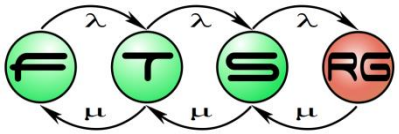
Modellierungsumgebungen, Kodegenerierung

Budapest University of Technology and Economics
Fault Tolerant Systems Research Group

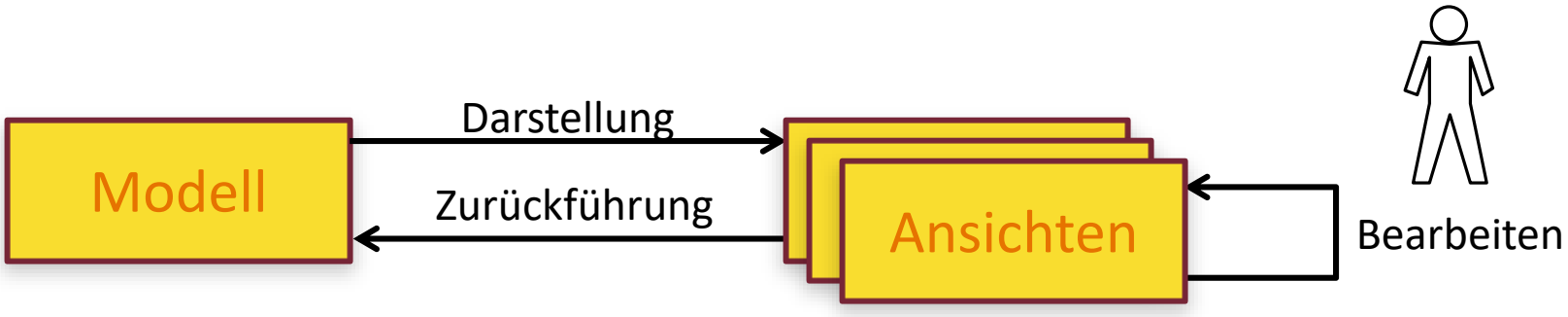


Inhalt

- **Funktionen**
- **Modellierungsumgebungen**
- **Kodegenerierung**

DIE FUNKTIONEN EINER MODELLIERUNGSUMGEBUNG

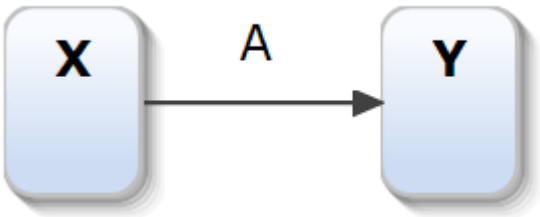
Funktionen einer Modellierungsumgebung



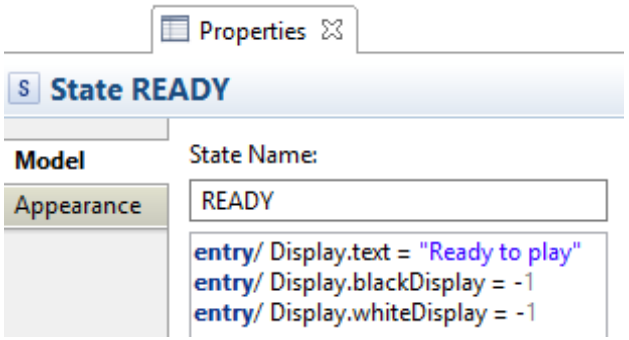
Textuell

Demo
internal: event A
event B

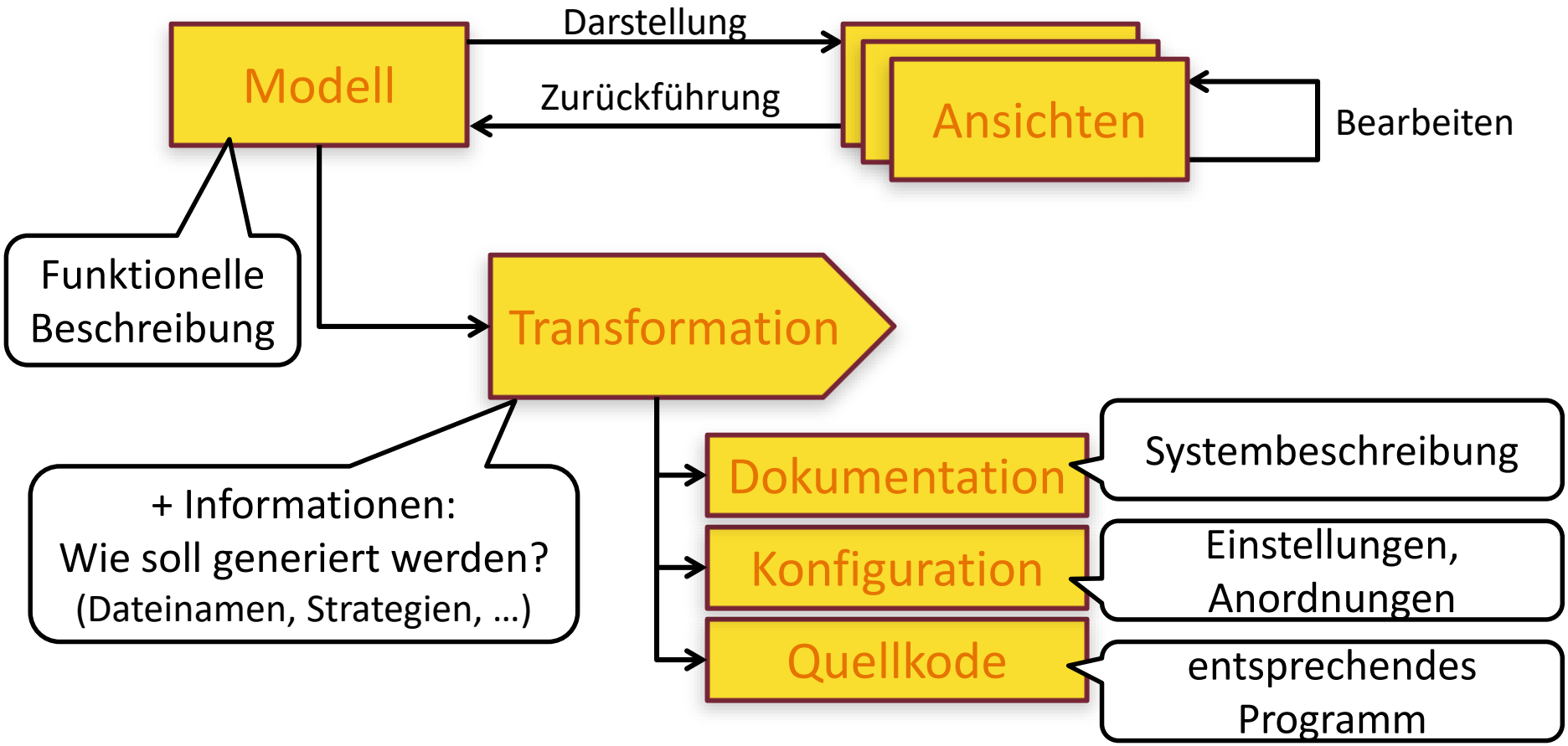
Graphisch



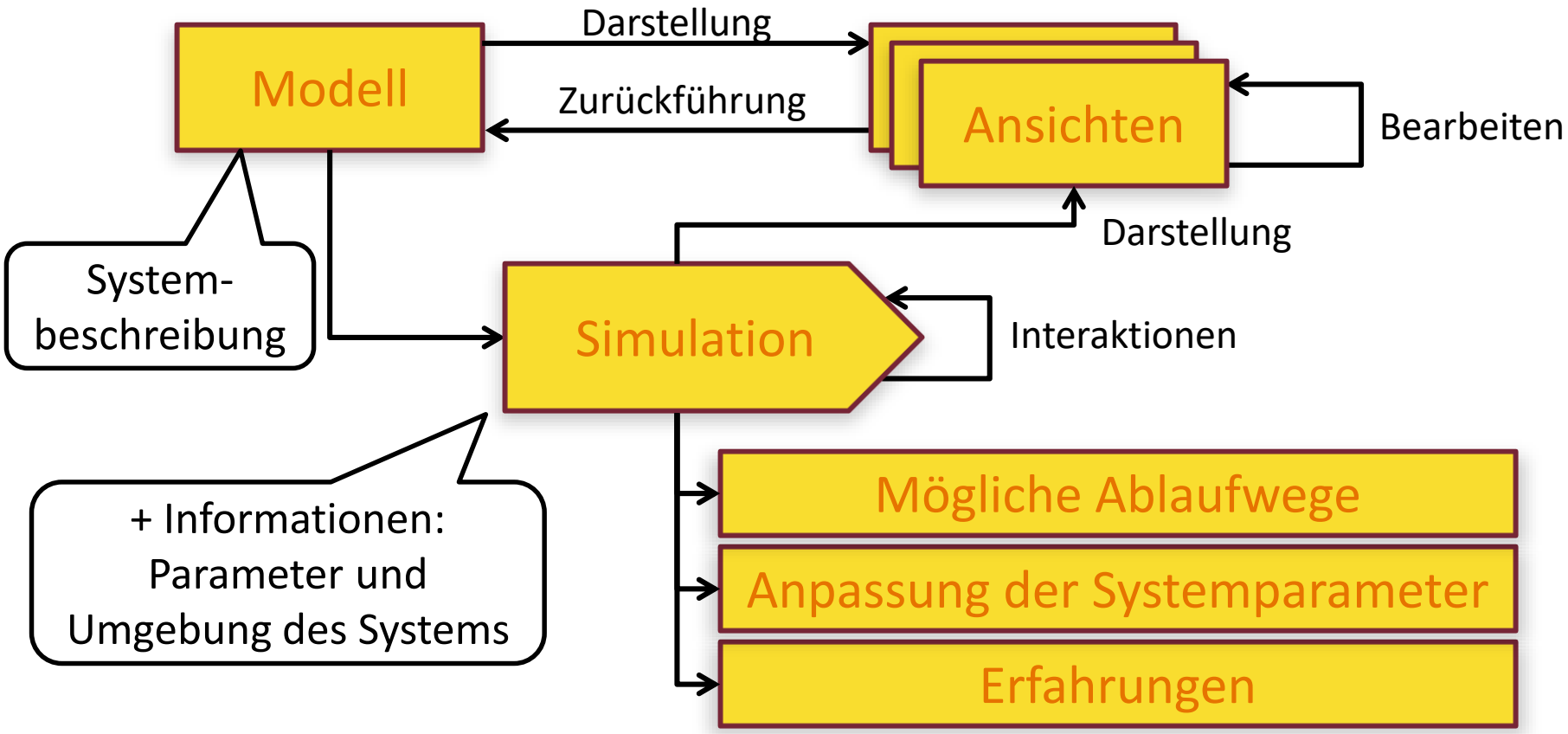
Strukturierte Schnittstellen



Funktionen einer Modellierungsumgebung



Funktionen einer Modellierungsumgebung

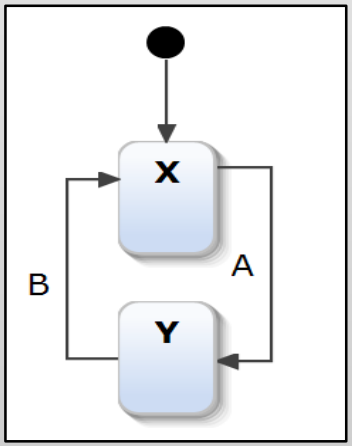


MODELLIERUNGSUMGEBUNGEN

Modellierungsfunktionen von Yakindu

Konkrete Syntax
(für den Benutzer)

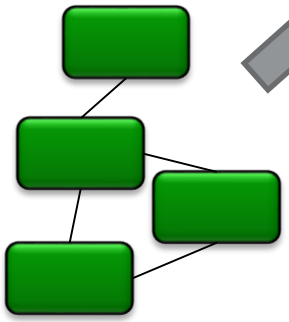
Graphisch:



Textuell:

```
Demo
internal:
event A
event B
```

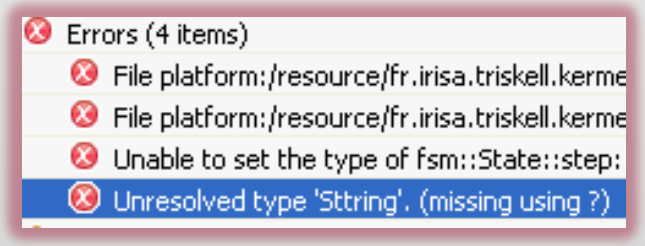
Syntax → Semantik



Modell
(Abstrakte Syntax)

Modellierungsfunktionen

Modellüberprüfung

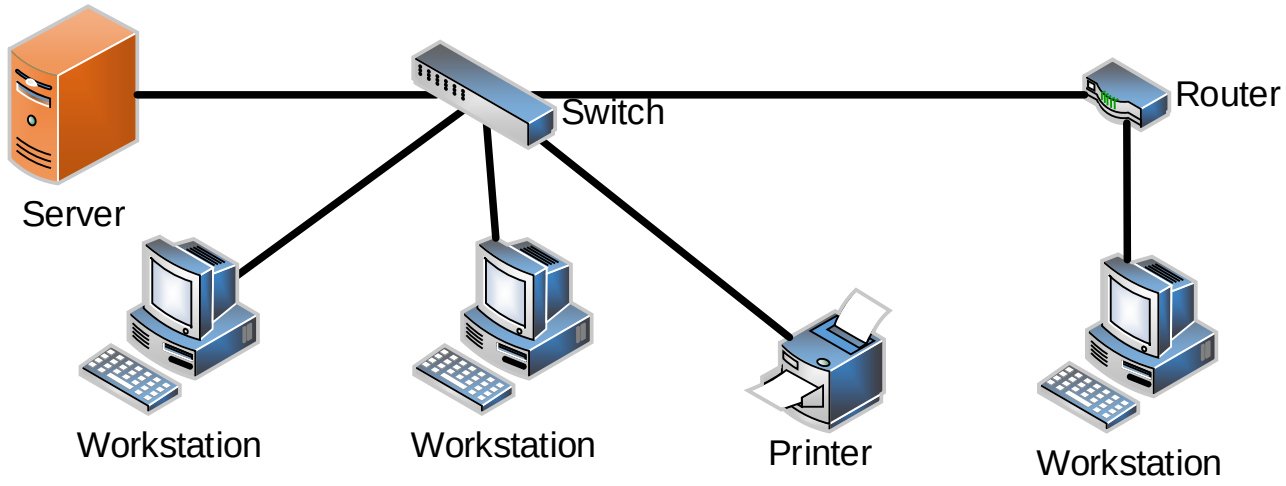


Kodegenerierung

(Quellenkode, Dokumentation,
Konfiguration)

Abstrakte Syntax

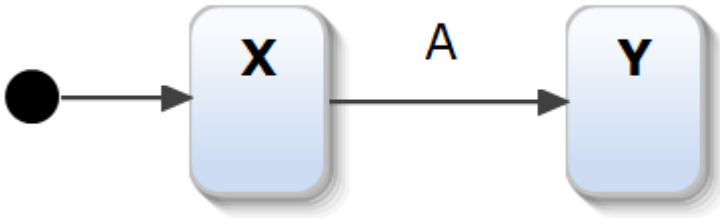
- **Definition:** Strukturelles Modell des zu editierenden Systemmodells
  - Strukturelles Modell eines Modells ???
- Wird von der Modellierungsumgebung verwaltet
- Zur Erinnerung: Strukturelles Modell = **Graph**
  - **Graph von: Knoten, Kanten, Eigenschaften**



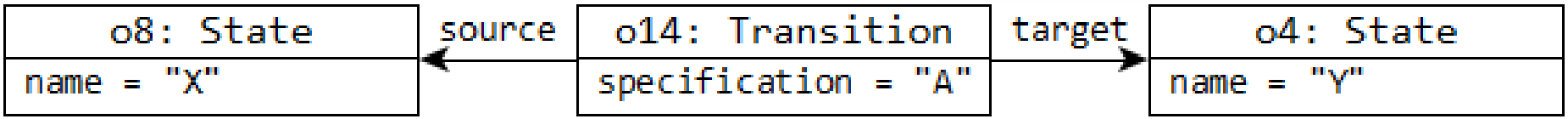
Beispiel – Abstrakte Syntax: Yakindu

Frage:
Wie würden wir eine Modelleirungsumgebung implementieren?

Beispiel: Yakindu Modell



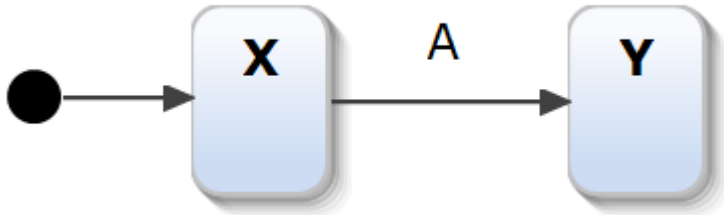
Abstrakte Syntax



Beispiel – Abstrakte Syntax: Yakindu

Frage:
Wie würden wir eine Modelleirungsumgebung implementieren?

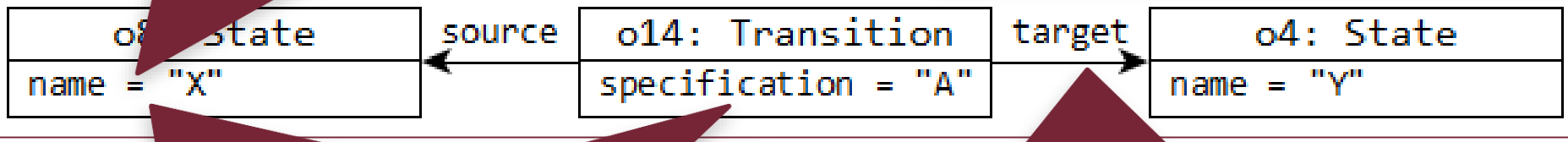
Beispiel: Yakindu Modell



Namen werden als String gespeichert

```
name = "X"
```

Abstrakte Syntax



Modellelemente als Objekte

Relationen als Referenzen

Antwort: Es ist ein einfaches objekt-orientiertes Program mit Extrafunktionen

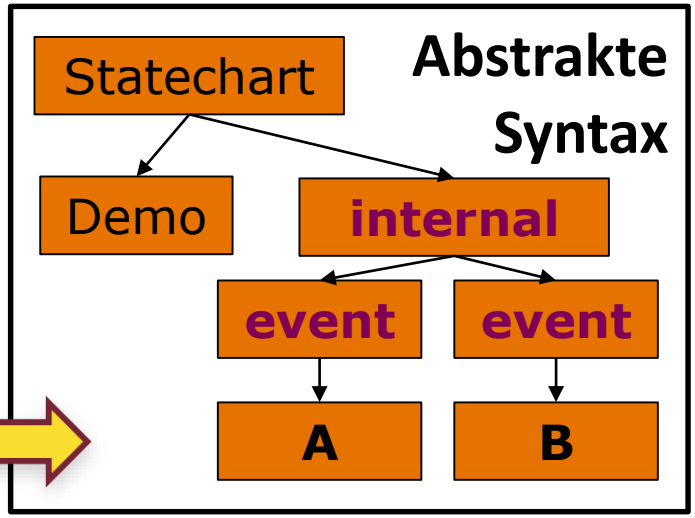
Konkrete Syntax: Textuelle Syntax

- **Ziel:** Repräsentation⇔ das Modell dahinter
- Textuelle Syntax (z.B. Programme)
  - Aufgabe: Text → Modell
  - Regelbasiert (sonst wird es schwierig!)

```
Demo
internal:
event A
event B
```

Grammatik

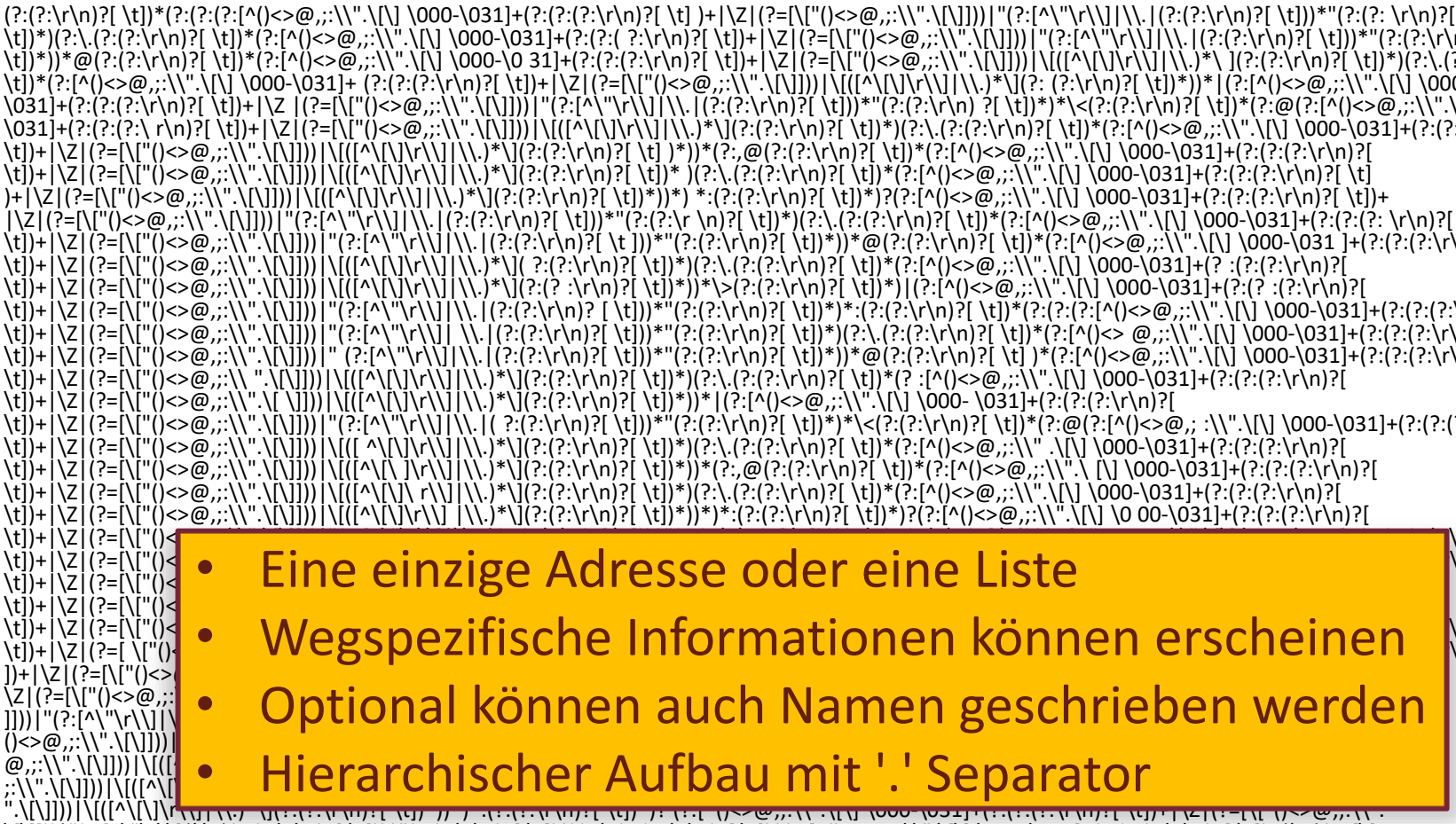
```
<Statechart>::= <Name> <Interface>*
<Interface> ::= ("internal" | <Name>)
                ":" <Event>*
<Event>     ::= "event" <Name>
<Name>      ::= …
```



Mit **geeigneten** Technologien (z.B. Xtext) kann **jeder** eine **eigene** Modellierungs-/Programmierungssprache implementieren!

Gegenbeispiel für „geeigneten Methoden”

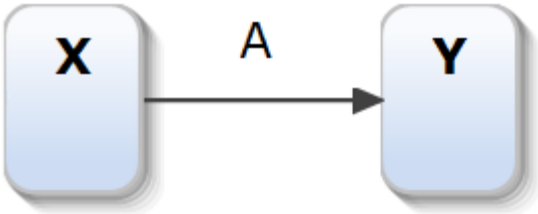
- **Aufgabe:** entscheiden ob ein Text eine Emailadresse ist



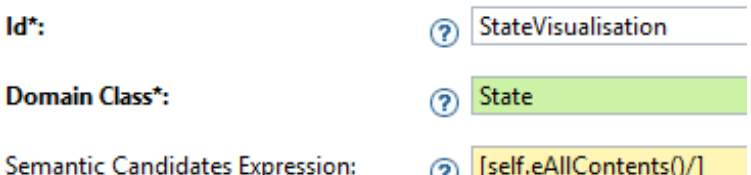
- Eine einzige Adresse oder eine Liste
- Wegspezifische Informationen können erscheinen
- Optional können auch Namen geschrieben werden
- Hierarchischer Aufbau mit '.' Separator

Konkrete Syntax: Graphische Syntax

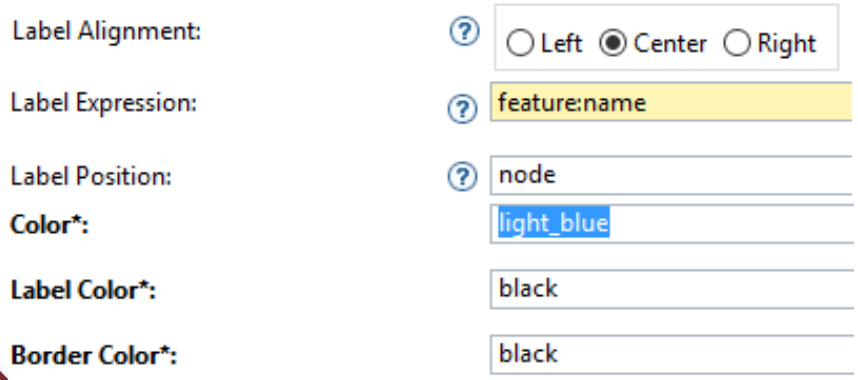
- **Ziel:** Repräsentation⇔ das Modell
- Graphische Syntax (z.B. Diagramm)
  - Aufgabe: Diagramm ↔ Modell
  - Übersichtlicher, schwieriger zu schreiben, regelbasiert



Kondition auf dem Modell



Anlegen des Diagrammelementes

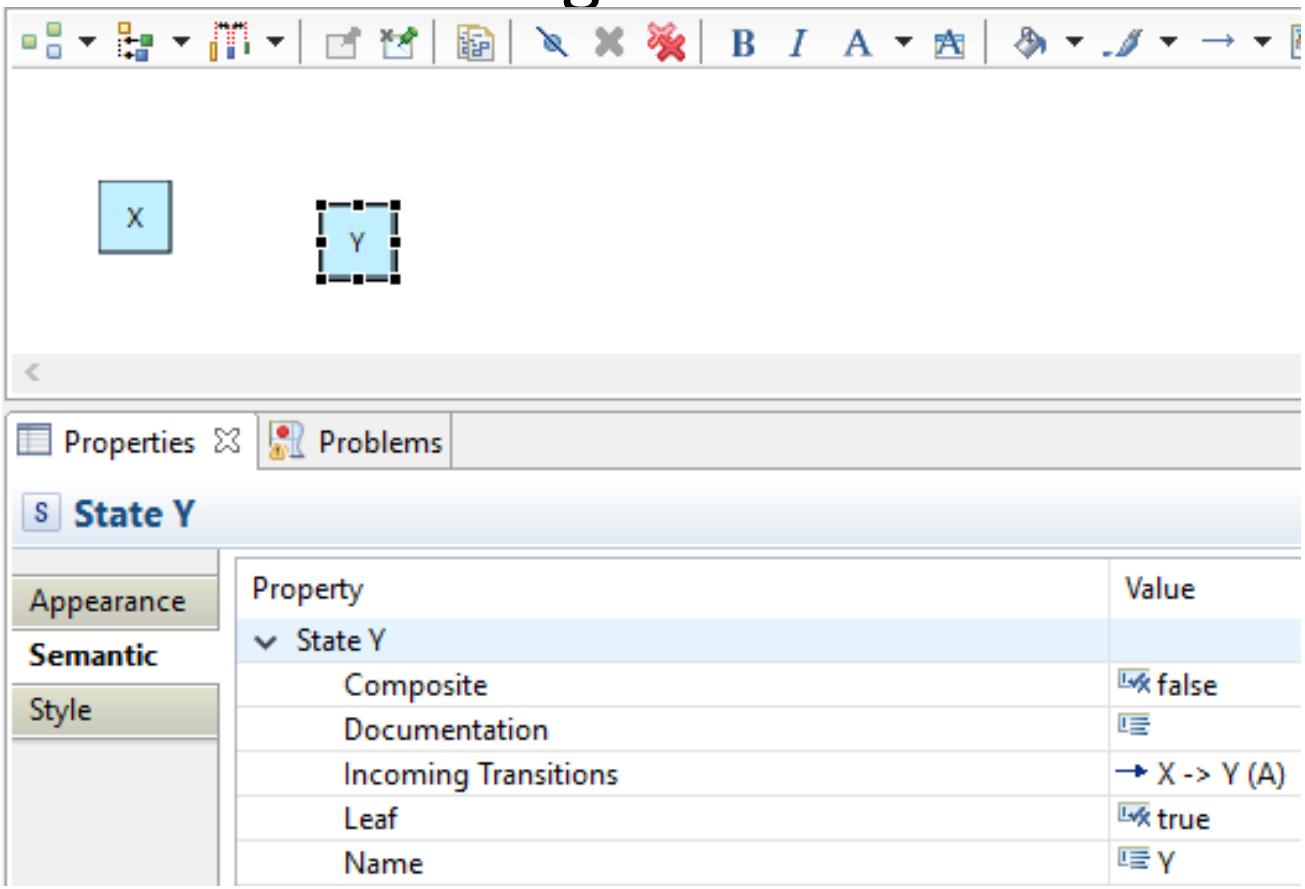


Kondition erfüllt → Diagrammelement wird angelegt

Diagramm wird geändert → Modell ändert sich auch

Konkrete Syntax: Graphische Syntax

Ergebnis:



Mit **geeigneten** Technologien (z.B. Sirius) kann **jeder** eine **eigene** Modellierungs-/Programmierungssprache implementieren!

Modellvalidierung: Syntaktische Überprüfung

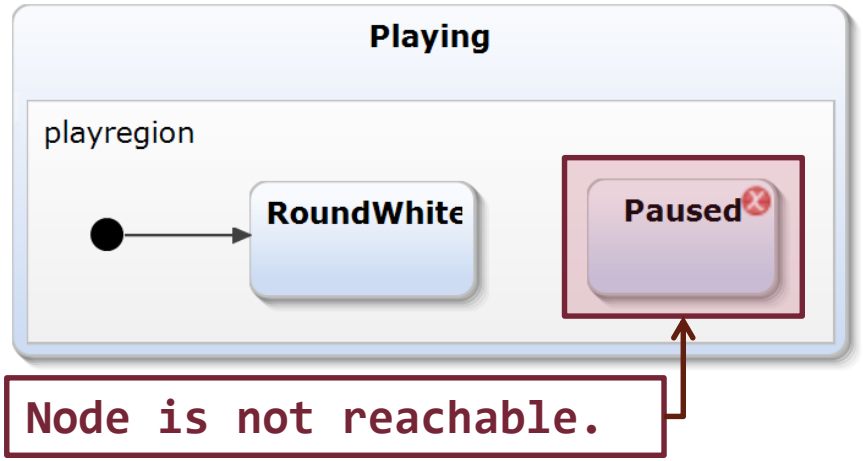
- Syntaktische Überprüfung: die Modellierungsumgebungen verbinden die logisch zusammengehörenden Elemente

- Syntaxgesteuerte Editoren
  - Fehler während Bearbeitung→ `Couldn't resolve reference`
  - Moderne Umgebungen: Vorschlagen der Kandidaten
- Kode+Diagram gemeinsam

- Programmieren: **fehlerhaft** während der Bearbeitung
  Modellieren: **korrekt** während der Bearbeitung

Modellvalidierung : Strukturelle Überprüfung

- Strukturelle Überprüfung: Untersuchung des Modellgraphen
- Suchen nach Fehlermuster während der Bearbeitung
- z.B. unerreichbarer Zustand:



- Weitere Überprüfungen: fehlender Anfangszustand, Verklemmung, fehlerhafte Wertzuornungen, etc.

KODEGENERIERUNG

Motivation

- Abkürzung der Entwicklungszeit:
  automatische Generierung der
  - Dokumentation
  - Quellencode
  - Konfiguration

  anhand der Modelle/Anforderungen/Pläne

- Beispiele: Systemmodellierung Hausaufgabe
  - Aufgabe: Parameter → Aufgabenbeschreibung (in verschiedenen Sprachen), Testfälle
  - Yakindu: Modell → Kode

Beispiel: Generierung der Aufgabenbeschr.

- Beispiel: Anfangsbedenkzeit der Spieler
- Parameter:
  - Anfangswert der Anfangsbedenkzeit
  - Änderung der Anfangsbedenkzeit
    - nicht möglich
    - gleichzeitig für beide Spieler
    - getrennt für die zwei Spieler
- LaTeX Kode:

```
Am Anfang sind die die Bedenkzeit messenden Uhren beider Spieler auf
die Anfangsbedenkzeit eingestellt. Der Wert der Anfangsbedenkzeit ist
\ifbool{setupInitialTimeSettable}{beim Einschalten der Schachuhr
}\setupInitialTimeDefault{} Sekunden für beide
Spieler\ifbool{setupInitialTimeSettable}{, er ist aber
\ifbool{setupInitialTimeIndividual}{für die zwei Spieler getrennt
}einstellbar (siehe Kapitel \refstruc{sec:settings-details})}.
```

Beispiel: Generierung der Aufgabenbeschr.

- LaTeX Kode:

```
Am Anfang sind die die Bedenkzeit messenden Uhren beider Spieler auf
die Anfangsbedenkzeit eingestellt. Der Wert der Anfangsbedenkzeit ist
\ifbool{setupInitialTimeSettable}{beim Einschalten der Schachuhr
}\setupInitialTimeDefault{} Sekunden für beide
Spieler\ifbool{setupInitialTimeSettable}{, er ist aber
\ifbool{setupInitialTimeIndividual}{für die zwei Spieler getrennt
}einstellbar (siehe Kapitel \refstruc{sec:settings-details})}.
```

- Die generierte Aufgabenbeschreibung

Am Anfang sind die die Bedenkzeit messenden Uhren beider Spieler auf die Anfangsbedenkzeit eingestellt. Der Wert der Anfangsbedenkzeit ist beim Einschalten der Schachuhr 150 Sekunden für beide Spieler, er ist aber einstellbar (siehe Kapitel 2.4).

Am Anfang sind die die Bedenkzeit messenden Uhren beider Spieler auf die Anfangsbedenkzeit eingestellt. Der Wert der Anfangsbedenkzeit ist beim Einschalten der Schachuhr 180 Sekunden für beide Spieler, er ist aber für die zwei Spieler getrennt einstellbar (siehe Kapitel 2.4).

Programierer vs. Kodegenerator

- Vorteile des **richtig geschriebenen** Kodegenerators
  - Generiert richtigen Kode → Zertifikation
  - Kode wird auf Knopfdruck generiert
    - Nach jeder Änderung gleich
  - Optimierbar auf Lesbarkeit, Effizienz, etc.

- Kodegenerator richtig zu schreiben ist schwierig

  → muss viel können, siehe oben

Aufgaben der Kodegenerierung

- **Aufgabe:** automatische Generierung von sich entsprechend verhaltenen Modellen
- Mehrere Möglichkeiten → Entwurfsentscheidung
  - **Interpretiert:** Modell wird eingelesen und ausgeführt
    Programmkode: Synthese der Quellenkode
  - **Programmiersprachen:** Java, C, C++, …
  - **Optimierung:** Speicher vs. Prozessor
    Beobachtbarkeit vs. Performanz
  - Verbindung des generierten Kodes mit eigenem Kode
- Kodegenerator: parametrisierbar + ergänzbar

Interpretieren vs. Generieren

- Dinamischer Interpreter
  - Schnelles Starten, Ergebnisse gleich
  - Im Allgemein Zusatz-kosten in Zeit/Speicher
  - Abhängigkeiten zur Ausführungszeit
  - Kann Änderung des Modelles während der Ausführung unterstützen
  - Verhalten entspricht immer dem Modell
- Kodegenerator
  - Starten erst nach Generieren und Kompilieren
  - Optimierbar auf effizientes Ausführen
  - Nach Änderung muss neu generiert werden
  - Kode manuell veränderbar
    → *Wollen wir das?*

Kodgenerator

- Was ist ein Kodegenerator?
  - Irrglaube:

    Programm, welches Programme schreibt …???
  - Realität:

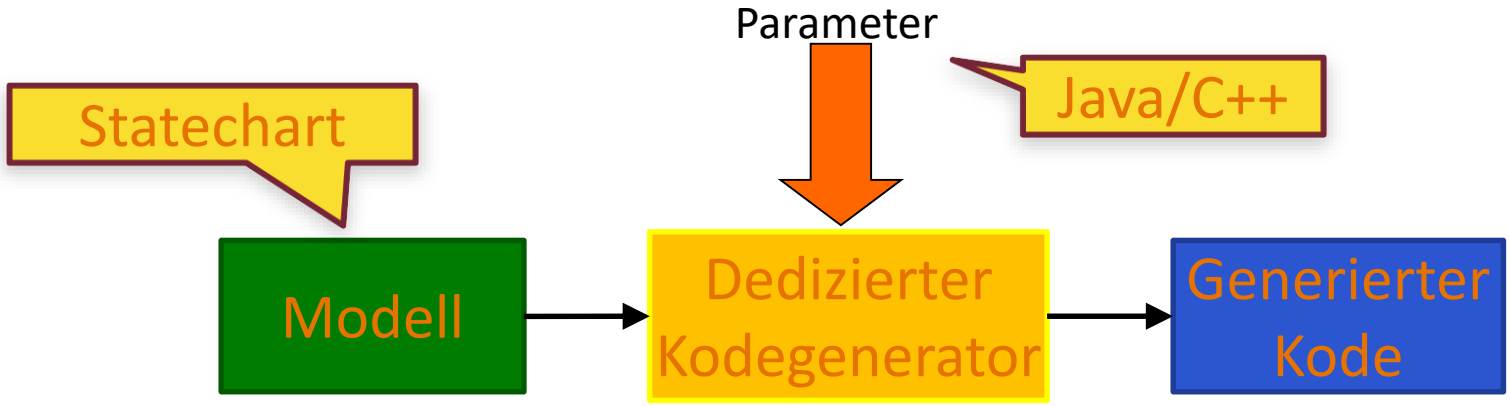
    Einfaches Programm mit viel, viel Ausschreiben

```
sourceFile.write("   temp = ((AIDA_PARTITION_TYPE*) selfModule.partitions.elements);\n" )
i = 0
for partition in partitions:
 numPorts = getNumberOfAllCommPorts_Partition(currModuleComm, interPartitionComm, partition.partitionName)
 sourceFile.write("   temp[" + str(i) + "].partition_id = " + str(partition.partitionID) + ";\n" )
 sourceFile.write("   strcpy( &temp[" + str(i) + "].partition_name[0], \"" + str(partition.partitionName) + "\");\n")
 sourceFile.write("   temp[" + str(i) + "].ports.type = CONST_AIDA_PORTS_TYPE;\n")
 sourceFile.write("   temp[" + str(i) + "].ports.elements = &mem_ports_" + str(partition.partitionName) + "[0];\n")
 sourceFile.write("   temp[" + str(i) + "].ports.numOfElements = " + str(numPorts) + ";\n")
 sourceFile.write("\n")
 i = i + 1
## end for
sourceFile.write("\n")
```

- Ausblick: *Quine* – Programm, welches seinen eigenen Kode ausschreibt

Kodegeneratortypen I.

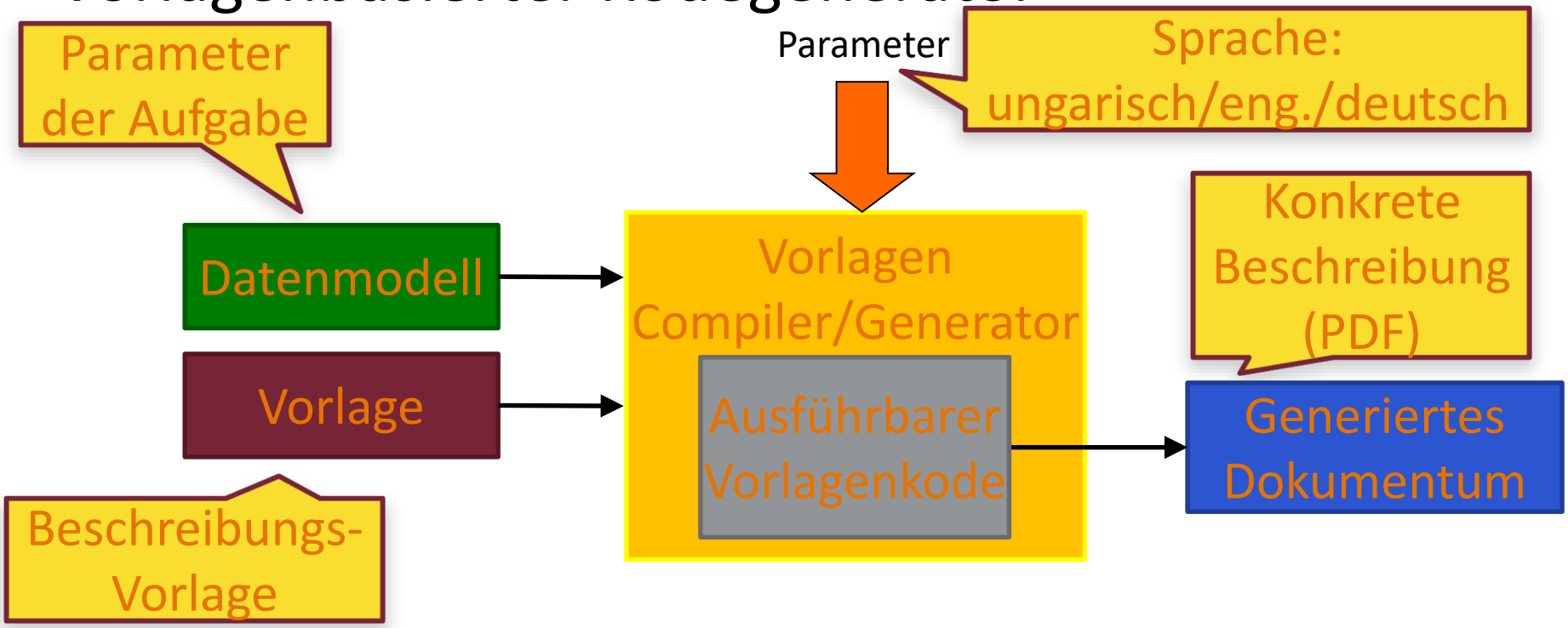
- Dedizierter Kodegenerator (z.B. Yakindu)



  - Teil einer Modellierungsumgebung
  - Zertifizierbar
  - Kann nicht richtig angepasst werden

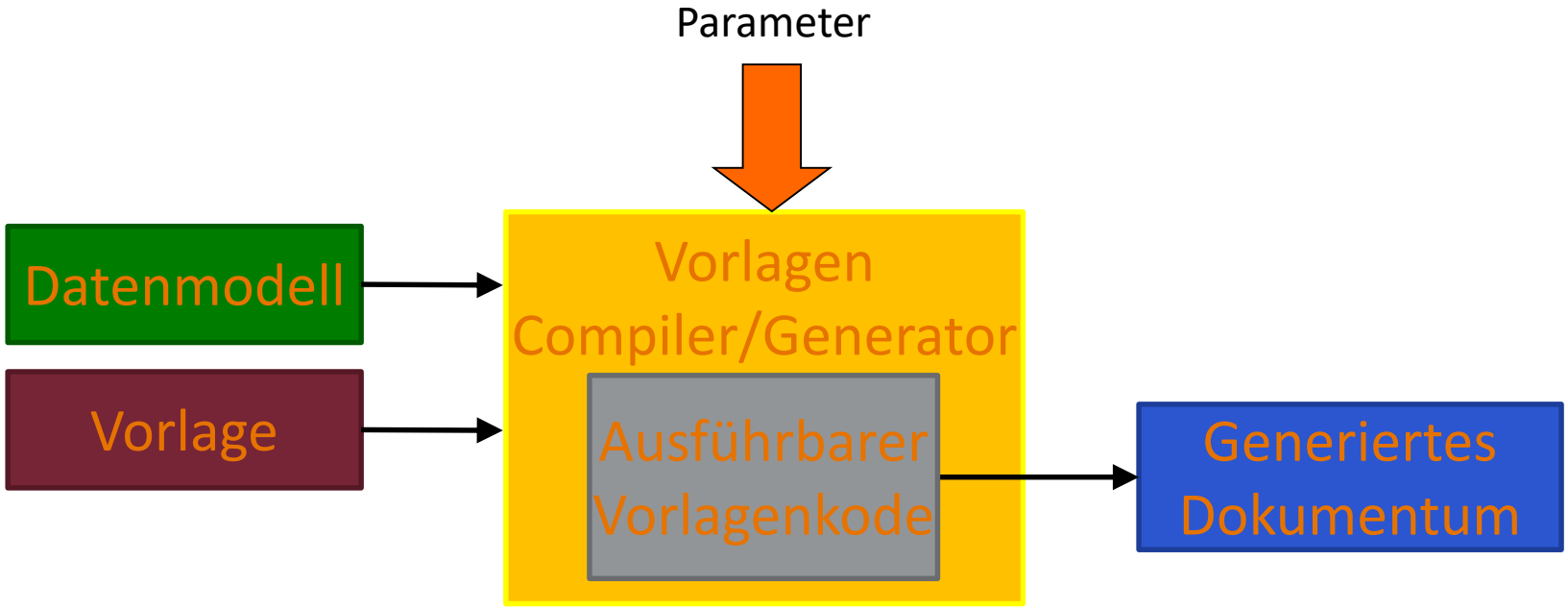
Kodegeneratortypen II.

- Vorlagenbasierter Kodegenerator



Kodegeneratortypen II.

- Vorlagenbasierter Kodegenerator



  - Schnellere Entwicklung
  - Komplexe Änderungen während des Lebenszyklus
    - Vorlage und Modell können unabhängig geändert werden

Funktionen

- Effizienz vs. Wiederverwendbarkeit, Lesbarkeit

- Nachverfolgbarkeit
  - Welches Modellelement ist zu einem gegebenen Kodeteil geführt?
  - Unterstützung der *Inkrementalität*: der Kode wird nur da geändert, wo das Modell geändert wird

→ Effizientes Neugenerieren

Anpassung des generierten Kodes

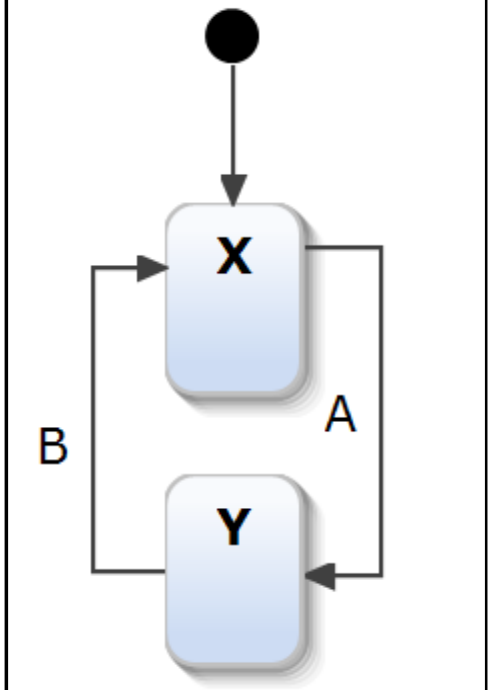
- Der vom Modell generierte Kode ist nur ein Teil der Anwendung. *Wird er zu den anderen beigelegt.* *Wie?*
  - Generierter Kode wird manuell nicht geändert
    → kann jederzeit neu generiert werden!
  - Im Modell/Vorlage würde es die Komplexität erhöhen

    Passt einfach nicht da hinein …
- Generierter und manuell geschriebener Kode kommt in getrennten Dateien/Verzeichnisse
- Anpassung: *Klebekode* (glue code)
  - Aufruf des generierten Kodes
  - Ableitung aus dem generierten Kode

Kodgeneratoren – ein Beispiel

- **Aufgabe:**
  Generiere C-Kode für eine Yakindu Zustandsmaschine

Demo
internal:
event A
event B



- Schreibe eine Funktion, die:
  → ein Modellobjekt nimmt
  ← einen Text zurückgibt
- Der Text wird in eine Datei „Demo.c” geschrieben
- Es wird von einem Compiler übersetzt

Vorlagenbasierter Kodegenerator (Xtend)

- Ziel: Zustände → Enum

```
enum states {
   StateX,
   StateY
};
```

Die Ausgabe wird in ein char* gesammelt, anstatt %s werden die Namen X,Y geschrieben

- Lösung: C Programm

```
sprintf(result,
   "enum states {\n\tState%s,\n\tState%s\n};",
   state1->name,
   state2->name);
```

☺ Funktioniert gleich!
☹ Unlesbar

- Vorlage (Xtend):

```
'''
enum states {
    State«state1.name»,
    State«state2.name»
}'''
```

Die variablen Stellen werden angegeben (*escape*)

Die Vorlage wird geschrieben

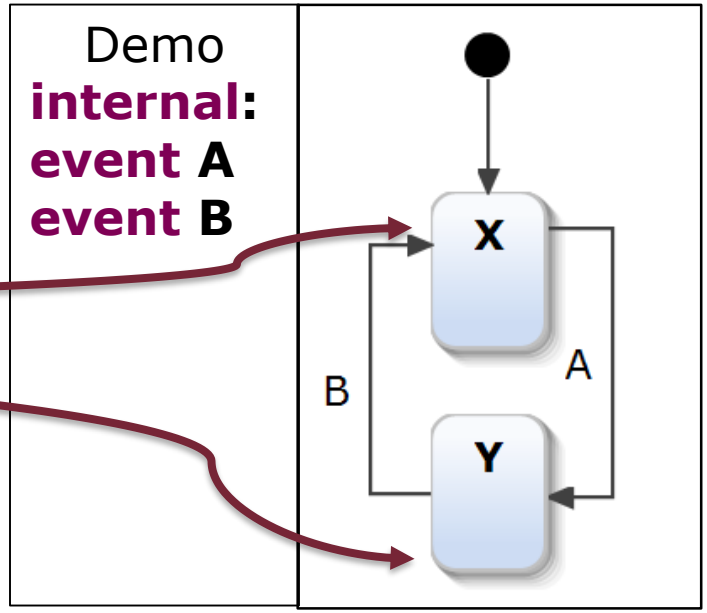
☺ Einfacher zu schreiben
☺ Übersichtlich
☺ Leicht zu modifizieren
☹ +1 Technologie

Kodegenerator Beispiel – Zustände

- Erwarteter C-Kode:

```
//States of the statemachine
enum states {
   StateX,
   StateY
};
```

Mögliche Zustände:
Als Enum aufgezählt



- Vorlage:

```
//States of the statemachine
enum states {
«FOR state : states»
   State«state.name»,
«ENDFOR»
};
```

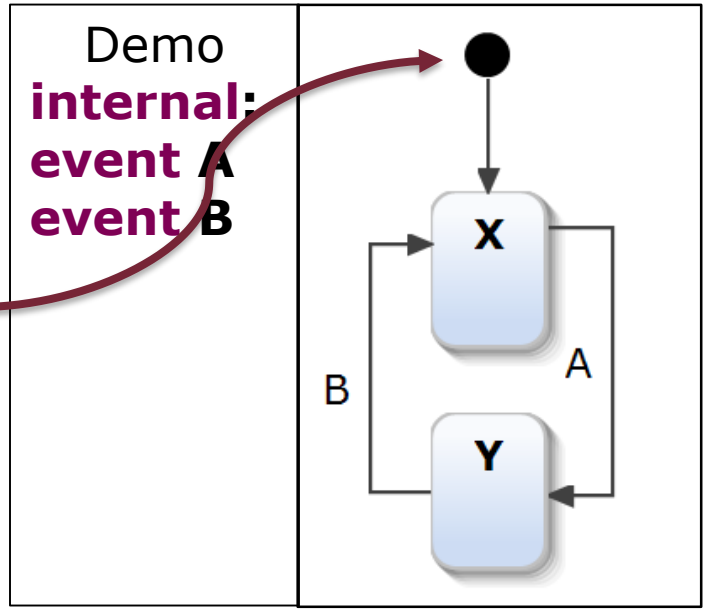
1. Wir iterieren durch alle Zustände
2. Ihre Namen werden mit Komma getrennt ausgeschrieben:
   `State«Name»` z.B.: `StateX`

Kodegenerator Beispiel – Anfangszustand

- Erwarteter C-Kode:

```
// The current state
// First = entry state.
enum states actualState = StateX
```

Aktueller Zustand = Anfangszustand



- Vorlage:

```
// The current state
// First = entry state.
enum states actualState = State«findEntry(states).name»
```

1. Wir suchen das Anfangselement
2. Sein Name wird ausgeschrieben

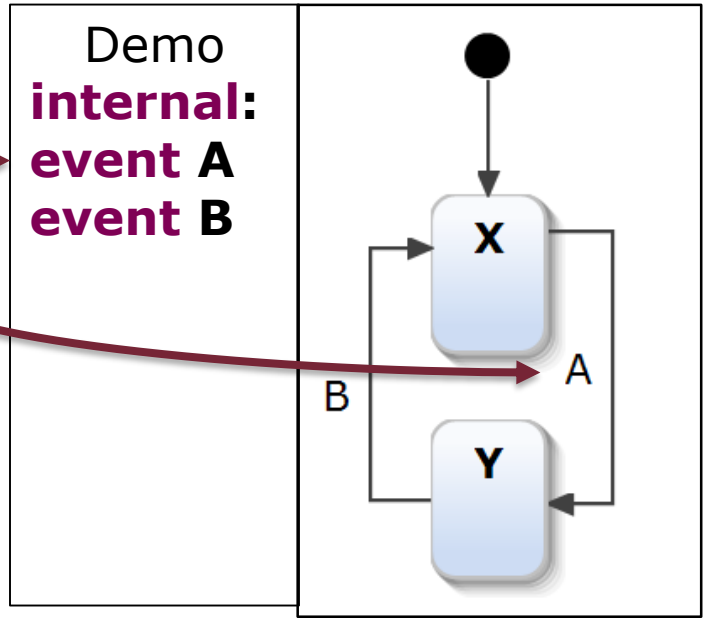
Kodegenerator Beispiel – Zustandsübergänge

- Erwarteter C-Kode:

```
// Execute "A" event
void doA{
switch(actualState) {
        case StateX:
            actualState = StateY;
            break;
        case StateY:
            break;
    }
}
```

A / X → Y

A / -



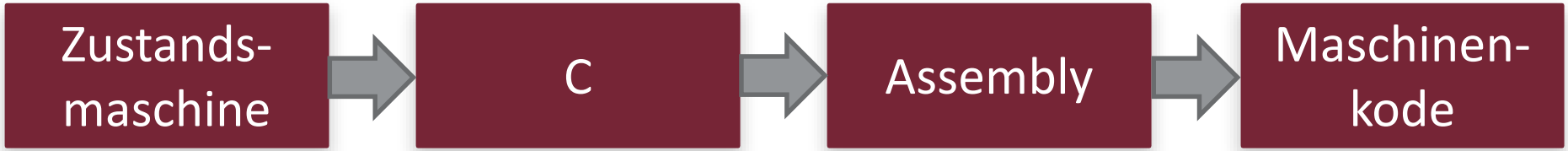
- Vorlage (skizziert):

1. Für jedes Ereignis eine Funktion `do«NameDesEreignisses»`
2. Der Funktioninhalt entspricht den Transitionen

Für eine (einfache) Zustandsmaschine ist der Kodegenerator nur soviel!

Kodegenerator – Zusammenfassung

- Kodegenerierung = Übersetzer
- Gleiche Schritte:



- Lösung in der Sprache des Problems: Produktivität ++
- Viele langweilige, komplizierte Kodierungsarbeit automatisiert Leistung ++
- Überprüfung in der Sprache des Problems: Zuverlässigkeit ++
- Projekte an unserem Lehrstuhl: bis zu **95%** generierter Kode

Kodegenerator – Zusammenfassung

- Kodegenerierung = Übersetzer
- Gleiche Schritte:

Zustands- maschin

Prophezeiung:
gestern Entwurfsmuster →
heute Funktion der Kodegeneratoren →
morgen Element der Modellierungssprachen

schinen- kode

- Lösung in der Sprache des Problems: Produktivität ++
- Viele langweilige, komplizierte Kodierungsarbeit automatisiert Leistung ++
- Überprüfung in der Sprache des Problems: Zuverlässigkeit ++
- Projekte an unserem Lehrstuhl: bis zu **95%** generierter Kode