Jan Schulz-Nigmann
14.09.2022

FMSE Zusammenfassung SoSe 2022

Inhalt

Jan Schulz-Nigmann
14.09.2022

Modellierung reaktiver Systeme

Endliche Automaten

Endliche Automaten bestehen aus endlich vielen Zuständen, Zustandsübergängen, einem Startzustand und beliebig vielen Endzuständen.

Transitionssysteme

Transitionssysteme sind angelehnt an endliche Automaten aber es können reaktive Systeme ohne „Endzustand“ modelliert werden. Zustandsübergänge werden mit Aktionen versehen und Zustände mit „atomaren“ Eigenschaften.

Definition: Transitionssysteme

Ein **Transitionssystem** TS ist ein Tupel $(S, Act, \rightarrow, I, AP, L)$, mit

- S einer Menge an Zuständen
- Act einer Menge an Aktionen
- $\rightarrow \subseteq S \times Act \times S$ einer Transitionsrelation
- I einer Menge an Anfangszuständen
- AP einer Menge an atomaren Propositionen und
- $L: S \rightarrow 2^{AP}$ einer Beschriftungsfunktion.

TS ist endlich, falls S, Act und AP endlich sind.

Mehrere Transitionen von einem Zustand können mit der gleichen Aktion nicht-deterministisch in unterschiedliche Zustände gehen. Nichtdeterminismus bedeutet, dass von einem Zustand mit einer Aktion in mehrere Folgezustände gewechselt werden kann. Dies dient zur Abstraktion und Modellierung von paralleler Ausführung.

Wir schreiben $s \xrightarrow{\alpha} s'$ für $(s, \alpha, s') \in \rightarrow$ und $L(S)$ die Menge der atomaren Propositionen, die in s erfüllt sind.

Definition: Ausführung

Eine **Ausführung** ρ eines TS kann zwei Arten haben:

1. Endliche Zustandsfolge $\rho = s_0 s_1 s_2 \dots s_n$

   mit $s_0 \in I$, für alle $i = 1, \dots n$ gibt es ein $\alpha_i \in$ Act mit $s_{i-1} \xrightarrow{\alpha_i} s_i$ und $(s_n, \alpha, s) \notin \rightarrow$ für alle $s \in S$ und $\alpha \in Act$ (d.h. s_n ist ein Terminalzustand)

1. Unendliche Zustandsfolge $\rho = s_0 s_1 s_2 \dots$

   mit $s_0 \in I$ und für alle $i \geq 1$ gibt es ein $\alpha_i \in Act$ mit $s_{i-1} \xrightarrow{\alpha_i} s_i$.

Datenabhängige Systeme

Datenabhängige Systeme sind eine Erweiterung der Transitionssysteme zu Programmgraphen durch hinzufügen von typisierten Variablen, Variablenbelegungen, Effekte von Aktionen auf Variablenbelegungen und bedingte Transitionen.

Jan Schulz-Nigmann
14.09.2022

1. Menge typisierten Variablen Var: Jeder Variable $x \in Var$ ist ein Wertebereich $dom(x)$ zugeordnet.
2. Menge an Variablenbelegung $Eval(Var)$ über Var: Eine Variablenbelegung weist einer Variable einen konkreten Wert zu.
3. Aktionen können Auswirkungen auf Variablen haben. Formal definieren wir eine Effekt-Funktion: $Effect: Act \times Eval(Var) \rightarrow Eval(Var)$
4. Wir verwenden Transitionen von Zustand s in s' falls eine boolesche Bedingung erfüllt ist. $s \xrightarrow{g:\alpha} s'$ Nur wenn g = guard erfüllt ist gelangen wir mit α= Aktion von Zustand s in Zustand s'. Die Menge aller booleschen Bedingungen über Var wir mit $Cond(Var)$.

Definition: Programmgraph

Ein **Programmgraph** PG über einer Menge typisierter Variablen $\boldsymbol{Var}$ ist ein Tupel $(\boldsymbol{Loc}, \boldsymbol{Act}, \mathbf{Effect}, \hookrightarrow, \boldsymbol{Loc_0}, \boldsymbol{g_0})$ mit

- $\boldsymbol{Loc}$ einer Menge an Positionen und $\boldsymbol{Act}$ einer Menge an Aktionen,
- $\mathbf{Effect}: \boldsymbol{Act} \times \mathrm{Eval}(\boldsymbol{Var}) \rightarrow \mathrm{Eval}(\boldsymbol{Var})$ einer Effekt-Funktion
- $\hookrightarrow \subseteq \boldsymbol{Loc} \times \mathrm{Cond}(\boldsymbol{Var}) \times \boldsymbol{Act} \times \boldsymbol{Loc}$ einer bedingten Transitionsrelation
- $\boldsymbol{Loc_0} \subseteq \boldsymbol{Loc}$ einer Menge an Initialpositionen
- $\boldsymbol{g_0} \in \mathrm{Cond}(\boldsymbol{Var})$ der Initialbedingung

Jeder Programmgraph PG kann als Transitionssystem TS(PG) interpretiert werden.

Definition: Transitionssystem über Programmgraph

Ein **Transitionssystem** TS(PG) über einem Programmgraph

$$\mathrm{PG} = (\boldsymbol{Loc}, \boldsymbol{Act}, \mathbf{Effect}, \hookrightarrow, \boldsymbol{Loc_0}, \boldsymbol{g_0})$$

über einer Menge typisierter Variablen $\boldsymbol{Var}$ ist ein Tupel $(\boldsymbol{S}, \boldsymbol{Act}, \rightarrow, \boldsymbol{I}, \boldsymbol{AP}, \boldsymbol{L})$ mit

- $\boldsymbol{S} = \boldsymbol{Loc} \times \mathrm{Eval}(\boldsymbol{Var})$
- $\rightarrow \subseteq \boldsymbol{S} \times \boldsymbol{Act} \times \boldsymbol{S}$ mit $(l, \eta) \xrightarrow{\alpha} (l', \mathbf{Effect}(\alpha, \eta))$, falls $l \xrightarrow{g:\alpha} l'$ und $\eta \vDash g$
- $\boldsymbol{I} = \{(l, \eta) | l \in \boldsymbol{Loc_0}, \eta \vDash \boldsymbol{g_0}\}$
- $\boldsymbol{AP} = \boldsymbol{Loc} \cup \mathrm{Cond}(\boldsymbol{Var})$
- $\boldsymbol{L}\big((l, \eta)\big) = \{l\} \cup \{g \in \mathrm{Cond}(\boldsymbol{Var}) | \eta \vDash g\}$

Nebenläufigkeit und Kommunikation

Nun sollen TS1,…,TSn parallel laufen. Wir wählen den Operator ||, so dass das Transitionssystem $TS = TS_1 || TS_2 || \ldots || TS_n$ das Verhalten der parallelen Komposition der n Systeme beschreibt. Wir betrachten verschiedene Kommunikationsarten:

1. Verschränkung paralleler Systeme: Aktionen unabhängiger Komponenten werden ineinander verschränkt (interleaved). Nebenläufigkeit wird durch eine nichtdeterministische Wahl der simultan agierenden Komponenten modelliert. Hierbei kommunizieren die Systeme nicht und konkurrieren nicht um die geteilten Variablen.

Jan Schulz-Nigmann
14.09.2022

Definition: Verschränkung von Transitionssystemen

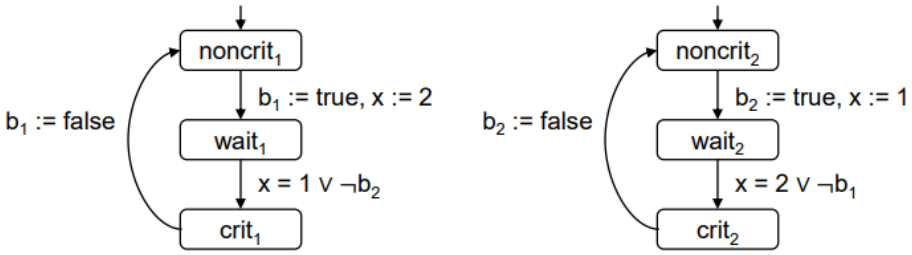
Seinen $\mathrm{TS}_i = (S_i, Act_i, \rightarrow_i, I_i, AP_i, L_i), i = 1{,}2$ zwei Transistionssysteme.
Dann ist $\mathrm{TS}_1 \,|||\, \mathrm{TS}_2$ definiert durch

$$TS_1 \,|||\, TS_2 = (S_1 \times S_2, Act_1 \cup Act_2, \rightarrow, I_1 \times I_2, AP_1 \cup AP_2, L)$$

mit

- $(s_1, s_2) \xrightarrow{\alpha} (s_1', s_2)$ falls $s_1 \xrightarrow{\alpha}_1 s_1'$ und
- $(s_1, s_2) \xrightarrow{\alpha} (s_1, s_2')$ falls $s_2 \xrightarrow{\alpha}_2 s_2'$, sowie
- $L\big((s_1, s_2)\big) = L_1(s_1) \cup L_2(s_2)$

2. Der Operator ||| zur Verschränkung von Transitionssystemen berücksichtigt auf der Ebene der TS nicht den Zugriff auf geteilte Variablen in einem Programmgraphen. Hierzu verschränkt man zunächst die Programmgraphen und konstruiert danach das Transitionssystem. Hierbei dürfen die parallelen Prozesse nicht gleichzeitig in einem kritischen Bereich sein. Hierzu nutzen wir Pettersons Algorithmus:



1. Setze b_i := true, x := 1 oder 2: Beide Prozesse wollen beide in den kritischen Zustand, geben aber jeweils anderen Prozess den Vortritt
2. Falls $x = 1 \vee \neg b_2$ oder $x = 2 \vee \neg b_1$: Wenn ich Priorität habe oder der andere Prozess nicht in den kritischen Bereich möchte, dann gehe in kritischen Bereich

Definition: Verschränkung von Programmgraphen

Seien $\mathrm{PG_i} = (\boldsymbol{Loc_i, Act_i, \mathrm{Effect}_i, \hookrightarrow_i, Loc_{0,i}, g_{0,i}})$, $i = 1{,}2$, zwei Programmgraphen über Var_i. Der Programmgraph $\mathrm{PG}_1 \,|||\, \mathrm{PG}_2$ über $Var_1 \cup Var_2$ ist definiert durch

$$(Loc_1 \times Loc_2, Act_1 \uplus Act_2, \mathrm{Effect}, \hookrightarrow, Loc_{0,1} \times Loc_{0,2}, g_{0,1} \wedge g_{0,2})$$

mit

- $(l_1, l_2) \xrightarrow{g:\alpha} (l_1', l_2)$ falls $l_1 \xrightarrow{g:\alpha}_1 l_1'$ in PG_1 und
  $(l_1, l_2) \xrightarrow{g:\alpha} (l_1, l_2')$ falls $l_2 \xrightarrow{g:\alpha}_2 l_2'$ in PG_2, sowie
- $\mathrm{Effect}(\alpha, \eta) = \mathrm{Effect}_i(\alpha, \eta)$ falls $\alpha \in Act_i$.

3. Handshake, synchroner Austausch von Nachrichten: Nebenläufige Prozesse können nur miteinander kommunizieren, wenn sie gleichzeitig eine Handschlagaktion aus der selben Handschlagmenge ausführen, dabei müssen die Aktionen der Handschlagmenge unabhängig

sein und autonom ausgeführt werden können. Mittels eines Schiedsrichtersystems kann verhindert werden, dass die TS gleichzeitig in einem kritischen Zustand landen.

Definition: Transitionssysteme mit Handschlagaktionen

Seien $\mathrm{TS}_i = (S_i, Act_i, \rightarrow_i, I_i, AP_i, L_i)$, $i = 1,2$, zwei Transistionssysteme und $H \subseteq Act_1 \cap Act_2$. Dann ist das Transistionssystem $\mathrm{TS}_1 ||_H \mathrm{TS}_2$ definiert durch $\mathrm{TS}_1 ||_H \mathrm{TS}_2 = (S_1 \times S_2, Act_1 \cup Act_2, \rightarrow, I_1 \times I_2, AP_1 \cup AP_2, L)$

mit $L((s_1, s_2)) = L_1(s_1) \cup L_2(s_2)$ und $\rightarrow$ definiert als

- **Verschränkung** für $\alpha \notin H$ gilt: $(s_1, s_2) \xrightarrow{\alpha} (s_1', s_2)$ falls $s_1 \xrightarrow{\alpha}_1 s_1'$ und $(s_1, s_2) \xrightarrow{\alpha} (s_1, s_2')$ falls $s_2 \xrightarrow{\alpha}_2 s_2'$,
- **Handschlag** für $\alpha \in H$ gilt: $(s_1, s_2) \xrightarrow{\alpha} (s_1', s_2')$ falls $s_1 \xrightarrow{\alpha}_1 s_1'$ und $s_2 \xrightarrow{\alpha}_2 s_2'$.

4. Kanalsysteme: Ein Kanalsystem hat einen Kanal c mit einem Kanaltypen $dom(c)$, welcher die Art der übermittelten Nachrichten angibt und einer Kapazität $cap(c)$, die das Speichervermögen des Kanals angibt. Das Kanalsystem besteht aus mehreren Prozessen, die jeweils durch einen Programmgraph beschrieben sind. Jeder Programmgraph besitzt zwei Typen von Transitionen, bedingte Transitionen und Kommunikationsaktionen. Es gibt zwei Kommunikationsaktionen: c! zum Schreiben ans Ende des Puffers des Kanals und c? zum Lesen einer Nachricht vom Anfang des Puffers des Kanals.

Definition: Kanalsystem

Ein Programmgraph über $(Var, Chan)$ ist ein Tupel

$$(Loc, Act, \mathbf{Effect}, \hookrightarrow, Loc_0, g_0)$$

gemäß der Definition Programmgraph, wobei im Unterschied zu dieser die Transitionen definiert sind wie folgt:

$$\hookrightarrow \subseteq Loc \times \mathrm{Cond}(Var) \times (Act \cup Comm) \times Loc.$$

Ein **Kanalsystem** $\mathrm{CS} = [\mathrm{PG}_1 | \ldots | \mathrm{PG}_n]$ über $(Var, Chan)$ besteht aus Programmgraphen PG_i über $(Var_i, Chan)$ für $i = 1, \ldots, n$ mit $Var = \bigcup_{i=1,\ldots,n} Var_i$.

Hierbei sei $Chan$ eine endliche Menge von Kanälen und einer Menge an Kommunikationsaktionen $Comm = \{c!\, v, c?\, x | c \in Chan, v \in dom(c), x \in Var, dom(x) \supseteq dom(c)\}$.

Explosion des Zustandsraums

- Zustandsraum von Programmgraphen: TS, die durch Auffalten von PG entstehen, bestehen aus Zuständen (Position, Variablenbelegung). Angenommen Wertebereich der Variablen ist endlich, dann besitzt Zustandsraum die Größe $|Loc| \cdot \prod_{x \in Var} |dom(x)|$.
- Zustandsraum bei Nebenläufigkeit: Der Zustandsraum des TS von nebenläufigen Systemen ergibt sich aus dem kartesischen Produkt der Zustandsräume der Komponenten. Für $TS = TS_1 ||| \ldots ||| TS_n$ ergeben sich also für TS $|S_1| \cdot \ldots \cdot |S_n|$ viele Zustände.
- Zustandsraum bei Kanalsystemen: Das Transitionssystem TS(CS) hat exponentiell viele Zustände in der Anzahl der Prozesse, der Anzahl der Variablen, der Anzahl der Kanäle und der Kapazität der Kanäle.

Jan Schulz-Nigmann
14.09.2022

Lineare Temporale Logik

Aussagenlogik

Aussagenlogische Formeln:

Syntax der Aussagenlogik:

- Menge von Variablen P (Beispiel: $P = \{p, q, r, s, \dots\}$)
- Junktoren: $\vee, \wedge, \neg, \leftrightarrow, \rightarrow$
- Konstanten: $\top, \bot$

Die Menge der aussagenlogischen Formeln ist die *kleinste* Menge für die gilt:

- Die Konstanten $\top, \bot$ sowie alle Variablen in P sind aussagenlogische Formeln.
- Sind φ und ψ aussagenlogische Formeln, so sind auch $\neg\varphi$, $(\varphi \vee \psi)$, $(\varphi \wedge \psi)$, $(\varphi \leftrightarrow \psi)$, $(\varphi \rightarrow \psi)$ aussagenlogische Formeln. (Induktive Definition)

Die Menge der aussagenlogischen Formen ist die *kleinste* Menge für die gilt:

- Die Konstanten $\top, \bot$ sowie alle Variablen in P sind aussagenlogische Formeln.
- Sind φ und ψ aussagenlogische Formeln, so sind auch $\neg\varphi$, $(\varphi \vee \psi)$, $(\varphi \wedge \psi)$, $(\varphi \leftrightarrow \psi)$, $(\varphi \rightarrow \psi)$ aussagenlogische Formeln. (Induktive Definition)

Definition: Interpretation

Abbildung $I\colon P \rightarrow \{\top, \bot\}$
weist jeder Variable entweder den Wahrheitswert TRUE oder FALSE zu.

Jan Schulz-Nigmann
14.09.2022

Definition: Auswertungsfunktion

Auswertungsfunktion v_I weist einer Formel einen Wahrheitswert zu.

$$v_I(p) = I(p) \quad v_I(\top) = \top \quad v_I(\bot) = \bot$$

$$v_I(\neg\varphi) = \begin{cases} \top & v_I(\varphi) = \bot \\ \bot & sonst \end{cases} \qquad v_I(\varphi \leftrightarrow \psi) = \begin{cases} \top & v_I(\varphi) = v_I(\psi) \\ \bot & sonst \end{cases}$$

$$v_I(\varphi \wedge \psi) = \begin{cases} \top & v_I(\varphi) = \top \; und \; v_I(\psi) = \top \\ \bot & sonst \end{cases}$$

$$v_I(\varphi \vee \psi) = \begin{cases} \top & v_I(\varphi) = \top \; oder \; v_I(\psi) = \top \\ \bot & sonst \end{cases}$$

$$v_I(\varphi \rightarrow \psi) = \begin{cases} \top & v_I(\varphi) = \bot \; oder \; v_I(\psi) = \top \\ \bot & sonst \end{cases}$$

Definition: Erfüllbarkeit

- Falls $v_I(\varphi) = \top$ schreiben wir auch $I \vDash \varphi$
  „ I erfüllt φ“
- Eine aussagenlogische Formel φ ist **erfüllbar**, wenn es eine Interpretation I gibt mit $I \vDash \varphi$.

Definition: Gültigkeit

- Eine Formel φ ist **gültig**, wenn für alle Interpretationen I gilt: $I \vDash \varphi$
- Wir schreiben dann: $\vDash \varphi$

Definition: Schlussfolgerung

ϕ folgt aus Γ, geschrieben ($\Gamma \vDash \phi$), falls für alle Interpretationen I gilt:
Wenn für alle $\psi \in \Gamma$ gilt $I \vDash \psi$, dann ist ebenso $I \vDash \phi$.

Formalisierung mittels Aussagenlogik

Ein Programm P kann durch eine Aussagenlogische Formel Φ_p aus, sodass jede Interpretation $I \vDash \Phi_p$ genau dann, wenn I einen möglichen Zustand von P beschreibt. Für eine gewünschte Eigenschaft ψ ist die Schlussfolgerung $\Phi_P \vDash \psi$ gültig, wenn ψ in allen erreichbaren Zuständen von P erfüllt ist. Hier erreichen wir die Grenzen der Aussagenlogik, denn diese kann keine Zeit modellieren und berücksichtigt keine zeitlichen Abhängigkeiten.

Computation Tree

Der Computation Tree fasst alle möglichen Pfade eines Transitionssystems zusammen, dies ist eine sehr kompakte Darstellung. Im Baum werden zwei Pfade mit dem gleichen Präfix zusammengefasst. Der Baum ist unendlich tief, da die Pfade unendlich lang sind.

Jan Schulz-Nigmann
14.09.2022

Temporale Logik LTL

Der Ausgangspunkt ist ein Computation Tree, die Zeitlogik erlaubt Aussagen über das temporale Verhalten eines Systems. Formeln werden immer bezüglich eines TS und einem Anfangszustand ausgewertet. Die LTL erlaubt Aussagen über alle Pfade, die in einem Zustand starten und erweitert die Aussagenlogik um folgende Zustandsquantoren:

- X (NEXT) „im nächsten Zustand gilt“
- F (FUTURE) „in der Zukunft gilt irgendwann“
- G (GLOBAL) „in der Zukunft gilt immer“
- U (UNTIL) „es gilt eine Eigenschaft, bis eine andere gilt“

Wir schreiben $M, s \vDash \varphi$, wenn Formel φ im Zustand s des TS M erfüllt ist.

Defintion: Syntax der temporalen Logik

Die Menge der LTL-Formeln über einer Menge an Variablen P ist die kleinste Menge für die gilt:

- Atomare Eigenschaften $p \in P$ sowie die Konstanten $\top, \bot$ sind LTL-Formeln.
- Sind φ und ψ LTL-Formeln, dann sind auch $\neg\varphi,\ \varphi \vee \psi,\ \varphi \wedge \psi,$ $\varphi \rightarrow \psi,\ \boldsymbol{X}\varphi,\ \boldsymbol{F}\varphi,\ \boldsymbol{G}\varphi,\ \varphi\boldsymbol{U}\psi$ LTL-Formeln.

Um Klammern zu sparen, gibt es eine Operatorpräzendenz (stark zu schwach): Unäre Operatoren: $\neg, G, F, X$, Operator Until U, logische Operatoren $\wedge$,$\vee$ und Implikationsoperator $\rightarrow$. Falls mehrere Operatoren einer Stufe verwendet werden, gilt rechts nach links Gebundenheit: $GFp = G(Fp)$.

LTL-Formeln werden über unendliche Pfade $\pi = s_0, s_1, s_2, \ldots$ ausgewertet. Wir schreiben $\pi \vDash \varphi$, wenn φ über dem Pfad π erfüllt ist. π^i beschreibt den Pfad ab s_i.

Definition: Semantik, Teil 1

$\pi \vDash \top, \pi \nvDash \bot$ für alle Pfade π	(Konstanten)
$\pi \vDash p$ falls $p \in L(s_0)$	(Atomare Aussagen)
$\pi \vDash \neg\varphi$ falls $\pi \nvDash \varphi$	(Negation)
$\pi \vDash \varphi \wedge \psi$ falls $\pi \vDash \varphi$ und $\pi \vDash \psi$	(Konjunktion)
$\pi \vDash \varphi \vee \psi$ falls $\pi \vDash \varphi$ oder $\pi \vDash \psi$	(Disjunktion)
$\pi \vDash \varphi \rightarrow \psi$ falls $\pi \nvDash \varphi$ oder $\pi \vDash \psi$	(Implikation)

Definition: Semantik, Teil 2

- $\pi \vDash \boldsymbol{G}\varphi$ falls für alle Indices $k \geq 0$ gilt $\pi^k \vDash \varphi$
- $\pi \vDash \boldsymbol{F}\varphi$ falls es einen Index $k \geq 0$ gilt $\pi^k \vDash \varphi$
- $\pi \vDash \boldsymbol{X}\varphi$ falls $\pi^1 \vDash \varphi$
- $\pi \vDash \varphi\boldsymbol{U}\psi$ falls es einen Index $k \geq 0$ gibt mit $\pi^k \vDash \psi$ und für alle Indices $0 \leq j < k$ gilt $\pi^j \vDash \varphi$

Jan Schulz-Nigmann
14.09.2022

Definition: Semantik, Teil 3

Eine LTL-Formel φ ist im Zustand s_0 eines Transitionssystems $TS = (S, Act, \rightarrow, I, AP, L)$ **erfüllt**, wenn für alle Pfade $\pi = s_0, s_1, s_2, s_3 \ldots$, die in s_0 starten gilt: $\pi \vDash \varphi$
Wir scheiben dann $TS, s_0 \vDash \varphi$

Ein Transitionssystem $TS = (S, Act, \rightarrow, I, AP, L)$ **erfüllt** eine LTL-Formel ϕ gdw. für alle $s \in I$ gilt $TS, s \vDash \phi$

Definition: Gültigkeit und Erfüllbarkeit

Eine LTL-Formel φ ist **gültig**, wenn für alle Transitionssysteme $TS = (S, Act, \rightarrow, I, AP, L)$ und **alle** initialen Zustände $s \in I$ gilt $TS, s \vDash \varphi$

Eine LTL-Formel φ ist **erfüllbar**, wenn es ein Transitionssystem $TS = (S, Act, \rightarrow, I, AP, L)$ und **einen** initialen Zustände $s \in I$ gibt mit $TS, s \vDash \varphi$

Eigenschaften der LTL

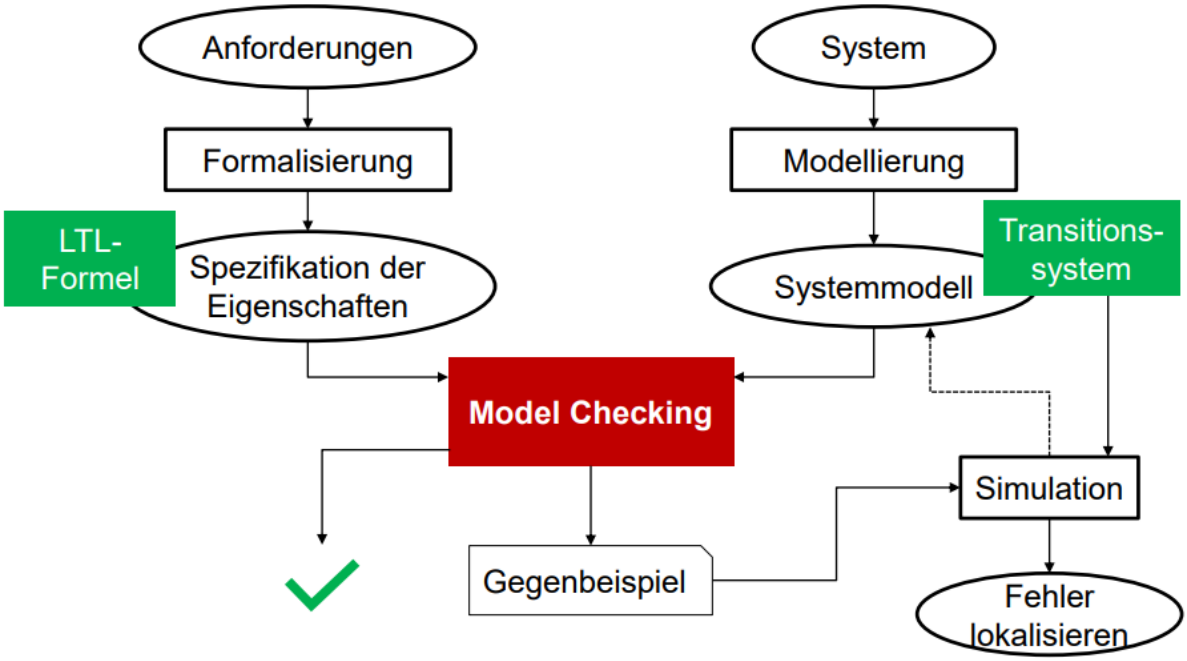
Safety-Eigenschaft: $G\varphi$ wird Safety-Eigenschaft genannt, diese modelliert Eigenschaften wie „Ein schlechtes Ereignis X tritt niemals ein“.

Liveness-Eigenschaft: $F\varphi$ wird Liveness-Eigenschaft genannt, diese modelliert Eigenschaften wie „Ein gutes Ereignis X tritt irgendwann einmal ein“.

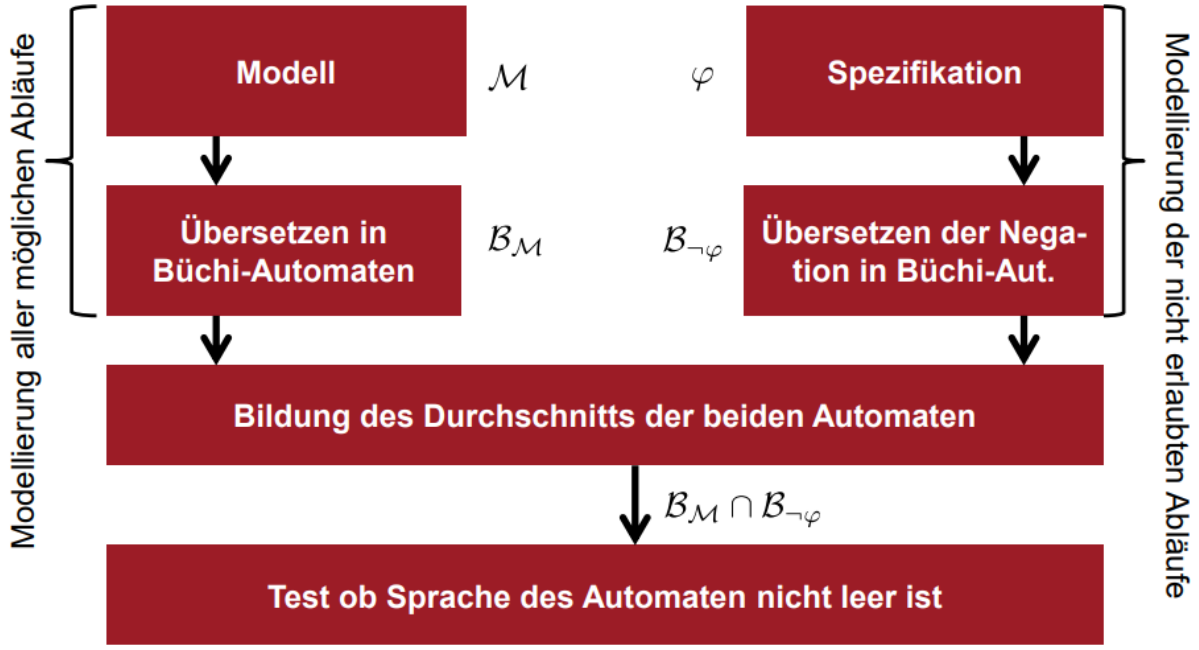
Diese beiden Eigenschaften lassen sich kombinieren um Aussagen wir „X ist unendlich oft erfüllt“ oder „X erreicht immer wieder einen Zustand…“ modellieren.

Jan Schulz-Nigmann
14.09.2022

Model Checking



LTL-Model Checking



Promela

Der Modelchecker ist Spin unter der Verwendung von Modellen in Promela. Promela (Process Meta-Language) ist eine Modellierungssprache für nebenläufige (verteilte) Systeme/Protokolle. Simuliert und verifiziert werden die Promela-Modelle mittels Spin. Promela ist keine klassische Programmiersprache und eignet sich nicht zur Programmierung realer Systeme, denn sie bietet keine Methoden, Libraries, GUI, Eingabe und verläuft nicht-deterministisch. Promela besteht aus Prozessen, Nachrichten-Kanälen und Variablen. Eine Synchronisation und ein Austausch von Nachrichten zwischen Prozessen sind möglich. Programme in Promela werden mit dem Keyword

proctype deklariert, ausgeführt können diese entweder mit dem Schlüsselwort active oder aus einem anderen Prozess mittels run.

Basisdatentypen

Name	Größe (bit)	Signed?	Wertbereich
bit	1	unsigned	0 … 1
bool	1	unsigned	0 … 1
byte	8	unsigned	0 … 255
mtype	8	unsigned	0 … 255
short	16	signed	-2^{15} … 2^{15} - 1
int	32	signed	$-2^{31} \dots 2^{31} - 1$

Komplexe Datentypen

Es gibt nativ nur eindimensionale Arrays mit konstanter Länge int x [5]. Es können aber auch eigene Datentypen deklariert werden, mit diesen lassen sich auch mehrdimensionale Arrays bilden, Keyword typedef name {}. Enumerations können mit Hilfe von mtype (message type) deklariert werden, pro Programm kann es nur einen mtype geben. Da es keine Methoden oder Funktionen in Promela gibt, kann man hier nur mit Hilfe von Makro-artigen inline Abkürzungen „Funktionen“ implementieren.

Kontrollstrukturen

If: nicht-deterministische Auswahl von Optionen, der erste Befehl nach der if-Klausel wird guard genannt. Mehrere Befehle, lassen sich mit ; voneinander trennen. Auch bedingte Ausdrücke sind möglich, hier sind Klammern wichtig: var = (guard -> then : otherwise).

```
if
:: (a < 5) -> commandA; commandA2
:: (a == 5) -> commandB
:: else -> commandC
fi
```

Anstelle von else (guard 3 wird nur gewählt, wenn die Abfragen davor false sind) kann auch true verwendet werden, dann würde der 3. Guard immer unabhängig von den Abfragen davor ausgeführt.

Schleifen: Promela bietet do und for Schleifen. Do-Schleifen werden durch einen break oder ein goto-Statement beendet, wenn aber keiner der guards erfüllt ist, blockiert die Schleife. For Schleifen können geschachtelt werden und mit ihnen sind Iterationen über Arrays möglich.

```
int a = 15, b = 20;
do
  :: a > b -> a = a - b
  :: b > a -> b = b - a
  :: a == b -> break
od
```

```
int i;
int sum = 0;
for(i : 1 .. N){
 sum = sum + 1
}
```

```
byte a[N];
byte i; byte sum = 0;
a[0] = 0; a[1] = 1; a[2] = 2; …
for(i in a){
  sum = sum + a[i]
}
```

Jan Schulz-Nigmann
14.09.2022

Sprünge: Sprünge können verwendet werden, um zu einem bestimmten Anweisungsblock zu „springen“, dies erfordert ein Label und die goto-Anweisung. Das Label sieht folgendermaßen aus: label1: Sprünge sind nur innerhalb eines Prozesses möglich und die Label müssen eindeutig sein.

Nachrichtenkanäle

Nachrichtenkanäle ermöglichen den Transport von Nachrichten einem Prozess zu einem anderen und werden als Queue implementiert. Wenn die Kapazität des Kanals 0 ist, muss die Nachricht direkt gelesen werden, sonst blockiert der Kanal.

```
Chan cname = [capacity] of {type_1,⋯,type_n}
```

Beim Senden von Nachrichten muss die Sequenz an Nachrichten entsprechend zur Initialisierung passen.

```
cname ! expr_1, ⋯, expr_n;
```

Beim Empfangen von Nachrichten gilt dasselbe.

```
cname ? var_1, ⋯, var_n;
```

Nicht-deterministische Modelle

Deterministische Promela-Modelle sind trivial, nicht-triviale Promela-Modelle sind nicht-deterministisch, wie zum Beispiel überlappende Guards oder mit dem Keyword select(i: LOW .. HIGH); welches nicht-deterministisch einen Wert für i festlegt. Der Nicht-Determinismus stammt aus zwei Quellen, den überlappenden Guards und dem scheduling nebenläufiger Prozesse.

Concurrency/Interleaving

Bei gemeinsamen Variablen kann unvorhergesehenes Verhalten entstehen. Der Prozess Scheduler bestimmt Reihenfolge, in der Statements verschiedener Prozesse ausgeführt werden.

Prozess Scheduling

Es können bis zu 255 Prozesse nebenläufig auf einem Prozessor laufen, der Scheduler wählt zufällig, aus welchem Prozess ein weiteres Statement ausgeführt werden soll. Ausführungen von Modellen sind entweder unendlich, terminierend oder blockierend. Das Prozess Scheduling kann begrenzt mit Atomicity und der Fairness-Anforderung gesteuert werden.

Atomicity: Ein Ausdruck oder ein Statement eines Prozesses, welcher komplett ohne die Möglichkeit von Interleaving ausgeführt wird, heißt atomar. Zuweisungen, Sprünge, skip, Ausdrücke und Conditional expressions sind automatisch atomar. Atomicity kann durch einen atomic-Block erzwungen werden.

Fairness: Eine Programmausführung wird weakly fair genannt, falls die folgende Bedingung gültig ist: Falls ein Statement immer ausgeführt werden kann, dann wird es irgendwann als Teil der Programmausführung ausgeführt

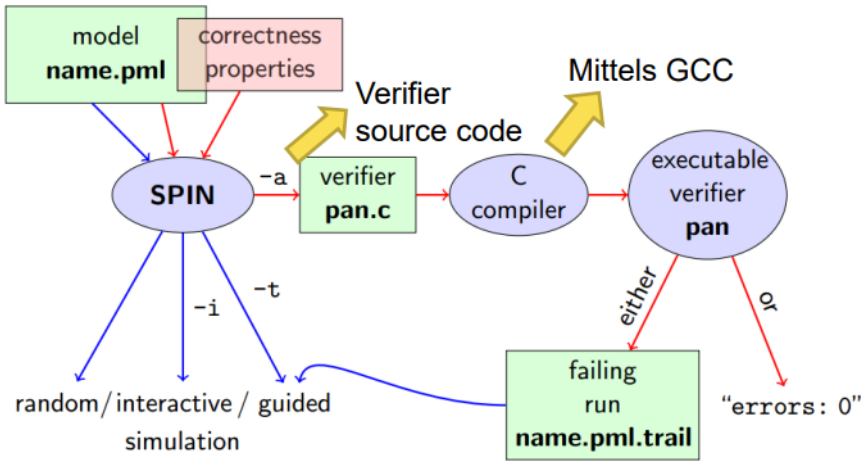
Anwendung

1. Modellierung der relevanten Features eines Systems mit Promella, Abstraktion von komplexen Berechnungen und Ersetzung von unbeschränkten Datenstrukturen durch endliche
2. Wähle Eigenschaften, die das Promela Modell erfüllen soll
    a. Allgemeine Eigenschaften: mutual exclusion und keine Deadlocks, keine Starvation
    b. System-spezifische Eigenschaften: Abwesenheit von Zuständen und Sequenz von Events

3. Verifiziere, dass alle Ausführungen des Modells die Eigenschaften erfüllen, meist sind mehrere Iterationen nötig um Modell und Eigenschaften korrekt zu formulieren, fehlgeschlagene Verifikation gibt Feedback in Form eines Gegenbeispiels

Spin

Spin (Simple Promela Interpreter) ist ein Werkzeug zur Simulation/Verifikation von Promela Modellen auf Fehler im Design (Deadlocks, Race Conditions, Verletzung von Aussagen, Safety und Liveness Eigenschaften). Prüft Modelle erschöpfend gegen Korrektheitseigenschaften und im Falle eines negativen Checks ein Gegenbeispiel liefert.



Korrektheitseigenschaften

Definition:
Korrektheit eines PROMELA Modells in Bezug auf Eigenschaft

Sei R_M eine Liste aller möglichen Ausführungen von M

- Für jede Korrektheitseigenschaft C_i ist R_{M,C_i} die Liste aller Ausführungen von M, die C_i erfüllen (es gilt $R_{M,C_i} \subseteq R_M$)
- M ist korrekt in Bezug auf $C_1, \dots, C_n$ falls gilt: $R_M = (R_{M,C_1} \cap \dots \cap R_{M,C_n})$
- Falls M nicht korrekt ist, dann ist $r \in (R_M / (R_{M,C_1} \cap \dots \cap R_{M,C_n}))$ ein Gegenbeispiel.

Die Korrektheitseigenschaften C können innerhalb oder außerhalb des Promela Modells M angegeben werden. Innerhalb: Assertion Statements und Meta Labels. Außerhalb: Never Claims und LTL-Logik Formeln.

Assertion Statements: Korrektheitseigenschaften können durch Assert-Statements überprüft werden: assert(expr) wobei expr ein beliebiger Promela Ausdruck vom Typ bool ist.

Meta Labels:

- End Labels: Nicht jeder Prozess muss immer terminieren, gültige Endzustände können mit End Labels markiert werden, um diese von Deadlocks zu unterscheiden. Endlabels beginnen mit dem Keyword end_
- Progress Labels: Erlaubt das Überprüfen, ob eine Code Zeile unendlich oft ausgeführt wird und beginnt mit dem Keyword progress:

Jan Schulz-Nigmann
14.09.2022

Never Claims: Spezifiziert einen Prozess der von Spin parallel zu jedem anderen Prozessschritt ausgeführt wird. In Never Claims darf es keine Zuweisungen geben.

LTL Formeln in Promela

Mit dem Schlüsselwort können LTL Formeln global spezifiziert werden: ltl [name] { formula }. LTL-Formeln können durch Spin in Never-Claims kompiliert werden.

Operatoren:

- [] Operator Global
- <> Operator Future
- ! Negation
- U Operator Until
- && oder /\ logisches AND
- || oder \/ logisches OR
- -> logische Implikation

LTL Model Checking

Büchi-Automaten

Endliche Automaten beschreiben Sprachen endlicher Wörter und entsprechen regulären Ausdrücken. Büchi-Automaten sind syntaktisch gleich zu endlichen Automaten aber akzeptieren unendlich lange Wörter und entsprechen omega-regulären Ausdrücken.

Definition: Wörter und Sprache

Sei Σ eine endliche Menge von Zeichen, dem sogenannten **Alphabet**. Dann bezeichnet Σ^* die **Menge aller endlichen Wörter** über Σ. Formal gilt:

$$\Sigma^* = \{\varepsilon\} \cup \bigcup_{i=1}^{\infty} \{a_1 a_2 a_3 \dots a_i | a_k \in \Sigma, 1 \leq k \leq i\}$$

$$\Sigma^+ = \bigcup_{i \in \mathbb{N}} \Sigma^i = \Sigma^* \setminus \{\varepsilon\}$$

Definition: Reguläre Ausdrücke - Syntax

Reguläre Ausdrücke über einem Alphabet Σ sind wie folgt definiert:

- $\emptyset$ (Symbol für die leere Menge) ist ein regulärer Ausdruck
- ϵ ist ein regulärer Ausdruck
- für alle $a \in \Sigma$ ist a ein regulärer Ausdruck
- sind x und y reguläre Ausdrücke, dann auch
  - $(x|y)$ – Alternative
  - (xy) – Konkatentation
  - (x^*) – Kleensche Hülle (endlich viele Wiederholungen von x)

Jan Schulz-Nigmann
14.09.2022

Definition: Reguläre Ausdrücke - Semantik

Es gilt:

- $L(\emptyset) = \emptyset$ Das Symbol für die leere Menge spezifiziert die leere Sprache
- $L(\epsilon) = \{\varepsilon\}$
- Für alle $a \in \Sigma$: $L(a) = \{a\}$
- Für alle regulären Ausdrücke x und y gilt
  - $L(x|y) = L(x) \cup L(y)$
  - $L(xy) = L(x) \cdot L(y)$
  - $L(x^*) = L^*(x)$

Auch hier verwenden wir eine Operator-Präzendenz um Klammern zu sparen (*>.>|) . ist die Konkatenation.

Definition: Unendliche Wörter

Σ^ωbezeichnet die **Menge aller unendlichen Wörter** über endlichem Alphabet Σ. Formal gilt:

$$\Sigma^\omega = \{a_1 a_2 a_3 a_4 \ldots | a_i \in \Sigma, i \geq 1\}$$

Sei $L \subseteq \Sigma^+$ und $L \neq \emptyset$, dann bezeichnet L^ω die Menge aller abzählbar unendlichen Konkatenationen von Wörtern aus L.

Definition: Endlicher Automat

Endlicher Automat ist definiert als Tupel $\mathcal{M} = (M, R, q, E, \Sigma)$, wobei folgendes gilt:

Formal:

- Endliche Menge an Zuständen M
- Endliches Alphabet Σ
- Zustandsübergänge: Relation $R \subseteq M \times \Sigma \times M$
- Startzustand $q \in M$
- Menge von Endzuständen: $E \subseteq M$

Ein endlicher Automat **akzeptiert** ein Wort $w_0 w_1 w_2 \ldots w_n$ mit $w_i \in \Sigma$ wenn:

- es eine Sequenz von Zuständen $q_0, q_1, q_2, \ldots, q_{n+1}$ gibt sodass
- der erste Zustand der Startzustand des Automaten ist $(q_0 = q)$
- der letzte Zustand ein Endzustand des Automaten ist $(q_{n+1} \in E)$ und ein „gültiger“ Zustandsübergang erfolgt: $(q_i, w_i, q_{i+1}) \in R$ für $0 \leq i \leq n$

Jan Schulz-Nigmann
14.09.2022

Definition: ω-reguläre Ausdrücke - Syntax

- Sei x ein regulärer Ausdruck, $L(x) \neq \emptyset$ und $\varepsilon \notin L$, dann ist x^ω ein ω-regulärer Ausdruck.
- Ist x ein regulärer Ausdruck und y ein ω-regulärer Ausdruck, dann ist xy ein ω-regulärer Ausdruck.
- Sind x, y ω-reguläre Ausdrücke, dann ist $x|y$ ein ω-regulärer Ausdruck.

Definition: ω-reguläre Ausdrücke - Semantik

- Für alle regulären Ausdrücke x: $L(x^\omega) = L^\omega(x)$
- Für alle regulären Ausdrücke x und ω-regulären Ausdrücke y:
$$L(xy) = L(x) \cdot L(y)$$
- Für alle ω-regulären Ausdrücke x, y: $L(x|y) = L(x) \cup L(y)$

Definition: Büchi-Automat

Ein Büchi-Automat $\boldsymbol{A} = (\boldsymbol{M}, \boldsymbol{R}, \boldsymbol{q}, \boldsymbol{E}, \boldsymbol{\Sigma})$ über einem endlichen Alphabet $\boldsymbol{\Sigma}$ besteht aus einer endlichen Menge $\boldsymbol{M}$ an Zuständen, einer Relation $\boldsymbol{R} \subseteq \boldsymbol{M} \times \boldsymbol{\Sigma} \times \boldsymbol{M}$, einem Startzustand $\boldsymbol{q} \in \boldsymbol{M}$ und einer Menge an Endzuständen $\boldsymbol{E} \subseteq \boldsymbol{M}$.

Wichtig: Syntax von Büchi Automaten gleich wie endlicher Automat. Unterschied liegt nur in der Art und Weise wie er Wörter akzeptiert.

Definition: Akzeptanzkriterium eines Büchi-Automat

Ein unendliches Wort $w = w_0 w_1 \ldots \in \Sigma^\omega$ entspricht einem Ablauf eines Büchi-Automaten $A = (M, R, q, E, \Sigma)$ wenn es eine Sequenz von Zuständen $m_0, m_1, m_2 \ldots \in M$ gibt mit $m_0 = q$ und $(m_{k-1}, w_{k-1}, m_k) \in R$ für alle $k \geq 1$. Der Automat **akzeptiert** dieses Wort, falls mindestens ein Zustand $m \in E$ unendlich oft im Ablauf $m_0, m_1, m_2 \ldots$ vorkommt.

Definition: Abschlusseigenschaften

Die Menge der omega-regulären Sprachen ist unter Vereinigung, Schnitt und Komplement **abgeschlossen**, d.h.

- sind die Sprachen L_1, L_2 omega-regulär, so sind es auch die Sprachen $L_1 \cup L_2$ und $L_1 \cap L_2$;
- ist die Sprache L_1 omega-regulär, so ist es auch das Komplement $\Sigma^\omega \backslash L_1$.

Bildung des Schnittes mit Hilfe der „Produktautomatenkonstruktion“: Zustände sind das kartesische Produkt der beiden Automaten + Bereiche, die Bereiche sind Bereich 0 (Initialzustände), Bereich 1 (Automat 1 war in Endzustand) und Bereich 2 (Automat 2 war in Endzustand). Die Endzustände beider Automaten müssen unendlich oft besucht werden.

Jan Schulz-Nigmann
14.09.2022

Idee der Produktautomatenkonstruktion

Gegeben: $A_1 = (M_1, R_1, q_1, E_1, \Sigma)$ und $A_2 = (M_2, R_2, q_2, E_2, \Sigma)$. Definiere einen neuen Büchi-Automaten mit Zuständen $M_1 \times M_2 \times \{0,1,2\}$, Startzustand $(q_1, q_2, 0)$, Endzuständen $M_1 \times M_2 \times \{2\}$ und der Relation:

- $(m_1, m_2, 0) \to (m_1', m_2', 0)$ falls $m_1 \to m_1', m_2 \to m_2', m_1 \notin E_1$
- $(m_1, m_2, 0) \to (m_1', m_2', 1)$ falls $m_1 \to m_1', m_2 \to m_2', m_1 \in E_1$
- $(m_1, m_2, 1) \to (m_1', m_2', 1)$ falls $m_1 \to m_1', m_2 \to m_2', m_2 \notin E_2$
- $(m_1, m_2, 1) \to (m_1', m_2', 2)$ falls $m_1 \to m_1', m_2 \to m_2', m_2 \in E_2$
- $(m_1, m_2, 2) \to (m_1', m_2', 0)$ falls $m_1 \to m_1', m_2 \to m_2'$

Entscheidbarkeit

Es ist effizient entscheidbar, ob die Sprache $L^{\omega}(\mathrm{A})$ eines Büchi-Automaten $\mathrm{A} = (M, R, q, E, \Sigma)$ die leere Menge ist.

Idee des Algorithmus: Betrachte Automaten als Graphen mit Knoten M und Kanten R, starke Zusammenhangskomponenten eines gerichteten Graphen sind ein Teilgraph in dem jeder Knoten durch Pfad von jedem anderen erreichbar ist. $L^{\omega}(A)$ ist nicht die leere Menge, wenn ein Endzustand in einer starken Zusammenhangskomponente liegt und vom Startzustand erreichbar ist. Die Komplexität dieses Algorithmus ist linear in der Anzahl an Knoten.

Transformation Transitionssystem in Büchi-Automaten

Modellierung der „gültigen“ Abläufe. Beim Umwandeln in einen Büchi-Automaten, werden die Kanten und Knoten übernommen, die Aktionen und Labels an den Knoten entfernt. Das Alphabet des Automaten ist die Potenzmenge P(AP) der atomaren Eigenschaften. Hinzufügen des neuen Startzustandes „init“ und beschriften der Kanten mit Labels des Zielknotens. Alle Knoten sind Endzustände.

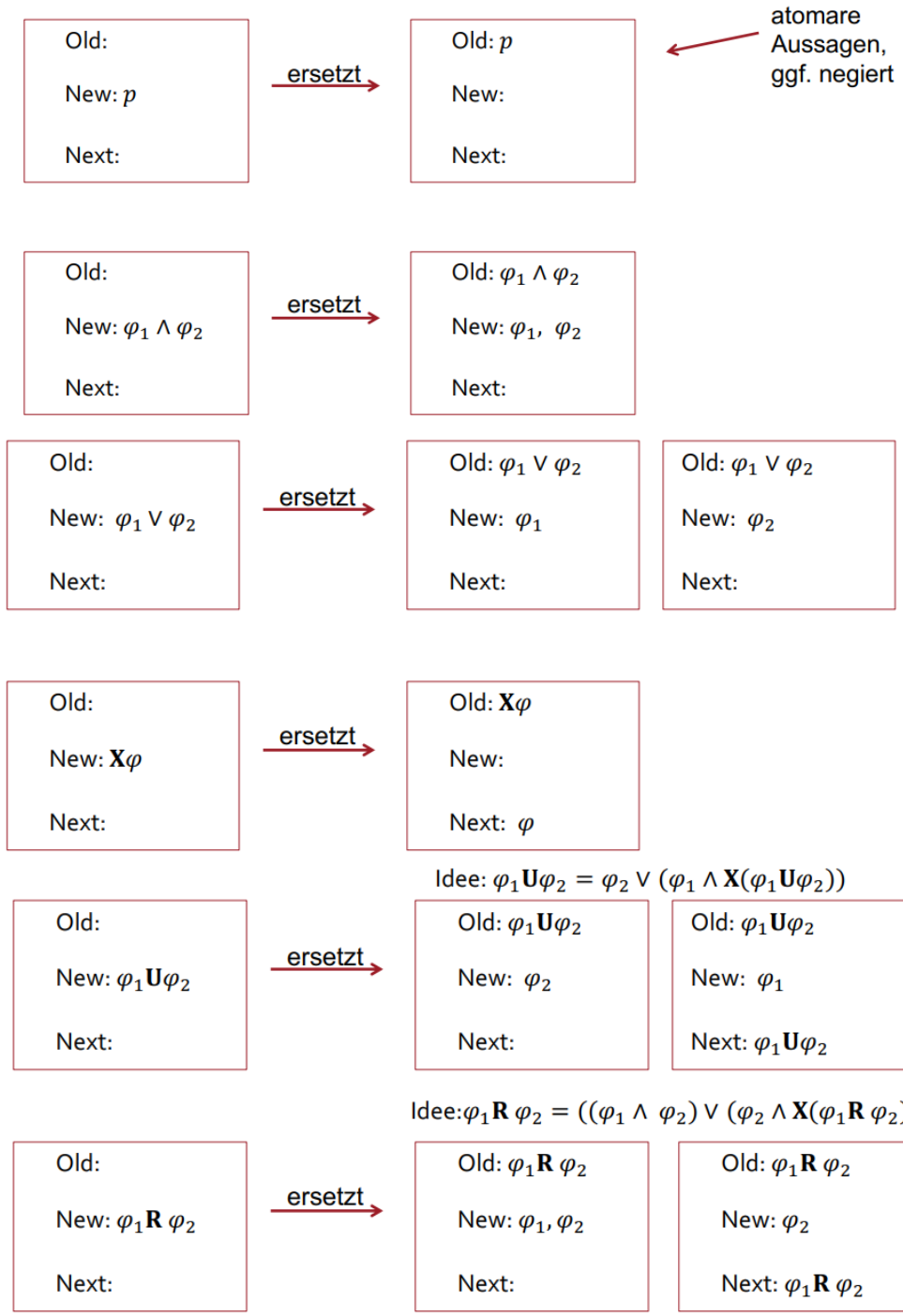
Transformation LTL-Formel in Büchi-Automaten

Modellierung der „nicht erlaubten“ Abläufe. Zu jeder LTL-Formel existiert ein Büchi-Automat, der alle Pfade akzeptiert, auf denen die Formel gültig ist. Zum Transformieren sind mehrere Schritte notwendig:

1. Schritt: Negationsnormalform: Alle Negationen sollen direkt vor atomaren Aussagen stehen. Folgende Umformungen sind möglich:
   - $\neg\neg\varphi = \varphi$
   - $\neg(\varphi_1 \vee \varphi_2) = \neg\varphi_1 \wedge \neg\varphi_2$
   - $\neg(\varphi_1 \wedge \varphi_2) = \neg\varphi_1 \vee \neg\varphi_2$
   - $\neg G\varphi = F\neg\varphi$
   - $\neg F\varphi = G\neg\varphi$
   - $\neg X\varphi = X\neg\varphi$
   - $\varphi_1 R\varphi_2 := \neg(\neg\varphi_1 U\neg\varphi_2)$ Neuer Release Operator R: φ_2 gilt einschließlich bis zur ersten Position, an der φ_1 gilt
   - $\neg(\varphi_1 U\varphi_2) = (\neg\varphi_1)R(\neg\varphi_2)$
   - $\neg(\varphi_1 R\varphi_2) = (\neg\varphi_1)U(\neg\varphi_2)$
   - $F\varphi = \top U\varphi$
   - $G\varphi = \bot R\varphi$
2. Schritt: Transformation in Graphen: Wir nehmen an, dass unsere Formel nur noch atomare (ggf. negierte) Aussagen, Konjuktionen, Disjunktionen und die Operatoren X, U und R enthält.

Jan Schulz-Nigmann
14.09.2022

Die Formel wird nun schrittweise expandiert und Knoten eines Graphen werden erzeugt bzw ersetzt. Jeder Knoten beinhaltet vier Mengen: „Incoming“ Vorgänger des Knoten im Automaten, „Old“ bereits bearbeitete Formeln, „New“ Formeln, die noch vom rekursiven Algorithmus bearbeitet werden müssen und „Next“ Formeln für den nächsten Knoten. Der Algorithmus startet mit der gesamten Formel in New und in Incoming befindet sich der init Knoten. Ist in einem Knoten das Feld New nicht leer, so wird der Knoten verfeinert oder ersetzt. Dies geschieht nach folgenden Verfeinerungsregeln:



3. Transformation in Automaten: Alphabet = alle atomaren Eigenschaften, Zustände = fertige vom Algorithmus erzeugten „fertigen“ Knoten und Startzustand, Zustandsübergänge = alle Übergänge, die in den Incoming-Feldern generiert wurden, werden beschriftet mit allen atomaren Eigenschaften, die die atomaren Eigenschaften im Old-Feld des Zielknotens erfüllt.

Falls Old-Feld leer, dann handelt es sich um einen True Übergang, Anfangszustand = artifizieller Init-Zustand.

Die Konstruktion liefert einen generalisierten Büchi-Automaten, bei dem es mehrere Akzeptanzmengen gibt; dieser akzeptiert ein unendliches Wort, falls in diesem Wort in jeder Akzeptanzmenge mindestens ein akzeptierender Zustand unendlich oft vorkommt. Generalisierte Büchi-Automaten können in Büchi-Automaten transformiert werden.

Definition Akzeptanzmenge: Für jede Teilformel der Form $\varphi_1 U \varphi_2$ wird eine Akzeptanzmenge erstellt. Die Menge enthält alle Knoten, in denen φ_2 in Old oder $\varphi_1 U \varphi_2$ nicht in Old vorkommt. Dies garantiert, dass irgendwann φ_2 gelten muss, sobald $\varphi_1 U \varphi_2$ gilt.

Prädikatenlogik

Die typisierte Prädikatenlogik erster Stufe besteht aus Funktionensymbolen, Konstanten, Relationen/Prädikaten, Variablen, Logischen Junktoren, logischen Konstanten True/False und Logischen Quantoren $\forall, \exists$ über Variablen. Sprache soll flexibel sein und beliebige Funktionen/Konstanten unterstützen (durch eine Signatur) und alle Variablen und Funktionen müssen typisiert sein.

Definition: Signatur

Eine Signatur Σ besteht aus

- einer Menge T_Σ von Typen,
- einer Menge F_Σ von Funktionssymbolen,
- einer Menge P_Σ von Relationssymbolen (Prädikaten) und
- einer Typfunktion α_Σ mit $\alpha_\Sigma(p) \in T_\Sigma^*$ für alle $p \in P_\Sigma$ und $\alpha_\Sigma(f) \in T_\Sigma^* \times T_\Sigma$ für alle $f \in F_\Sigma$

Die **Arität** eines Prädikates ist definiert durch $|\alpha_\Sigma(p)|$, die **Arität** einer Funktion ist $|\alpha_\Sigma(f)| - 1$. Wir schreiben f/n und p/n, um die Arität n von Funktionssymbolen f und Relationssymbolen p anzugeben.

Ein **Term** des Typs $t \in T_\Sigma$ ist entweder

- eine Variable $v \in V$ des Typs $\alpha_\Sigma(v) = t$ oder
- ein Term $f(t_1, \ldots, t_n)$, wobei das Resultat den Typ t besitzt sowie alle Terme t_i den korrekten Typ nach $\alpha_\Sigma(f)$ besitzen.

Definition: Atomare Formeln

Atomare Formeln sind

- die Konstanten $\top, \bot$,
- Prädikate $p(t_1, \ldots, t_n)$, wobei $p \in P_\Sigma$ und alle Terme t_i sind korrekt nach α_Σ typisiert, oder
- Ausdrücke der Form $t_1 = t_2$ für zwei Terme t_1, t_2 gleichen Typs.

Jan Schulz-Nigmann
14.09.2022

Definition: Prädikatenlogische Formeln

Die Menge der typisierten prädikatenlogischen Formeln über der Signatur Σ ist die kleinste Menge für die gilt:

- Atomare Formeln sind prädikatenlogische Formeln
- Sind φ, ψ prädikatenlogische Formeln, so sind es auch:
  - $\neg\varphi, \varphi \vee \psi$,
  - $\varphi \wedge \psi$,
  - $\varphi \rightarrow \psi$,
  - $\varphi \leftrightarrow \psi$,
  - $\forall \tau\, x : \varphi$,
  - $\exists \tau\, x : \varphi$,
    wobei x eine Variable vom Typ τ ist.

Prädikatenlogik – Semantik

Eine Interpretation einer Formel umfasst eine typisierte Menge an Elementen (Universum), eine Spezifikation der Funktionen, eine Repräsentation der Relationen und eine Zuweisung von Elementen an Variablen. Eine nicht-leere Menge D bezeichnet man als Universum. Jedem Element des Universums wird mit der Funktion $\delta: D \rightarrow T$ ein Typ zugewiesen. Wir schreiben für alle Elemente des Universums eines bestimmten Typs τ: $D^{\tau} = \{d \in D | \delta(d) = \tau\}$. Wir nehmen an, dass jedes Element nur einen Typ haben kann und dass von jedem Typ mindestens ein Element vorhanden sein muss.

Interpretation:

Für eine Funktion f mit $\alpha_{\Sigma}(f) = (\tau_1, \tau_2, \ldots, \tau_n, \tau)$ ist eine Interpretation gegeben durch eine Funktion $I(f): D^{\tau_1} \times D^{\tau_2} \times \ldots \times D^{\tau_n} \rightarrow D^{\tau}$

Für eine Relation p mit $\alpha_{\Sigma}(p) = (\tau_1, \ldots, \tau_n)$ ist eine Interpretation gegeben durch eine Relation $I(p) \subseteq D^{\tau_1} \times \ldots \times D^{\tau_n}$

Wir bezeichnen das Tupel $S = (D, \delta, I)$ als einen prädikatenlogischen Zustand („first order state“).

Zum Zuweisen von Werten an Variablen nutzen wir die Abbildung $\beta: V \rightarrow D$. Diese Funktion muss den Typ der Variablen respektieren, für eine Variable x des Typs τ muss gelten: $\beta(x) \in D^{\tau}$. Zum Zuweisen auf eine andere Variable schreiben wir $\beta_y^d(x) = \begin{cases} \beta(x), x \neq y \\ d, x = y \end{cases}$.

Evaluation einer Formel

1. Schritt: Terme

Jan Schulz-Nigmann
14.09.2022

Evaluation von Termen:

Sei $S = (D, \delta, I)$ ein prädikatenlogischer Zustand und $\beta : V \to D$. Wir evaluieren Terme mit einer Funktion $val_{S,\beta} : Term \to D$ rekursiv:

- Variablen: $val_{S,\beta}(x) = \beta(x)$
- Funktionen: $val_{S,\beta}(f(t_1, \dots, t_n)) = I(f)(val_{S,\beta}(t_1), \dots, val_{S,\beta}(t_n))$

2. Schritt

Evaluation von prädikatenlogischen Formeln:

Sei $S = (D, \delta, I)$ ein prädikatenlogischer Zustand und $\beta : V \to D$. Wir evaluieren eine Formel rekursiv durch eine Funktion $val_{S,\beta} : FO \to \{\top, \bot\}$

- **Prädikate:** $val_{S,\beta}(p(t_1, \dots, t_n)) = \top$, falls
  $$(val_{S,\beta}(t_1), \dots, val_{S,\beta}(t_n)) \in I(p)$$
- **Logische Ausdrücke:** $val_{S,\beta}(\varphi \wedge \psi) = \top$, falls
  $$val_{S,\beta}(\varphi) = \top \text{ und } val_{S,\beta}(\psi) = \top$$
  - Ähnliche Regeln für Oder, Implikation, Äquivalenz
- **Allquantor:** $val_{S,\beta}(\forall \tau\, x : \varphi) = \top$ falls
  $$\text{für alle } d \in D^\tau \text{ gilt: } val_{S,\beta^d_x}(\varphi) = \top$$
- **Existenzquantor:** $val_{S,\beta}(\exists \tau\, x : \varphi) = \top$ falls
  $$\text{es ein } d \in D^\tau \text{ gibt mit: } val_{S,\beta^d_x}(\varphi) = \top$$

Die Formel φ ist **erfüllbar**, wenn es einen Zustand S und eine Abbildung für Variablen β gibt, sodass $val_{S,\beta}(\varphi) = \top$

Die Formel φ ist in S **erfüllt**, wenn für alle Abbildungen β gilt: $val_{S,\beta}(\varphi) = \top$
Wir schreiben dann $S \vDash \varphi$

Die Formel φ ist **gültig**, wenn für alle Zustände S und alle Abbildungen β gilt: $val_{S,\beta}(\varphi) = \top$. Wir schreiben dann $\vDash \varphi$.

Jan Schulz-Nigmann
14.09.2022

Sequenzkalküle

Definitionen: Kalkül und Sequenz

Ein **Kalkül** umfasst eine Menge von **Transformationsregeln**, durch deren Anwendung auf gegebene Aussagen (=**Axiome**) sich andere Aussagen ableiten lassen.

Wir verwenden Großbuchstaben $A, B, C, \dots$ für einzelne Formeln und griechische Großbuchstaben Γ, Δ für eine Menge von Formeln.

Eine **Sequenz** ist eine Zeichenfolge $A, \Gamma \vdash B, \Delta$ mit $\Gamma = \{\varphi_1, \dots, \varphi_n\}$ und $\Delta = \{\psi_1, \dots, \psi_m\}$ hat die gleiche Bedeutung wie die Formel

$$(A \wedge \varphi_1 \wedge \cdots \wedge \varphi_n) \rightarrow (\mathrm{B} \vee \psi_1 \vee \cdots \vee \psi_m)$$

Das Symbol $\vdash$ (engl.: *turnstile*, TeX: \vdash) wird als "beweist" oder "bedingt" gelesen.

Axiome: Ein Axiom ist ein als wahr angenommener Grundsatz. Zu diesen gehören die Sequenz $\Gamma, A \vdash A, \Delta$, die Sequenz $\Gamma \vdash \top, \Delta$ und die Sequenz $\Gamma, \bot \vdash \Delta$.

Transformationsregeln: Regeln zum Umformen einer Sequenz. Eine Regel ist korrekt, wenn die Gültigkeit der Voraussetzungen die Gültigkeit der Schlussfolgerung impliziert. Um die Gültigkeit der Schlussfolgerung zu beweisen, müssen alle Voraussetzungen bewiesen werden. Es gibt folgende Regeln:

Negation:

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}(\neg l) \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta}(\neg r)$$

Konjunktion:

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge \mathrm{B} \vdash \Delta}(\wedge l) \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge \mathrm{B}, \Delta}(\wedge r)$$

Disjunktion:

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \vee \mathrm{B} \vdash \Delta}(\vee l) \qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta}(\vee r)$$

Implikation:

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \rightarrow \mathrm{B} \vdash \Delta}(\rightarrow l) \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \rightarrow B, \Delta}(\rightarrow r)$$

Identität:

$$\frac{}{\Gamma, A \vdash A, \Delta}(I)$$

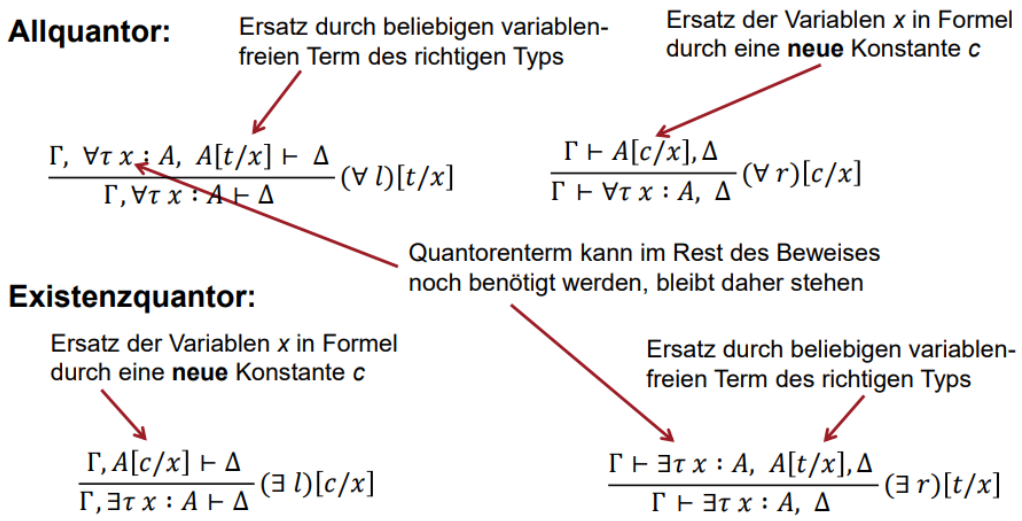
Notation: Ersatz des Terms z durch Term t_2

Gleichheit:

$$\frac{\Gamma, t_1 = t_2, \phi[t_1/z], \phi[t_2/z] \vdash \Delta}{\Gamma, t_1 = t_2, \phi\,[t_1/z] \vdash \Delta}(= l) \qquad \frac{\Gamma, t_1 = t_2 \vdash \phi[t_2/z], \phi[t_1/z], \Delta}{\Gamma, t_1 = t_2 \vdash \phi\,[t_1/z], \Delta}(= r)$$

$$\frac{\Gamma, t_2 = t_1 \vdash \Delta}{\Gamma, t_1 = t_2 \vdash \Delta}(= symL)$$

Jan Schulz-Nigmann
14.09.2022



Beweis der Gültigkeit einer Formel

Um zu beweisen, dass eine FO-Formel (First-Order Logic Formel) gültig ist, müssen wir diese zunächst in eine Sequenz transformieren und dann die Transformationsregeln anwenden, um schrittweise logische Operatoren zu eliminieren, bis nur noch Axiome vorhanden sind.

Dynamische Logik

Prädikatenlogik erster Stufe kann nur statische Aussagen modellieren und betrachtet nur einen Programmzustand. Die dynamische Logik kann Aussagen über das dynamische Verhalten eines Programms treffen und Aussagen über Zustände vor und nach Ausführung eines Programms verbinden. Die Logik kann Aussagen über Terminierung und Korrektheit treffen. Die Dynamische Logik ist eine Typisierte Prädikatenlogik erster Stufe unter Hinzunahme der Modalitäten $\langle p\rangle\varphi$ und $[p]\varphi$, wobei p ein Programm in einer bestimmten Programmiersprache und φ eine DL-Formel ist. $\langle p\rangle\varphi$ = Programm p terminiert und danach gilt φ. $[p]\varphi$ = Falls Programm p terminiert, gilt danach φ.

Definition: Signatur der dynamischen Logik

Die Signatur Σ der Dynamischen Logik besteht aus:

- einer Menge T_Σ von Typen
- einer Menge F_Σ an Funktionssymbolen („rigid“)
- einer Menge P_Σ an Relationssymbolen
- einer Menge PV_Σ an Programmvariablen
- einer Typisierungsfunktion α_Σ

Wobei wir zwei Typen von Variablen unterscheiden: Logische Variablen (Menge V) typisiert, lokal definiert in einem Ausdruck mit Quantor, darf nicht in einem Programm vorkommen und Programmvariablen (Menge PV) können nicht mit Quantor versehen werden aber dürfen in Programmen und Formeln vorkommen.

Terme in DL sind definiert wie Terme in Prädikatenlogik, zusätzlich sind Programmvariablen Terme des entsprechenden Typs.

Jan Schulz-Nigmann
14.09.2022

Definition: DL-Formeln

Die Menge der DL-Formeln ist die kleinste Menge für die gilt:

- Alle Formeln in Prädikatenlogik sind DL-Formeln.
- Ist p ein Programm und φ eine DL-Formel, so sind auch $\langle p \rangle \varphi$ und $[p]\varphi$ DL-Formeln.
- Die Menge der DL-Formeln ist unter den logischen Verknüpfungen sowie unter All- und Existenzquantoren abgeschlossen.

Dynamische Logik – Semantik

Zustände beinhalten Interpretation aller Programmvariablen, Funktions- und Relationssymbole. Eine Ausführung des Programms ändert Zustände der Programmvariablen. Semantik daher als partielle Funktion auf der Menge der Zustände interpretierbar. Die Transitionsrelation $\rho: Program \rightarrow (States \rightarrow States)$ beschreibt $\rho(p)(s_1) = s_2$ genau dann, wenn „Programm p in Zustand s_1 normal terminiert und den finalen Zustand s_2 annimmt." Diese Funktion ist partiell, da Programm nicht terminieren muss. Der Zustand ist modelliert als Tupel $S = (D, \delta, I)$, wobei Interpretation I von Programmvariablen abhängt. Die Menge aller Zustände ist die Menge States.

Gültigkeit der DL-Modalitäten:

- $S \vDash \langle p \rangle \varphi$ gilt dann und nur dann, wenn $\rho(p)(S)$ definiert ist **und** $\rho(p)(S) \vDash \varphi$
- $S \vDash [p] \varphi$ gilt dann und nur dann, wenn $\rho(p)(S) \vDash \varphi$ sobald $\rho(p)(S)$ definiert ist.

DL-Sequenzen

Erweiterung des Kalküls für Prädikatenlogik um Modalitäten und Programme. Programme werden durch syntaktisch einfachere (aber längere) Programme ersetzt. Ausdrücke eines Programms werden in ihrer „natürlichen" Reihenfolge ausgewertet (dem Kontrollfluss folgend). Symbolische Repräsentation des Zustands falls Werte der Variablen nicht bekannt. Betrachte immer die „erste" Instruktion des Programms sowie den Rest: $\langle statement; rest \rangle \varphi$, $[statement; rest]\varphi$. Regeln evaluieren immer die erste Instruktion, dadurch wird das Programm schrittweise „vereinfacht".

IF-Anweisungen:

$$\frac{\Gamma, b = \top \vdash \langle p; \text{rest} \rangle \varphi, \Delta \qquad \Gamma, b = \bot \vdash \langle q; \text{rest} \rangle \varphi, \Delta}{\Gamma \vdash \langle \text{if (b) \{p\} else \{q\}; rest} \rangle \varphi, \Delta}$$

Abrollen von Schleifen:

$$\frac{\Gamma \vdash \langle \text{if (b) \{p ; while (b) \{p\} \} ; rest} \rangle \varphi, \Delta}{\Gamma \vdash \langle \text{while (b) \{p\} ; rest} \rangle \varphi, \Delta}$$

Jan Schulz-Nigmann
14.09.2022

Schleifeninvarianten: Einfaches abrollen von Schleifen reicht nicht. Alternative Lösung von Schleifeninvarianten. Schleifeninvarianten sind Aussagen, die vor und nach der Ausführung der Schleife gelten, dadurch kann Schleife bei dem Beweis „umgangen“ werden.

Einfache Schleifenregel:

Invariante gilt immer

Invariante gilt nach Ausführung eines Schleifendurlaufs

$$\frac{\Gamma \vdash U\ Inv, \Delta \quad Inv, b = \top \vdash [p]Inv \quad Inv, b = \bot \vdash [\pi\ \omega]\varphi}{\Gamma \vdash U[\pi \text{ while(b) p}\omega]\varphi, \Delta}$$

Regeln für Zuweisungen: Zuweisungen können über Updates der entsprechenden Variablen modelliert werden. Updates verändern den aktuellen Zustand S. Ein Update hat die Form $\{v \coloneqq t\}$, wobei v eine Variable ist, t ein beliebiger first-order logischer Term und β eine Variablenzuweisung für die first-order logischen Variablen in t ist.

Parallele Updates: Ein paralleles Update ist ein Ausdruck der Form $\{v_1 \coloneqq t_1 \parallel v_2 \coloneqq t_2 \parallel \cdots \parallel v_n \coloneqq t_n\}$, wobei jedes $\{v_i \coloneqq t_i\}$ ein einfaches Update ist. Updates aller Programmvariablen werden gleichzeitig ausgeführt. Bei einem Konflikt $v_i = v_j, t_j \neq t_j$ gewinnt das spätere Update. Mengen von Updates schreiben wir als U.

Zuweisung über Updates: $$\frac{\Gamma \vdash \{x \coloneqq t\}\langle rest\rangle\varphi,\ \Delta}{\Gamma \vdash \langle x = t;\ rest\rangle\varphi,\ \Delta}$$

Einfache Regel: Ersatz des Funktionsaufrufs durch Code der Funktion selbst, davor werden die Parameter evaluiert und neuen Variablen zugewiesen.

Komplexere Regel: „Method Contract Rule”: Jede Methode beschreibt Bedingungen, die vor der Ausführung gelten müssen und Bedingungen, die nach dem Aufruf sicher gelten. Benutzt werden diese „Kontrakte“ in der Verifikation.

Java Modeling Language

JML ist eine auf Java zugeschnittene Spezifikationssprache. JML Annotationen werden direkt als spezielle Kommentare in den Java Quellcode hinzugefügt.

Komponentenspezifikation ist vergleichbar mit unit testing. Methodenspezifikation muss folgende Aspekte berücksichtigen: Ergebniswert, Initialwerte der formalen Parameter und zugängliche Bereiche des Pre-/Post-Zustand. Unsere Spezifikation wird als Vertrag zwischen Entwickler und Benutzer bzw. Aufrufer und aufgerufene Methode formalisiert. Aufgerufene Methode versichert bestimmte Resultate, falls Aufrufer Vorbedingung erfüllt. Der Vertrag gibt an, was garantiert unter welcher Bedingung gilt.

Definition: Vor- und Nachbedingungen

Ein Paar aus Vor-/Nach-Bedingung für eine Methode m wird von einer Implementierung von m erfüllt, wenn gilt:

Falls die Methode in einem Zustand, in dem die Vorbedingung gilt, aufgerufen wird, so gilt in jedem Zustand nach Termination die Nachbedingung.

Jan Schulz-Nigmann
14.09.2022

Die Nachbedingung muss nur nach Termination gelten und es gibt keine Garantien, dass die Vorbedingung erfüllt ist. Die Terminierung kann auch über Exceptions erfolgen. Die formale Spezifikation gibt Verträge von Komponenten mit mathematischer Genauigkeit an.

JML Syntax

Kommentare in Java werden mit der Form /*begonnen und mit */ beendet. Java Kommentare die mit @ starten werden von JML interpretiert, hierbei muss nach Kommentarbeginn @ das erste Zeichen sein.

Eine public Spezifikation ist zugreifbar von allen Klassen und Interfaces und kann sich nur auf public Methoden/Felder beziehen. Jedes Schlüsselwort, dass mit _behavior aufhört startet ein Spezifikationsfall. Normal_behavior garantiert, dass die Methode keine Exception wirft, falls die Vorbedingung des Spezifikationsfalls erfüllt ist. Ein Spezifikationsfall kann mehrere Vorbedingungen haben, diese beginnen mit dem Keyword requires und sind boolsche JML-Ausdrücke. Ein Spezifikationsfall kann mehrere Nachbedingungen haben, diese sind ebenfalls JML-Ausdrücke und haben das Keyword ensures. Um mehrere Spezifikationsfälle zu definieren, nutzt man das Keyword also und kann nach diesem einen neuen Spezifikationsfall beginnen. Im Spezifikationsfall können auch auf Werte des Pre-Zustands in den Nachbedingungen zugegriffen werden. \old(E) bedeutet dass E im Pre-Zustand ausgewählt wird. E kann ein beliebiger JML-Ausdruck sein. In der Nachbedingung müssen alle sichtbaren Felder aller sichtbaren Klassen angegeben werden. JML nimmt an, dass die Umgebung unverändert ist, falls nicht explizit angegeben.

Zuweisbare Felder

Es ist effizienter explizit die Liste aller Felder anzugeben, die geändert werden können. Die assignable Angabe gibt explizit an, welche Felder verändert werden dürfen, und insbesondere dürfen alle anderen Felder nicht verändert werden (temporäre Änderungen sind erlaubt). Assignable \nothing gibt an, dass keine Felder verändert werden dürfen und assignable \everything gibt an, dass alle Felder verändert werden dürfen (Standard).

JML Modifiers

Spec_public: Da die public Spezifikation nur auf public Felder Zugreifen kann, kann man Felder mit einem JML-Kommentar mit dem Modifier spec_public versehen, so kann man die Sichtbarkeit für die Spezifikation anpassen und auch private/protected Felder verwenden.

Pure: Eine Spezifikation kann kompakter gestaltet werden, indem Methodenaufrufe innerhalb der JML-Annotationen verwendet werden. Die Spezifikation darf jedoch nicht den Zustand verändern. Eine java Methode ist pure, genau dann, wenn sie keine Seiteneffekte hat und immer terminiert. JML-Ausdrücke können nur pure Methoden aufrufen. Der Entwickler muss überprüfen, dass eine pure Methode tatsächlich pure ist, an dieser Stelle ist Formale Verifikation möglich, pure impliziert assignable \nothing.

Jan Schulz-Nigmann
14.09.2022

JML Ausdrücke

Definition: JML Ausdrücke (1)

- jeder seiteneffektfreie Java Ausdruck ist ein JML Ausdruck
  - jeder Methodenaufruf muss zu einer `pure` Methode sein
  - z.B. `i++` ist kein JML Ausdruck
- Falls *E* ein seiteneffektfreier Java Ausdruck ist, dann ist `\old(E)` ein JML Ausdruck
- Falls `a` und `b` **boolesche** JML Ausdrücke sind, dann auch
  - `!a` („nicht a“)
  - `a && b` („a und b“)
  - `a || b` („a oder b“)
  - `a ==> b` („a impliziert b“)
  - `a <==> b` („a is äquivalent zu b“)

Das ist noch nicht ausreichend!

Definition: JML Ausdrücke (2)

- …
- Falls `a` und `b` **boolesche** JML Ausdrücke sind, dann auch
  - …
  - `(\forall t x; a)` („für alle Werte `x` des Typs `t`, a ist wahr“)
  - `(\exists t x; a)` („es existiert ein Wert `x` des Typs `t`, sodass `a`“)
  - `(\forall t x; a; b)` („für alle Werte `x` des Typs `t`, **die `a` erfüllen**, `b` ist wahr“)
  - `(\exists t x; a; b)` („es existiert ein Wert `x` des Typs `t`, **der `a` erfüllt**, sodass `b` wahr ist“)

Definition: Bereichsprädikat

In den JML Ausdrücke `(\forall t x; a; b)` und `(\exists t x; a; b)` wird das **boolesche** `a` als Bereichsprädikat bezeichnet.

Bereichsprädikate sind syntaktischer Zucker. (\forall t x; a; b) ist äquivalent zu (\forall t x; a ==> b). Bereichsprädikate können genutzt werden, um den Bereich x näher als nur via des Typs t einzuschränken. Mittels \created lässt sich ein Bereich auf nur erzeugte Objekte einschränken.

Jan Schulz-Nigmann
14.09.2022

Rückgabewerte

Pure Methoden haben keine Auswirkung auf den Zustand und dürfen insbesondere keine Exceptions werfen. Wir müssen also den Rückgabewert spezifizieren. In Nachbedingungen bezieht sich \result auf den Rückgabewert der Methode.

Spezifikation von Zustandseinschränkungen

Zustandseinschränkungen sind global, alle Methoden müssen sie bewahren. Zustandseinschränkungen können Einschränkungen des Zustands einer Klasse spezifizieren. Die Klasseninvariante kann an jeder Stelle in der Klasse stehen und hat das Keyword invariant. Methodenverträge hingegen müssen direkt vor der Methode stehen.

Instanzinvarianten können sich auf Instanzfelder des this Objekts beziehen, Keyword instance invariant

Statische Invarianten: Können sich nicht auf Instanzfelder des this Objekts beziehen, Keyword static invariant

Semantik von JML/KeY Klasseninvarianten

Für jede Methode m() einer Klasse C: Für jede Invariante I von C (inklusive Invarianten von Superklassen): Füge I zu Vor/Nachbedingung des Vertrags von m() zu, falls m() ein Konstruktur ist, muss I für das neue Objekt sichergestellt werden.

Spezifikation von Ausnahme-Verhalten

Normal_behavior Spezifikationsfall: Angenommen Vorbedingung P ist erfüllt, verbietet Methode eine Ausnahme zu werfen, falls Vorzustand P erfüllt.

Exceptional_behavior Spezifikationsfall: Angenommen Vorbedingung P ist erfüllt, erfordert, dass Methode eine Ausnahme wirft, falls Vorzustand P erfüllt. Signals_only spezifiziert den Typ der Ausnahme. Ein Ausnahme-Spezifikationsfall kann höchstens eine Klausel der folgenden Form haben signals_only $E_1, E_2, \dots, E_n$ wobei E die Ausnahmetypen sind und die geworfene Ausnahme muss einen dieser Typen annehmen. Keyword signals spezifiziert den Post-Zustand, abhängig vom Typ der geworfenen Ausnahme. Ein Ausnahme-Spezifikationsfall kann mehrere Klauseln der folgenden Form haben singals (E) b; wobei E Ausnahmetyp und b ein boolescher JML-Ausdruck ist. Falls eine Ausnahme vom Typ E geworfen wird, dann gilt b im Post-Zustand.

Die normale/Ausnahme-Spezifikationen müssen durch geeignete (disjunkte) Vorbedingung getrennt werden. Standardmäßig gilt für beide Spezifikationsfälle, dass Terminierung erzwungen wird. In jedem Spezifikationsfall kann eine Nicht-Terminierung mit der Klausel diverges true; erlaubt werden, aber nur wenn die Vorbedingungen erfüllt sind.

JML erweitert Java-Modifikationen um weitere Modifikatoren für Klassenfelder, Methodenparameter und Methodenrückgabetypen. Diese können mit nullable (kann null oder nicht null sein) oder non_null (darf nicht null sein) versehen werden. Non-Null ist Standard in JML und wird für jedes Feld angenommen.

JML-Verträge, also Spezifikationsfälle und Klasseninvarianten werden von Oberklassen an Unterklassen vererbt, diesen können also mit also weitere Spezifikationsfälle hinzugefügt werden.

KeY

KeY ist ein Theorembeweiser für Prädikatenlogik und Dynamische Logik über Java.

Jan Schulz-Nigmann
14.09.2022

KeY benötigt als Eingabe die Spezifikation von Typen, Funktionen, Prädikaten und die zu beweisende Formel (Problemstellung).

Weiteres siehe VL 11.

Finden einer Schleifeninvariante

```
n >= 0 & n = m ==>
{i := 0}\[{
    while (i < n)
    {
        i = i+1;
    }
}\](i = m)
```

- Betrachte zuerst die Endbedingung: `(i = m)`
- Welche Bedingung, abgesehen vom negierten Guard (i >= n), wird benötigt um die Nachbedingung (i = m) zu gewährleisten? `(i = m)`
- Gilt `(i = m)` vor und nach Ausführung der Schleife? Leider Nein!

Betrachte Vorbedingungen und Updates im Schleifenrumpf:

- Wenn `(n = m)` gilt, reicht `(i <= n)` aus um zeigen, dass `(n = m & i <= n & i >= n) => (i = m)` gilt.