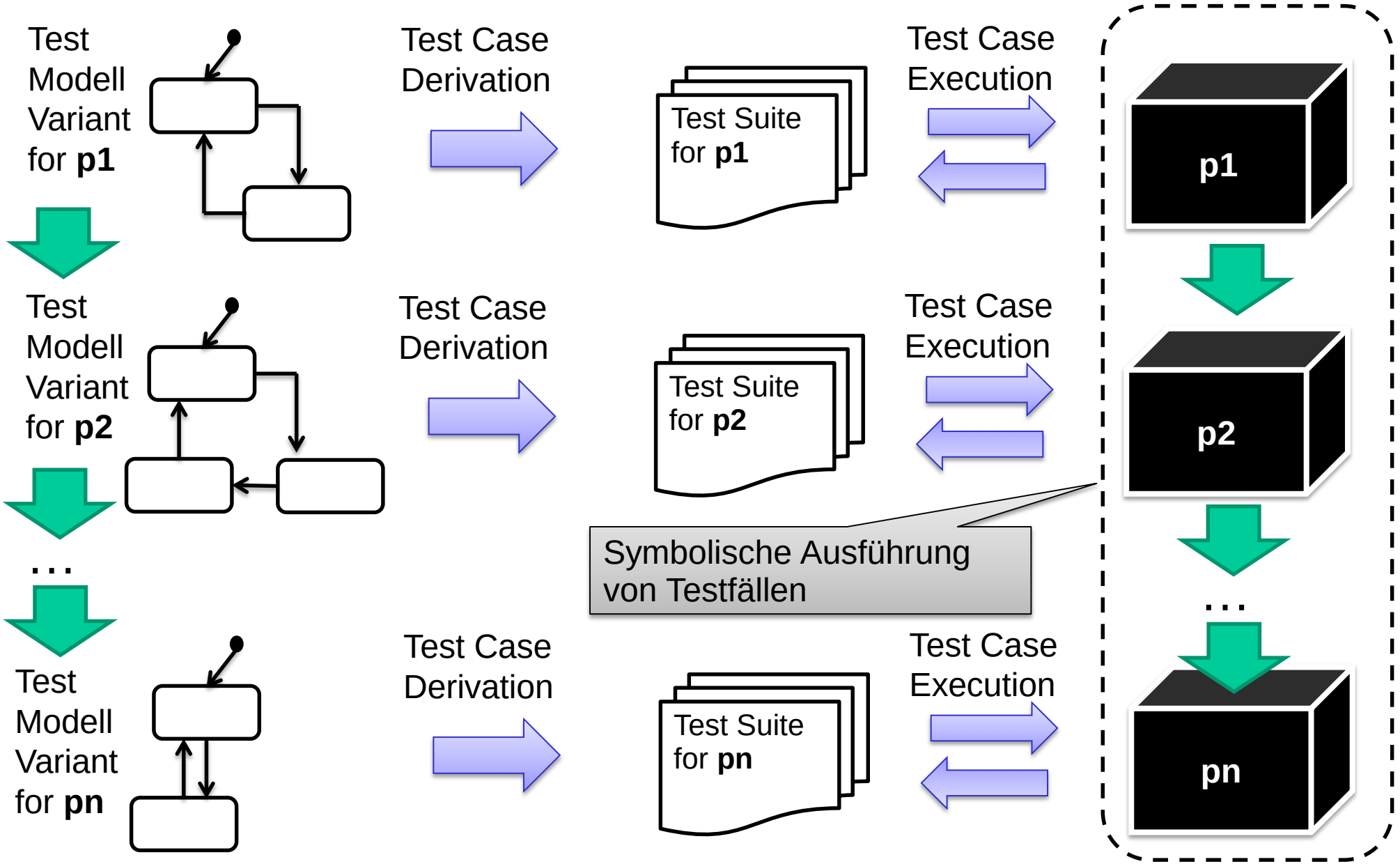


Beispiel: BCS SPL

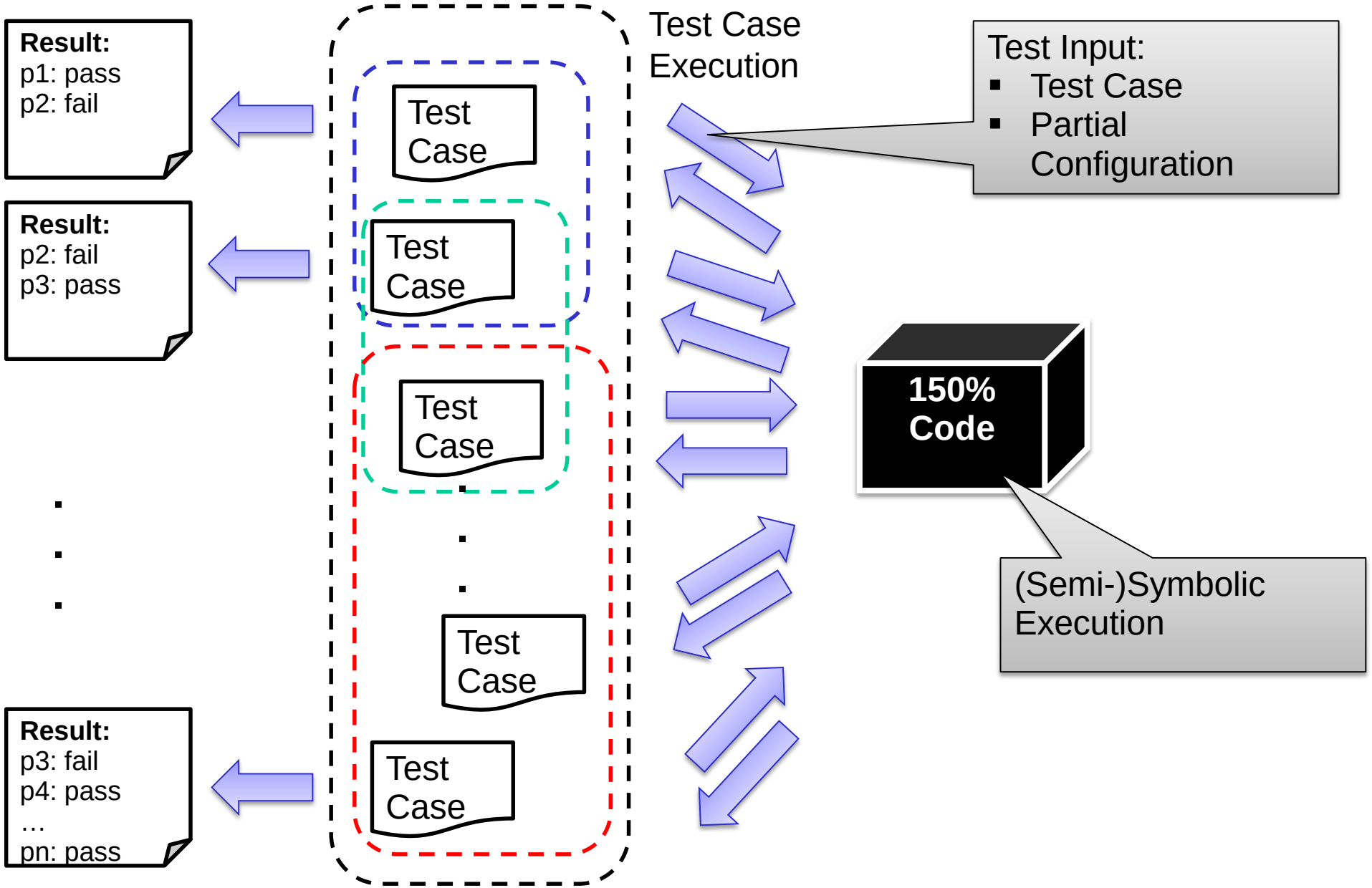


- Pairwise Sampling: 17 Products-Under-Test
- Product-by-product SPL Testing: ~ 64 Testfälle pro Produkt generiert und ausgeführt
- Incremental SPL Testing: ~ 10 Testfälle pro Produkt generiert, ~ 9 Testfälle pro Produkt ausgeführt

Family-based Test Execution



Family-based Test Execution



Fazit

- Kombination verschiedener SPL-Testansätze möglich:

    1. Sample erzeugen
    2. Erzeugen einer SPL Test Suite für den Sample
        - Family-based SPL Test Suite Generierung für Teilmenge der Testziele und/oder Produktmenge
        - Inkrementelle Verfeinerung der SPL Test Suite
    3. Symbolische Ausführung der Testfälle auf dem Sample

- Je nach Ansatz eignen sich verschiedene SPL Testmodellierungstechniken
    - Sample-based: Kompositionsbasiertes SPL Testmodell für Detektion von Feature-Interaktionen
    - Family-based: Annotatives SPL Testmodell
    - Incremental: Delta-orientiertes SPL Testmodell

Referenzen (1/2)

- *B. Stevens, E. Mendelsohn:* **Efficient Software Testing Protocols. In: Conference of the Centre of Advanced Studies on Collaborative Research.** *IBM Press, 1998*
- *M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa:* **A Methodology for Controlling the Size of a Test Suite.** ACM Trans. Softw. Eng. Methodol., 2(3):270–285, July 1993.
- *Malte Lochau, Sebastian Oster, Ursula Goltz, and Andy Schürr:* **Model-based Pairwise Testing for Feature Interaction Coverage in Software Product Line Engineering.** Software Quality Journal, pages 1–38, 2011.
- *Harald Cichos, Sebastian Oster, Malte Lochau, and Andy Schürr:* **Model-based Coverage-Driven Test Suite Generation for Software Product Lines.** In 14th International Conference on Model Driven Engineering Languages and Systems (MoDELS), 2011.
- *Malte Lochau, Ina Schaefer, Jochen Kamischke, and Sascha Lity:* **Incremental Model-based Testing of Delta-oriented Software Product Lines.** In 6th International Conference on Tests & Proofs, 2012.
- *Malte Lochau and Ursula Goltz:* **Feature Interaction Aware Test Case Generation for Embedded Control Systems.** ENTCS, 264:37–52, 2010.

Referenzen (2/2)

- *Malte Lochau and Ursula Goltz:* **Feature Interaction Aware Test Case Generation for Embedded Control Systems.** ENTCS, 264:37–52, 2010.
- *Jochen Kamischke, Malte Lochau, and Hauke Baller:* **Conditioned Model Slicing of Feature-Annotated State Machines.** In 4th International Workshop on Feature-Oriented Software Development (FOSD), 2012.
- *Malte Lochau and Jochen Kamischke:* **Parameterized Preorder Relations for Model-based Testing of Software Product Lines.** In 5th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, 2012.
- *Sebastian Oster, Ivan Zorcic, Florian Markert, and Malte Lochau:* **MoSo-PoLiTe - Tool Support for Pairwise and Model-Based Software Product Line Testing.** In 5th International Workshop on Variability Modeling of Software-intensive Systems (VaMoS), 2011.
- *Harald Cichos, Malte Lochau, Sebastian Oster, and Andy Schürr:* **Reduktion von Testsuiten für Software-Produktlinien.** In Software Engineering (SE), pages 143–154, 2012.