Softwaretechnik und Programmierparadigmen

VL 08: Service Logic Graphs und Model Checking

Prof. Dr. Sabine Glesner
FG Programmierung eingebetteter Systeme
Technische Universität Berlin

Ein Teil dieser Folien wurde uns von Prof. Dr. Tiziana Margaria-Steffen zur Verfügung gestellt.

Ankündigung:

- Im nächsten Tutorium:
  - Arbeiten mit jABC
  - Wenn möglich Laptop mit jABC mitbringen

  - Tool Download unter

  http://ls5-www.cs.tu-dortmund.de/projects/jabc/index.php

Motivation

- Bisher: Semi-formale Modellierungstechniken
  - Formale Syntax
  - Informelle Semantik (außer OCL)
- Problem: Automatische Analyse auf Modellebene erfordert formale Semantik
- Service Logic Graphs
  - **Ähnlich zu Aktivitätsdiagrammen**
  - Formale Syntax und Semantik
  - Spezifikation von Eigenschaften mit Temporallogik
  - Model Checking in jABC

Übersicht

- Einführung
- Service Independent Blocks (SIBs)
- Service Logic Graphs (in jABC)
- jABC Plugins
- Model Checking

Übersicht

- Einführung
- Service Independent Blocks (SIBs)
- Service Logic Graphs (in jABC)
- jABC Plugins
- Model Checking

Modellgetriebene Entwicklung

(Model Driven Development - MDD)

- Modelle als primäre Artefakte in der Softwareentwicklung
- Weitgehend automatische Modelltransformationen, möglichst bis zum Maschinencode
- Oft Verwendung von domänenspezifischen (Modellierungs-)Sprachen (**DSL**)

ähnlich Abstraktion „Assembler ⟶ höhere Programmiersprachen“

- leichtere Verständlichkeit
- Konsistenz von Modellen und Code
- größere Unabhängigkeit von (wechselnden) Implementierungstechnologien

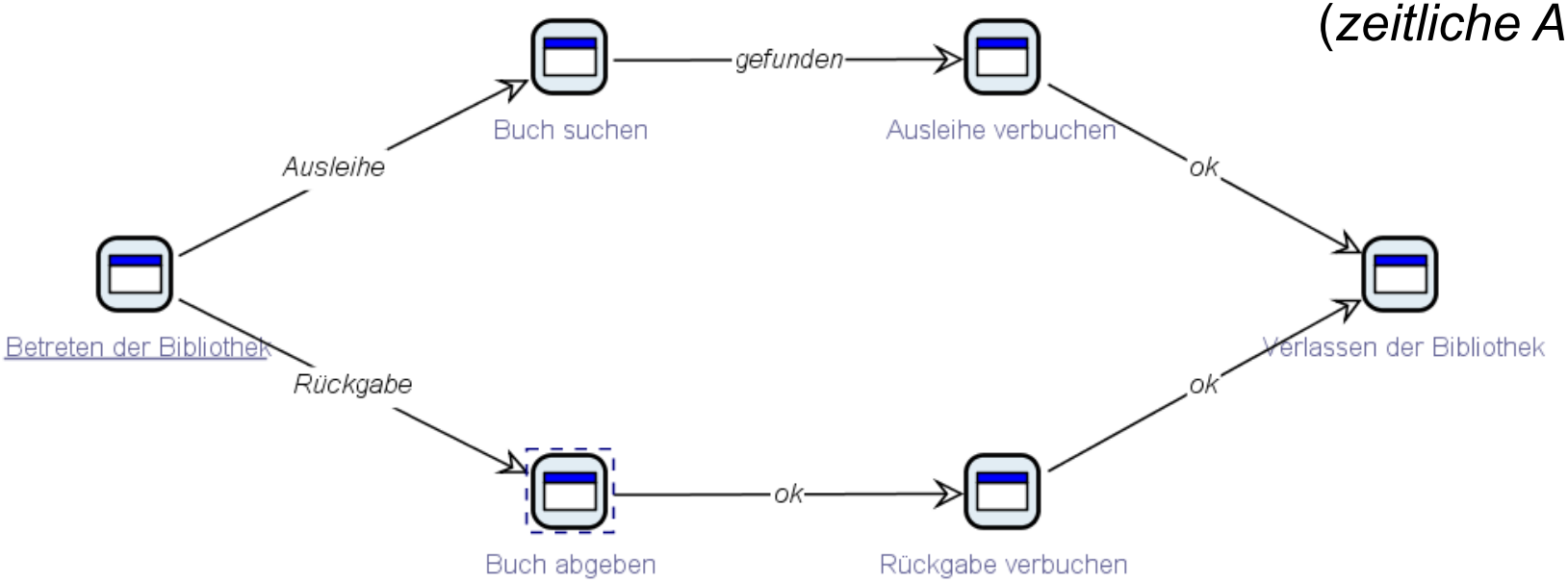
Modellgetrieben:
Systemänderungen werden nur auf der Modellebene vorgenommen
Modell als das einzige & zentrale Entwicklungsartefakt, das schrittweise verfeinert wird („One-Thing Approach“ [OTA])

Beispiel eines modellgetriebenen Ansatzes

Service Logic Graphs (SLGs)
als ausführbare Prozessmodelle

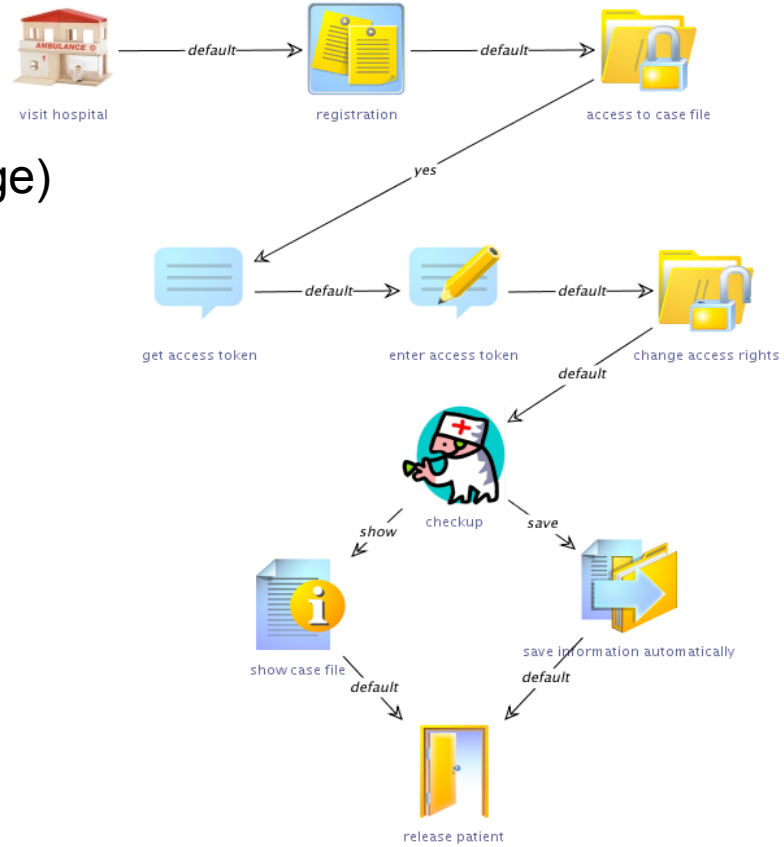
Ein Geschäftsprozess in einer Bibliothek

Knoten: Basisaktivitäten
Kanten: Kontrollfluss
(*zeitliche Abfolge*)



Services in jABC

- Prozessknoten repräsentieren **Services**
  - elementare Standardservices (z.B. Dialoge)
  - Legacy-Systeme, COTS
  - externe Services (z.B. Web-Services)
  - neue Implementierungen



- **Schrittweise Verfeinerung**
  von Benutzermodellen
  (z.B. Geschäftsprozessmodellen)
  zu technischeren Modellen,
  bis hin zu elementaren Services
  - wiederverwendbar
  - voneinander unabhängig
- „**Orchestrierung**“ dieser elementaren Services zu einem ganzheitlichen Prozess

Entwicklungswerkzeug jABC

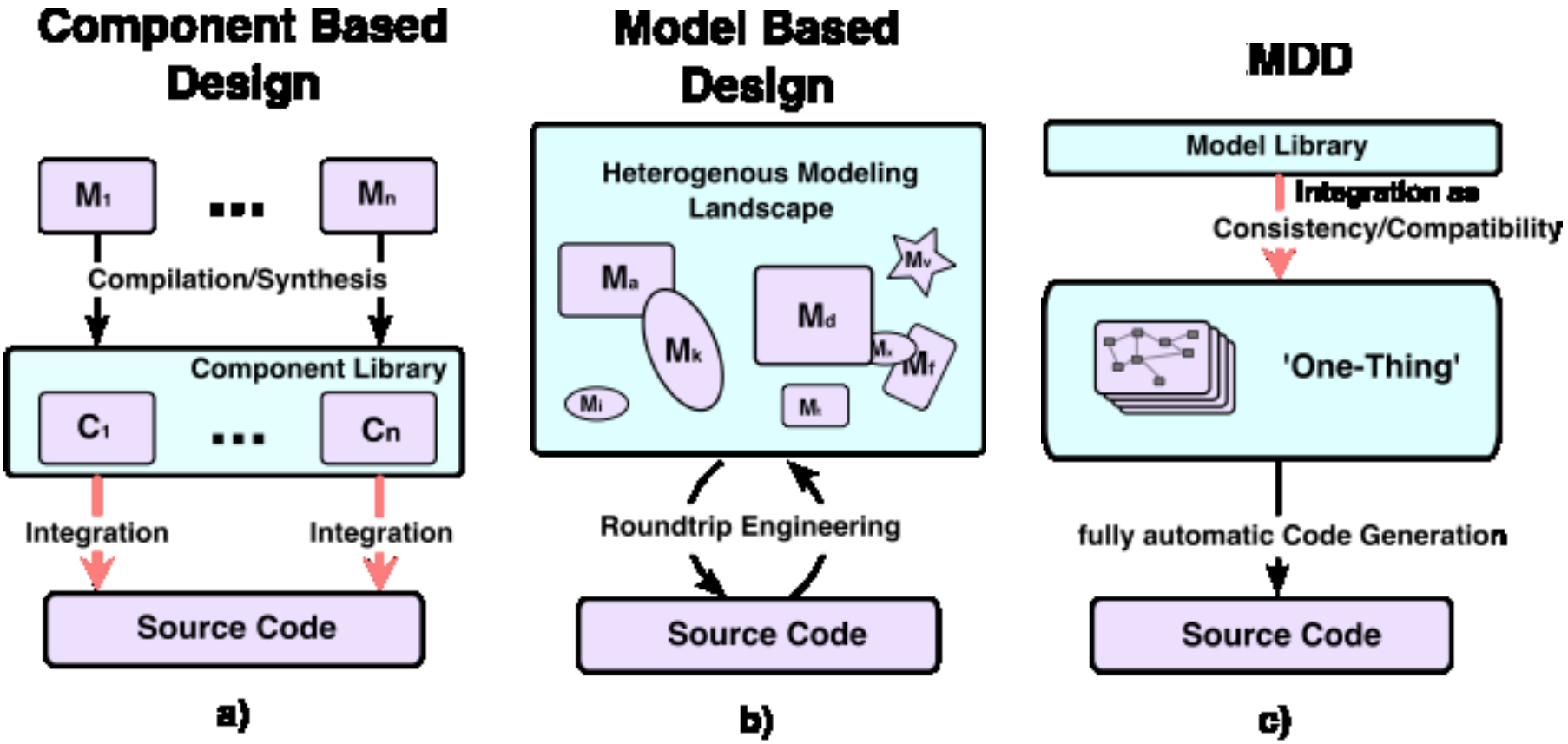
- **SLG-Editor**
  - intuitive Syntax erlaubt Einbeziehen von Auftraggebern/Anwendern
- **Tracer:** Plugin zur Ausführung und zum Debugging
  - frühes Testen

➡ unterstützt *agiles Vorgehen*

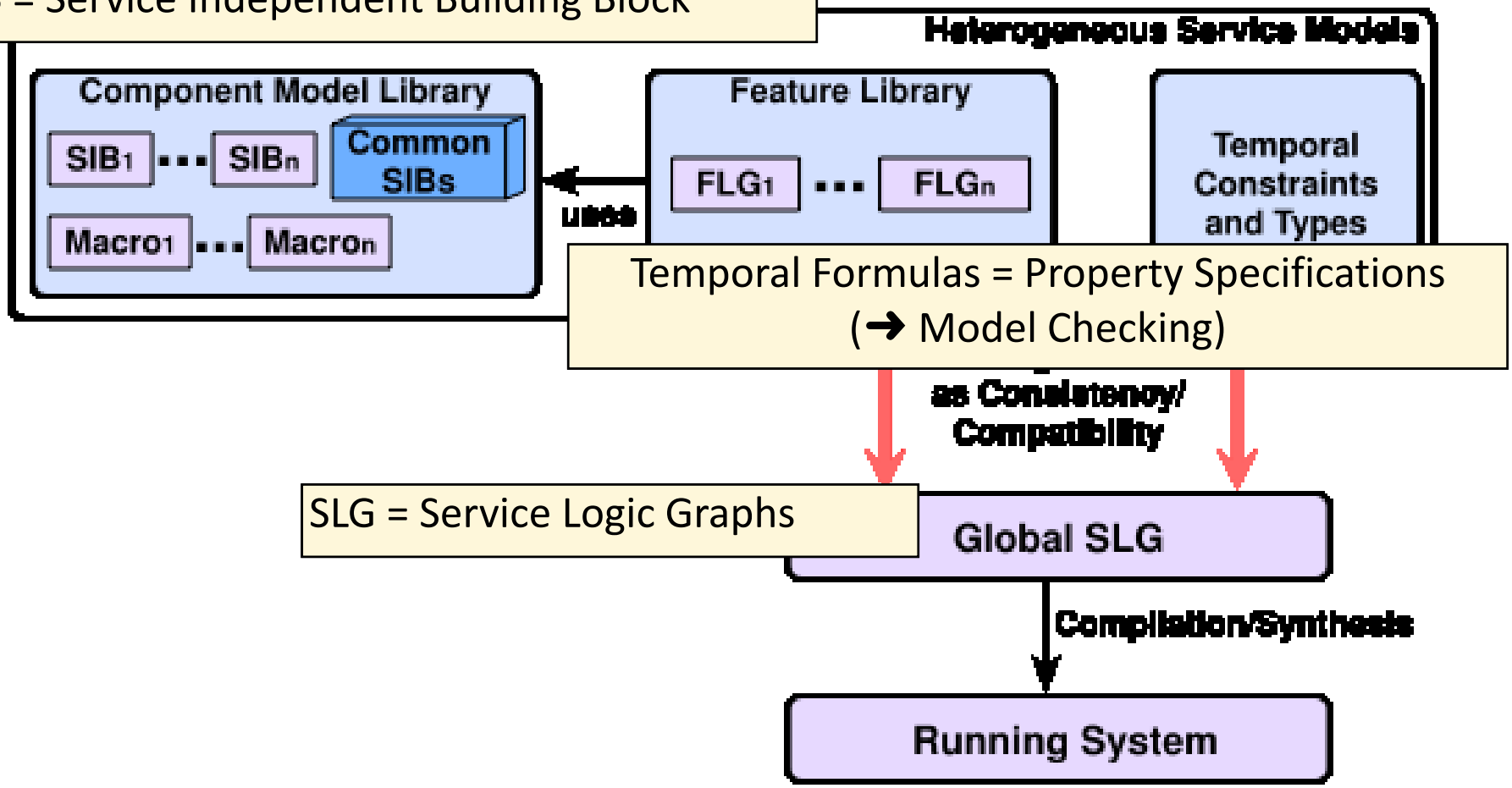
- **Local Checker:** Plugin zur Überprüfung der korrekten Verwendung der Knoten und Kanten (in Abhängigkeit vom Knotentyp)
- **GEAR:** Plugin zum **Model Checking** (Überprüfung globaler Eigenschaften des gesamten Prozessmodells)
- **Genesys:** Code-Generator

➡ unterstützt *modellgetriebenes Vorgehen*

Entwicklungsansätze



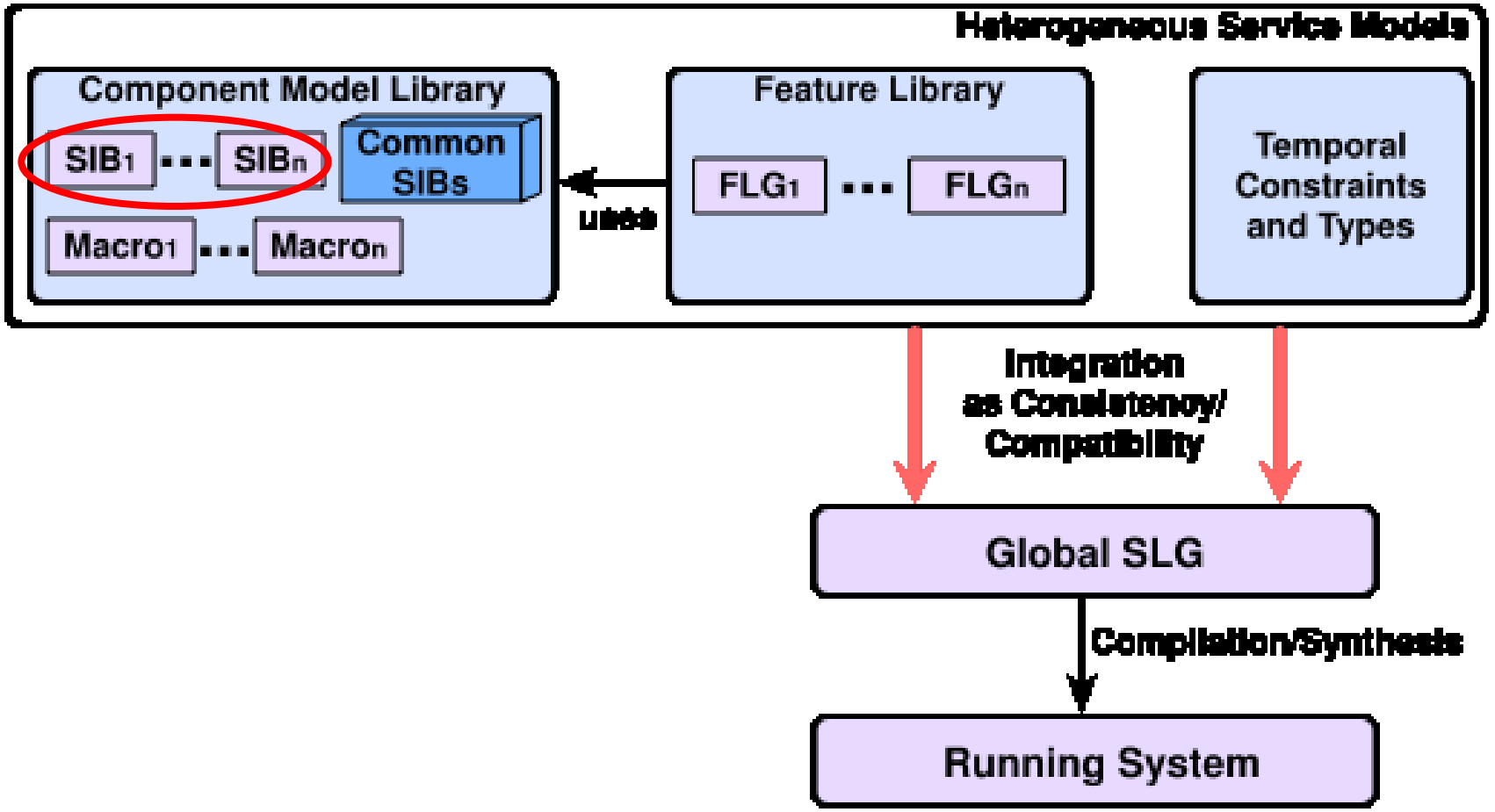
SIB = Service Independent Building Block



➪ SIBs

Übersicht

- Einführung
- Service Independent Blocks (SIBs)
- Service Logic Graphs (in jABC)
- jABC Plugins
- Model Checking


Heterogeneous Service Models
Component Model Library
SIB1
SIBn
Common SIBs
Macro1
Macron
Feature Library
FLG1
FLGn
Temporal Constraints and Types
Integration as Consistency/ Compatibility
Global SLG
Compilation/Synthesis
Running System

Service Independent Building Blocks (SIBs)

- Repräsentieren fertiggestellte, elementare Services
- Werden zur Zusammenstellung eines Systems verwendet
- Konfigurierbar & wiederverwendbar
- Ausführbar
- Intern als einfache Java-Klassen implementiert

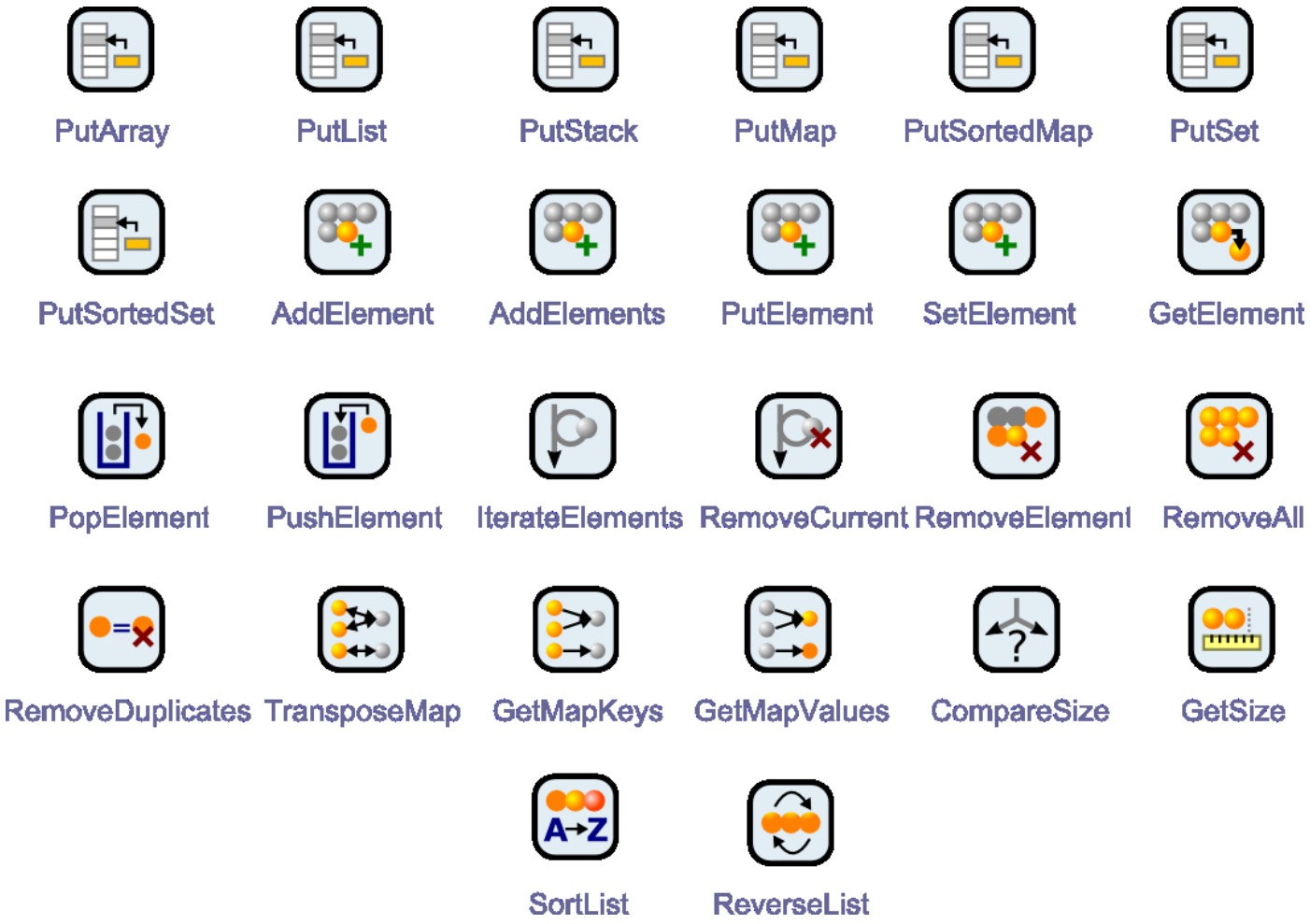
Übliche SIBs

- Basisbibliothek von SIBs bei jABC dabei
- Sollten einen einfachen Start ermöglichen
- Dokumentation:
  - in jABC (Mauszeiger über SIB)
  - http://www.jabc.de/sib

Basis-SIBs


CheckVariable
PutFile
PutInteger
PutFloat
PutString
PutUUID
PutPassword
PutExpression
PutBoolean
SwapElements
CopyElement
RemoveElement
RemoveElements
MergeStrings
CompareNumber
SwitchString
MatchString
EvaluateCondition
GetOS
CheckThreadExitBranch
Noop
SleepThread
ExtractPattern
ReplaceString
RepeatLoop
LogMessage
LockElements
UnlockElements
GetSystemInfo
new();
InstantiateObject
f(x);
InvokeMethod
ThrowException
i++
UpdateCounter
Prototype
GetTimeStamp
StartTimeMeasurement
StopTimeMeasurement
ChooseRandomBranch
GenerateRandomNumber

Collection-SIBs



GUI-SIBs


Beep


ShowBranchingDialog


ShowConfirmDialog


ShowInputDialog


ShowMessageDialog


ShowFileChooser


ShowImageDialog


ShowEmailDialog


ShowLoginDialog


ShowTextDialog

IO-SIBs


CheckPath
ConvertPathToURI
GetTempDirectory
GetCurrentDirectory
GetUserHomeDirectory
PrintMessage
PrintException
CopyFile
ScpFile
DeleteFile
WriteTextFile
ReadTextFile
ReadImageFile
WriteImageFile
ExportResource
LoadProperties
ScanDirectory
CreateTempDirectory
UnzipFiles
ExecuteCommand

SIBs (2)

- Bestehen aus:
  - UID: global eindeutiger Bezeichner
  - Name: lokal eindeutiger Bezeichner (innerhalb eines Modells)
  - Label: beliebiges Label
  - Parameter: zur Konfiguration
  - Branches: ≅Ausführungsergebnisse oder „exits“
    - fixed: Können nicht durch Anwender geändert werden
    - mutable: Können durch Benutzer geändert werden

SIBs (3)

- Bestehen aus:
  - Symbol
  - Dokumentation
  - Einem oder mehreren Service-Adaptern, die das Ausführungsverhalten des SIB beschreiben

SIBs: Beispiel

Show a Dialog

SIB ShowBranchingDialog

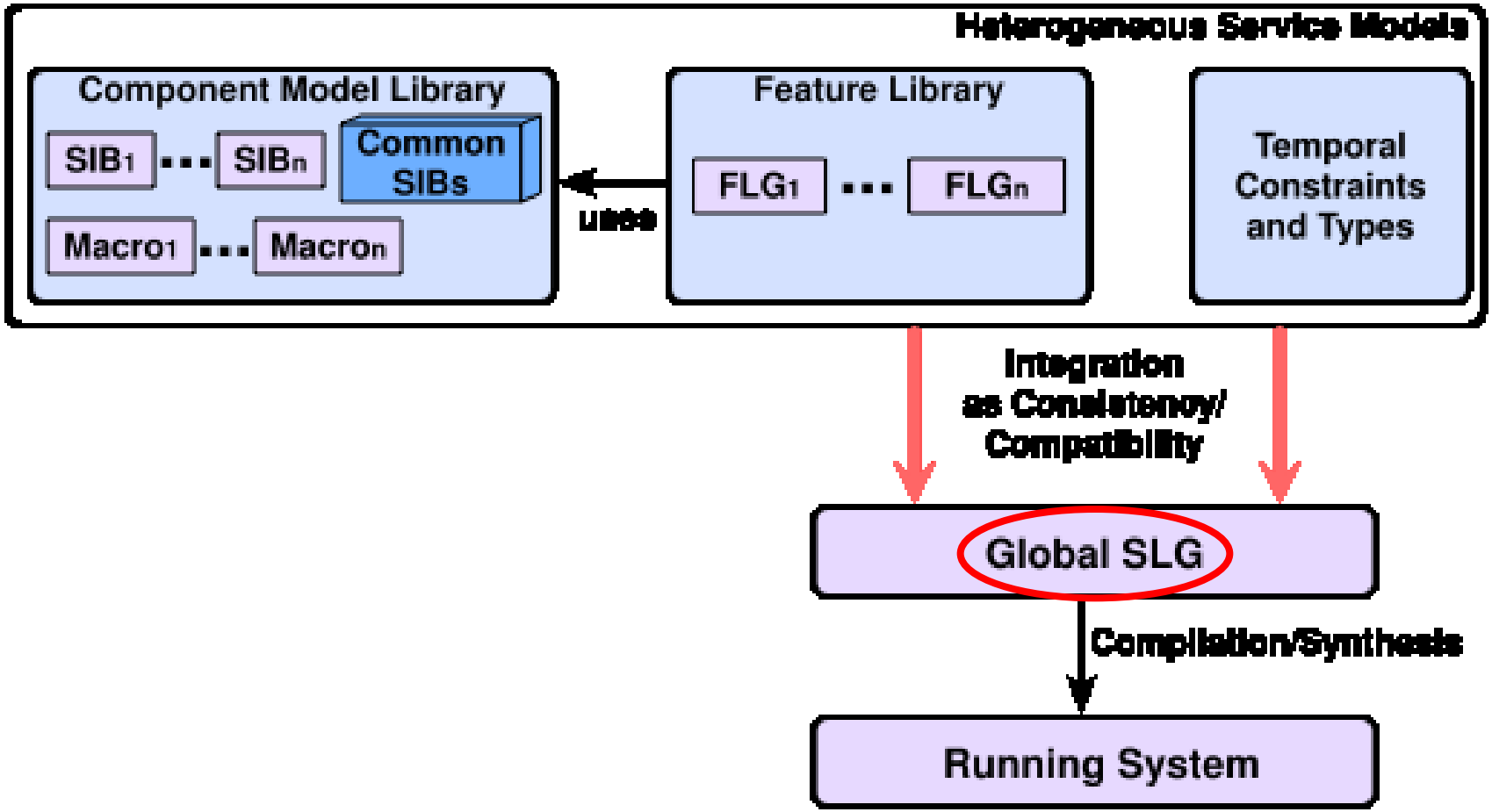
- Dokumentation:

Shows a message in a popup dialog. The dialog features exactly one button for each mutable branch configured for this SIB. The execution will finish with the branch for which the corresponding button on the dialog has been selected by the user.

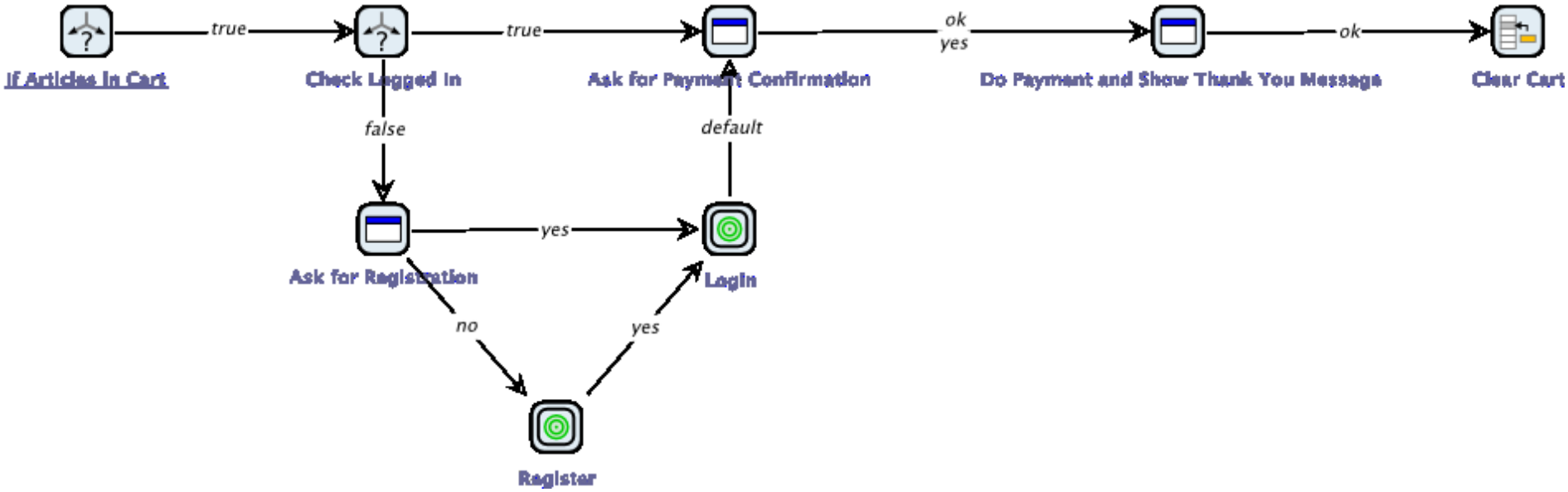
- UID: gui-sibs/ShowBranchingDialog
- Name, z.B.: ShowBranchingDialog
- Parameter:
  - dialogTitle (String)
  - message (String)
  - messageType (ListBox: "Plain", "Information", "Question", "Warning", "Error")
- Branches:
  - fixed: „cancel“, „error“
  - mutable: „ok“

Übersicht

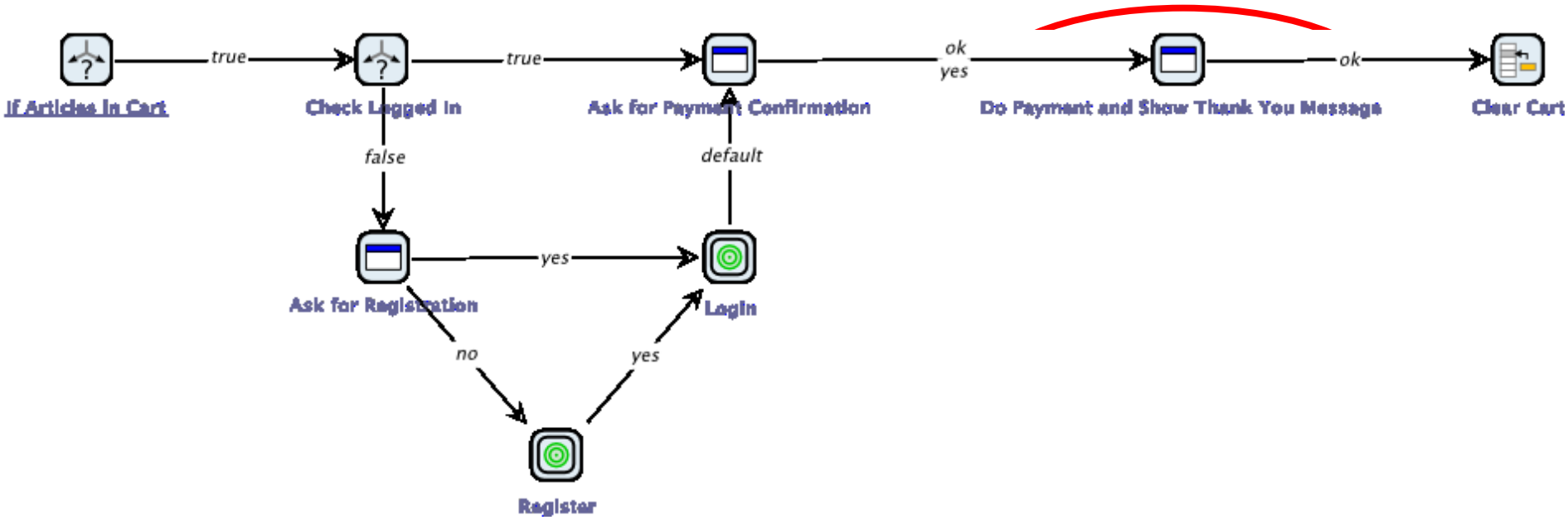
- Einführung
- Service Independent Blocks (SIBs)
- Service Logic Graphs (in jABC)
- jABC Plugins
- Model Checking


Heterogeneous Service Models
Component Model Library
SIB1
SIBn
Common SIBs
Macro1
Macron
uses
Feature Library
FLG1
FLGn
Temporal Constraints and Types
Integration as Consistency/ Compatibility
Global SLG
Compilation/Synthesis
Running System

SLGs in jABC

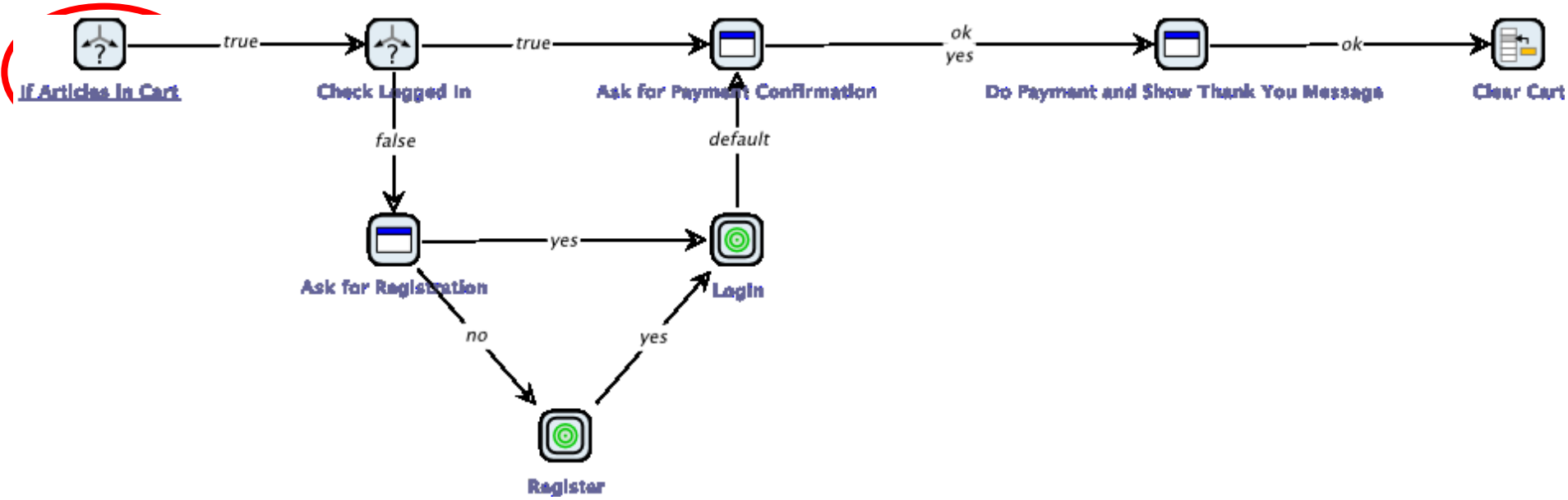

true
true
ok
yes
ok
If Articles in Cart
Check Logged In
Ask for Payment Confirmation
Do Payment and Show Thank You Message
Clear Cart
false
default
yes
Ask for Registration
Login
no
yes
Register

SLGs in jABC

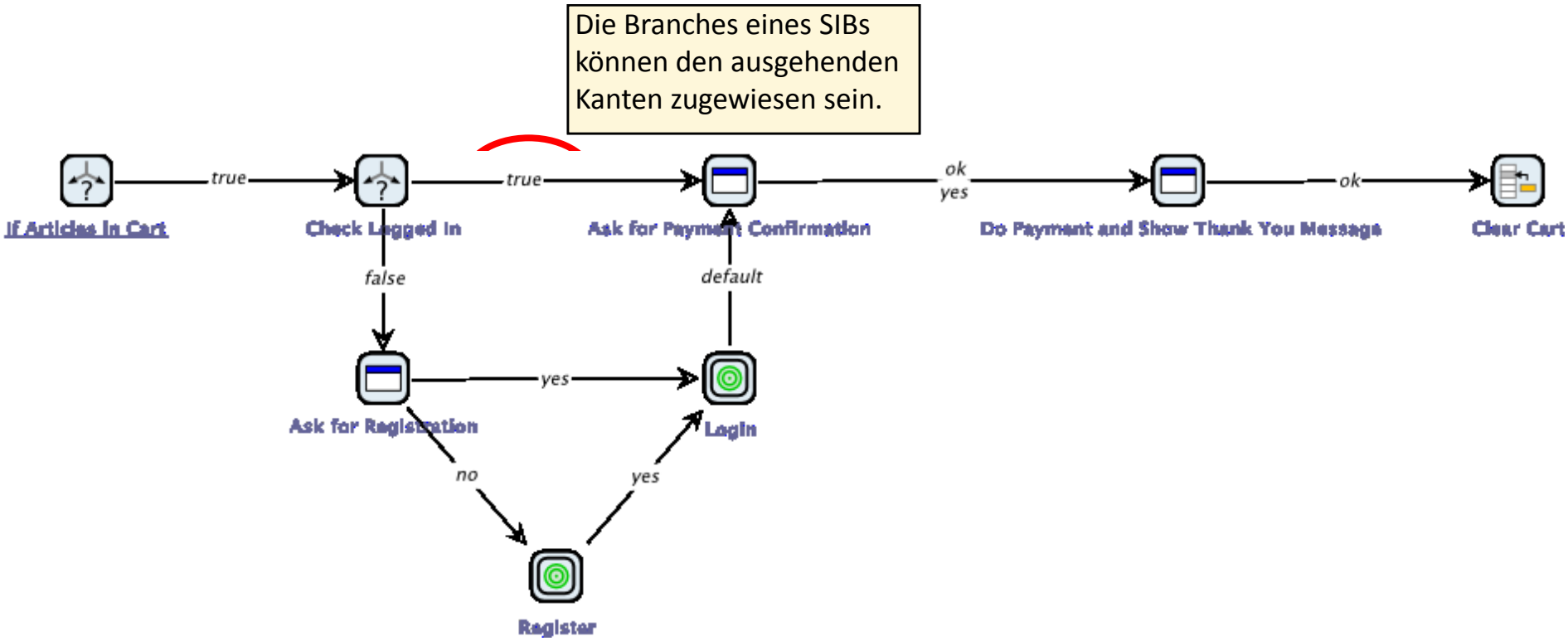

true
true
ok
yes
ok
false
default
yes
no
yes
If Articles in Cart
Check Logged In
Ask for Payment Confirmation
Do Payment and Show Thank You Message
Clear Cart
Ask for Registration
Login
Register

SLGs in jABC

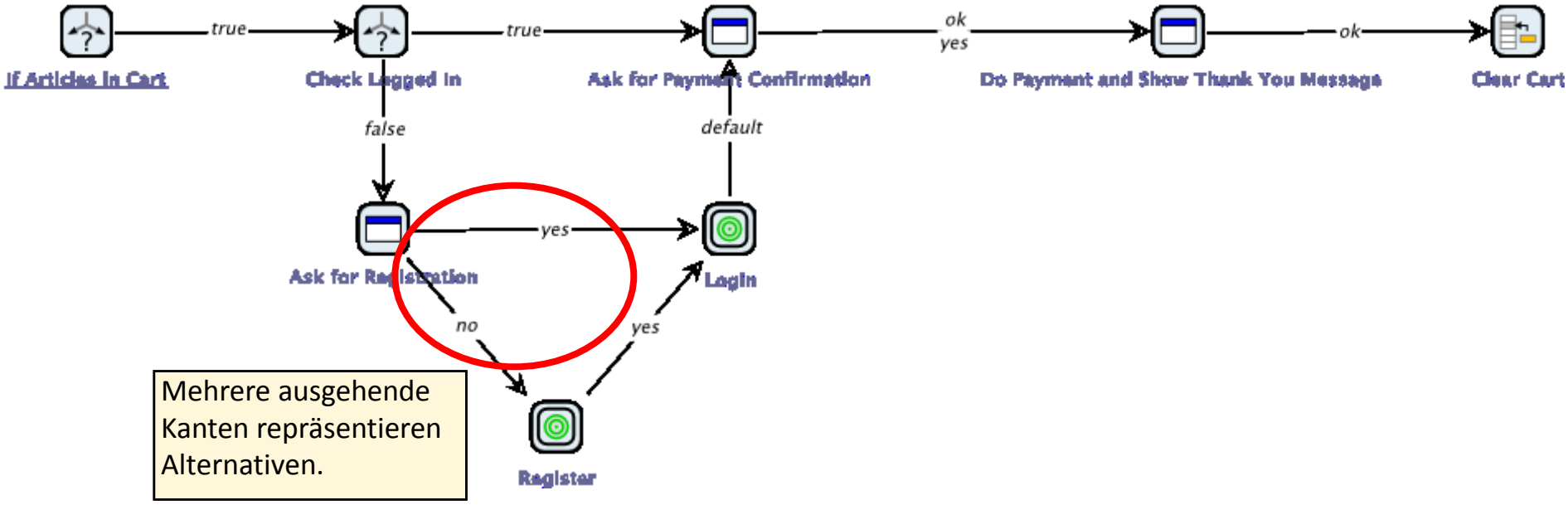
Für die Ausführbarkeit muss ein SIB als Start-SIB markiert sein.



SLGs in jABC


Die Branches eines SIBs können den ausgehenden Kanten zugewiesen sein.
true
true
ok
yes
ok
If Articles in Cart
Check Logged In
Ask for Payment Confirmation
Do Payment and Show Thank You Message
Clear Cart
false
default
yes
Ask for Registration
Login
no
yes
Register

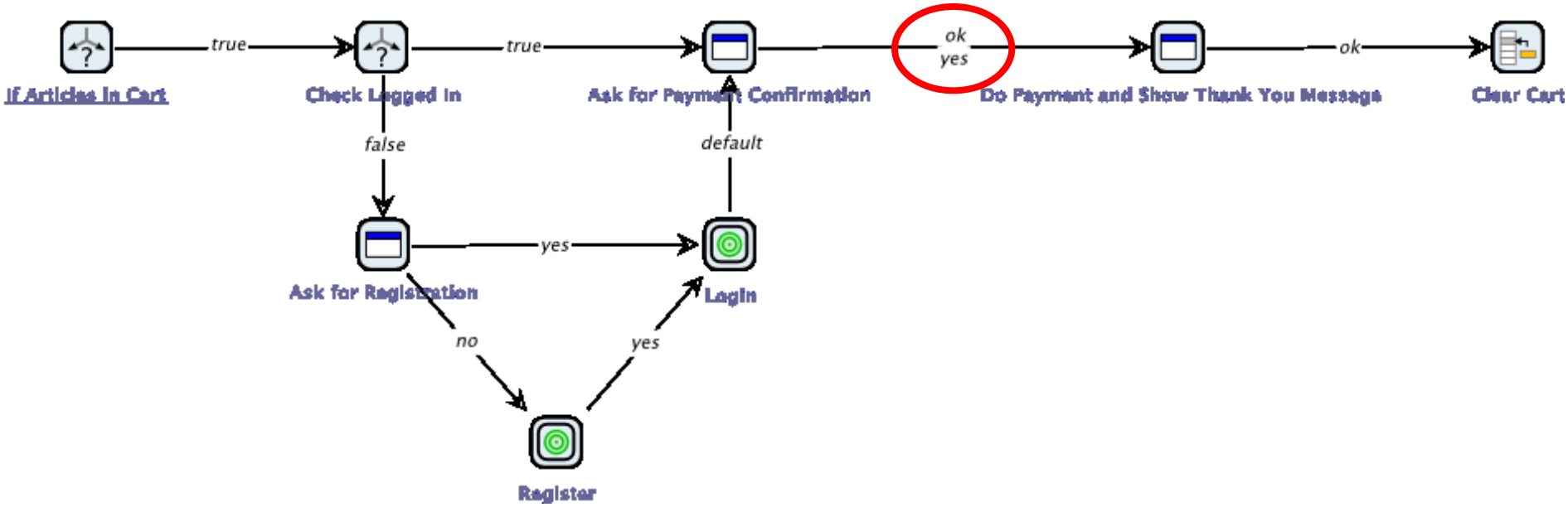
SLGs in jABC



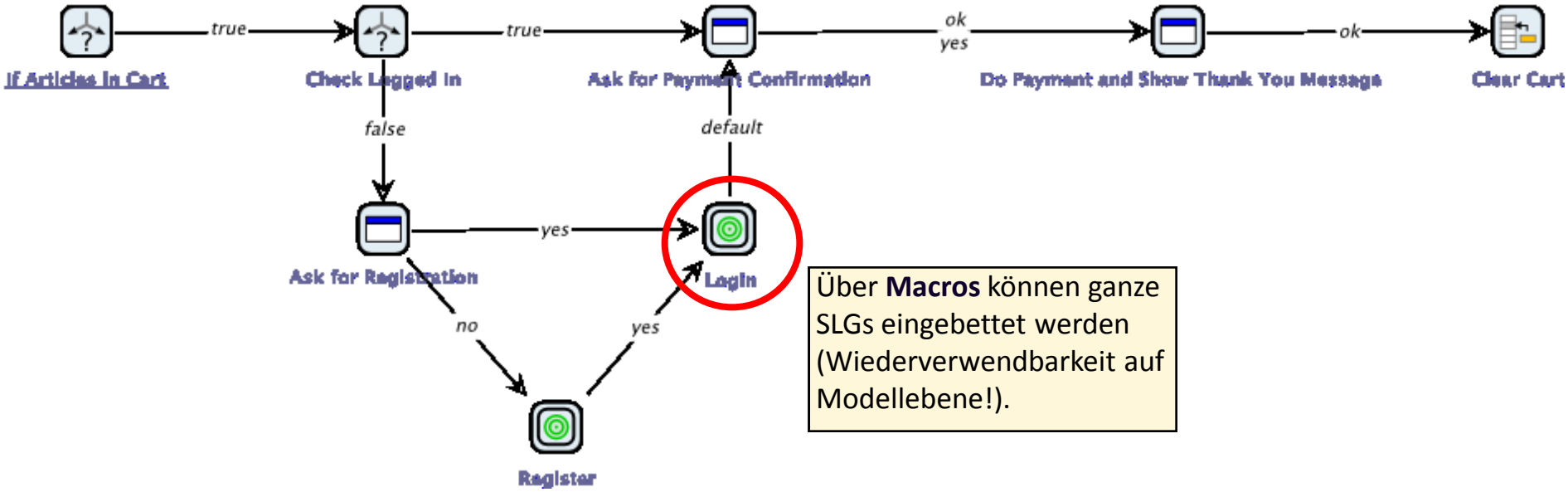
Mehrere ausgehende Kanten repräsentieren Alternativen.

SLGs in jABC

Als Abkürzung können mehrere Branches eines SIBs einer Kante zugeordnet sein.

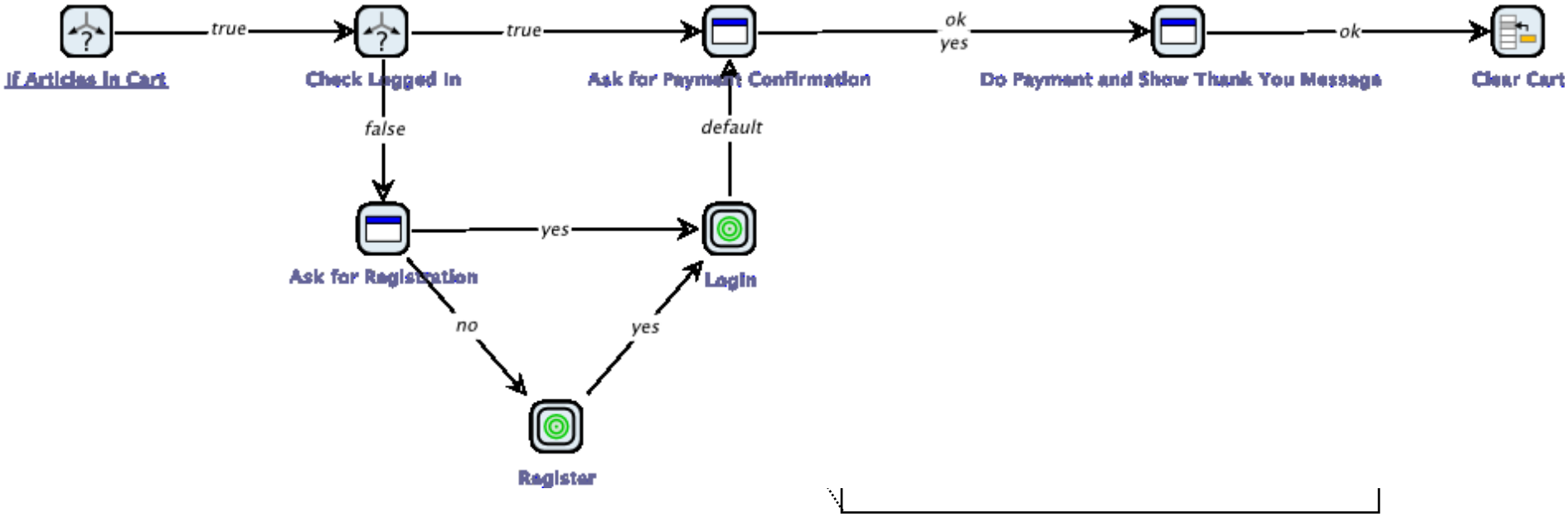


SLGs in jABC

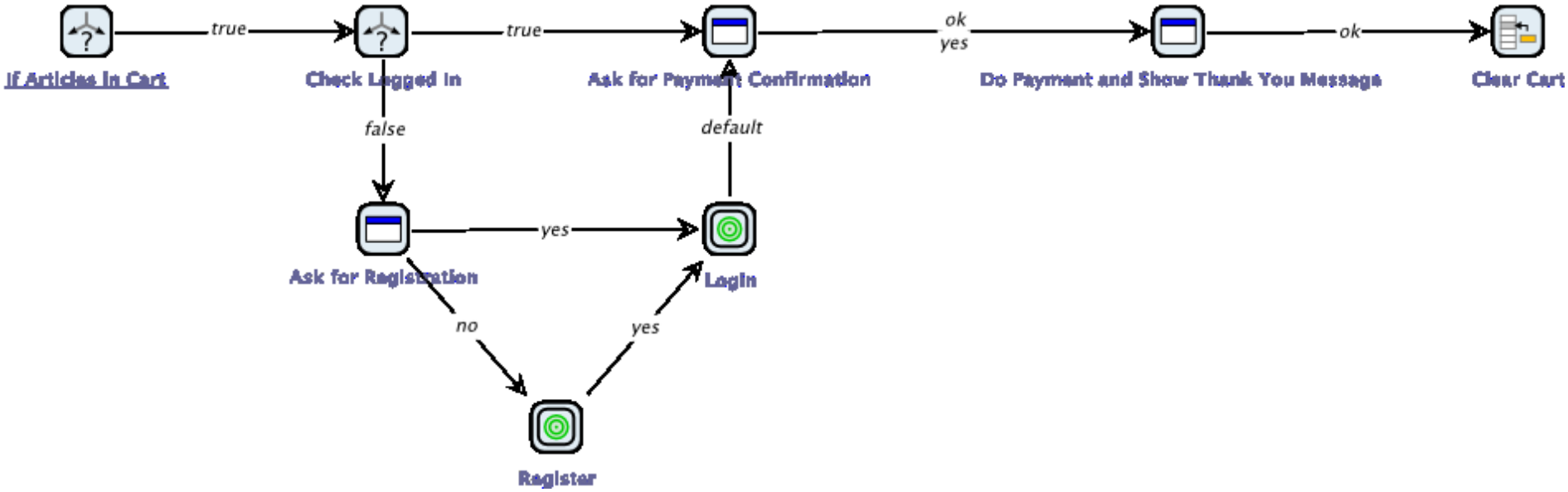


Über **Macros** können ganze SLGs eingebettet werden (Wiederverwendbarkeit auf Modellebene!).

SLGs in jABC


true
true
ok
yes
ok
If Articles in Cart
Check Logged In
Ask for Payment Confirmation
Do Payment and Show Thank You Message
Clear Cart
false
default
yes
Ask for Registration
Login
no
yes
Register

SLGs in jABC

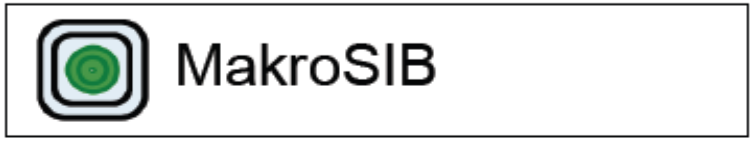


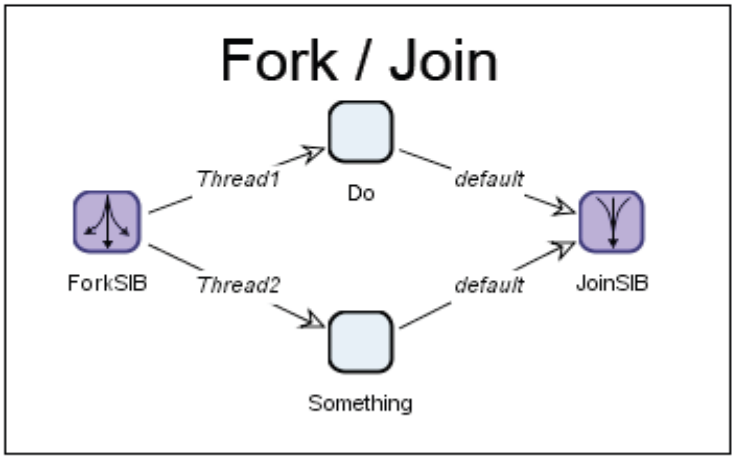
Show Login Dialog Do Login

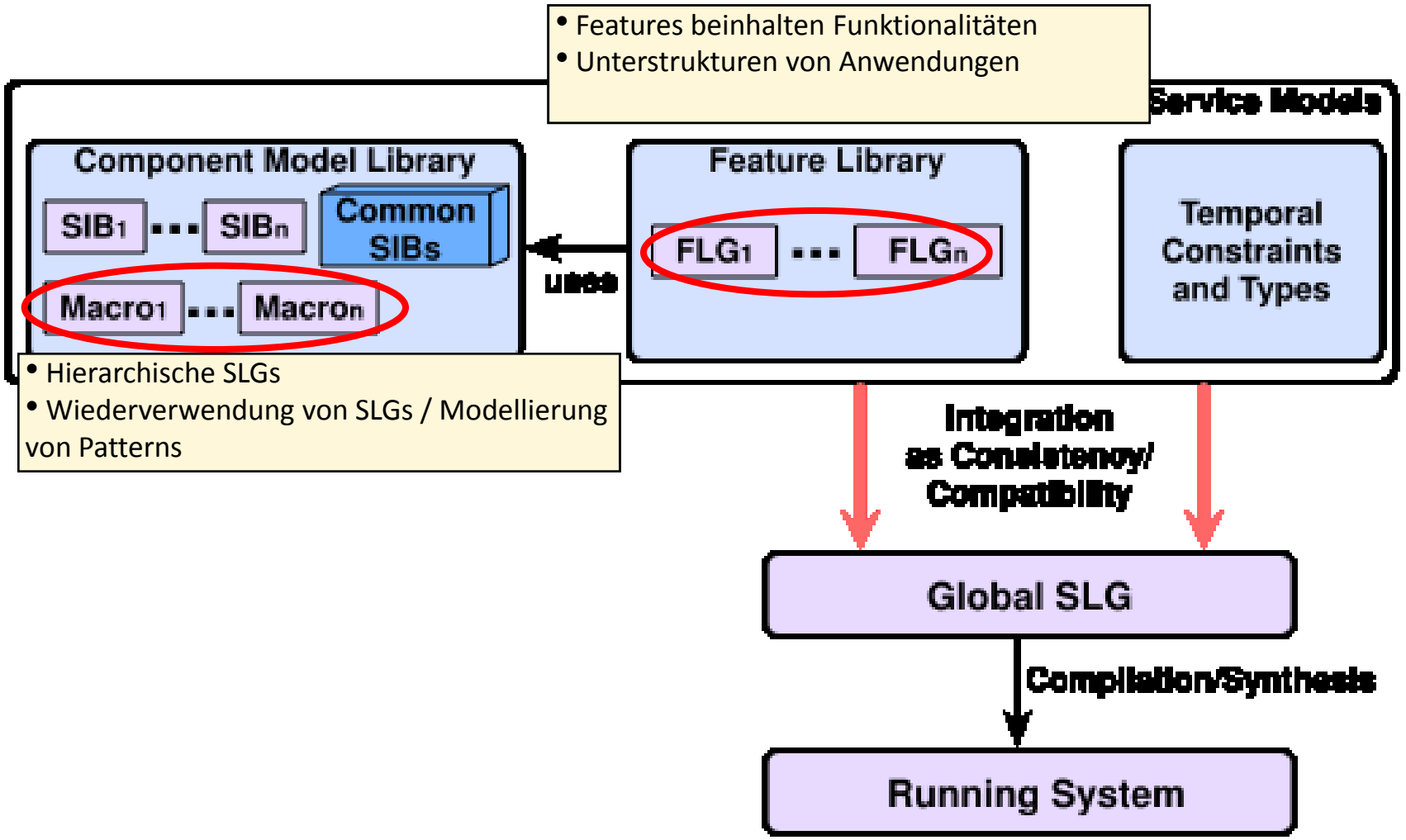
Grafisch nicht-sichtbare **Model Branches** spezifizieren „exits“ von SLGs. Der Kontrollfluss fährt mit dem Eltern-SLG in der Hierarchie fort. Das gleiche Konzept existiert für Parameter (**Model Parameters**).

Control SIBs

- Spezielle SIBs, die den Standardkontrollfluss während der Ausführung ändern können






• Features beinhalten Funktionalitäten
• Unterstrukturen von Anwendungen
Service Models
Component Model Library
SIB1
SIBn
Common SIBs
Macro1
Macron
Feature Library
FLG1
FLGn
Temporal Constraints and Types
• Hierarchische SLGs
• Wiederverwendung von SLGs / Modellierung von Patterns
Integration as Consistency/ Compatibility
Global SLG
Compilation/Synthesis
Running System

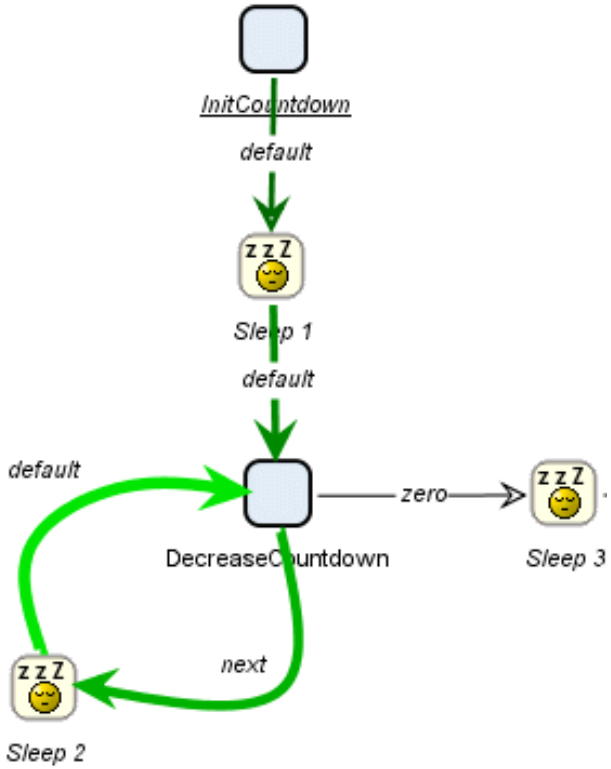
Übersicht

- Einführung
- Service Independent Blocks (SIBs)
- Service Logic Graphs (in jABC)
- jABC Plugins
- Model Checking

jABC Plugins

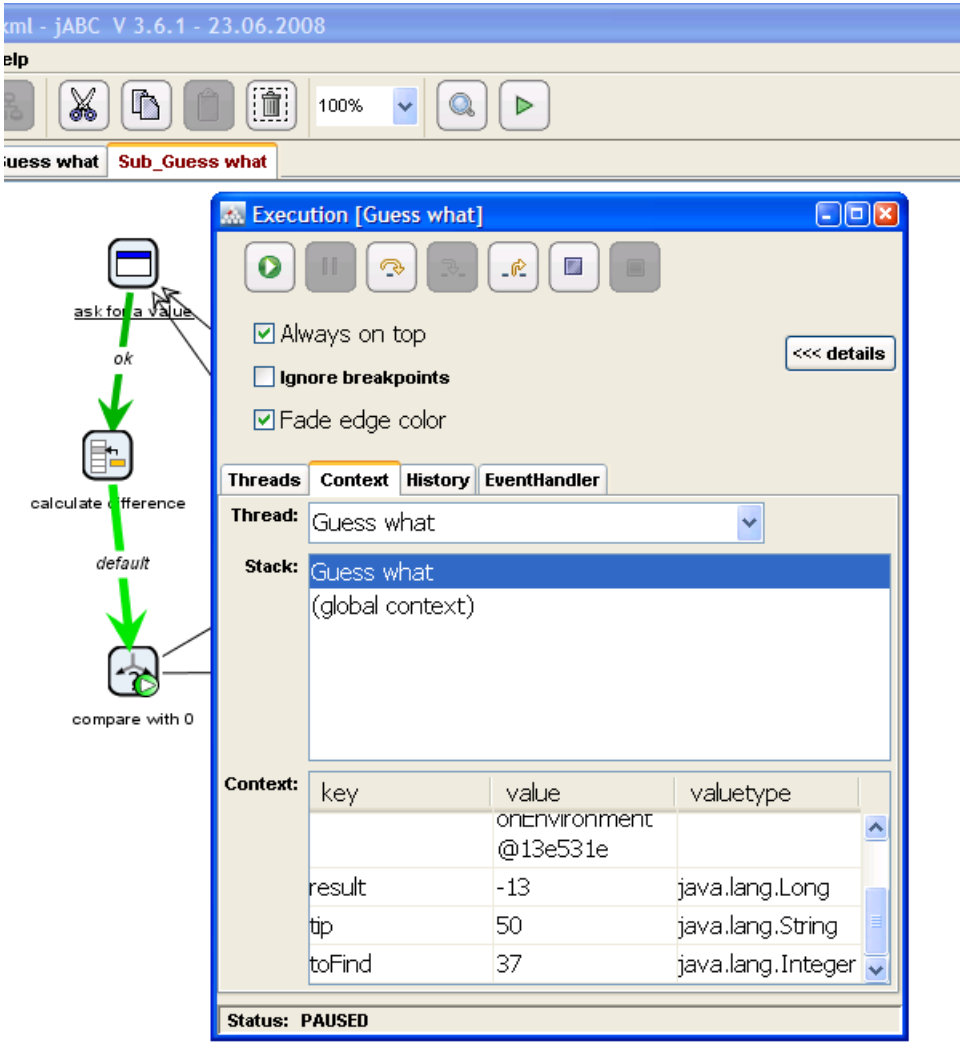
- Reichern SLGs mit Semantik an, indem sie sie interpretieren, z.B. als:
  - Kontrollflussgraphen
  - Syntaxbaum einer Formel
  - Datenschema
  - ...
- Essentiell, um Entwicklungsaktivitäten zu unterstützen wie z.B.:
  - Ausführung, Debugging, Rapid Prototyping
  - Verifikation
  - Code Generierung
  - ...

Ausführung: Tracer

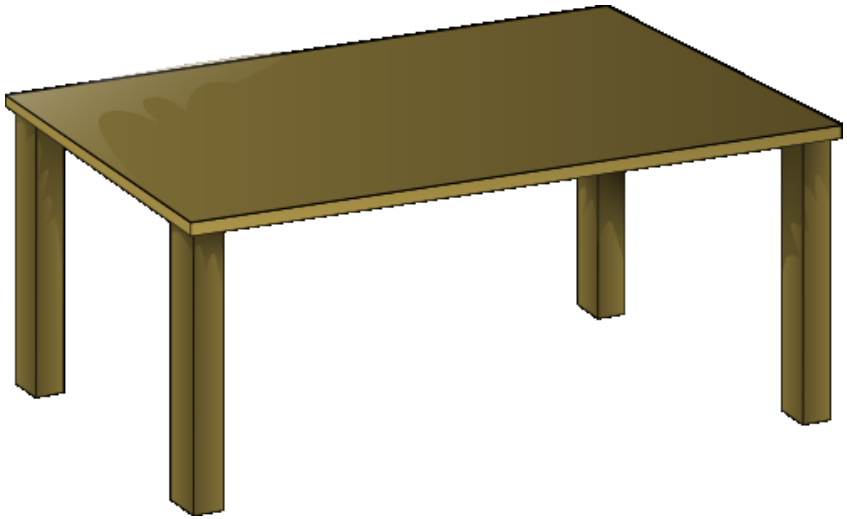


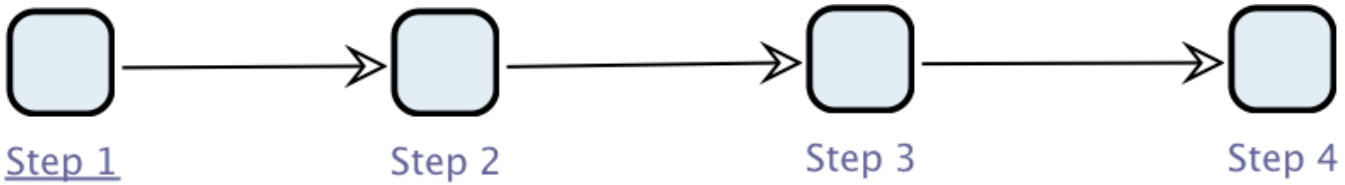
Weitere Tracer Features

- Debugging
  - Breakpoints
  - Betrachtung der Kontextinformationen
  - Schrittweise Ausführung
- Multi-threading mit fork & join
- Event-handling

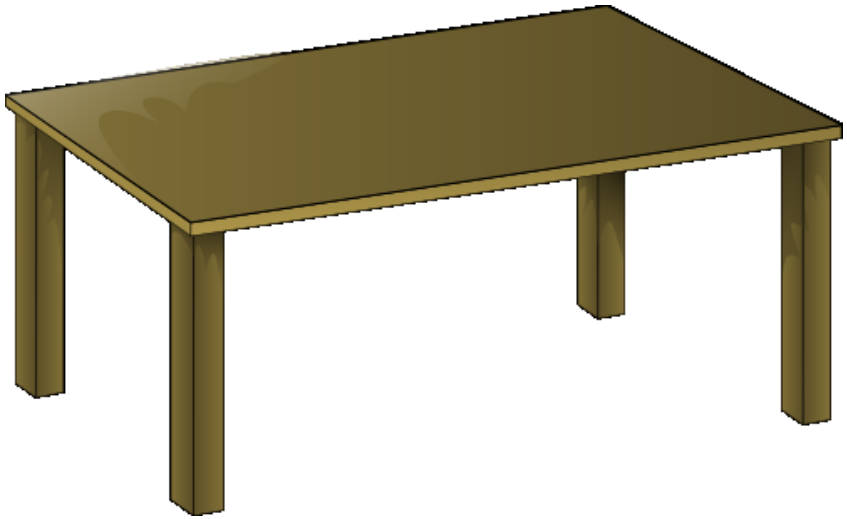


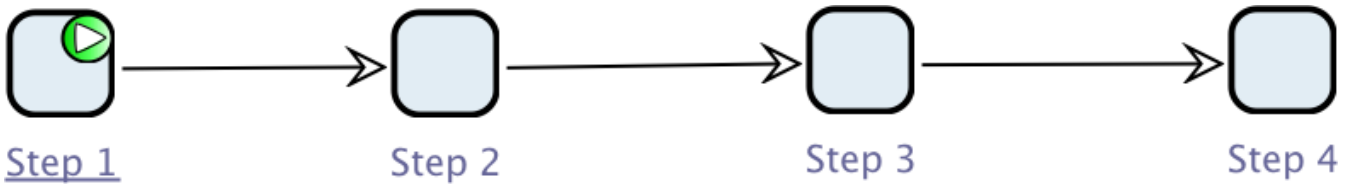
Ausführungskontext



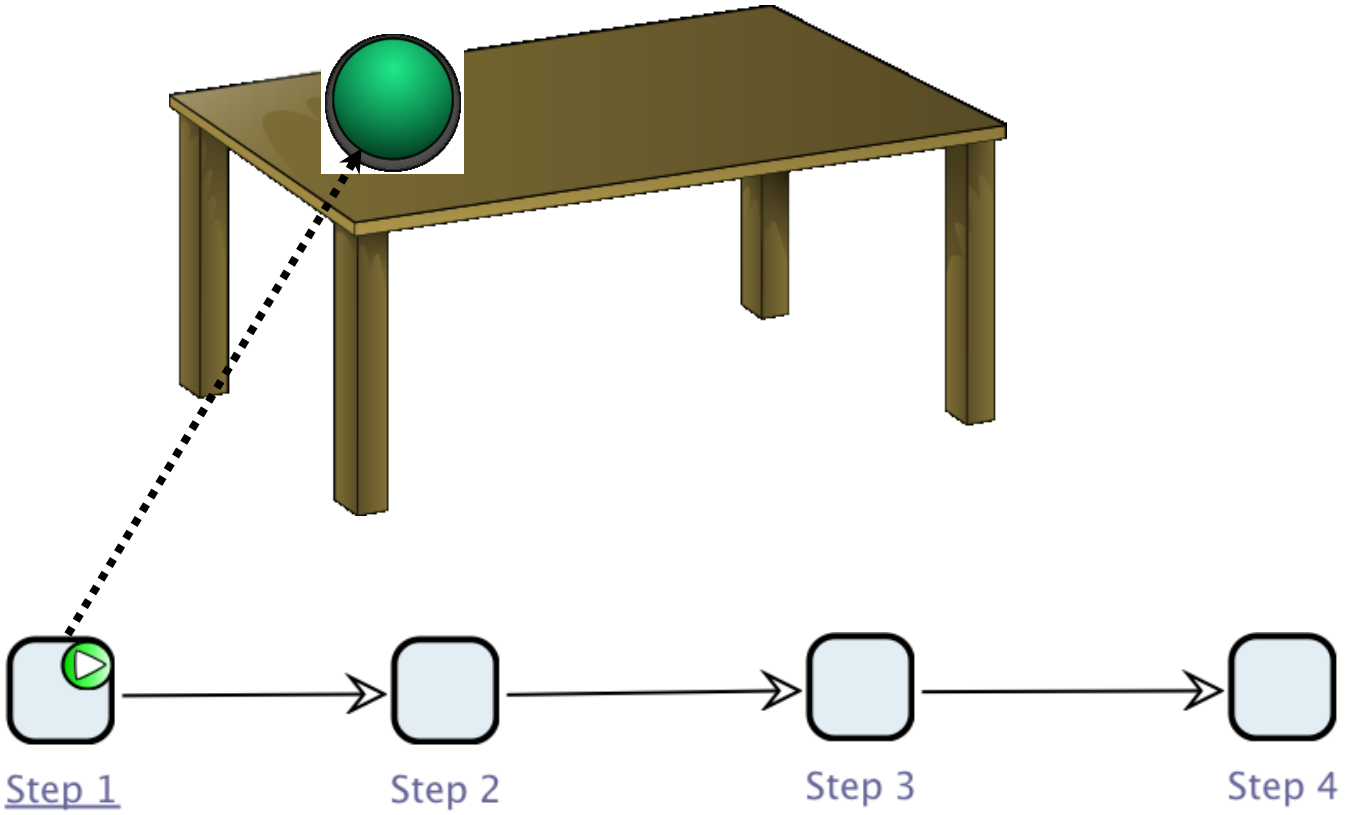


Ausführungskontext

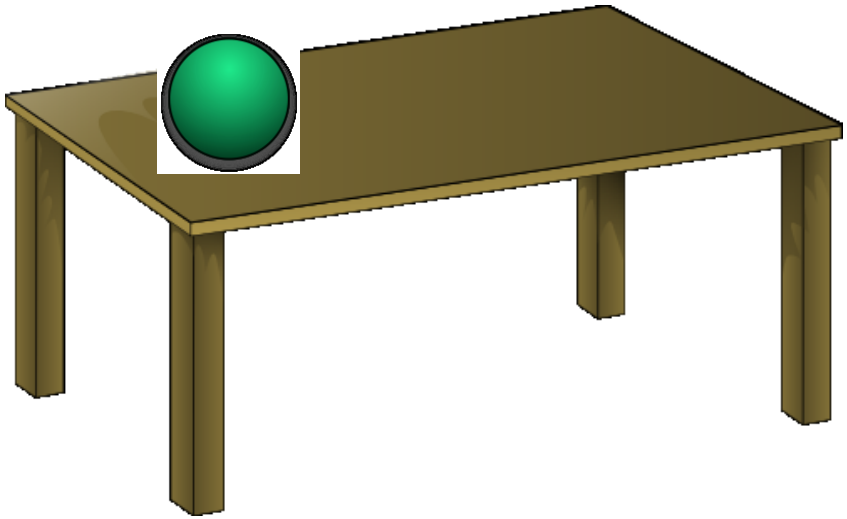


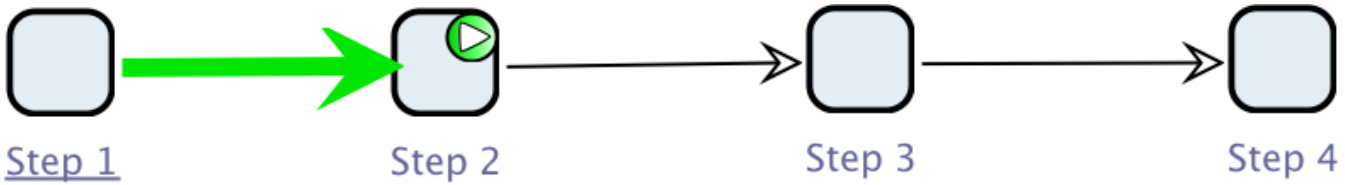


Ausführungskontext

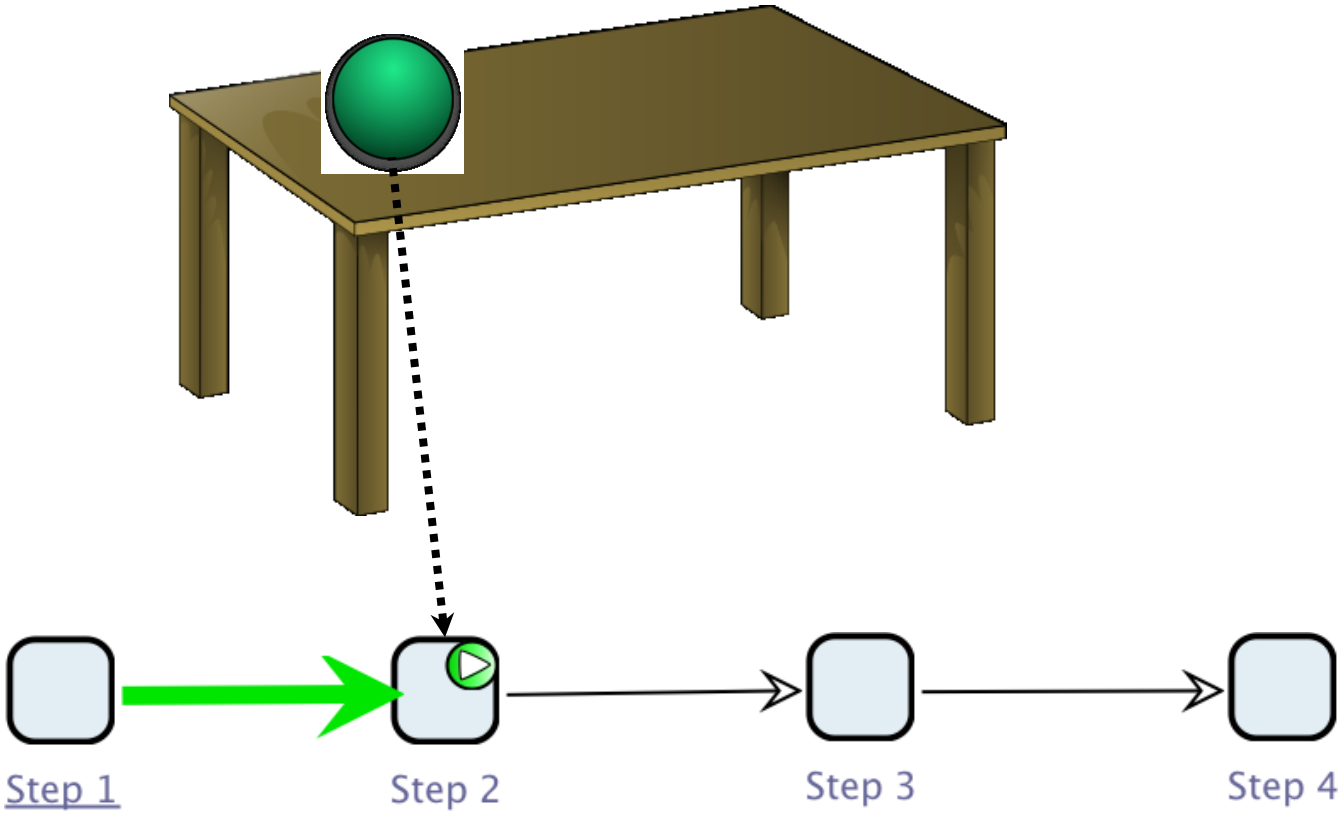


Ausführungskontext

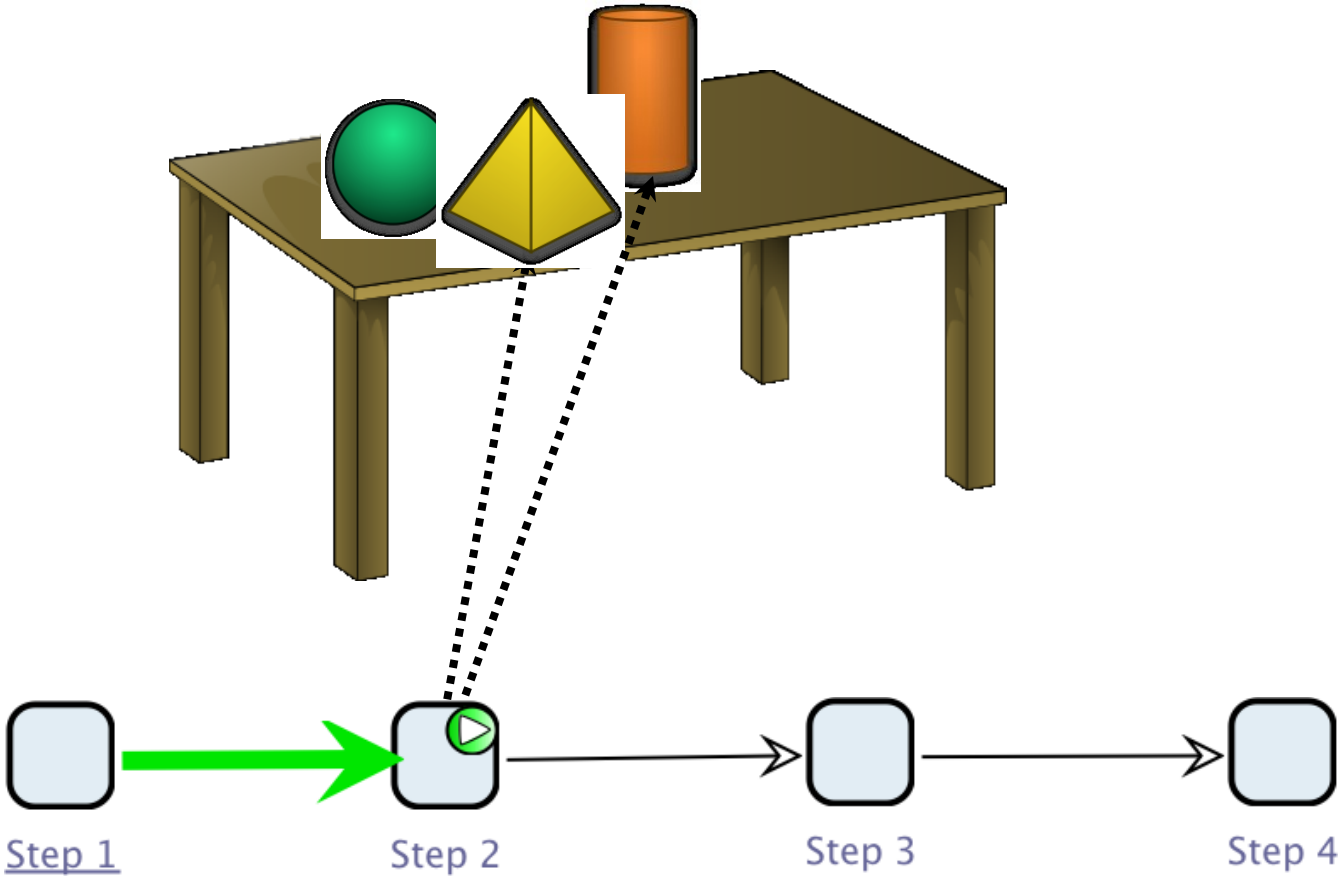




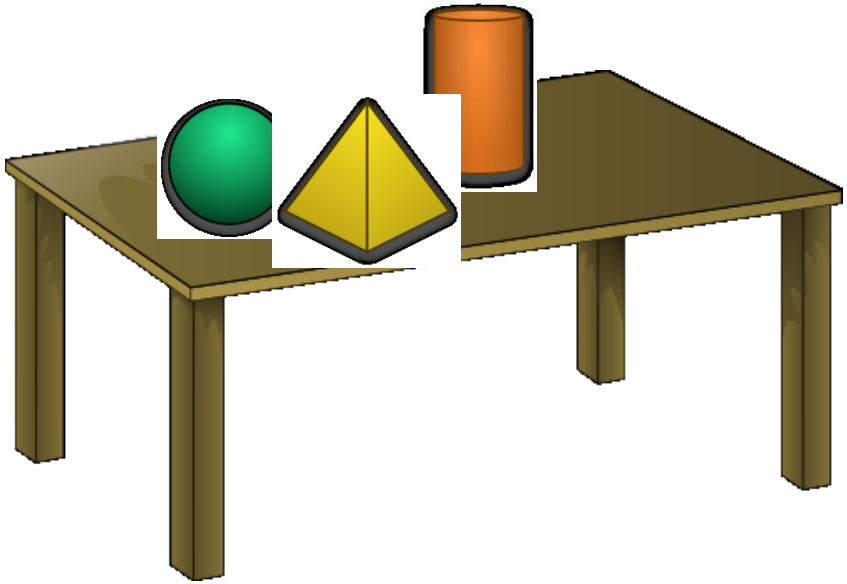
Ausführungskontext

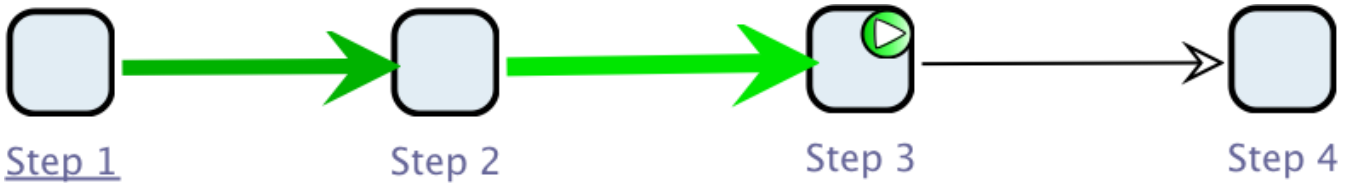


Ausführungskontext

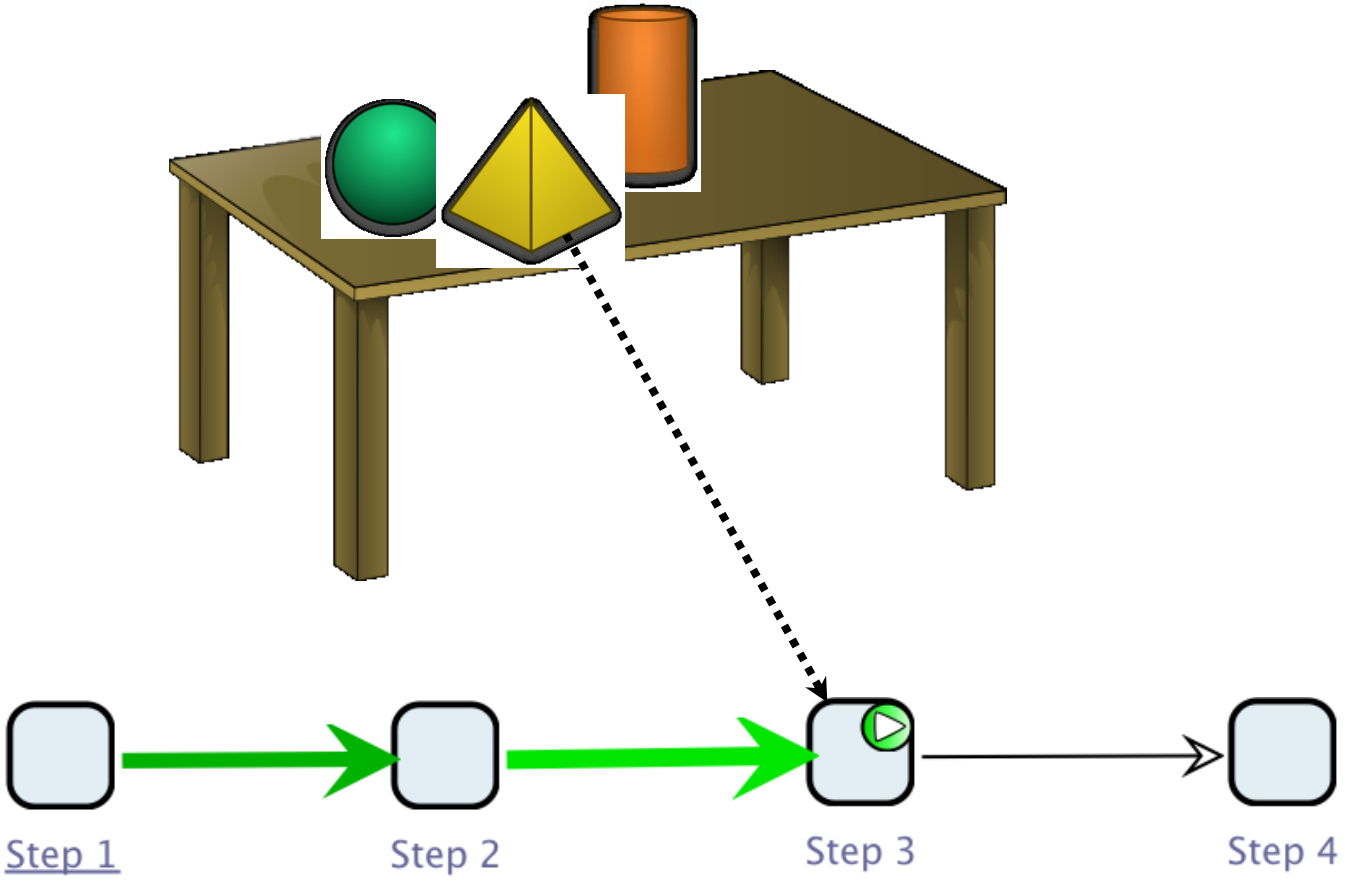


Ausführungskontext

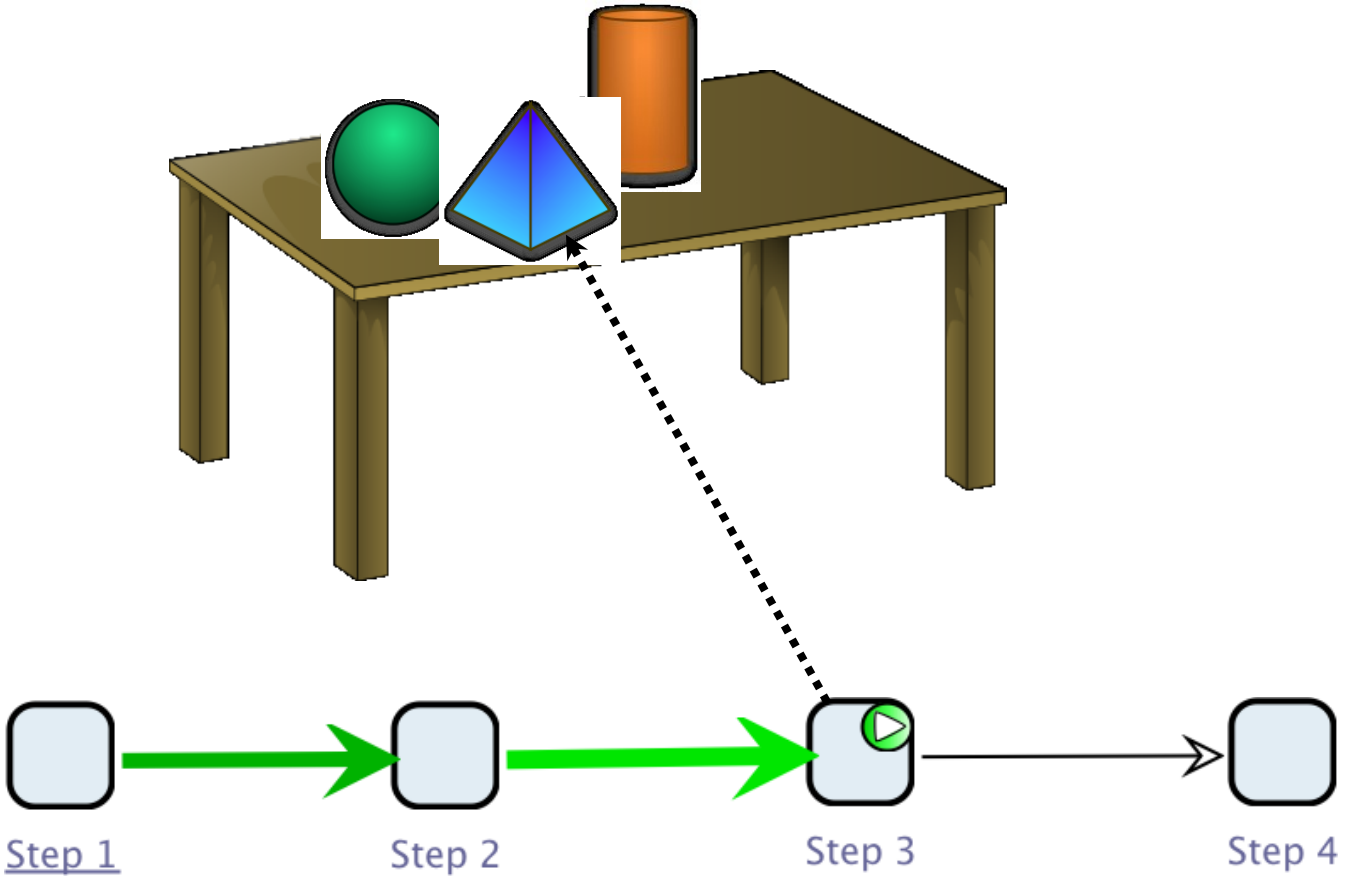




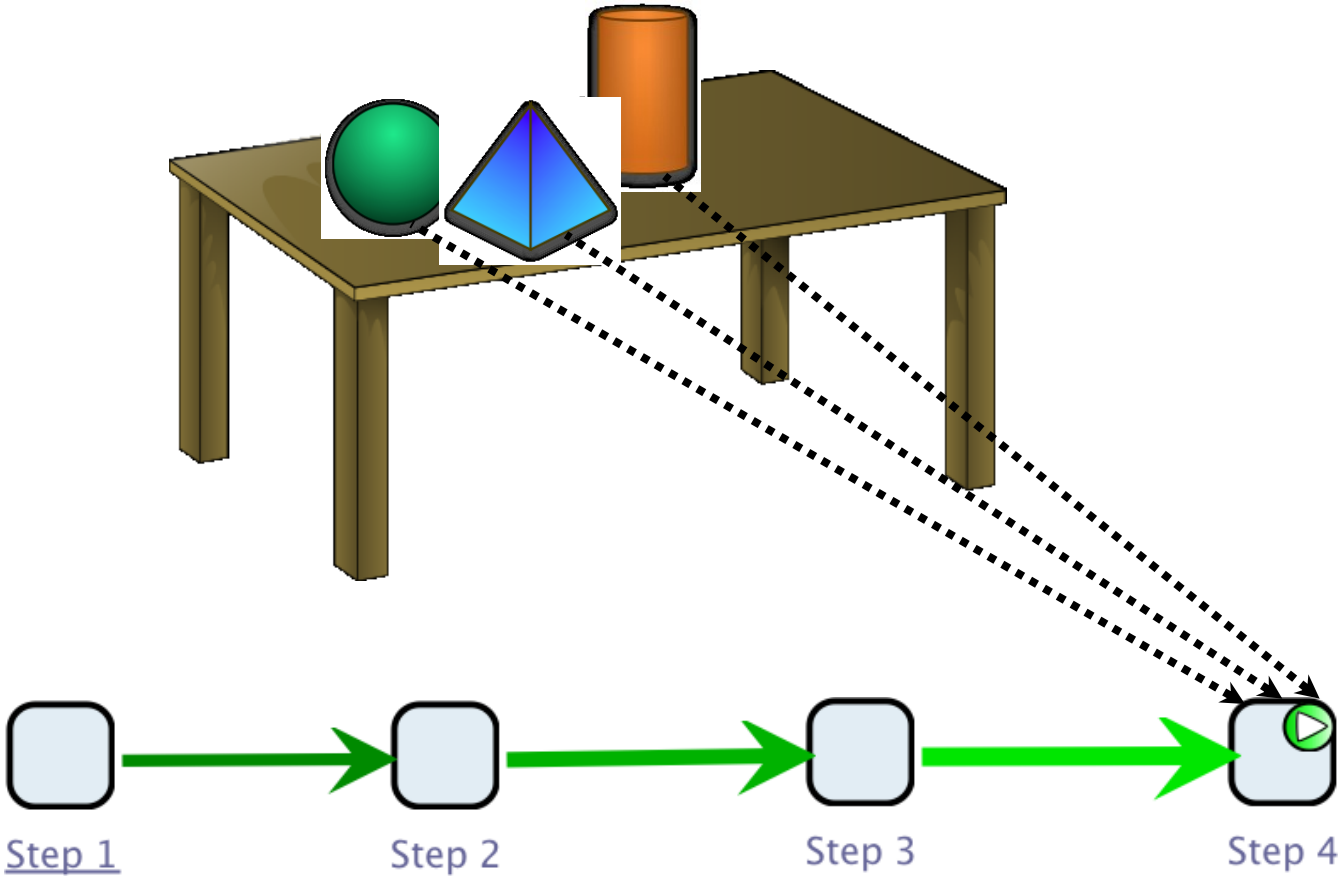
Ausführungskontext



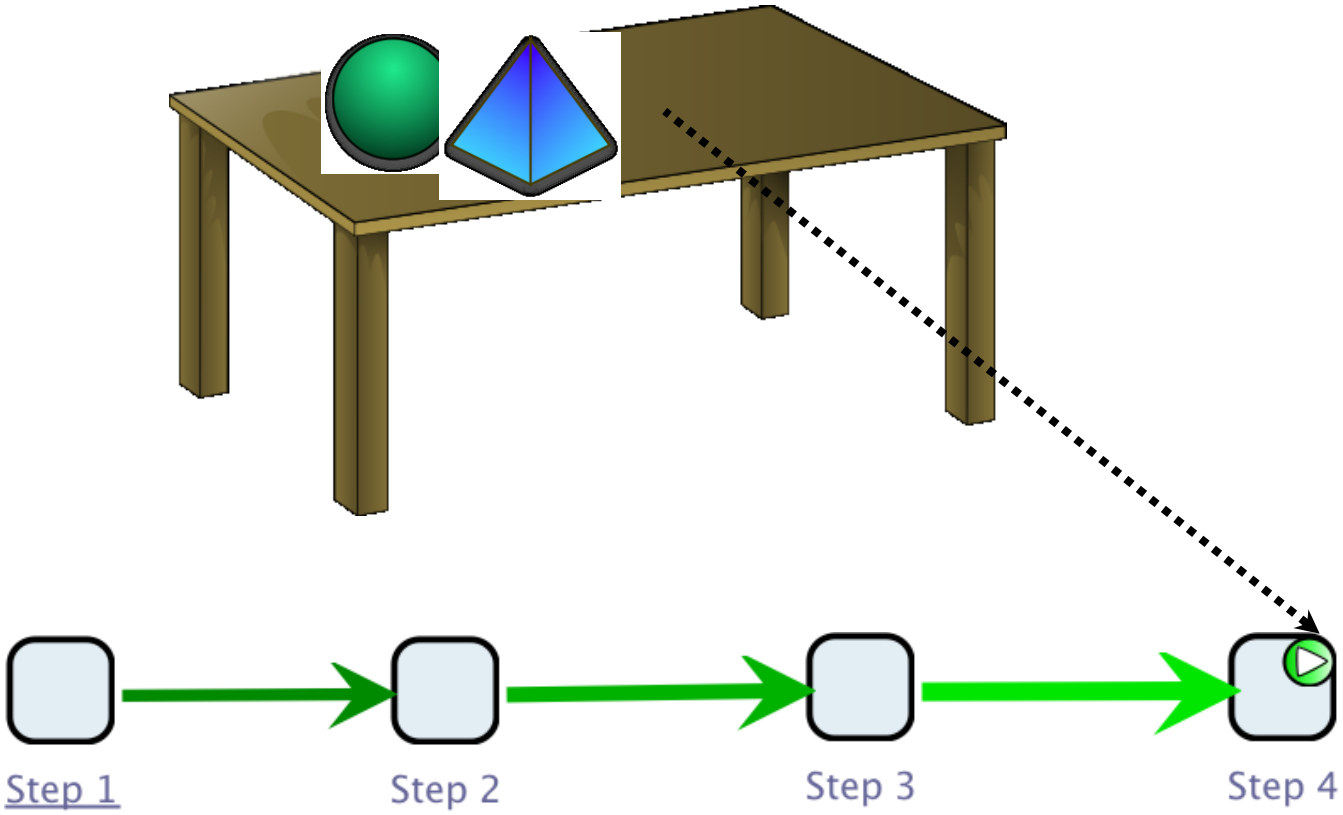
Ausführungskontext



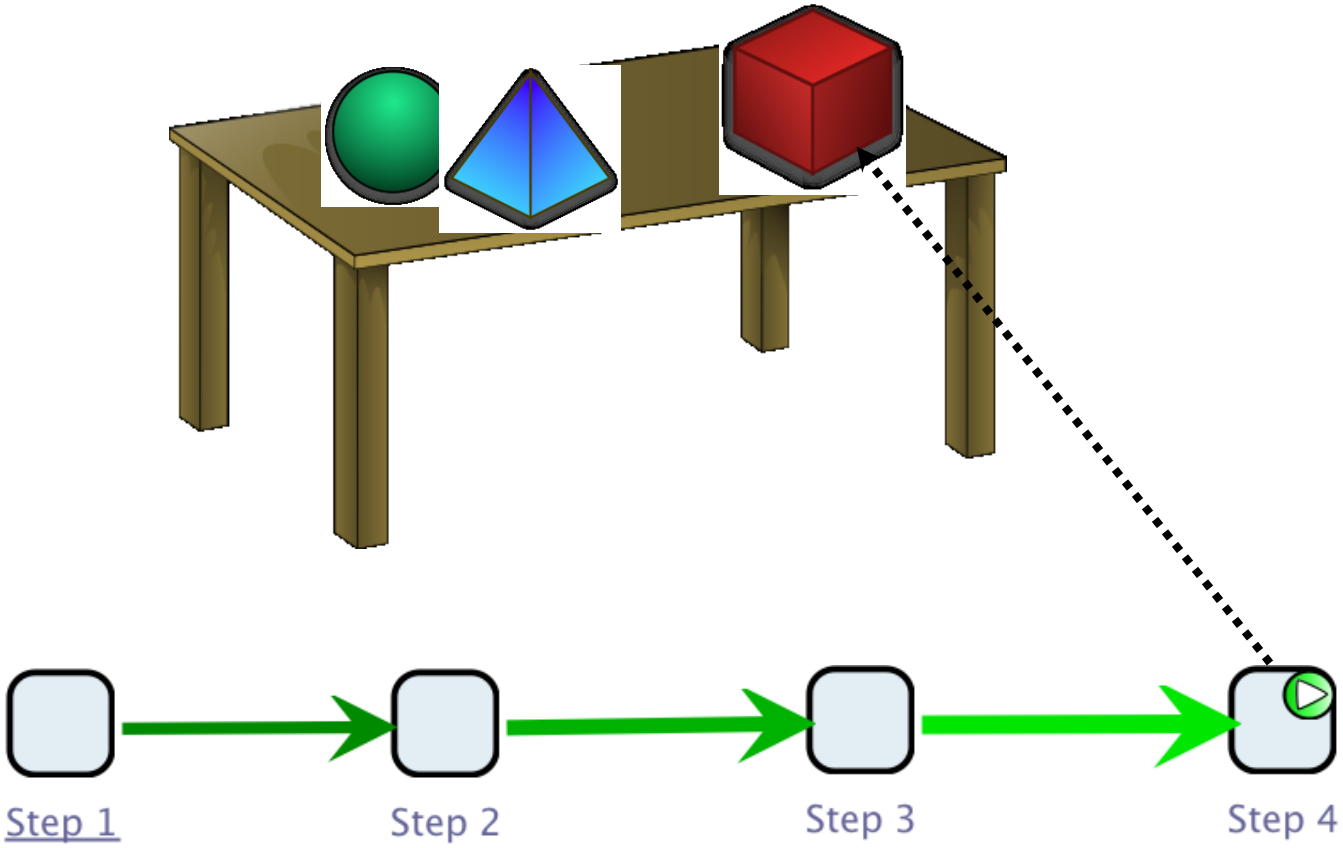
Ausführungskontext



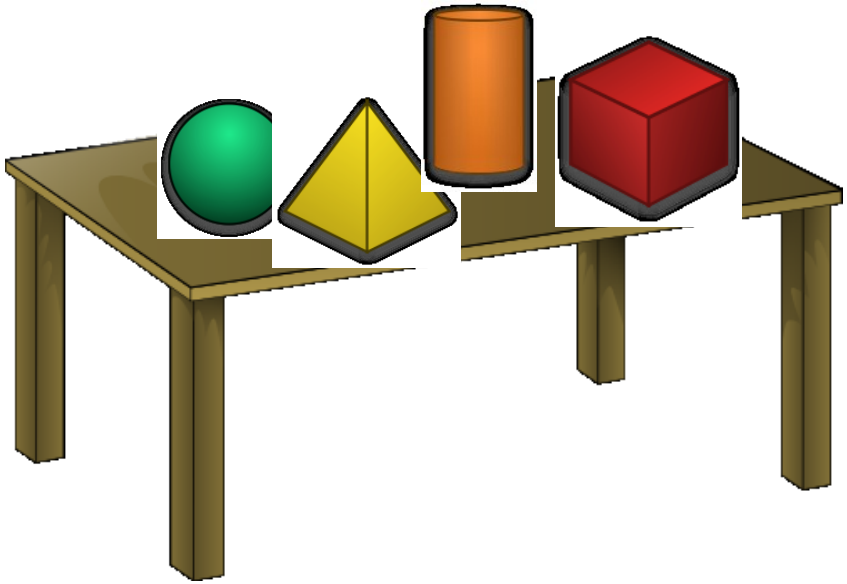
Ausführungskontext



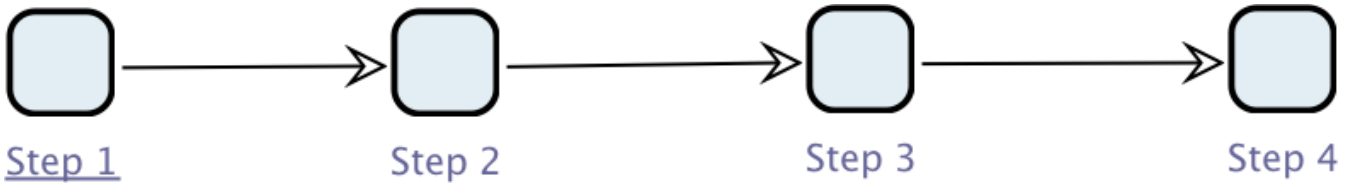
Ausführungskontext



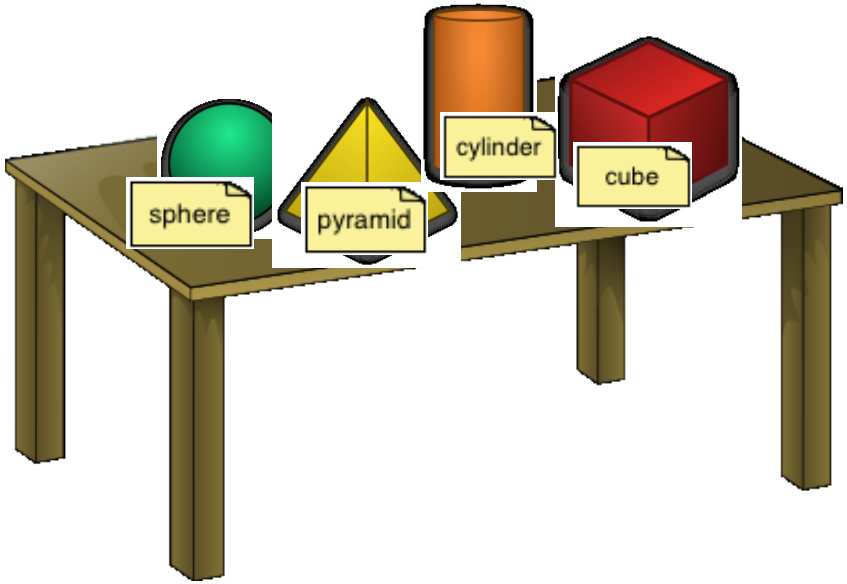
Ausführungskontext

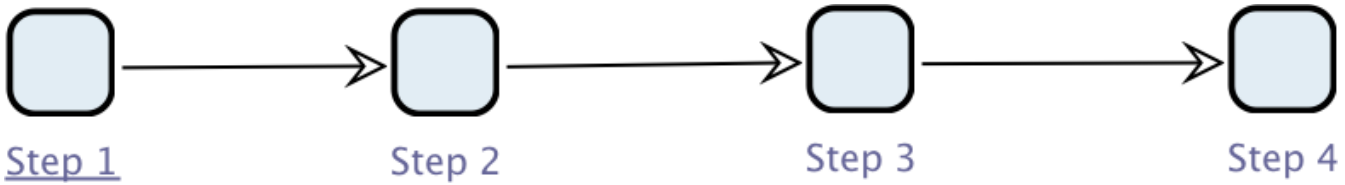


Frage:
Wie können die Objekte auf dem Tisch identifiziert werden?

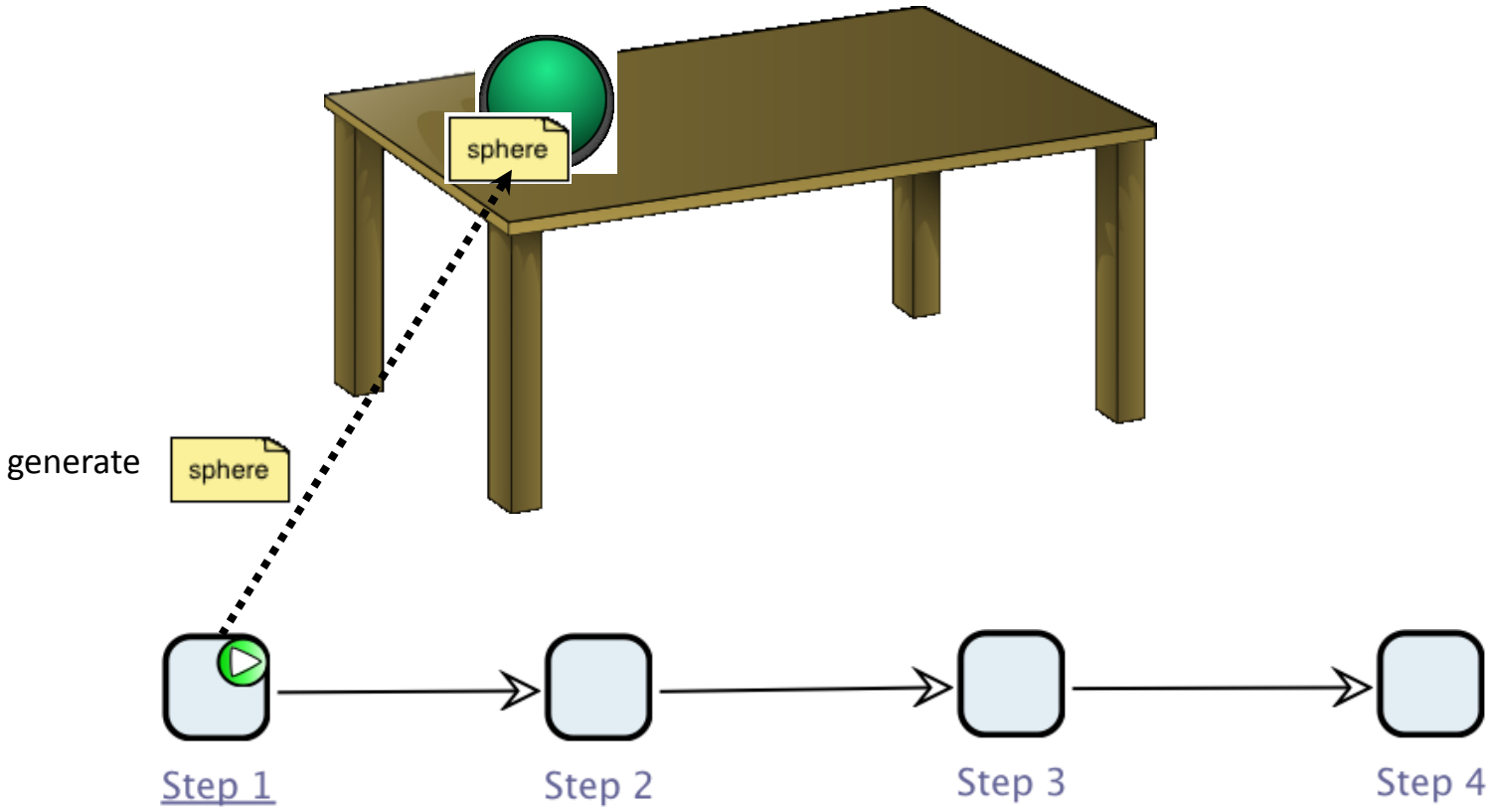


Antwort: Context Keys

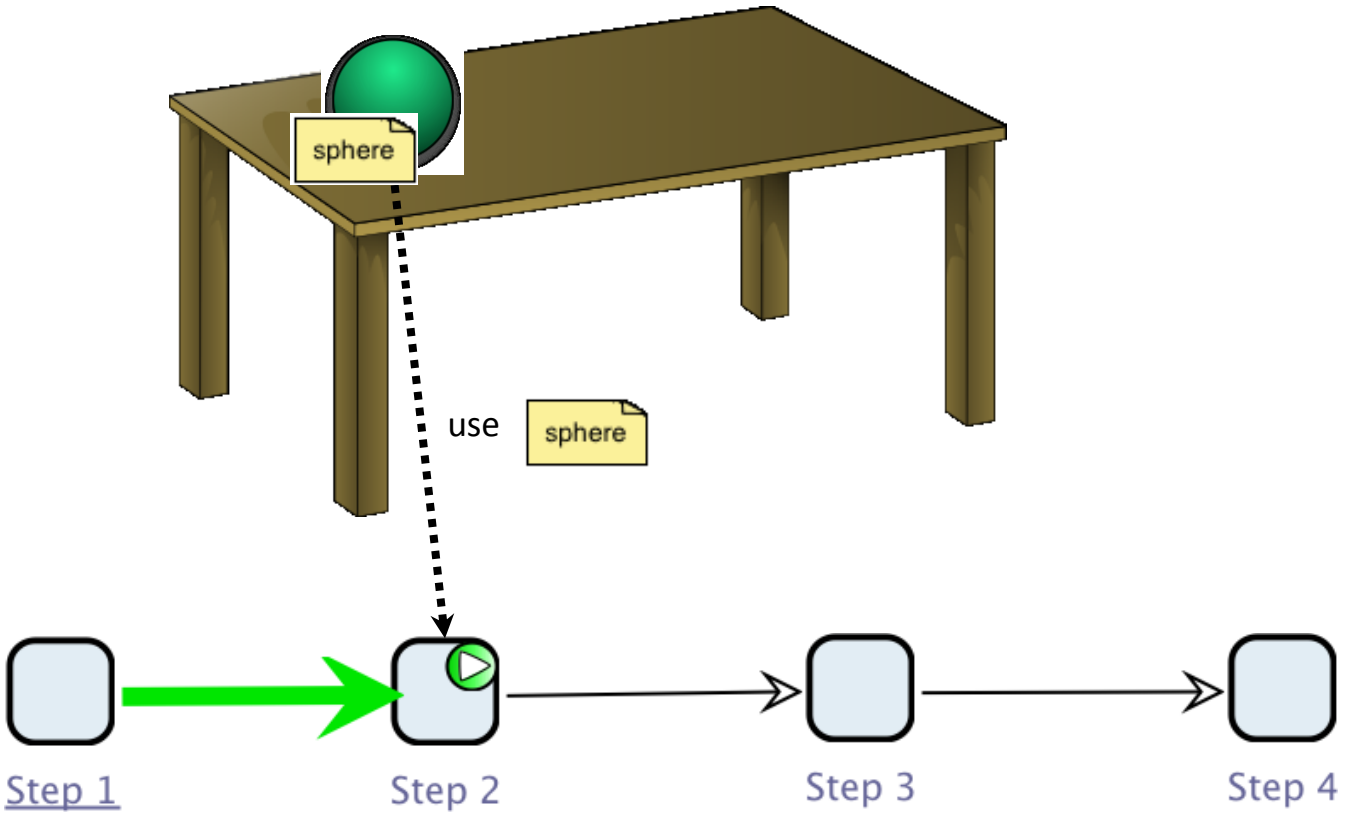




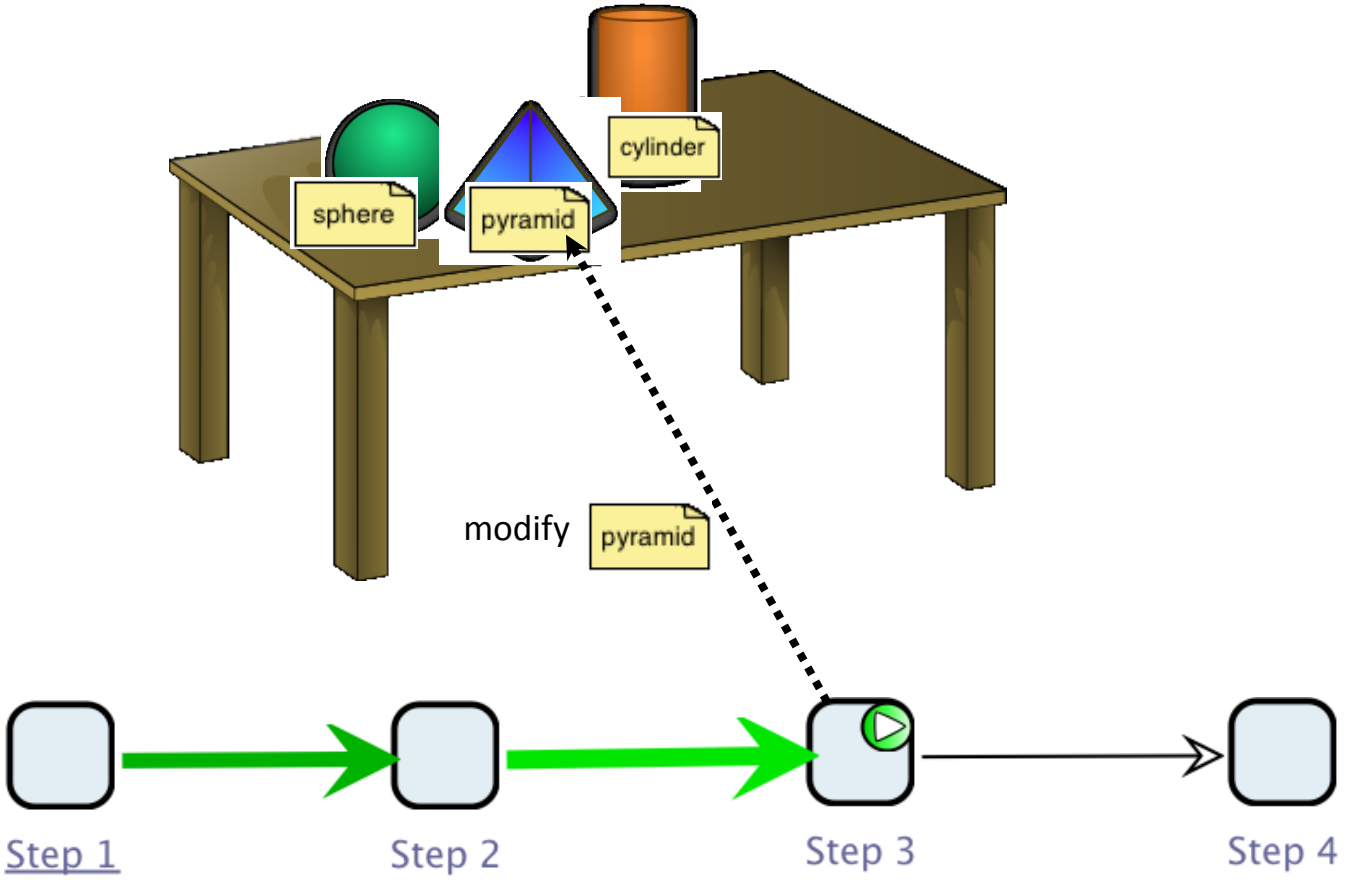
Ausführungskontext



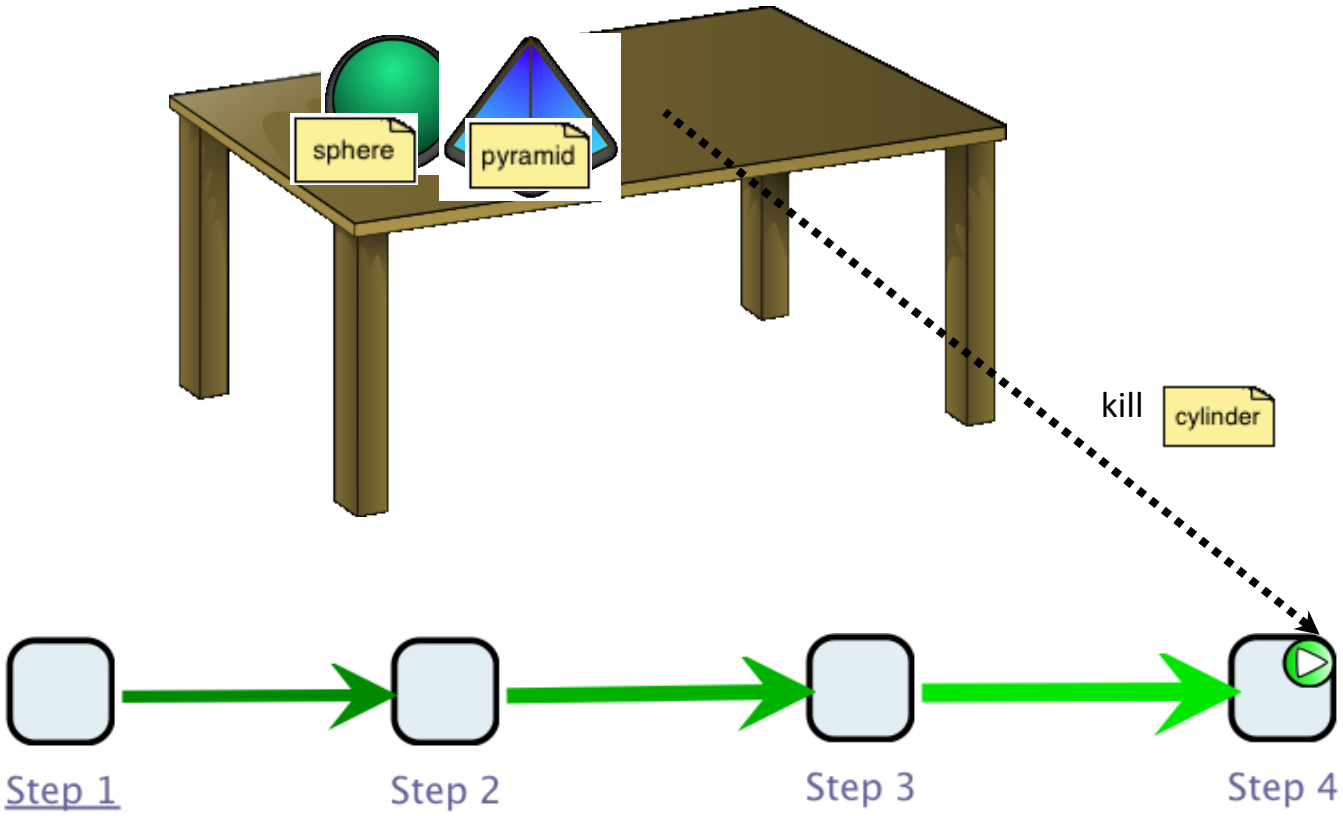
Ausführungskontext

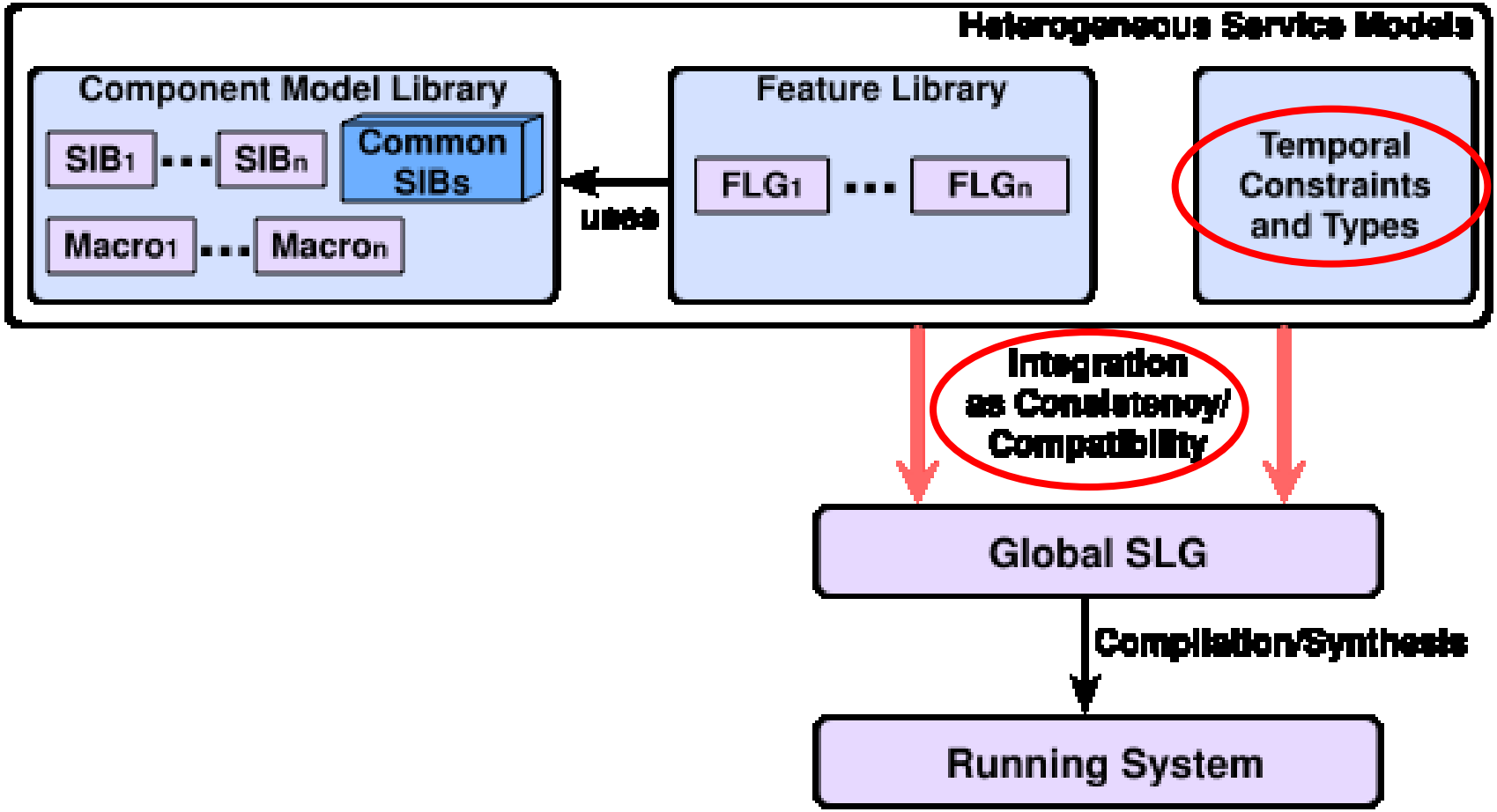


Ausführungskontext



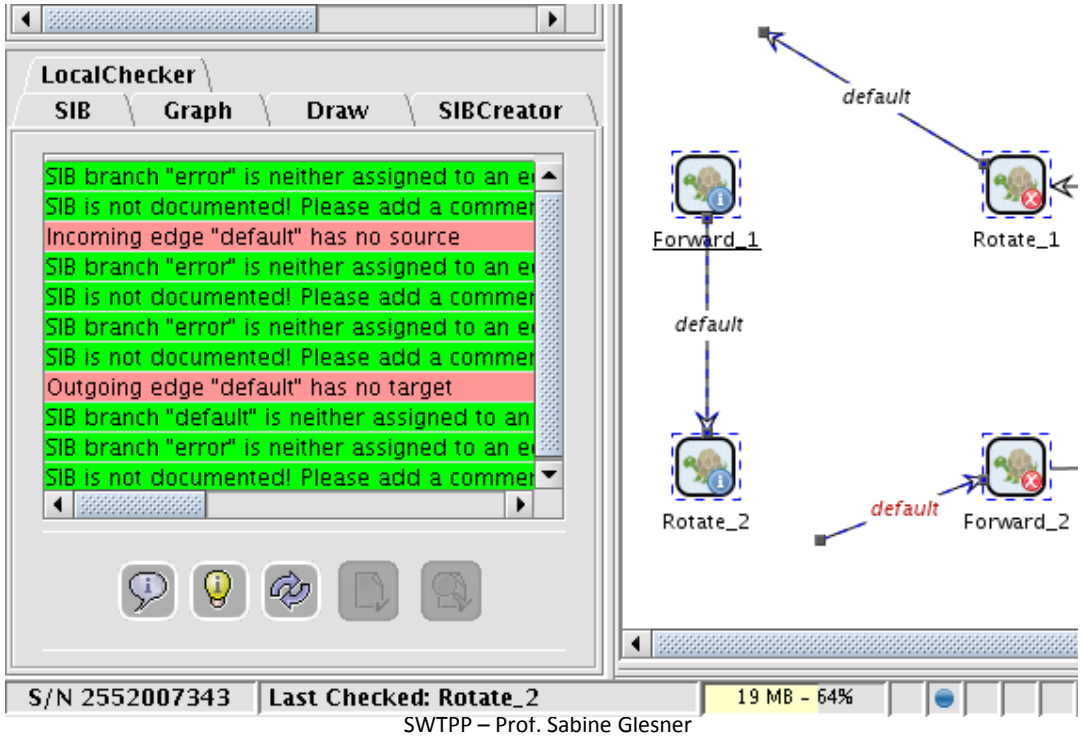
Ausführungskontext




Heterogeneous Service Models
Component Model Library
SIB1
SIBn
Common SIBs
Macro1
Macron
uses
Feature Library
FLG1
FLGn
Temporal Constraints and Types
Integration as Consistency/ Compatibility
Global SLG
Compilation/Synthesis
Running System

Verifikation: Local Checker

- Prüfen von lokalen Bedingungen, die die korrekte Verwendung von SIBs spezifizieren (Parametrisierung, unbehandelte Branches, etc.)

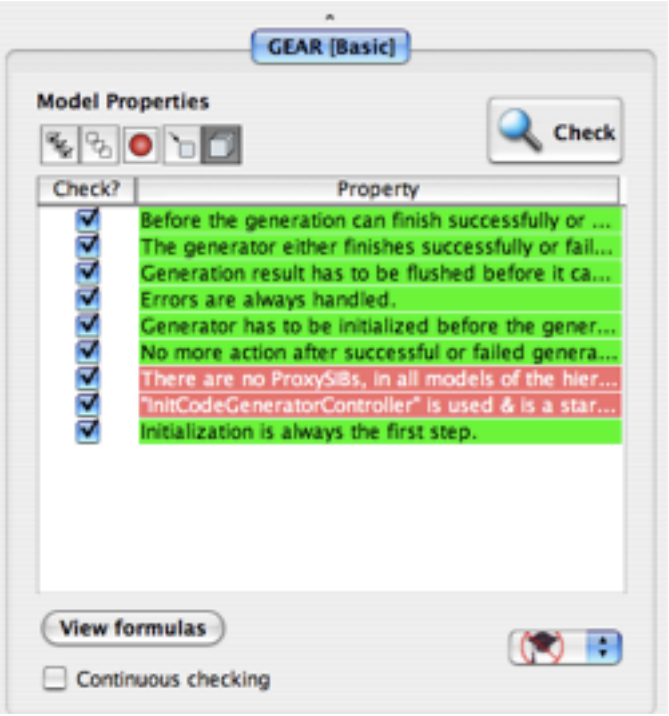


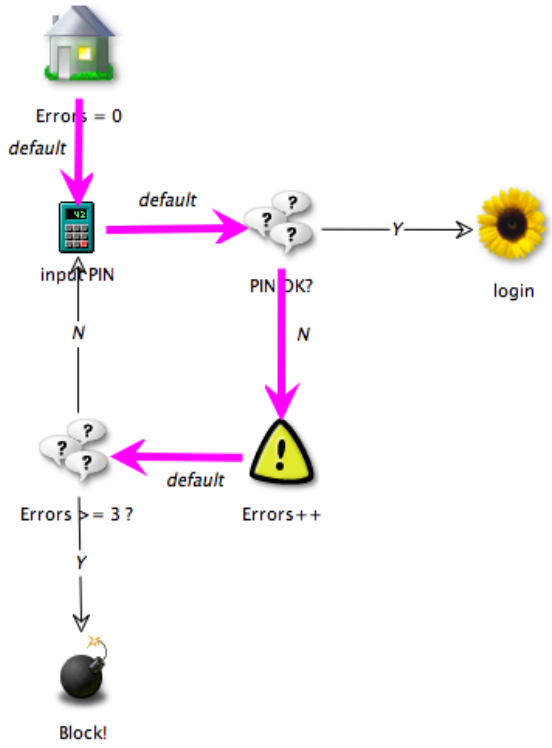
Übersicht

- Einführung
- Service Independent Blocks (SIBs)
- Service Logic Graphs (in jABC)
- jABC Plugins
- Model Checking

Verifikation: Model Checker „GEAR“

- Prüfen von globalen Bedingungen auf dem gesamten SLG
- Z.B.: „Ein Benutzer muss sich immer erst einloggen, bevor er sich ausloggen kann.“



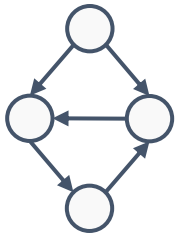


Verifikation: Model Checker „GEAR“

- Annotation von atomaren Aussagen an SIBs
- Spezifikation von temporalen Anforderungen mit CTL (Computation Tree Logic)
- Automatische Verifikation, ob Eigenschaft erfüllt ist oder Finden eines Gegenbeispiels

Model Checking

Endliches Modell



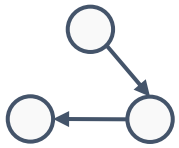
G c

G (a ⇒ X b)

Eigenschaft in Temporallogik

Model Checker (automatisch)

Gegenbeispiel



Eigenschaft erfüllt

Gegenbeispiel

- Gegenbeispiel = Trace, der die gegebene Eigenschaft verletzt
- Das Finden eines Gegeneispiels ist häufig schneller als zu zeigen, dass die Eigenschaft erfüllt ist
- Die Verletzung einer Eigenschaft kann auf einen Fehler im Modell oder der Anforderungsspezifikation zurückzuführen sein

Model Checking

- Mit Model Checking können wir prüfen, ob ein Modell $\mathcal{M}$ eine Eigenschaft φ erfüllt: $\mathcal{M} \models \varphi$
- Das Modell $\mathcal{M}$ ist eine Kripke Struktur (beschrifteter Transitionsgraph)
- Eigenschaften werden in Temporallogik ausgedrückt

Kripke Struktur

- Eine Kripke Struktur ist ein Tupel (S, T, S_0, L) mit
  - S ist eine Menge von Zuständen
  - T $\subseteq S \times S$ ist eine Transitionsrelation
  - $S_0 \subseteq S$ ist eine Menge von Initialzuständen
  - $L: S \rightarrow 2^{AP}$ ist eine Beschriftungsfunktion
    ($L(s)$: Menge von atomaren Aussagen, die in s wahr sind)

- Um Model Checking zu ermöglichen muss S endlich sein!

Eigenschaften

- Eigenschaften werden über Berechnungspfade definiert
- Beispiele:
  - Es gibt höchstens einen Prozess im kritischen Bereich
  - Ein Fehlerzustand wird nie erreicht
  - Wenn a gilt, dann ist ein Zustand erreichbar, in dem b gilt
  - Bei einer Anfrage wird das System irgendwann antworten

Klassifizierung von Eigenschaften

- Sicherheit: “something bad will never happen”
  - Es werden niemals zwei Prozesse gleichzeitig im kritischen Bereich sein.
  - Ein Fehlerzustand wird niemals auftreten.
- Lebendigkeit: “something good will eventually happen“
  - Eine bestimmte Eigenschaft wird irgendwann erfüllt sein.
  - Bei einer Anfrage wird das System irgendwann antworten.
- Fairness: Wenn eine Aktion oft genug verfügbar ist, wird sie auch ausgeführt.
  - Wenn ein Prozess unendlich oft den kritischen Bereich betreten kann, dann wird er das auch unendlich oft tun (und nicht für immer den kürzeren ziehen).

Temporallogik

- “Temporal logics have proven to be useful for specifying concurrent systems, because they can describe the ordering of events in time without introducing time explicitly”
- [Clarke, Grumberg and Peled 1999]

- Temporallogik drückt Eigenschaften wie die folgenden aus „auf allen Pfaden …” oder „auf mindestens einem Pfad …”
- Temporallogik beschreibt Änderungen von Werten über die Zeit „x ist immer …“ oder „x wird irgendwann“
- Zusammen: „auf allen Pfaden gilt immer …“, „auf mindestens einem Pfad gilt irgendwann …“

LTL vs. CTL

- Zwei Interpretationen von “temporal”
  - Lineare Zeit:
    - lineare (total geordnete) Sequenz von Zeitpunkten
      ⇨ Linear-time Temporal Logic (LTL)
  - Verzweigende Zeit:
    - Mehrere Nachfolgezustände möglich
    - Die Menge der möglichen (verzweigenden) Pfade kann als Berechnungsbaum beschrieben werden
      ⇨ Computation Tree Logic (CTL)

Linear-time Temporal Logic (Syntax)

- Induktive Definition von LTL Formeln:
  - ϕ ::= $\bot \mid \top \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \Rightarrow \phi) \mid$
    $X\,\phi \mid F\,\phi \mid G\,\phi \mid \phi 1\ U\ \phi 2 \mid \phi 1\ W\ \phi 2$
  - $\bot$ und $\top$ bezeichnen „false“ und „true“
  - p bezeichnet eine atomare Formel (Aussage)
  - Die logischen Operatoren sind die üblichen
  - Temporale (auch modale genannt) Operatoren:

$X\,\phi$	“ϕ muss im nächsten Zustand gelten”
$F\,\phi$	“ϕ muss irgendwann in einem zukünftigen Zustand gelten (kann auch der aktuelle Zustand sein)”
$G\,\phi$	“ϕ muss global in allen zukünftigen Zuständen gelten (betrifft auch den aktuellen Zustand)”
$\phi 1\ U\ \phi 2$	“$\phi 2$ gilt irgendwann, bis dahin muss $\phi 1$ in allen Zuständen gelten”
$\phi 1\ W\ \phi 2$	“$\phi 1$ gilt durchgängig bis $\phi 2$ gilt oder $\phi 1$ gilt in allen Zuständen” (Weak until)

LTL-Formeln

- Welche der folgenden LTL-Formeln X *p*, F *p* und G *p* sind erfüllt?

	0	**1**	**2**		**0**	**1**	**2**		**0**	**1**	**2**		**0**	**1**	**2**	
p	F	T	F	*p*	F	F	T	*p*	T	T	F	*p*	T	T	T	…

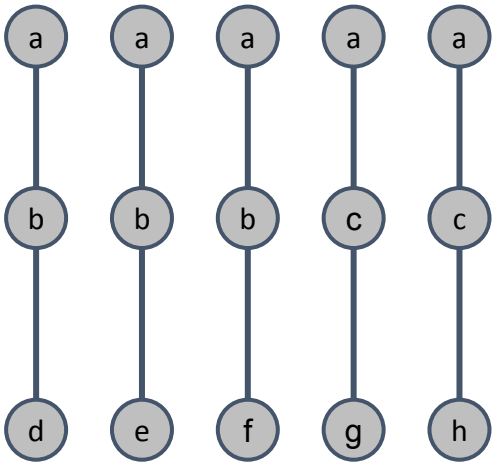
- Welche der folgenden LTL-Formeln *p* U *q* und *p* W *q* sind erfüllt?

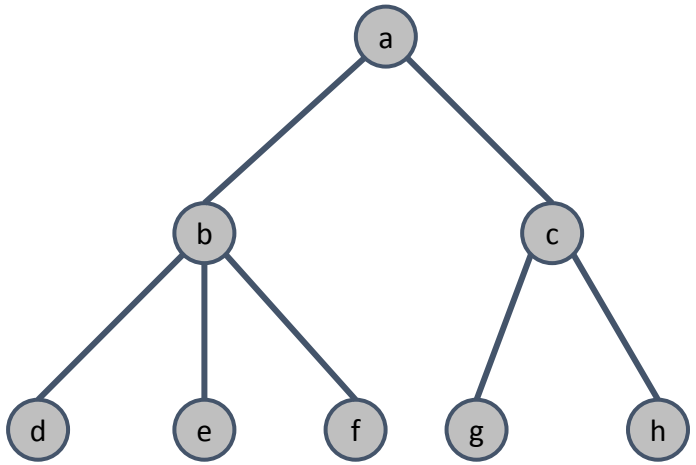
	0	**1**	**2**		**0**	**1**	**2**		**0**	**1**	**2**
p	T	T	F	*p*	T	F	T	*p*	T	F	T
q	F	F	T	*q*	F	F	T	*q*	F	F	F

Computation Tree Logic (Syntax)

- Induktive Definition von CTL Formeln:
  - ϕ ::= $\bot \mid \top \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \Rightarrow \phi) \mid$
    $AX\,\phi \mid EX\,\phi \mid AG\,\phi \mid EG\,\phi \mid AF\,\phi \mid EF\,\phi \mid$
    $A\,[\phi 1\ U\ \phi 2] \mid E\,[\phi 1\ U\ \phi 2] \mid A\,[\phi 1\ W\ \phi 2] \mid E\,[\phi 1\ W\ \phi 2]$
  - $\bot$ und $\top$ bezeichnen „false“ und „true“
  - p bezeichnet eine atomare Formel (Aussage)
  - Die logischen Operatoren sind die üblichen
  - Temporale (bzw. modale) Operatoren wie in LTL:
  - $X\,\phi$, $F\,\phi$, $G\,\phi$, $\phi 1\ U\ \phi 2$, $\phi 1\ W\ \phi 2$
  - Pfadquantoren
    - A “auf allen Pfaden” (Always)
    - E “auf mindestens einem Pfad” (Exists)

LTL vs CTL in Bildern

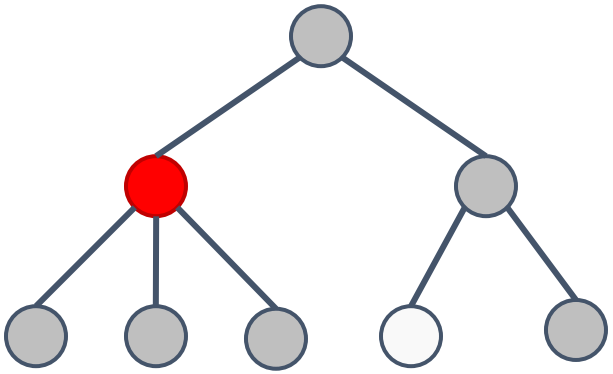




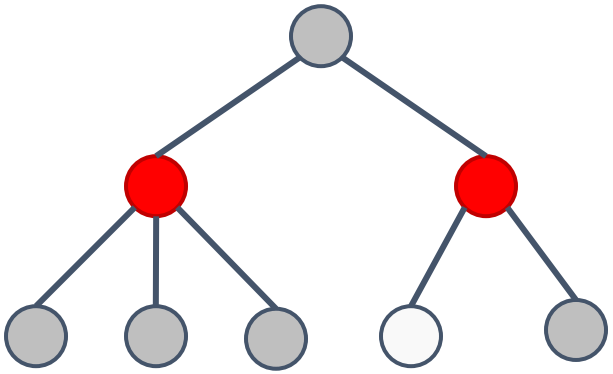
LTL betrachtet eine Menge von Pfaden durch das Programm

CTL betrachtet den Berechnungsbaum

CTL Operatoren 1/4

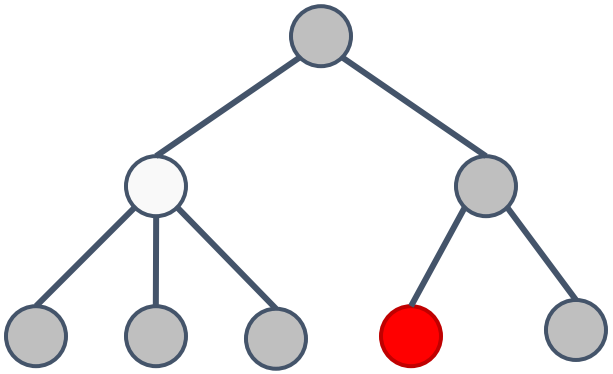


EX p
Es existiert mindestens ein Pfad, auf dem p im nächsten (next) Zustand gilt

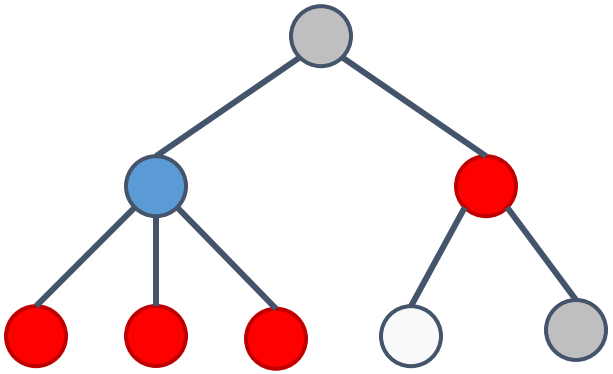


AX p
Auf allen Pfaden gilt p im nächsten (next) Zustand

CTL Operatoren 2/4

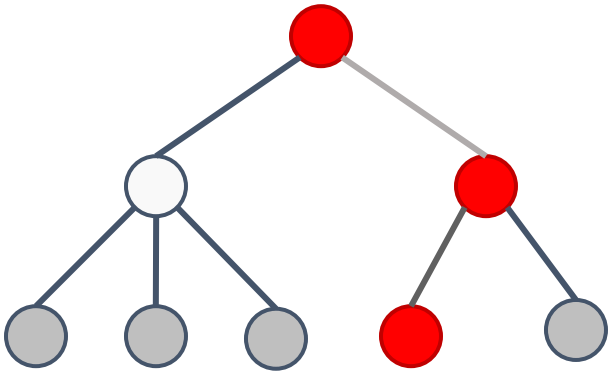


EF p
Es existiert mindestens ein Pfad, auf dem p irgendwann in einem zukünftigen (future) Zustand gilt.

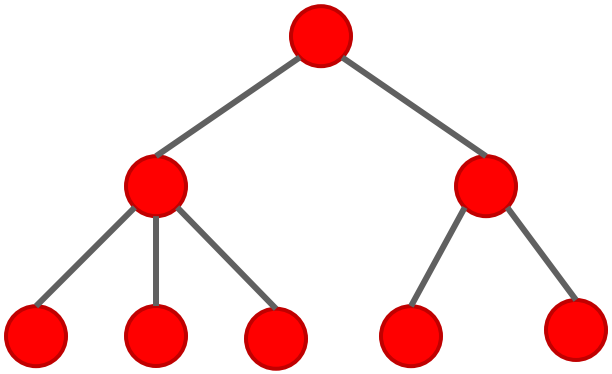


AF p
Auf allen Pfaden gilt p irgendwann in einem zukünftigen (future) Zustand.

CTL Operatoren 3/4

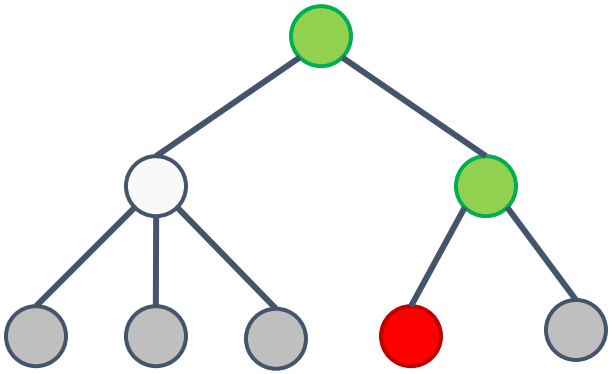


EG p
Es existiert mindestens ein Pfad, auf dem p global in allen zukünftigen Zuständen gilt.

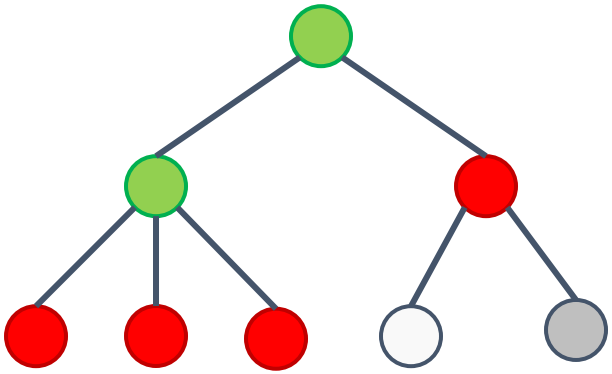


AG p
Auf allen Pfaden gilt p global in allen zukünftigen Zuständen.

CTL Operatoren 4/4



E [p U q]
Es existiert mindestens ein Pfad, auf dem p gilt bis irgendwann q gilt.

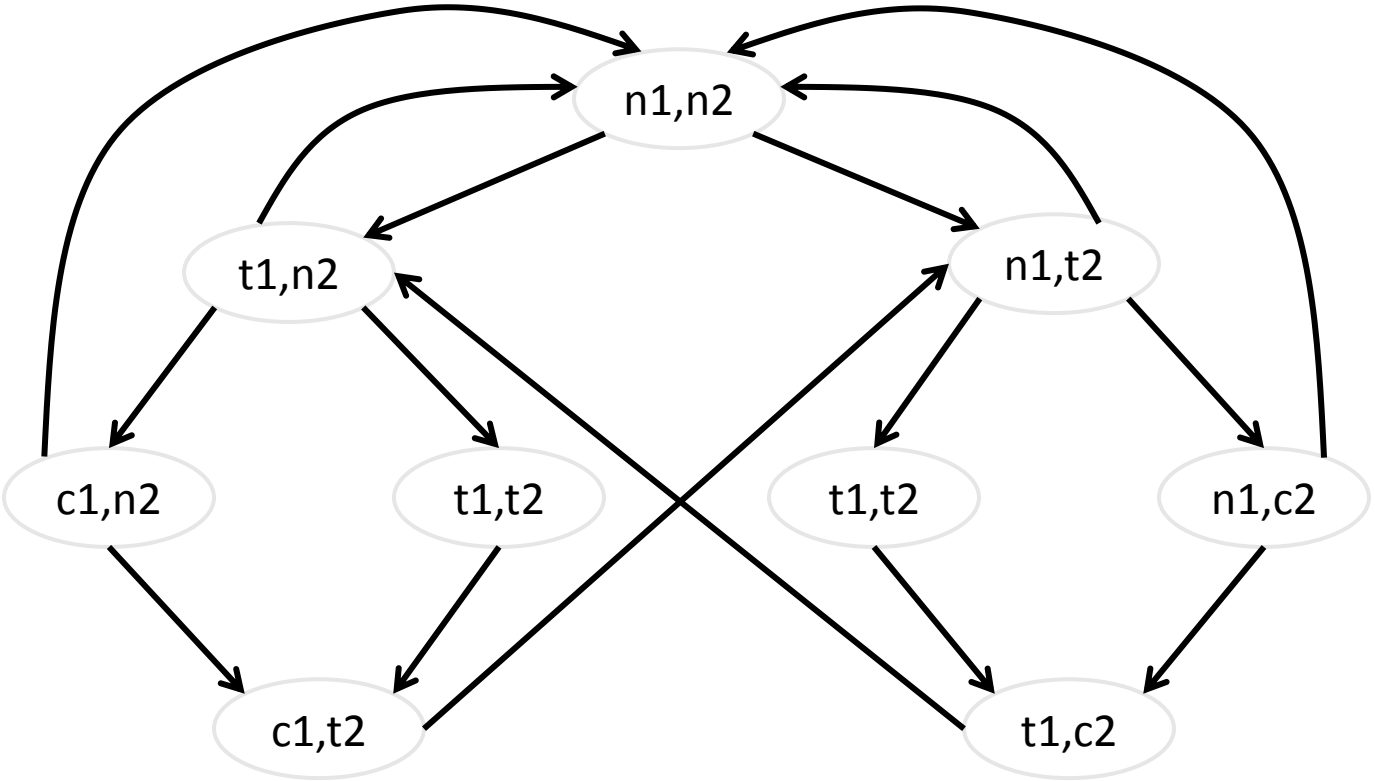


A [p U q]
Auf allen Pfaden gilt p bis irgendwann q gilt.

Model Checking: Beispiel

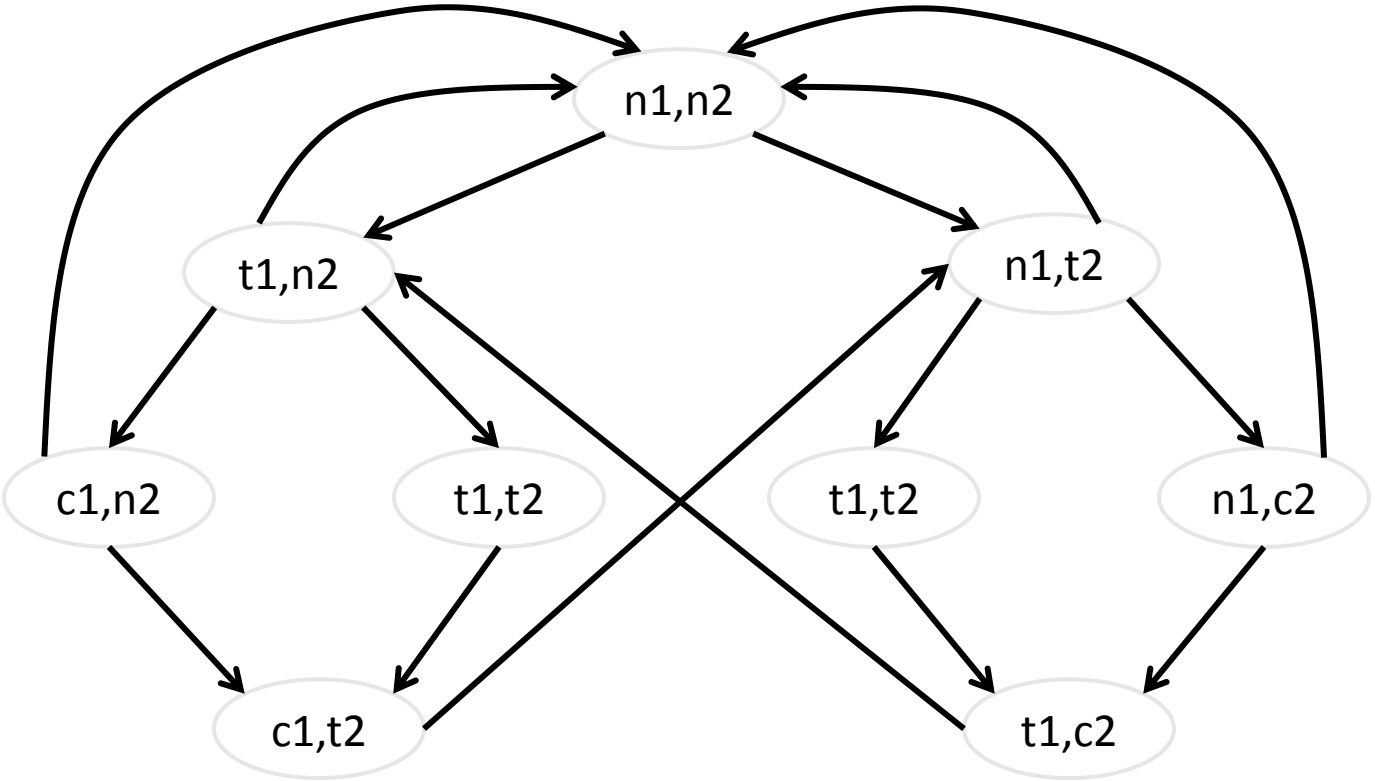
- Beispiel: Gegenseitiger Ausschluss
  - Zwei Prozesse, i = 1,2
  - Globaler Zustandsraum S1 … S9
  - Nebenläufige Semantik
  - Atomare Aussagen:
    - ni: Prozess i ist nicht im kritischen Bereich
    - ti: Prozess i versucht den kritischen Bereich zu betreten
    - ci: Process i befindet sich im kritischen Bereich

Model Checking: Beispiel



- Anforderung: Es werden niemals beide Prozesse im kritischen Bereich sein
- Formalisierung: AG $\neg$ (c1 $\wedge$ c2)

Model Checking: Beispiel



- Verifikationsidee: Betrachte jeden Zustand und prüfe, ob die Eigenschaft erfüllt ist.

Große Kripke Strukturen

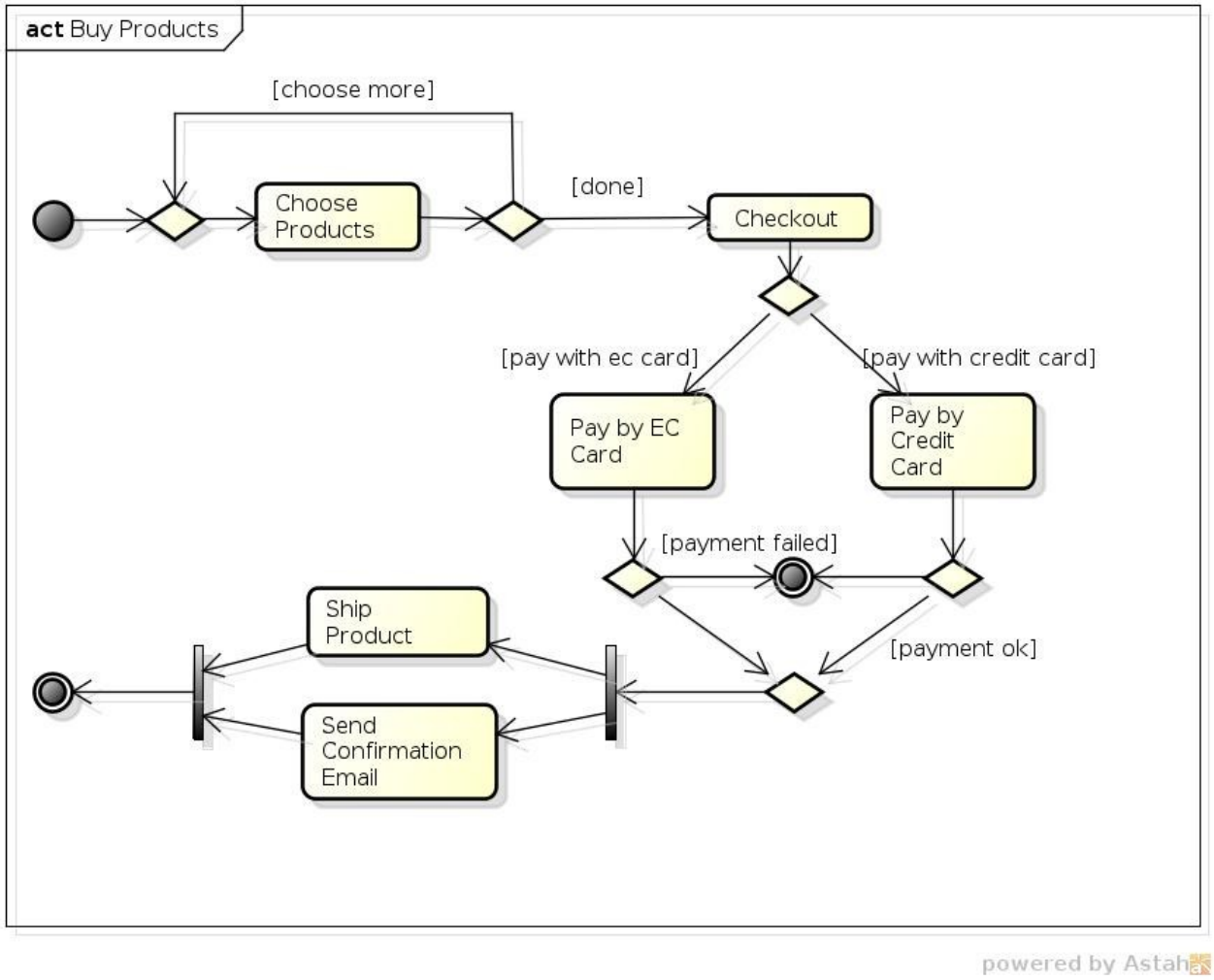
- Problem: Zustandsexplosion

- Typische Hardware Designs: 21000000 Zustände
- Kripke Struktur wächst üblicherweise exponentiell im Vergleich zum Programm
  - Nebenläufige Prozesse (kombinatorische Zustandsexplosion)
  - Anzahl der Variablen

Model Checking Zusammenfassung

- LTL zur Beschreibung von Eigenschaften auf einzelnen Pfaden
- CTL zur Beschreibung von Eigenschaften auf Berechnungsbäumen
- Model Checking ist das algorithmische Prüfen, ob ein Modell (endliche Kripke-Struktur) eine Eigenschaft erfüllt
- In jABC verwendet: CTL

Beispiel - Wiederholung



Beispiel - Repräsentation in jA

Und in der Schleife jeweils um eins erhöht.

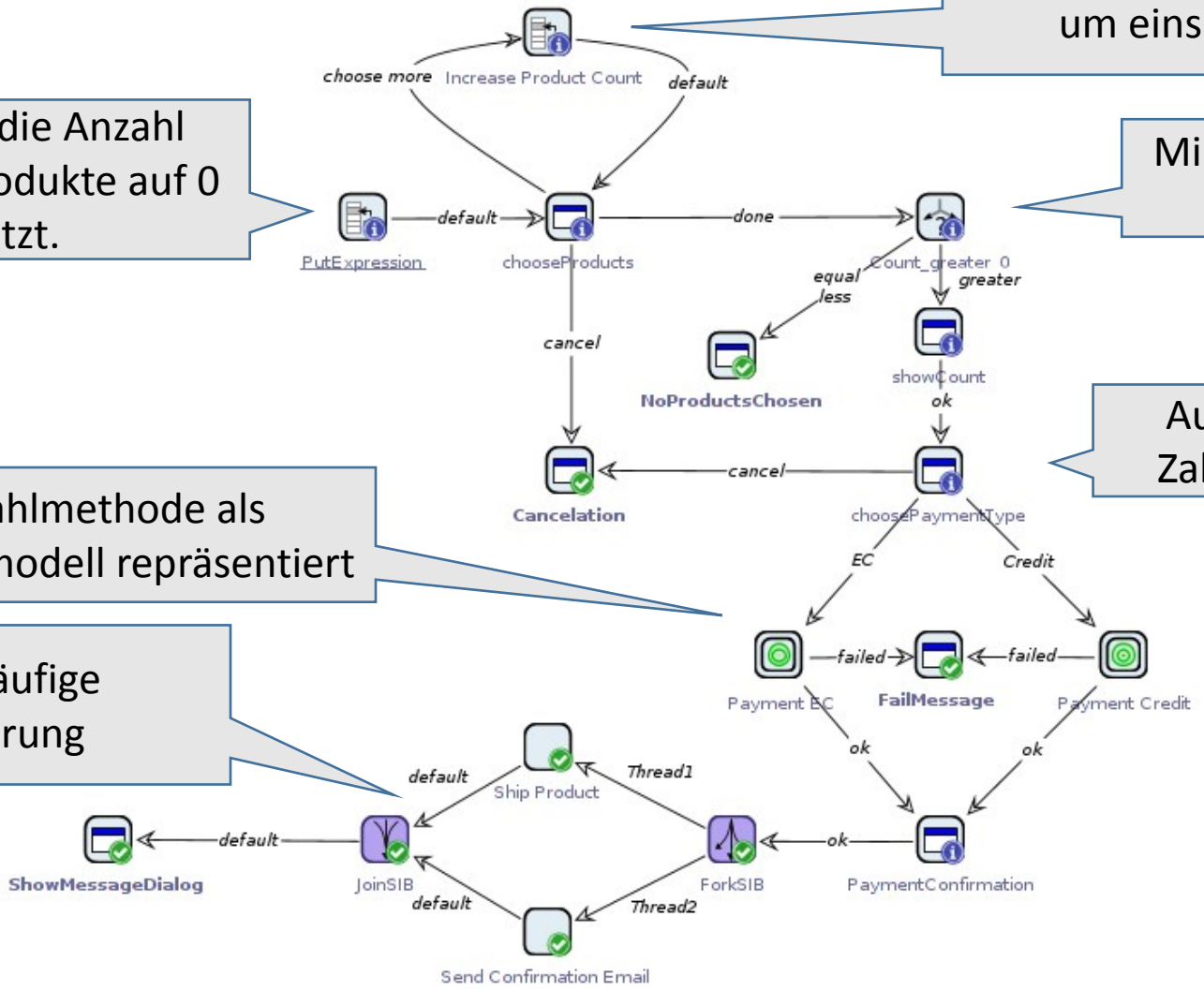
Initial wird die Anzahl gewählter Produkte auf 0 gesetzt.

Mindestens ein Produkt?

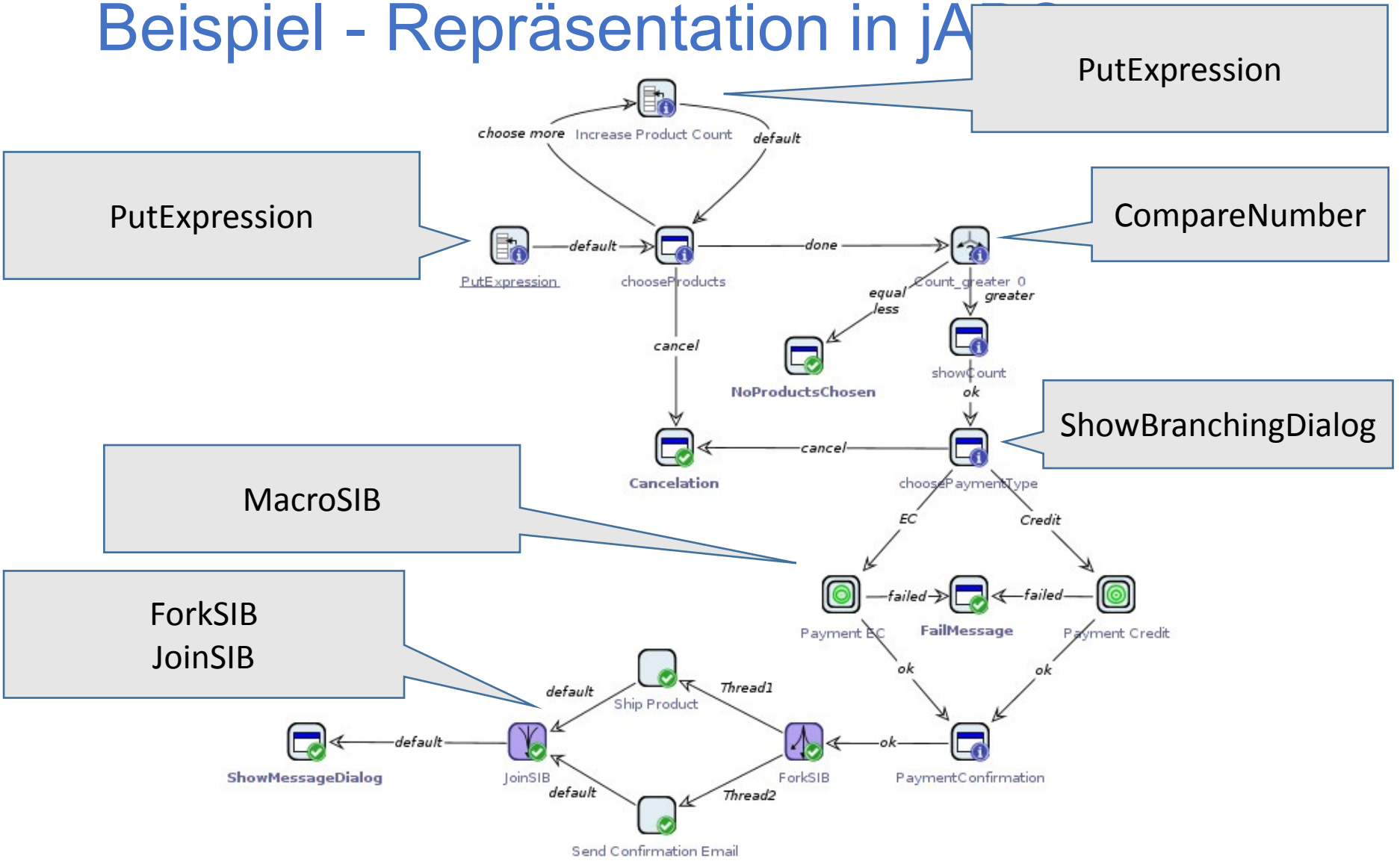
Auswahl der Zahlmethode

Zahlmethode als Untermodell repräsentiert

Nebenläufige Ausführung



Beispiel - Repräsentation in jA

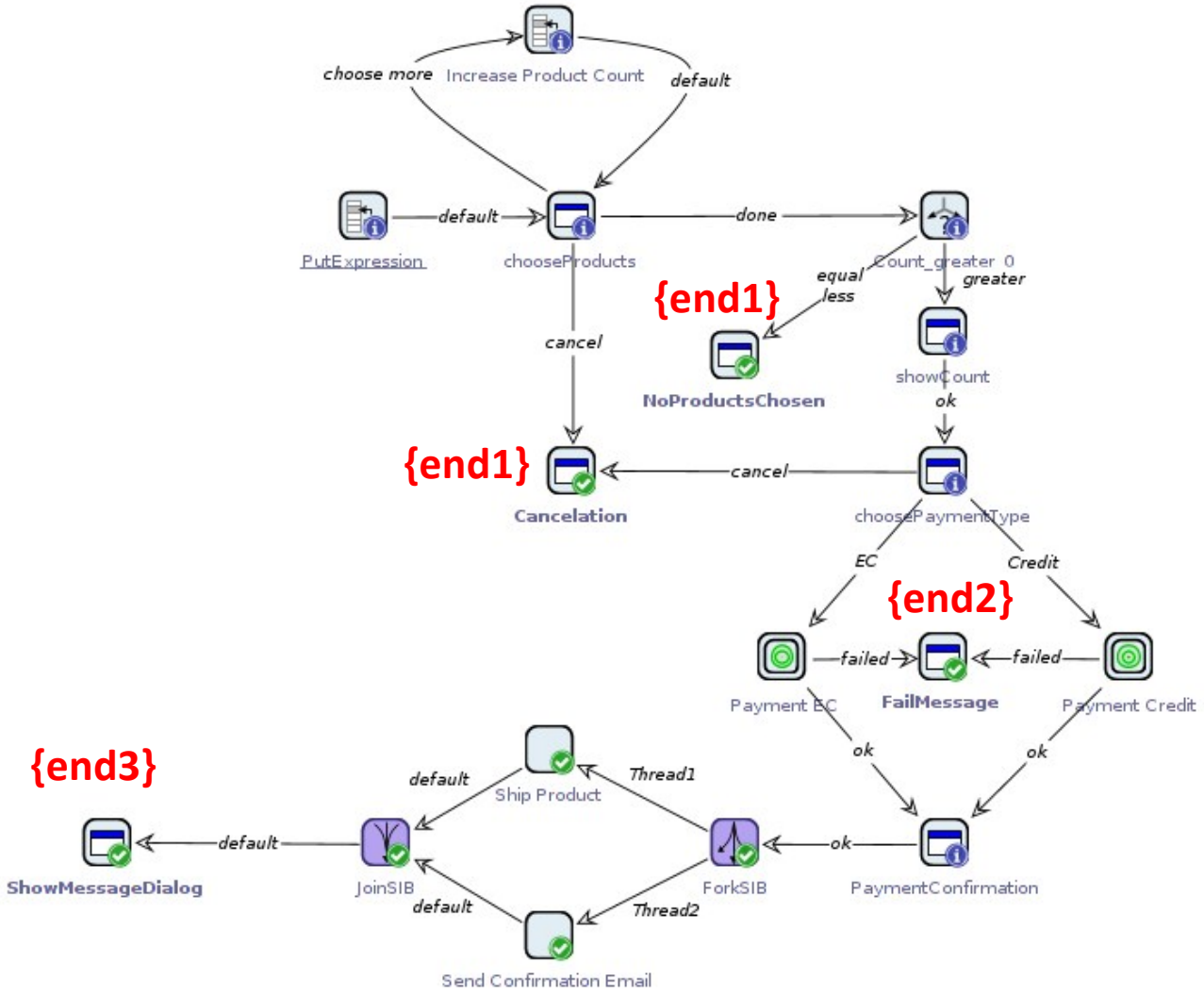


Beispiel – Untermodell Payment Credit

Hier durch eine nicht-deterministische Auswahl (chooseRandomBranch) repräsentiert – Könnte später verfeinert werden.

Die manuellen Branches „ok“ und „failed“ sind als Modellbranches festgelegt.

Beispiel – Deklaration von Präpositionen



Beispiel - Eigenschaft

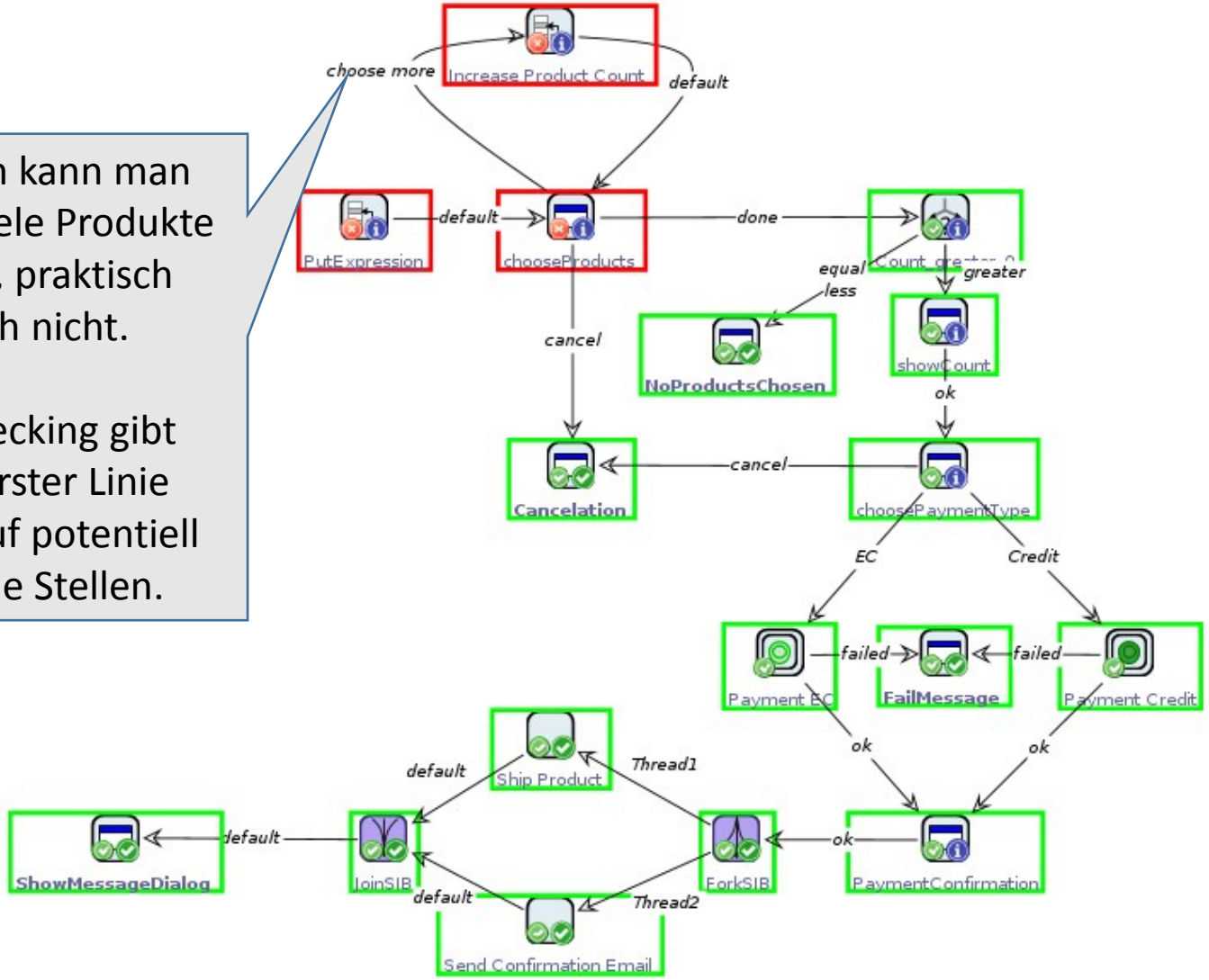
- Wir drücken aus, dass das Modell irgendwann einen terminalen Zustand erreichen wird
- **AF (end1 | end2 | end3)**

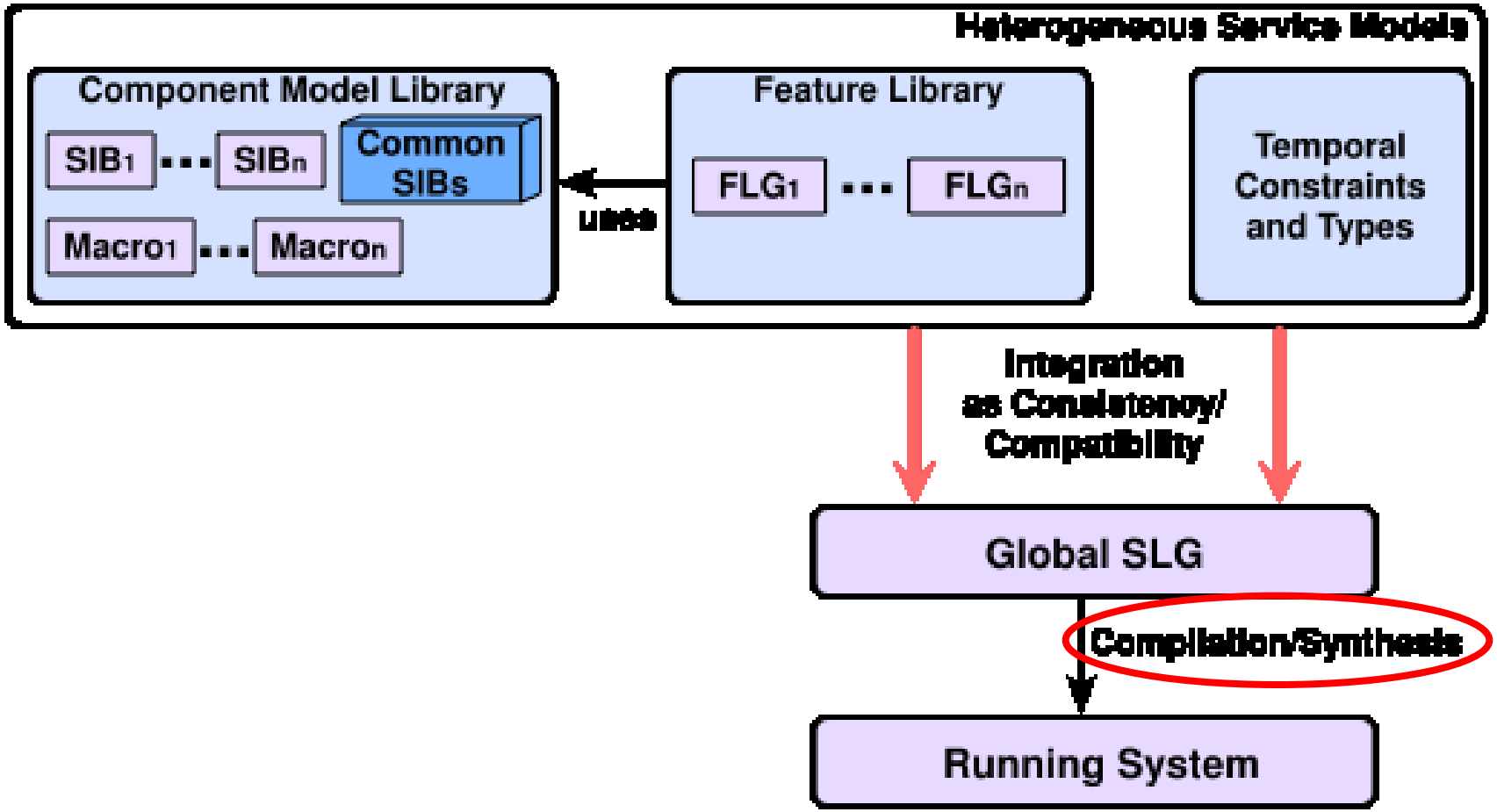
- In welchen Zuständen ist diese Eigenschaft erfüllt?

Beispiel – Model Checking

Theoretisch kann man unendlich viele Produkte einkaufen, praktisch natürlich nicht.

Model Checking gibt somit in erster Linie Hinweise auf potentiell gefährliche Stellen.




Heterogeneous Service Models
Component Model Library
SIB1
SIBn
Common SIBs
Macro1
Macron
uses
Feature Library
FLG1
FLGn
Temporal Constraints and Types
Integration as Consistency/ Compatibility
Global SLG
Compilation/Synthesis
Running System

Code-Generierung: Genesys

Yet another Plugin …

Referenzen

- [XMDD] Tiziana Margaria, Bernhard Steffen: Agile IT: Thinking in User-Centric Models. In: Proc. ISoLA 2008, CCIS N.17, Springer, pp 493–505
- [OTA] Tiziana Margaria, Bernhard Steffen: Business Process Modeling in the jABC: The One-Thing Approach. In: Handbook of research on business process modeling, IGI Global, 2008.
- [jABC] www.**jabc**.de