# Testfallgenerierung für regelbasierte Übersetzer

Martin Horsch

20. Juli 2005

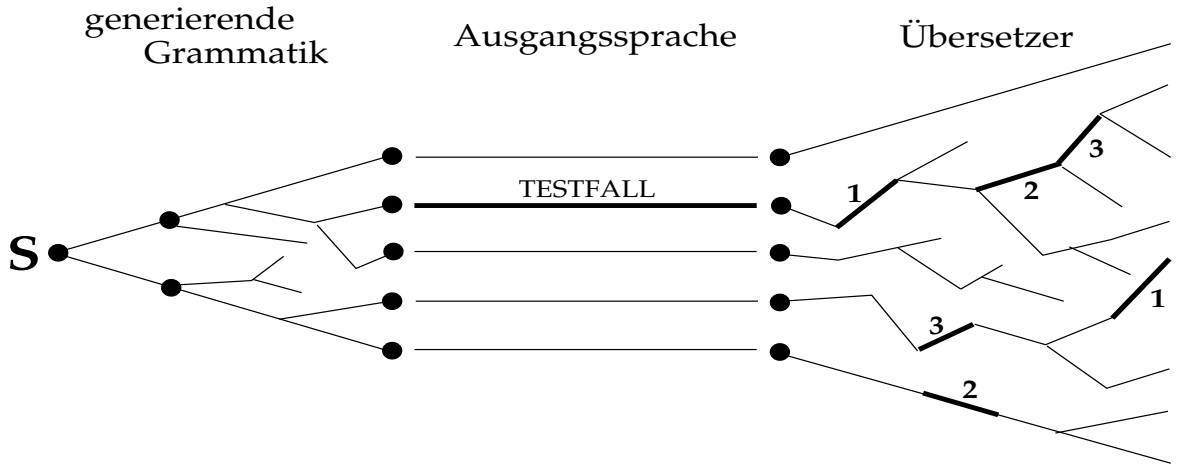
**Übersetzer als Transformationssysteme über Hypergraphen**

Darstellung:
Eingabe und Ausgabe des Übersetzers sind Hypergraphen.

Modell:
Die Ausgangssprache wird von einer (Graph-)Grammatik erzeugt, die *Spezifikation* des Übersetzers ist ein *nichtdeterministisches* Transformationssystem.

Ziel:
Aus der Spezifikation sollen Testfälle erzeugt werden, die es erlauben, bestimmte *Kombinationen* von Ersetzungsschritten auszuführen.

## Vorgehensweise zur Testfallgenerierung




**1.** berechne die Entfaltung (kompakte Darstellung des Zustandsraums) der Grammatiken von Ausgangssprache und Übersetzer.

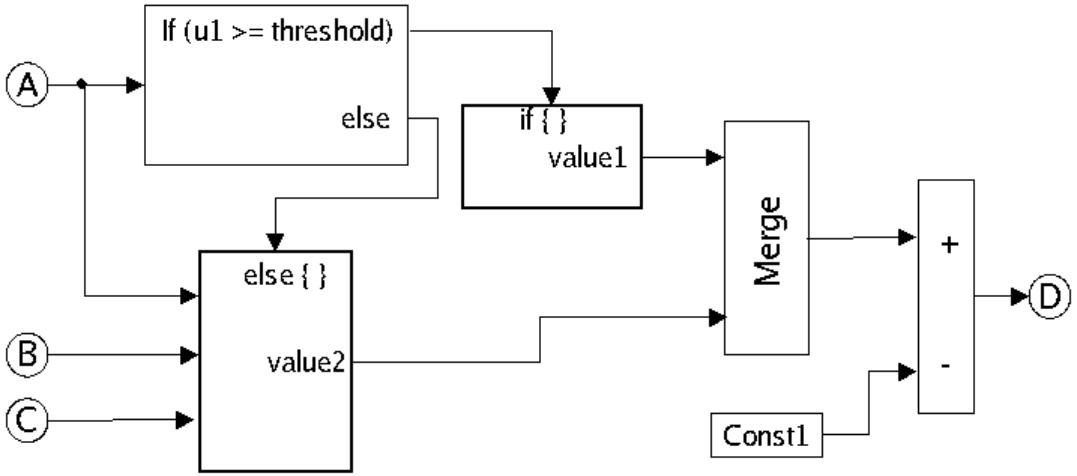
**2.** markiere alle Anwendungen der zu testenden Regeln.

**3.** suche einen Ablauf, in dem alle diese Regeln vorkommen.

**4.** erzeuge das entsprechende Element der Ausgangssprache.

## Beispiel: Codegenerierung aus Simulink-Modellen

Eingabe:
Programmrepräsentation in der Modellierungssprache Simulink

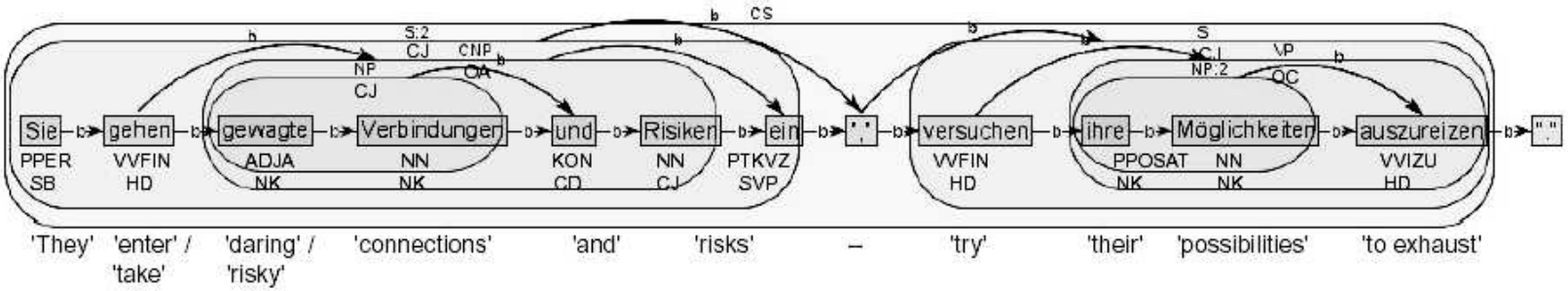



Ausgabe: entsprechender C-Code

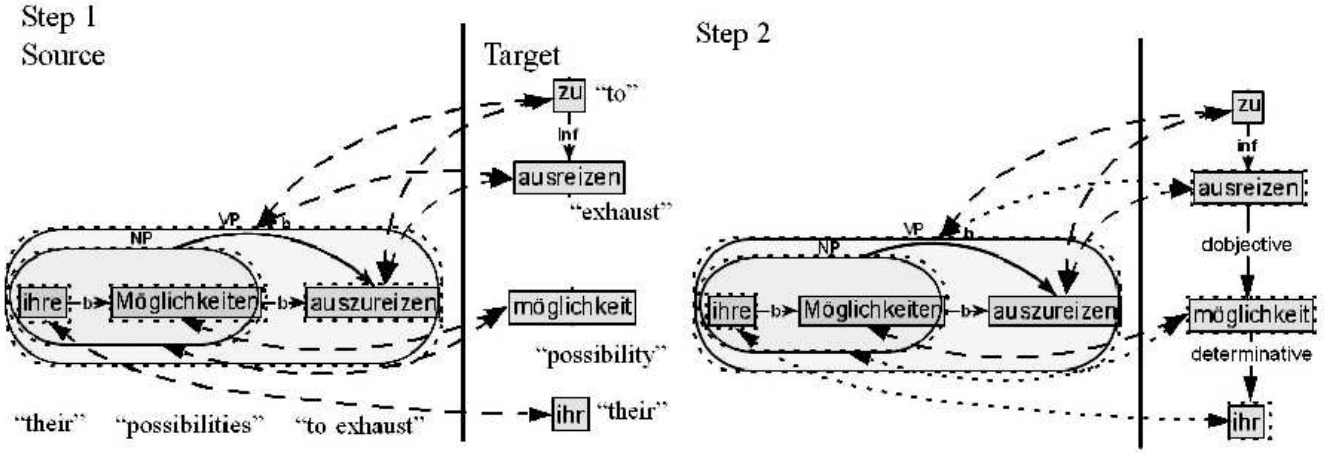
mehr Information unter http://www.mathworks.com/

## Beispiel: Analyse natürlichsprachiger Phrasenstrukturen

Eingabe: Phrasenstruktur **PS**




Ausgabe: Dependenzstruktur **DS**

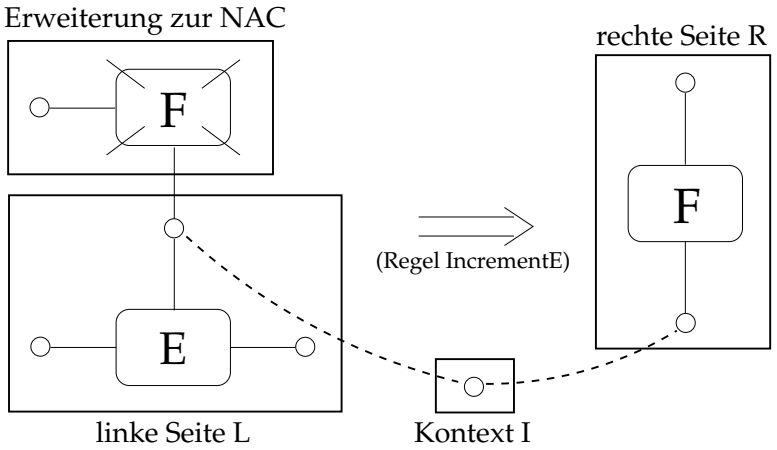



Quelle: Bernd Bohnet (IIS), Mapping Phrase Structures to Dependency Structures . . . , 2003.

## Graphersetzungsregeln für kantenbeschriftete Hypergraphen

$$r = (I, L, R, \varphi_L, \varphi_R, \nabla)$$

$\varphi_L : I \to L$ und $\varphi_R : I \to R$ sind injektive *Graphmorphismen*.
$\nabla$ ist eine Menge *negativer Anwendungsbedingungen* (NAC).




NAC: L mit *einer* zusätzlichen Inhibitorkante
Graphentransformationssystem: Menge von Ersetzungsregeln
Graphgrammatik: GTS mit Startgraph

## Testfälle für regelbasierte Übersetzer

Anwendung der Regelsequenz $s$ auf Graph G mit Ergebnis H:

$$G \Rightarrow_s H$$

betrachtetes System:
Startgraph $S$, *generierende* Regeln $P_0$, *übersetzende* Regeln $P_1$

Die beiden GTS werden *sequentiell* angewandt.

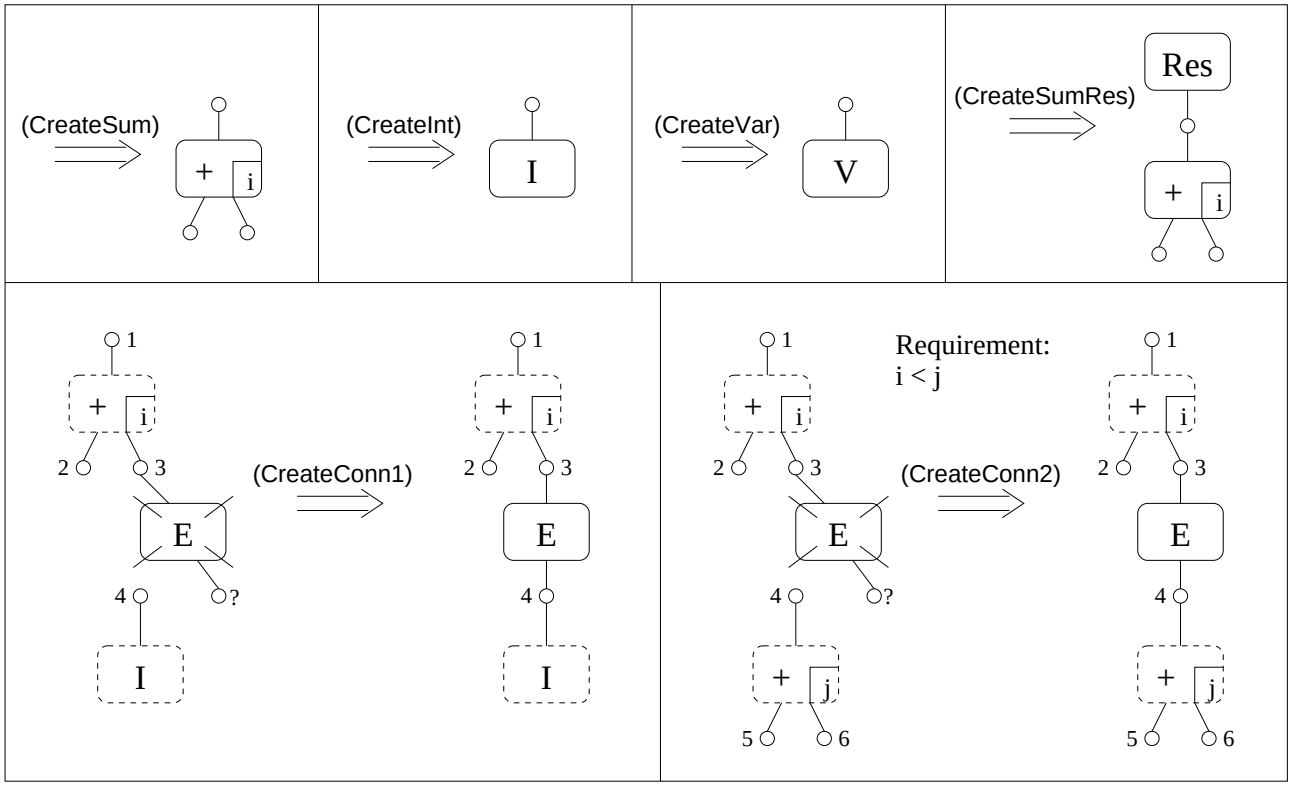
Die Regelmenge $P_T \subseteq P_1$ des Übersetzers soll getestet werden.
Graph C ist ein *Testfall* für $P_T$, gdw. $\exists s \in P_0^*$, $s' \in P_1^*$:

$$\forall r \in P_T : | s' |_r \geq 1 \quad \wedge \quad S \Rightarrow_s C \Rightarrow_{s'} .$$

Der Testfall C erlaubt die Anwendung aller zu testenden Regeln.

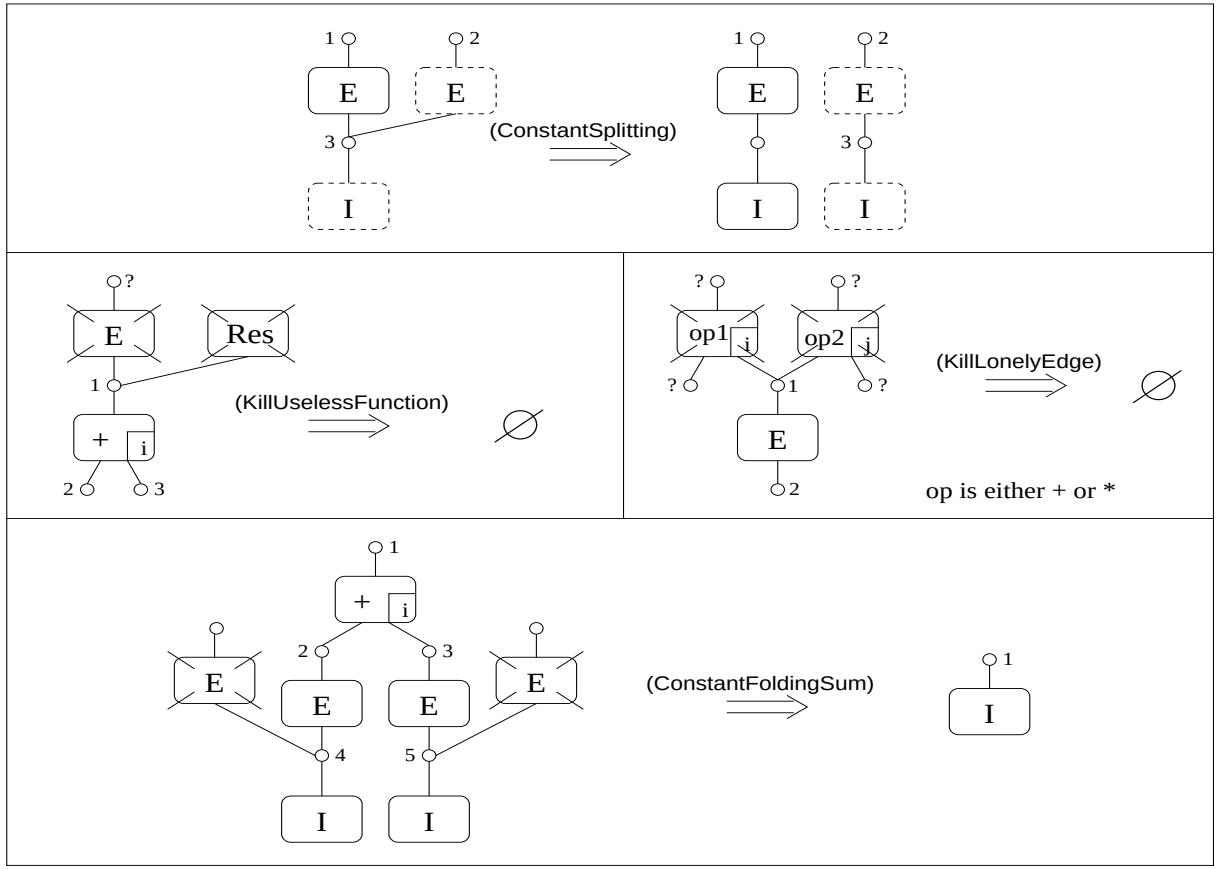
# Beispiel: Vereinfachung arithmetischer Ausdrücke

Startgraph ist der leere Hypergraph.
Generierende Regeln:

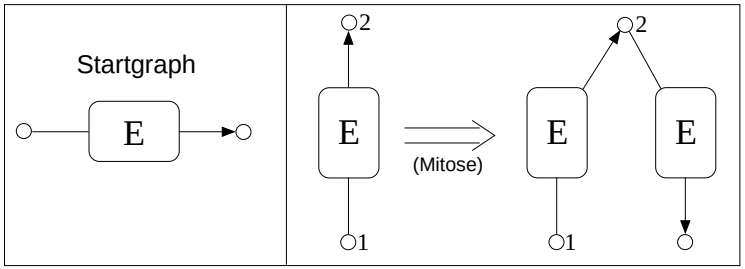


# Beispiel: Vereinfachung arithmetischer Ausdrücke

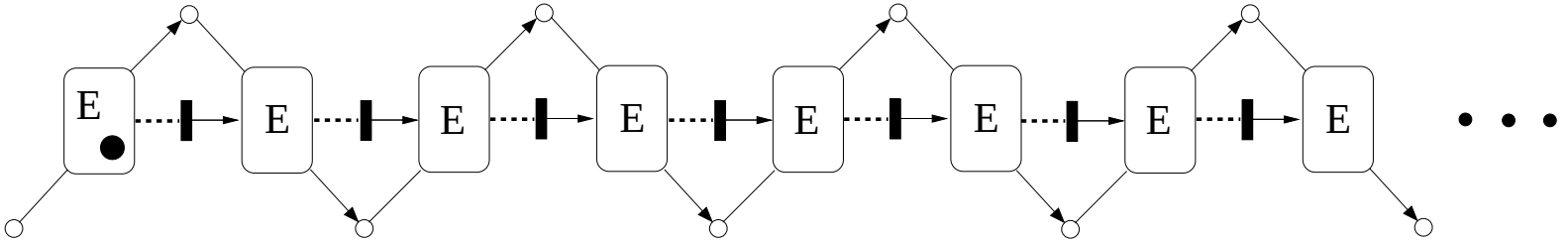
Übersetzende Regeln:



Die **Entfaltung einer Graphgrammatik** ist ein *Petri-Graph* $(H, N)$: Hypergraph $H$, dessen *Kanten* die *Stellen* des P/T-Netzes $N$ sind.




Feuersequenz von $N$ $\equiv$ Regelsequenz der Grammatik



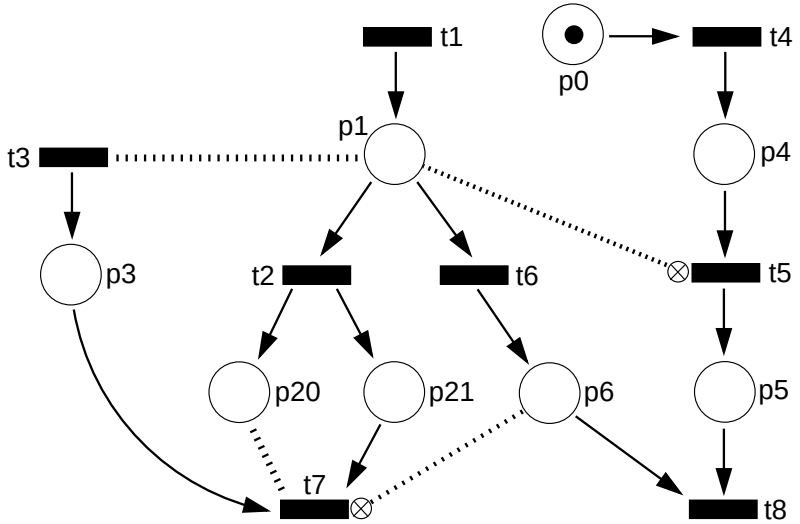

Markierung von $N$ $\equiv$ Subgraph von $H$

Kontext und den NAC einer Regel entsprechen *Lese-* und *Inhibitor-kanten*. Für unendliche Entfaltungen berechnen wir ein *endliches Präfix*. Einschränkungstypen: *depth* und *width* restriction

## Kontextabhängige P/T-Netze

. . . sind Tupel $N = (S, T, {}^\bullet(\cdot), (\cdot)^\bullet, \underline{\underline{(\cdot)}}, \multimap \cdot)$

mit Stellen $S$, Transitionen T und vier Abbildungen $T \to 2^S$ für Preset ${}^\bullet t$, Postset $t^\bullet$, *Kontext* $\underline{\underline{t}}$ und *Inhibitorstellen* $\multimap t$ von $t \in T$.

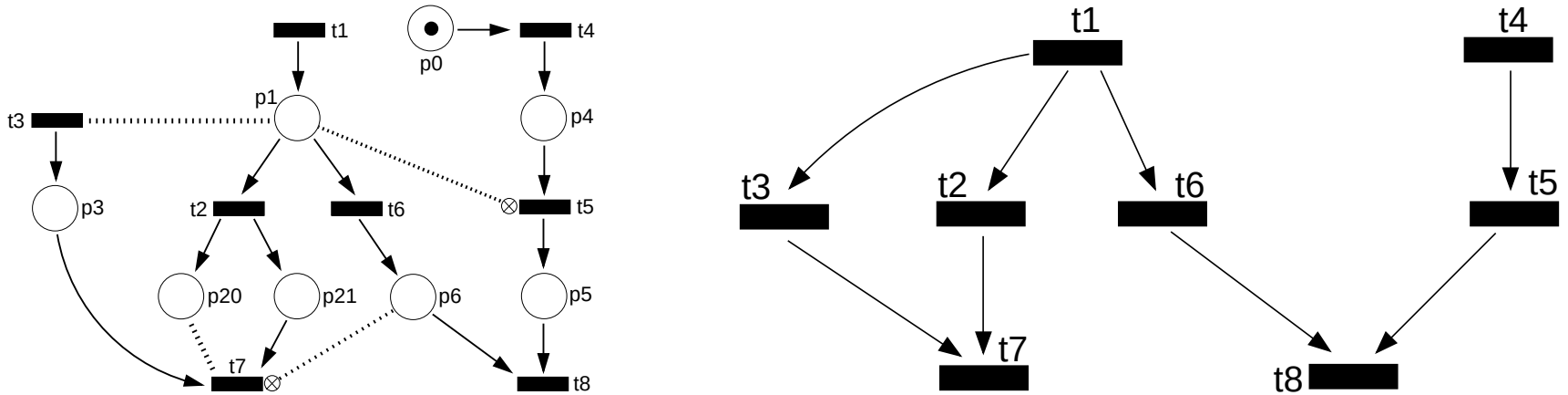



Eine Transition t kann feuern, wenn alle Stellen aus ${}^\bullet t \cup \underline{\underline{t}}$ und keine aus $\multimap t$ markiert sind.

## Abhängigkeit zwischen Transitionen

Die *Kausalitätsrelation* $(\cdot < \cdot) \subseteq T \times T$
ist die kleinste transitive Relation mit der Eigenschaft

$$\forall t, \tau \in T: \quad t^\bullet \cap ({}^\bullet\tau \cup \underline{\underline{\tau}}) \neq \emptyset \;\Rightarrow\; t < \tau.$$




Die *Ursache* von $\tau \in T$ ist $\lfloor \tau \rfloor = \{t \in T \mid t < \tau\} \cup \{\tau\}$.

Der (asymmetrische) *Konflikt* zwischen Transitionen ist definiert durch $t \nearrow \tau \iff ({}^\bullet t \cup \underline{\underline{t}}) \cap {}^\bullet\tau \neq \emptyset$.

## Occurrence Nets

Ein kontextabhängiges P/T-Netz $N = (S, T, {}^\bullet(\,\cdot\,), (\,\cdot\,)^\bullet, \underline{\underline{(\,\cdot\,)}}, \multimap\,\cdot)$
ist ein occurrence net, falls es die folgenden Bedingungen erfüllt:

1) N ist (1-)sicher.
2) die Kausalitätsrelation $<$ von N ist azyklisch.
3) alle Stellen $s \in S$ haben Eingangsgrad $0$ oder $1$.
4) genau die Stellen mit Eingangsgrad $0$ sind anfangs markiert.

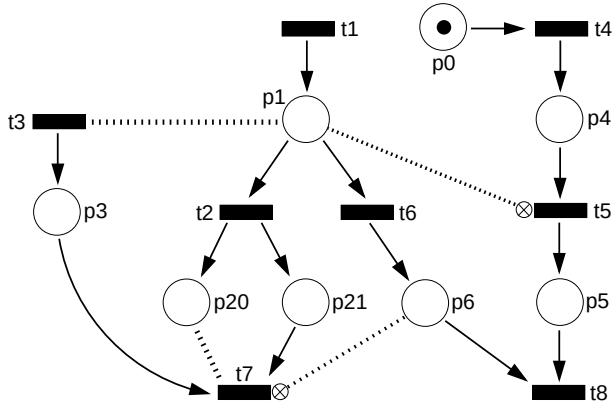
Für Transitionen $t$ und $\tau$ eines occurrence net gilt:
$t < \tau \implies \tau$ kann *nur nach* $t$ gefeuert werden.
$t \nearrow \tau \implies t$ kann *nicht nach* $\tau$ gefeuert werden.

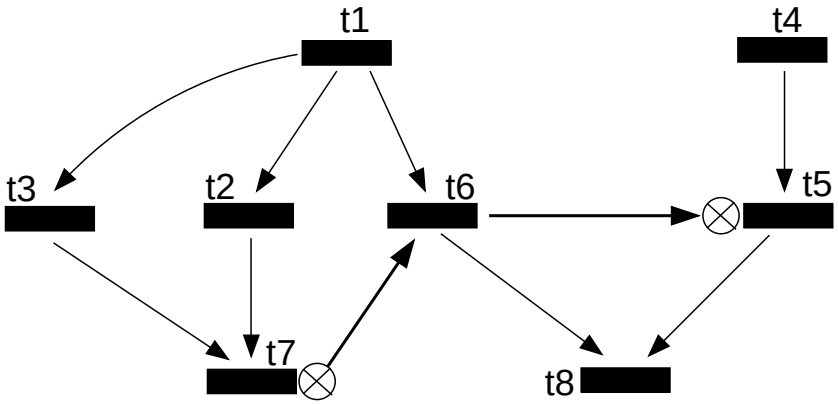
Sei N ein beliebiges Netz und $N'$ ein occurrence net, das die gleichen Feuersequenzen zulässt wie N. Dann ist $N'$ eine *Entfaltung* von N.

## Konfigurationen

Eine Konfiguration $(\cdot <_{\mathbf{C}} \cdot) \subseteq \mathrm{T} \times \mathrm{T}$ eines occurrence net erfüllt die Eigenschaft, dass $(\cdot < \cdot) \cup (\cdot <_{\mathbf{C}} \cdot)$ azyklisch ist und für alle $s \multimap \mathrm{t}$:

$$({}^\bullet s = \{{}'s\} \ \wedge \ \tau <_{\mathbf{C}} {}'s) \ \ \vee \ \ \exists \mathrm{t} \in s^\bullet : \ \mathrm{t} <_{\mathbf{C}} \tau \qquad \textit{(Konfiguration von t)}$$

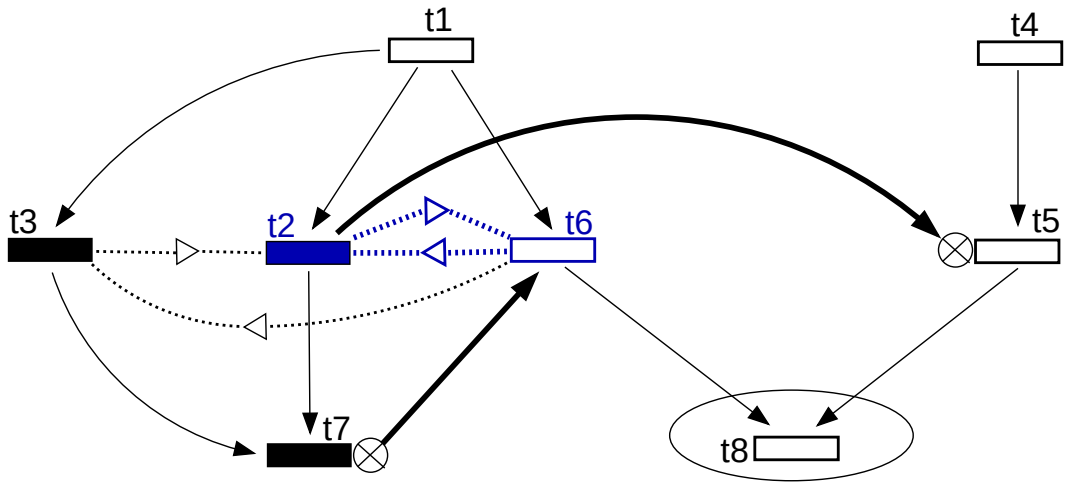



Die Anzahl der Konfigurationen wächst i.A. exponentiell mit der Zahl der Inhibitorkanten. (Hier gibt es 6 verschiedene.)

Eine Menge von Transitionen kann in der Konfiguration C nicht gefeuert werden, wenn sie einen Zyklus von $(\nearrow \cup < \cup <_C)$ enthält.

$\lfloor \tau \rfloor_C$, die C-Ursache von $\tau$, ist die kleinste Menge mit **a)** $\lfloor \tau \rfloor \subseteq \lfloor \tau \rfloor_C$, die **b)** für jedes ihrer Elemente dessen Ursache und **c)** für jede Konfiguration $t <_C t'$ *eines ihrer Elemente* mit $t'$ auch $t$ enthält.




$\lfloor t_8 \rfloor = \{t_1, t_4, t_5, t_6, t_8\}$ und $\lfloor t_8 \rfloor_C = \lfloor t_8 \rfloor \cup \{t_2\}$.
$t_8$ kann in dieser Konfiguration nicht gefeuert werden.

## Aufgabenstellung der Studienarbeit

**1)** Dokumenttyp für Graphgrammatiken um NAC erweitern

Grundlage: *GraphCondition*-Element der GTXL DTD

**2)** NAC in einer Entfaltung als Inhibitorkanten berücksichtigen

Grundlage: Entfaltung einer Graphgrammatik ohne NAC durch *aunfold*

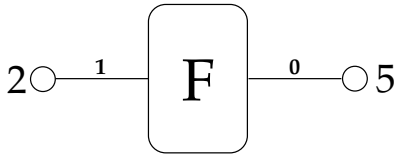
**3)** Konfigurationen der Entfaltung berechnen

Grundlage: *Model Checking Kit* für sichere Petrinetze

**4)** Implementierung in C++

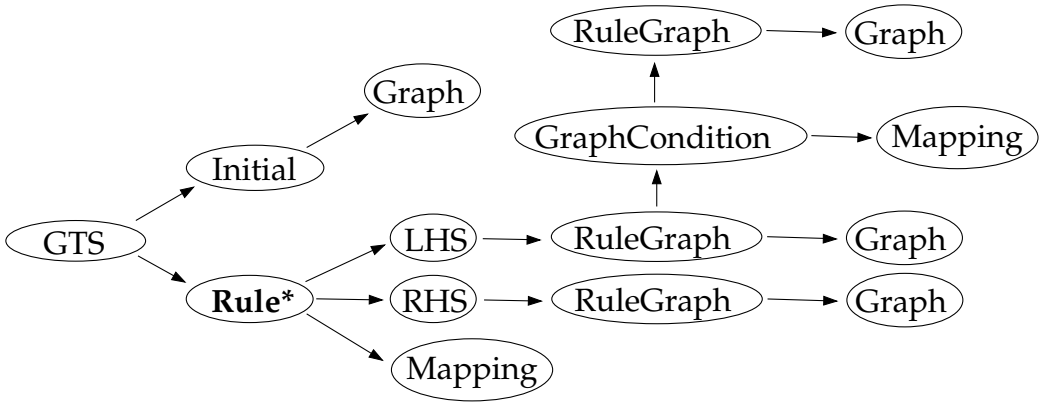
Grundlage: Code für *AUGUR I*

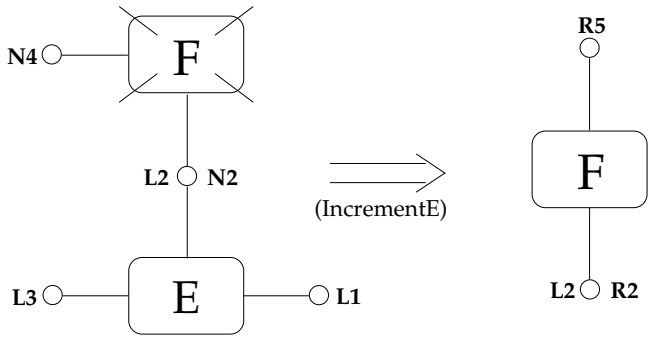
## **G**raph e**X**change **L**anguage: Hypergraphen und Petri-Graphen




```
<Graph hypergraph="true" edgemode="undirected">
     <node id="2"/><node id="5"/>
     <rel><attr name="label"><string>F</string></attr>
          <relend target="5" startorder="0"/>
          <relend target="2" startorder="1"/>
     </rel>
</Graph>
```

## **G**raph **T**ransformation e**X**change **L**anguage: Graphgrammatiken

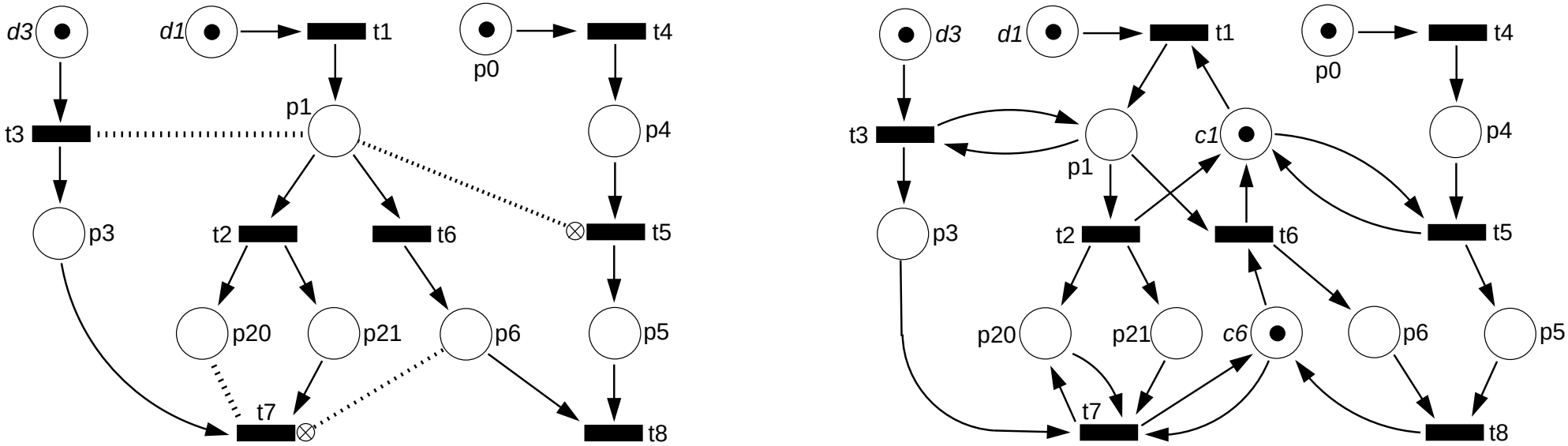





```
<Rule id="IncrementE">
     <LHS>
          <RuleGraph>
               <Graph hypergraph="true" edgemode="undirected">
                    <node id="L1"/><node id="L2"/><node id="L3"/>
                    <rel><attr name="label"><string>E</string></attr>
                         <relend target="L1"/><relend target="L2"/><relend target="L3"/></rel>
               </Graph>
               <GraphCondition>
                    <RuleGraph><Graph hypergraph="true" edgemode="undirected">
                         <node id="N2"/><node id="N4"/>
                         <rel><attr name="label"><string>F</string></attr>
                              <relend target="N2"/><relend target="N4"/></rel>
                    </Graph></RuleGraph>
                    <Mapping><MapElem from="L2" to="N2"/></Mapping>
               </GraphCondition>
          </RuleGraph>
     </LHS>
     <RHS><RuleGraph><Graph ... /Graph></RuleGraph></RHS>
     <Mapping><MapElem from="L2" to="R2"/></Mapping>
</Rule>
```

## Konvertierung in ein einfaches Petri-Netz

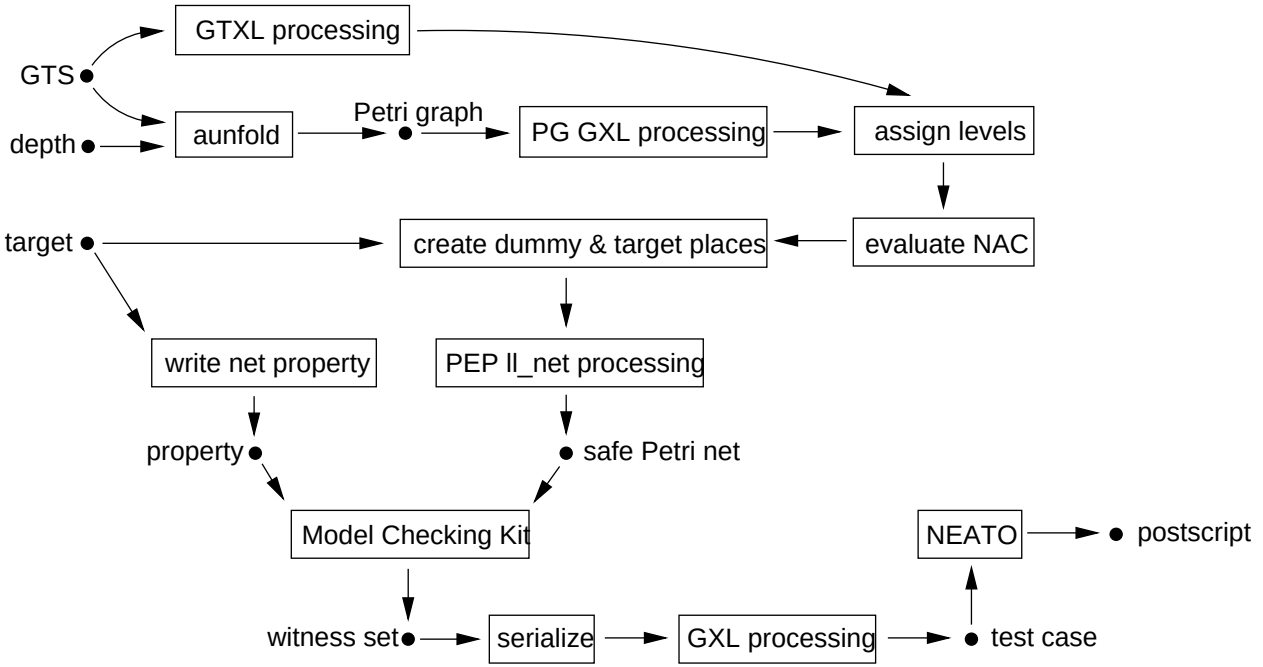
Wir ersetzen Lesekanten durch *write-back arcs* und Inhibitorkanten durch write-back arcs zu neuen *Komplementärstellen*:




Für sichere Netze sind das kontextabhängige Original und das konvertierte Netz äquivalent.

## Programmfluss des implementierten Testfallgenerators

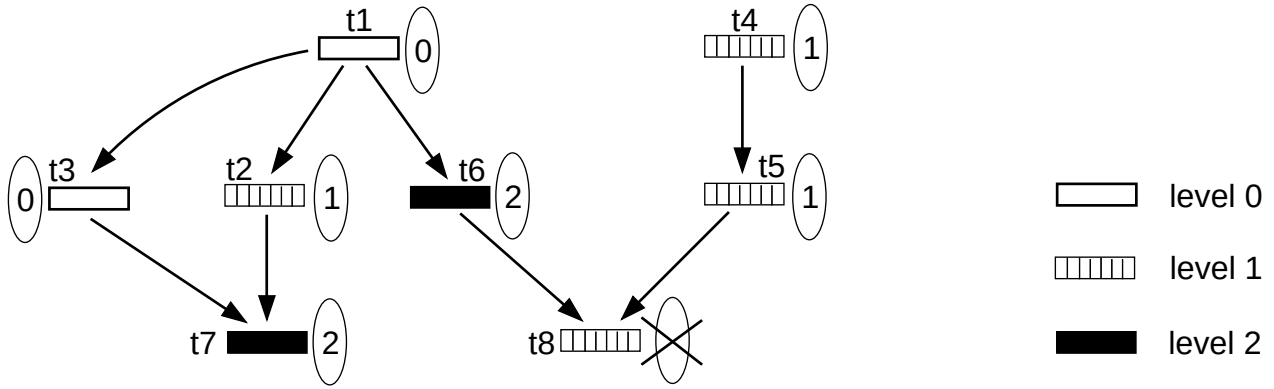
Der AUGUR test case generator **atcg** ruft die externen Komponenten *aunfold*, *check*, *pg2neato* und *neato* auf . . .




. . . und erzeugt einen Testfall in GXL und postscript.

## Beschriftung der Transitionen mit Levels

Levels der Ersetzungsregeln entsprechen der Abfolge der GTS.




Transitionen werden entfernt, wenn in ihrer Ursache ein höheres Level als das ihrer eigenen Regel vorkommt. Das genügt aber nicht, um sicherzustellen, dass die Transitionen auch in der Reihenfolge ihrer Levels gefeuert werden.

**weitere Bearbeitung der Entfaltung**

1. nach Auftreten der NAC suchen und entsprechend Inhibitorkanten ergänzen (die Entfaltung wird mit aunfold erzeugt, ohne NAC zu berücksichtigen)

2. leere Presets um je eine *Dummy-Stelle* ergänzen

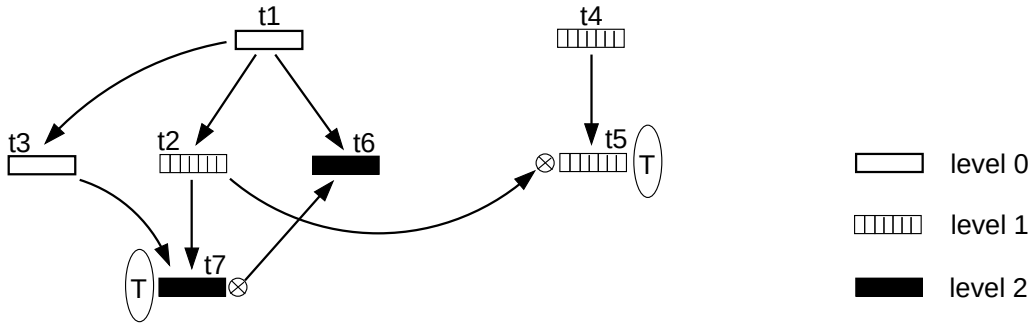
3. für jede zu testende Regel eine *Ziel-Stelle* ergänzen, die in das Postset jeder entsprechenden Transition aufgenommen wird

Intern wird das (theoretisch vorgesehene) kontextabhängige P/T Netz direkt als das äquivalente Petri-Netz ohne Lese- und Inhibitorkanten gespeichert. Weitere Arbeitsschritte: Export in das PEP llnet Format und Übergabe an das Model Checking Kit.

## Auswertung der erhaltenen Feuersequenz

Der Aufruf des *Model Checking Kit* mit dem Algorithmus *mcs-check* ergibt ggf. eine Feuersequenz, die alle Ziel-Stellen markiert. Die so erhaltene Markierung des Graphen ist der gesuchte Testfall.

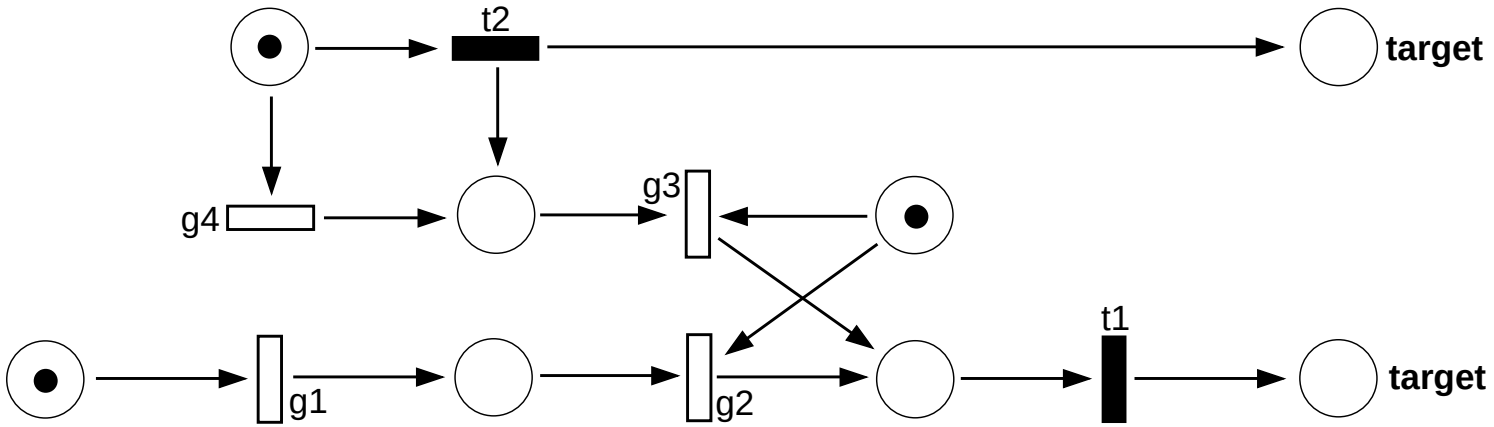
Falls die Sequenz nicht der Abfolge von *generierendem* und übersetzendem GTS entspricht, kann sie evtl. umsortiert werden.




unzulässige Feuersequenz: $\mathbf{t_4 - t_1 - t_3 - t_2 - t_7 - t_5}$
zulässige Serialisierung: $\mathbf{t_1 - t_3 - t_4 - t_2 - t_5 - t_7}$

## Nicht serialisierbare Feuersequenzen

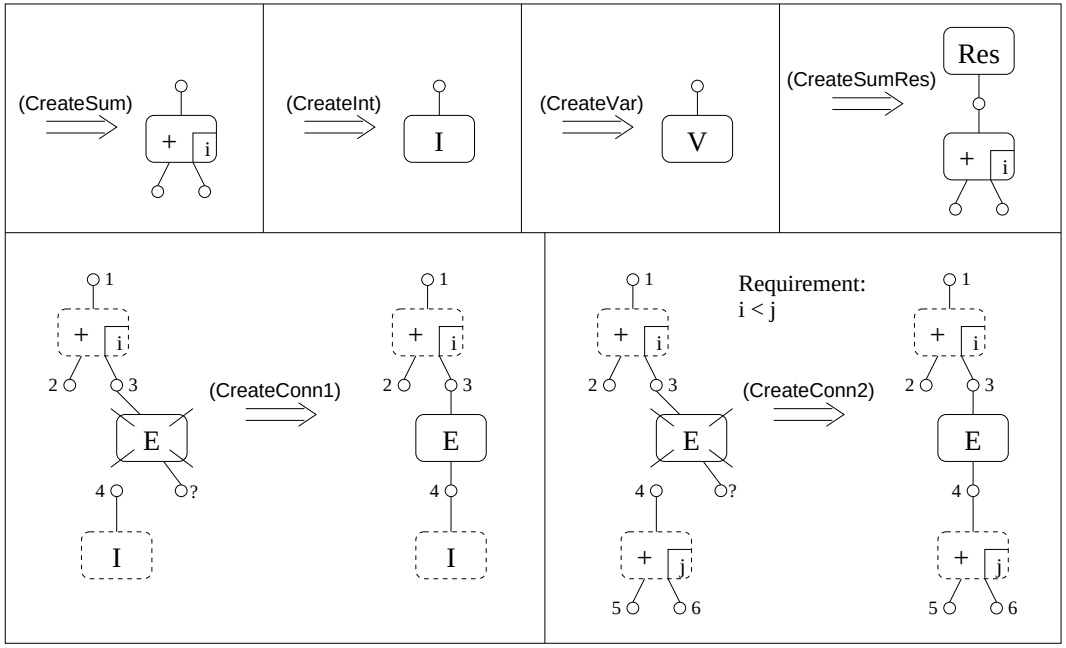
Bei diesem Verfahren kann es vorkommen, dass ein Testfall für bestimmte Regelmengen übersehen wird:

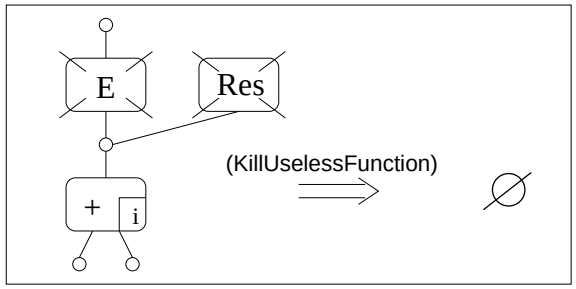



richtige Lösung: $\mathbf{g_1 - g_2 - t_1 - t_2}$
kürzere, aber *nicht serialisierbare* Sequenz: $\mathbf{t_2 - g_3 - t_1}$

# Vereinfachung arithmetischer Ausdrücke

Startgraph ist der leere Graph.




Suche einen Testfall für die Regel **KillUselessFunction**.

## Schlussfolgerungen über den verfolgten Ansatz

Vorteile:
(+) systematische, schrittweise Vorgehensweise
(+) Einsatz bestehender Komponenten (aunfold, check, . . . )
(+) modulare Struktur erlaubt es, Komponenten auszutauschen.

Nachteile:
(−) kein Einsatz von Heuristiken
(−) durch den Konvertierungsschritt geht Information verloren.
(−) depth und width müssen im Voraus festgelegt werden.
(−) hoher Speicherbedarf (z.B. im Vergleich zu Backtracking)
(−) geringer Informationsfluss zwischen den Komponenten