Modellierung von Echtzeitsystemen

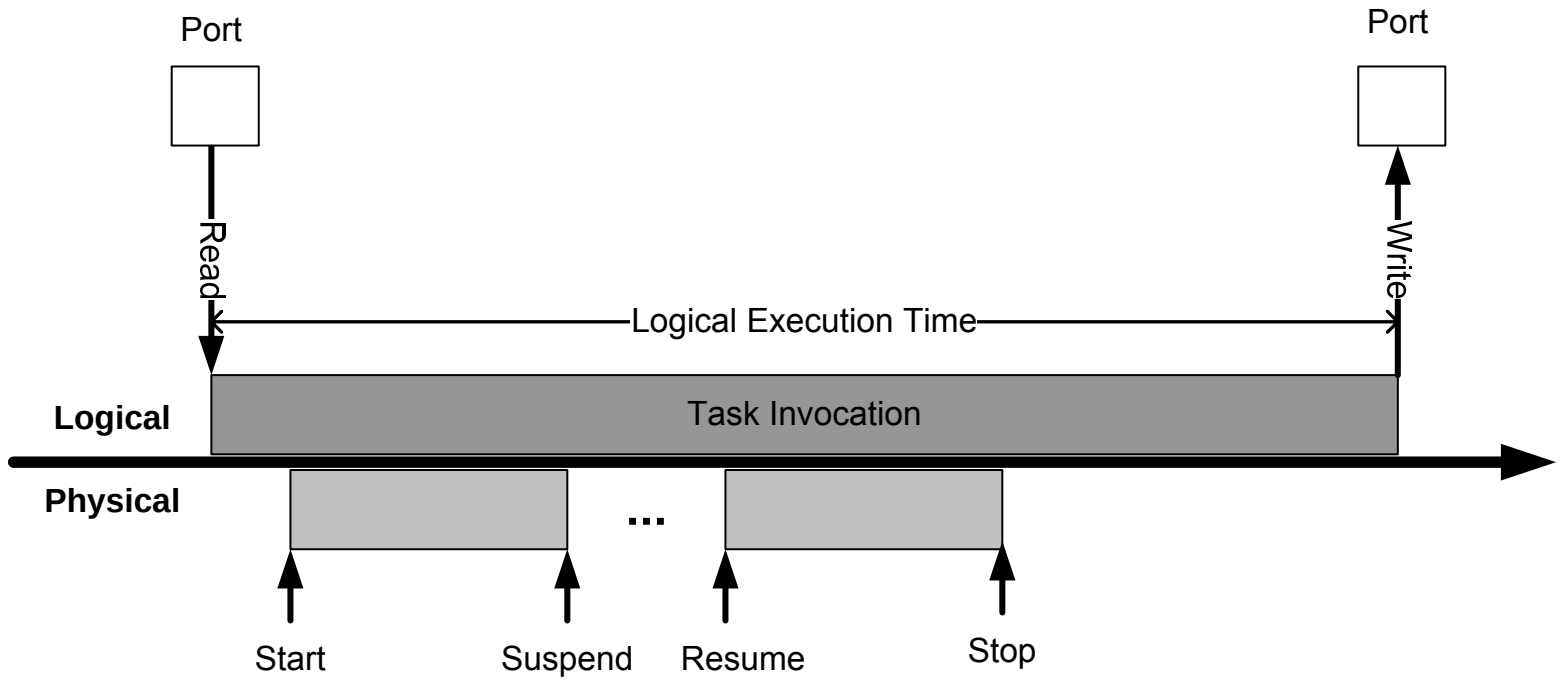
Zeitgesteuerte Systeme

Werkzeug: Giotto

Giotto: Hintergrund

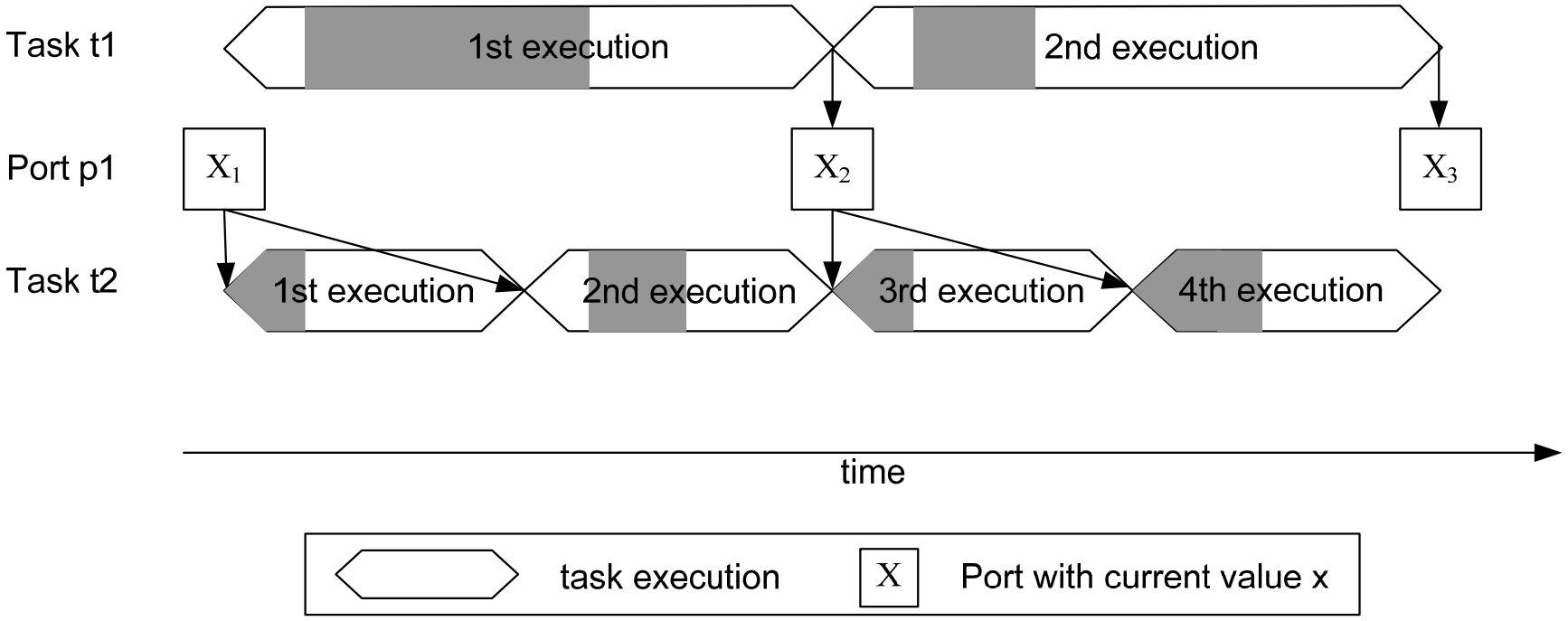
- Programmierumgebung für eingebettete Systeme (evtl. ausgeführt im verteilten System)
- Ziel:
  - strikte Trennung von plattformunabhängiger Funktionalität und plattformabhängigen Scheduling und Kommunikation
  - temporaler Determinismus
- Hauptkonzept: Logische Ausführungszeiten
- Aktoren:
  - Tasks
    - Programmblock aus sequentiellen Code
    - keine Synchronisationspunkte, blockende Operationen erlaubt
    - Schnittstellen: Ports
  - Drivers: realisieren die Kommunikation zwischen Ports
  - Flexibilität durch Modes/Guards
- Ausführung durch virtuelle Maschinen:
  - Embedded Machine: Reaktion der Tasks auf physikalische Ereignisse
  - Scheduling Machine: physikalisches Scheduling
- http://embedded.eecs.berkeley.edu/giotto/

Logische Ausführungszeit



Motivation siehe http://www.cs.uic.edu/~shatz/SEES/henzinger.slides.ppt

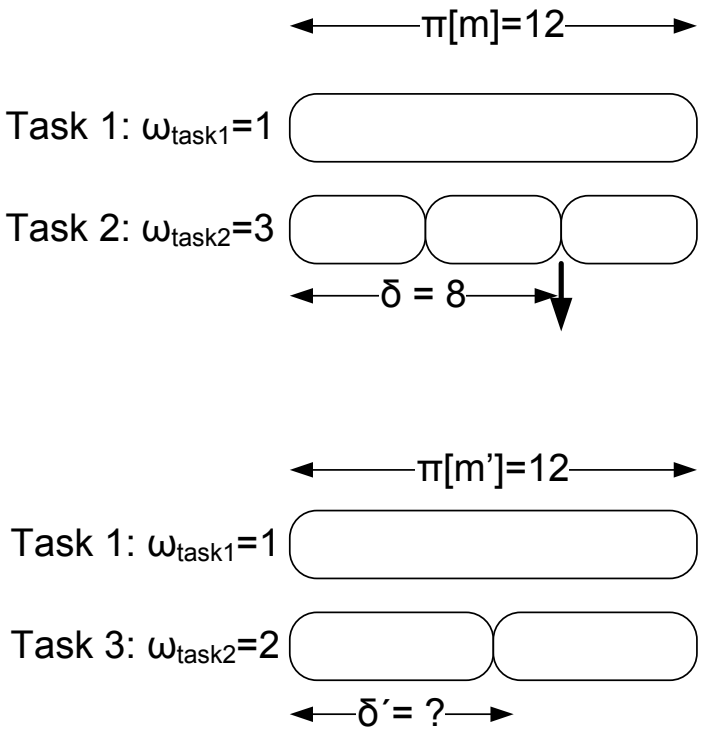
Kommunikation zwischen Tasks



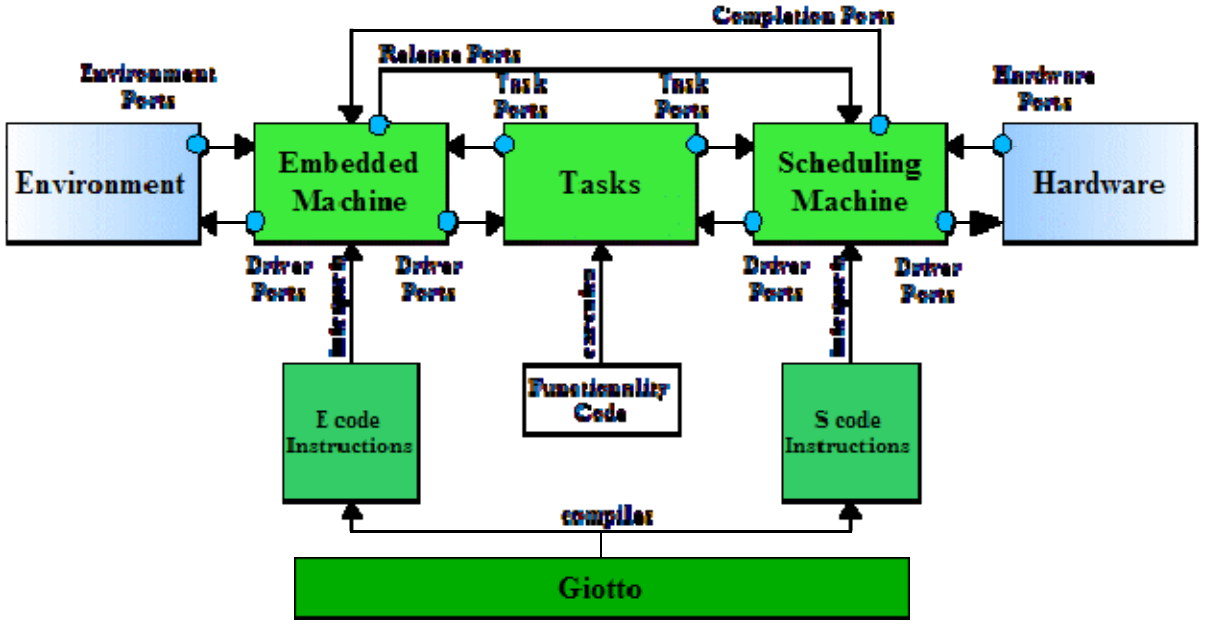
Modes / Guards

- Um die Ausführung flexibel zu gestalten, bietet Giotto Modes und Guards an
  - Guards: Boolsche Funktion, die über die Ausführung eines Tasks entscheidet (wird vor Start des Tasks aufgerufen)
  - Mode: Menge von Tasks und Drivers die zeitgleich ausgeführ werden, es kann immer nur ein Mode aktiv sein.
- Nicht-harmonischer Moduswechsel (Unterbrechung eines laufenden Modes):
  - Voraussetzung: $\pi[m]/\omega_{task} = \pi[m']/\omega'_{task}$
    m: Quellmodus, m': Zielmodus, $\pi[m]$: Modusdauer m, ω_{task}: Taskfrequenz $\Rightarrow$ Logische Ausführungszeit muss gleich sein
  - Wechselmechanismus:

    γ = LCM $\{\pi[m]/\omega_{task} | \{\omega_{task}, t,) \in$ Invokes[m],
    $\delta' = \pi[m']-(\varepsilon-\delta)$ mit $\varepsilon = n^* \gamma \geq \delta$
    LCM: least common multiple, δ: aktuelle Rundenzeit, δ': neue Rundenzeit in m', $\varepsilon - \delta$: Zeit bis zum nächsten gleichzeitigen Beendigungspunkt



Ausführungsumgebung



Zusammenfassung

- Das Konzept der logischen Ausführungszeiten erlaubt eine Abstrahierung von der physikalischen Ausführungszeit und somit die Trennung von plattformunabhängigem Verhalten (Funktionalität und zeitl. Verhalten) und plattformabhängiger Realisierung (Scheduling, Kommunikation)
- Die Ausführung erfolgt über zwei virtuelle Maschinen:
  - E-Machine: Interaktion mit der Umgebung (reaktiv)
  - S-Machine: Interaktion mit der ausführenden Plattform (proaktiv), Vorteil: Schedule kann vorab berechnet werden
- Weitere Literaturhinweise:
  - Henzinger et al.: Giotto: A time-triggered language für embedded programming, Proceedings of the IEEE, vol.91, no.1, pp. 84-99, Jan 2003
  - Henzinger et al.: Schedule-Carrying Code, Proceedings of the Third International Conference on Embedded Software (EMSOFT), 2003

Modellierung von Echtzeitsystemen

Verifikation von Echtzeitsystemen - Einsatz von Formalen Methoden

Problemstellung

„As soon as we started programming, we found to our surprise that it wasn‘t as easy to get programs right as we had thought.
Debugging had to be discovered.
I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.“

Maurice Wilkes.
(Turing Award 1967)

Verifikation & Validierung

- Verifikation: Um die Korrektheit von Programmen in Bezug auf die Spezifikation zu garantieren, wird eine formale Verifikation benutzt. Dazu werden mathematische Korrektheitsbeweise durchgeführt.
- Validierung: Durch eine Validierung kann überprüft werden, dass das System als Modell hinreichend genau nachgebildet wird. Techniken:
  - Inspektion
  - Plausibilitätsprüfung
  - Vergleich unabhängig entwickelter Modelle
  - Vergleichmessung an einem Referenzobjekt

Übersicht über formale Methoden

- Deduktive (SW-)Verifikation
  - Beweissysteme, Theorem Proving
- Model Checking
  - für Systeme mit endlichem Zustandsraum
  - Anforderungsspezifikation mit temporaler Logik
- Testen
  - spielt in der Praxis eine große Rolle
  - sollte systematisch erfolgen $\rightarrow$ ausgereifte Methodik
  - ... stets unvollständig

Verifikation in der Realität

- In der Industrie wird der Begriff Verifikation häufig im Zusammenhang mit nicht funktionalen Methoden verwendet:
  - Testen, Strategien:
    - 100% Befehlsabdeckung (Statement Coverage)
    - 100% Zweigüberdeckung (Branch Coverage)
    - 100% Pfadüberdeckung (Path Coverage)
    - Siehe auch http://www.software-kompetenz.de/?10764
  - Code reviews
  - Verfolgbarkeitsanalysen

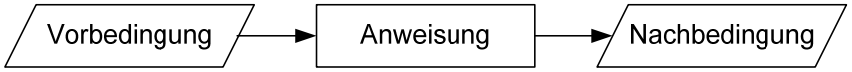
Testen

Mit Testen ist es möglich die Existenz von Fehlern nachzuweisen, nicht jedoch deren Abwesenheit.

- Testen ist von Natur aus unvollständig (non-exhaustive)
- Es werden nur ausgewählte Testfälle / Szenarien getestet, aber niemals alle möglichen.

Deduktive Methoden

- Nachweis der Korrektheit eines Programms durch math.-logisches Schließen
- Anfangsbelegung des Datenraums ⇒ Endbelegung
- Induktionsbeweise, Invarianten
  - klass. Bsp: Prädikatenkalkül von Floyd und Hoare, Betrachten von Einzelanweisungen eines Programms:



- Programmbeweise sind aufwändig, erfordern Experten
- i.A. nur kleine Programme verifizierbar
- Noch nicht vollautomatisch, aber es gibt schon leistungsfähige Werkzeuge

Temporale Logik

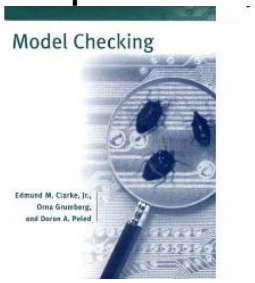
- Mittels Verifikation soll überprüft werden, dass:
  - Fehlerzustände nie erreicht werden
    - Der Aufzug soll nie mit offener Tür fahren.
  - ein System irgendwann einen bestimmten Zustand erreicht (und evtl. dort verbleibt)
    - Nach einer endlichen Initialisierungsphase, geht der Aufzug in den Betriebsmodus über.
  - Zustand x immer nach Eintreten des Zustandes y auftritt.
    - Nach Drücken des Tasters im Stockwerk wird der Aufzug in einem späteren Zustand auch dieses Stockwerk erreichen.
- Um solche Aussagen auch für Rechner lesbar auszudrücken, kann temporale Logik, z.B. in Form von LTL (linear time temporal logic), verwendet werden.
- In LTL wird Zustandsübergänge und damit auch die Zeit als diskrete Folge von Zuständen interpretiert.

Kripke-Struktur

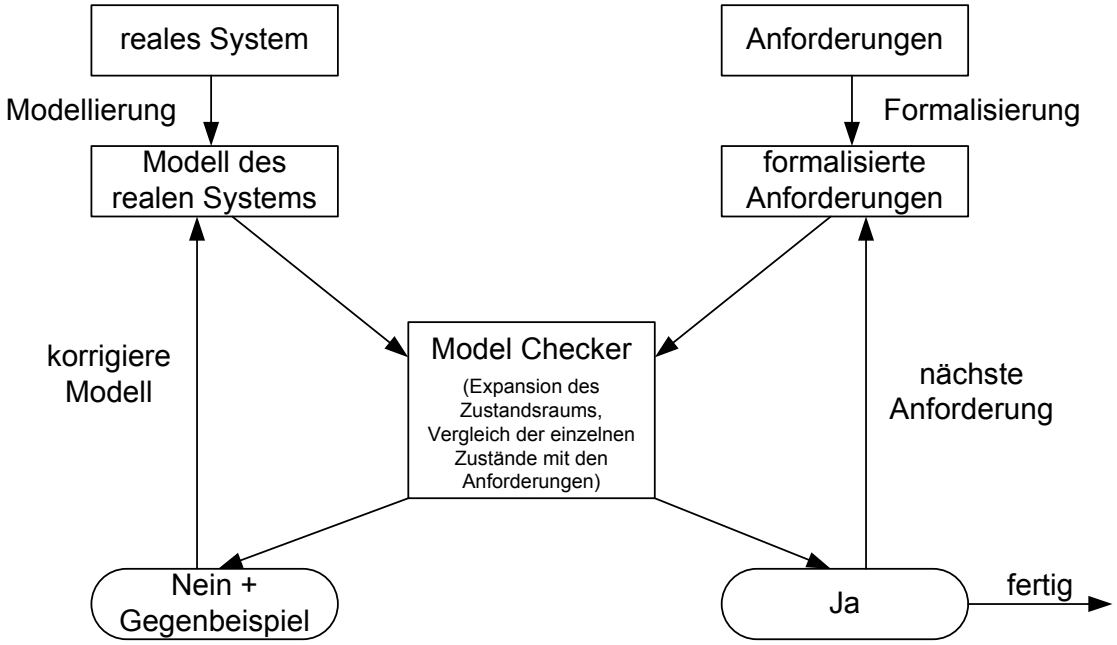
- Zur Darstellung eines Systems werden Kripke-Strukturen K = (V,I,R,B) und eine endliche Menge P von atomaren logischen Aussagen verwendet.
  - V: Menge binärer Variablen (z.B. Tür offen, Aufzug fährt)
  - Die Zustandsmenge S ergibt sich aus allen möglichen Kombinationen über V, somit gilt $S = B^V$
  - Menge der möglichen Anfangszustände $I \subseteq S$
  - R: Transitionsstruktur $R \subseteq S \times S$
  - B: Bewertungsfunktion $S \times P \rightarrow \{true,false\}$ zur Feststellung, ob ein Zustand eine Eigenschaft auf P erfüllt
- Mittels Model-Checking muss nun nachgewiesen werden, dass eine gewisse Eigenschaft P ausgehend von den Anfangszuständen
  - immer gilt
  - schließlich erfüllt wird
  - …

Explizites Model Checking: Verfahren

- Ausgehend von den Startzuständen exploriert der Model Checker mögliche Nachbarzustände:
  - Auswahl eines noch nicht evaluierten Zustandes
  - Prüfung aller möglichen Zustandsübergange:
    - bereits bekannter Zustand: verwerfen
    - unbekannter Zustand, Eigenschaft prüfen
      - falls Eigenschaft nicht erfüllt, Abbruch und Präsentation eines Gegenbeispiels
      - falls erfüllt, zur Menge der nicht evaluierten Zustände hinzufügen
  - Abbruchbedingung: alle erreichbaren Zustände wurden überprüft
- Problem: Zustandsexplosion
- Literaturhinweis: Edmund M. Clarke, Orna Grumberg, Doron A. Peled, *Model Checking*, 1999, MIT Press



Umgang mit Model Checking



Weitere Strategien

- Symbolische Model Checker:
  - Grundidee: durch eine einfache Formel können viele Zustände zu einem Zustand gekapselt werden
  - Verwendung von binären Entscheidungsdiagrammen (binary decision diagrams – BDD)
- Bounded Model Checker:
  - Grundidee: durch Abstraktion können viele Zustände zusammengefasst werden (z.B. Aufteilung der ganzen Zahlen in positive, negative Zahlen und 0)
  - Häufig sind diese Model Checker pessimistisch (Präsentation von Gegenbeispielen, die keine sind)

Probleme mit formalen Methoden

- Entwickler empfinden formale Methoden häufig als zu kryptisch
- Beispiel TLA:

$$
\begin{aligned}
HCini &\triangleq \quad \wedge \quad hr \in \{0, \ldots, 23\} \\
HCnxt &\triangleq \quad \wedge \quad hr' = IF\ hr \neq 23\, THEN\ hr + 1\ ELSE\ 0 \\
HC &\triangleq \quad \wedge \quad HCini \\
&\qquad\quad \wedge \quad \Box HCnxt
\end{aligned}
$$

- Neue Ansätze: Erweiterung der Programmier / Modellierungssprachen, automatische Übersetzung

1. Beispiel: Verifikation in Esterel Studio

- Esterel Studio bietet eine eingebaute Verifikationsfunktionalität zur einfachen Verifikation von Programmen
- Zur Modellierung der verschiedenen Eigenschaften kann das Schlüsselwort `assert` verwendet werden.
- Im Verifikationsmodus können die Eigenschaften dann getestet werden, dabei stehen Methoden zum unbegrenzten / in der Testtiefe begrenzten Modell Checking, sowie zum symbolischen Model Checking zur Verfügung.
- Grundsätzliche Vorgehensweise:
  - Finden von Fehlern in den Annahmen / Modellen mit begrenztem Model Checker
  - Nachweis der Korrektheit des verbesserten Modells in Bezug auf die korrigierten Eigenschaften mit unbegrenztem Model Checking / symbolischen Model Checking
- Details siehe Demonstration

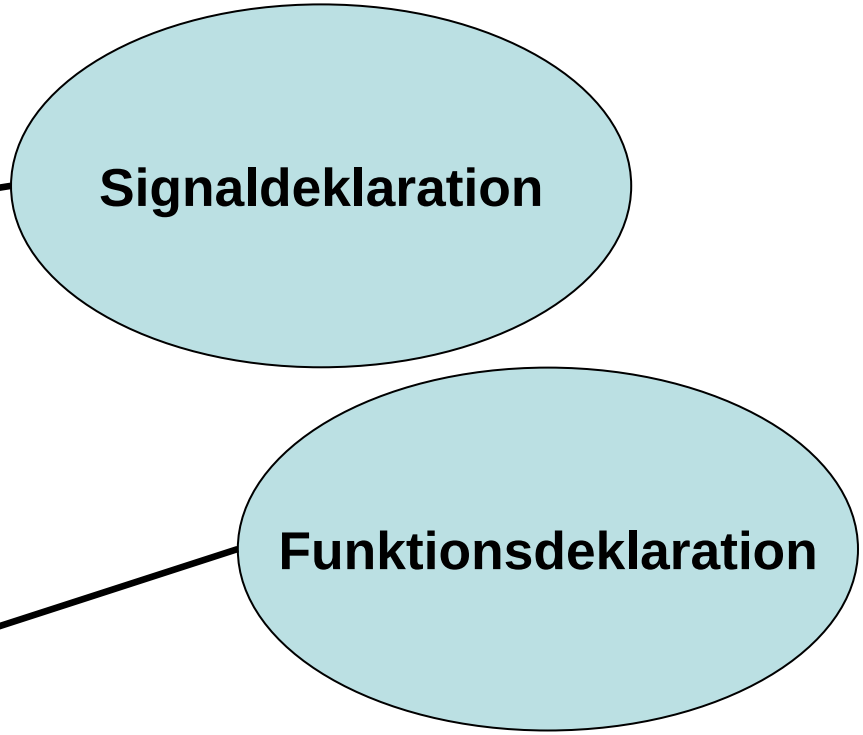
2. Beispiel: BoogiePL in Kombination mit Z3

- Grundidee: Verifikation von C# Programmen durch Erweiterung Spec# von Microsoft
- Das Spec#-Programm wird in Zwischensprache BoogiePL übersetzt. Die geforderten Eigenschaften werden dann mit Hilfe des SMT-Solvers Z3 nachgewiesen.
- Grundkonstrukte (Ausschnitt):
  - assert: Annahmen die durch den Beweiser verifiziert werden müssen
  - assume: Annahme durch den Benutzer, Zuständsübergänge, die der Annahme widersprechen werden vom Beweiser ignoriert
  - havoc: Zuweisung eines beliebigen Wertes an eine Variable (z.B. zur Simulation der Umgebung)

Boogie-Programm: Türbeispiel vereinfacht

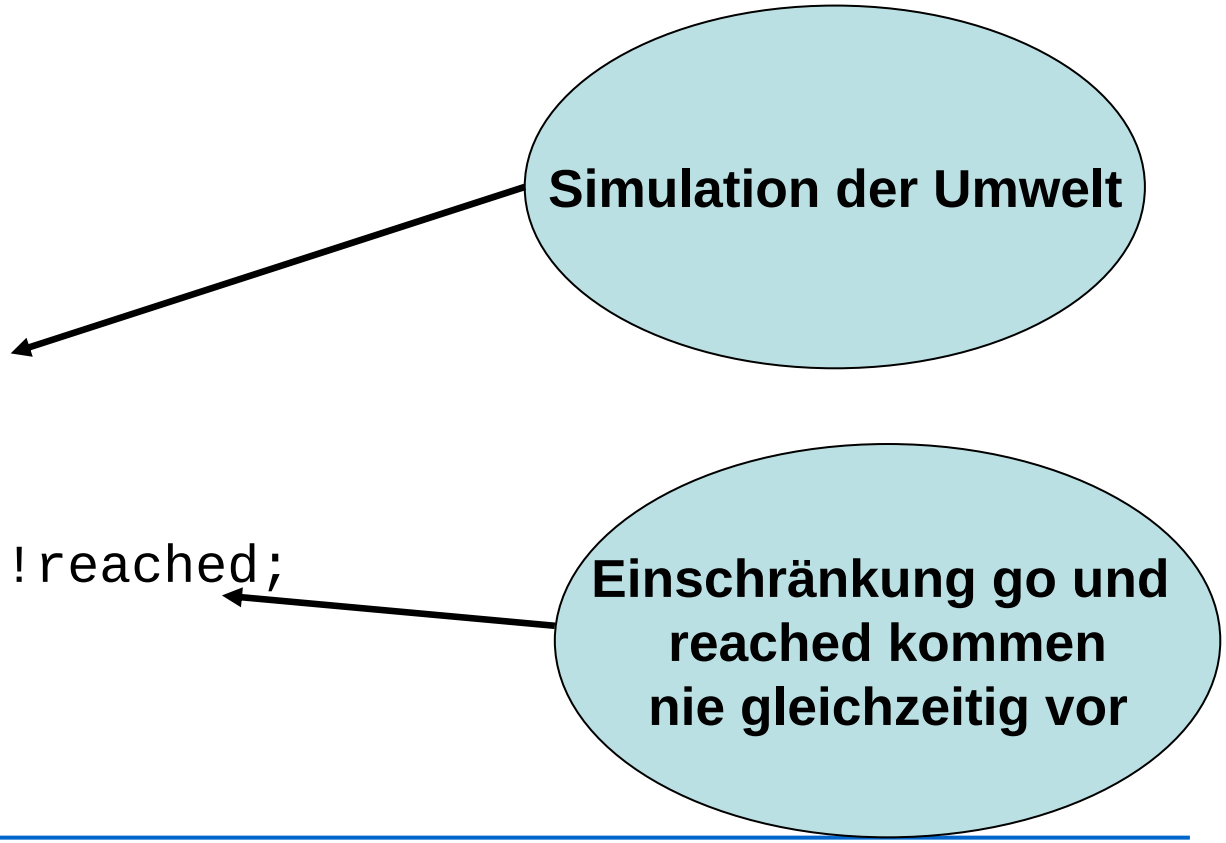
```
var open: bool;
var close: bool;
var go: bool;
var reached: bool;
var sig_open: bool;
var sig_close: bool;

procedure Door();
  modifies open,close,go,reached,sig_open,sig_close;
```



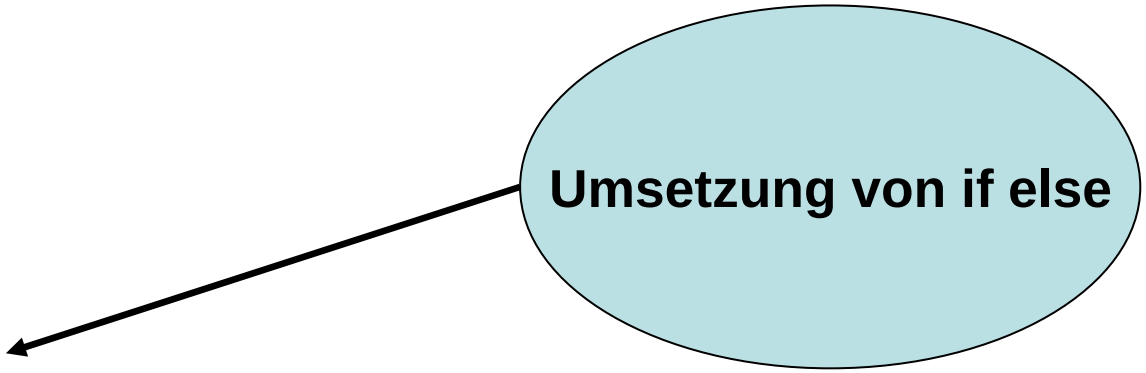
Boogie-Programm: Türbeispiel vereinfacht

```
implementation Door()
{
Begin:
   havoc open;
   havoc close;
   havoc go;
   havoc reached;
   assume !go && !reached;
```



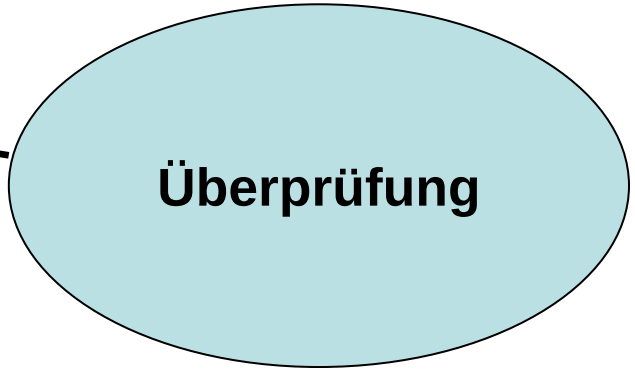
Boogie-Programm: Türbeispiel vereinfacht

```
  goto Open,Close;
Open:
  assume open;
  sig_open:=true;
  goto End;
Close:
  assume !open && close;
  sig_close:=true;
  goto End;
```



Boogie-Programm: Türbeispiel vereinfacht

```
End:
   assert open ==> sig_open;
   assert close ==> sig_close;
   goto Begin;
}
```



Boogie-Ergebnis:

```
D:\boogie>boogie door.bpl -enhancedErrorMessages:1
Spec# Program Verifier Version 0.87, Copyright (c) 2003-2007, Microsoft.
Information extracted from prover model:
close == True
sig_close == False
Failing assertion: close ==> sig_close
door.bpl(35,2): Error BP5001: This assertion might not hold.
Execution trace:
    door.bpl(13,1): Begin
    door.bpl(23,1): Open
    door.bpl(33,1): End

Spec# Program Verifier finished with 0 verified, 1 error
```