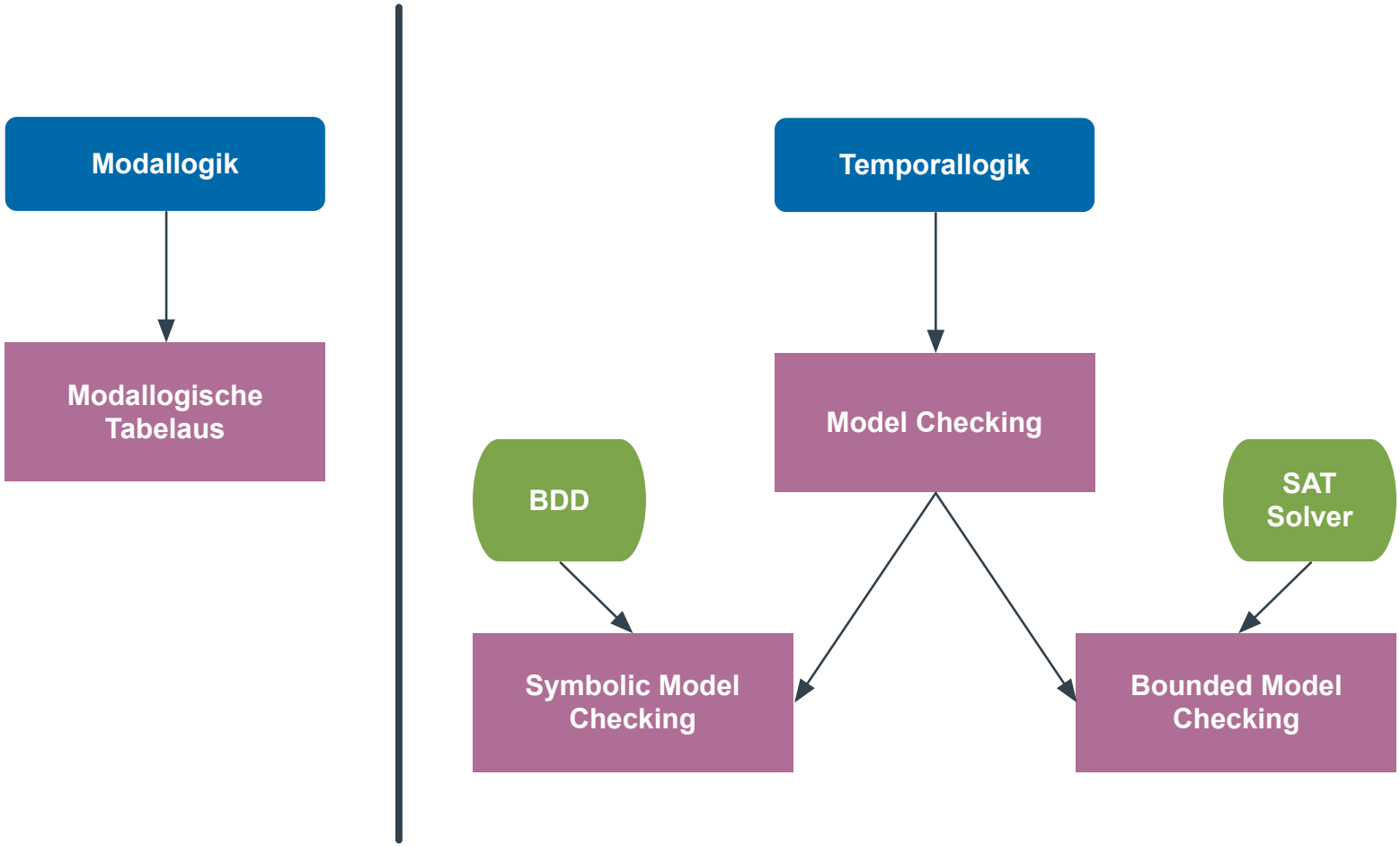
Automatisches Beweisen—Vertiefung
Symbolic & Bounded Model Checking

Christoph Zengler

Arbeitsbereich Symbolisches Rechnen
Prof. Dr. Wolfgang Küchlin
Universität Tübingen

17. Januar 2012

Der Plan für die nächsten drei Wochen



Motivation

Was bisher geschah...

- Modellierung eines Systems als Kripke Struktur
- CTL & LTL Model Checking auf dieser Struktur
- CTL: Labelling Algorithmus (Annotation der Zustände)
- LTL: Konstruktion eines Automaten für die Formel

⚠ Problem: State Explosion

Die Anzahl der Zustände eines Modells kann exponentiell in der Anzahl der Atomaren Aussagen sein.

Idee!

Repräsentiere die Kripke Struktur nicht explizit, sondern codiere sie in eine Boolesche Formel.

Zwei verschiedene Ansätze

Symbolic Model Checking für CTL

- Kam Anfang der 90er auf
- Idee: Repräsentiere die Kripke Struktur als Boolesche Formel in einem BDD
- Durchbruch für das Model Checking, sehr viel größere Systeme konnten verifiziert werden ($> 10^{20}$ Zustände)

Bounded Model Checking für LTL

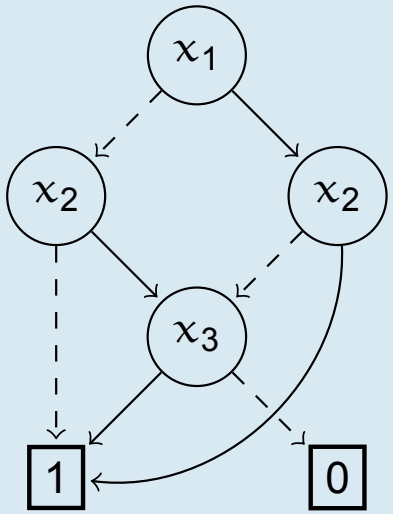
- Kam Ender der 90er auf
- Idee: Benutze einen SAT Solver zum Schlussfolgern über den Booleschen Formeln
- Wickle Modelle nur endlich oft ab

Kurze Wiederholung: BDD

- Knowledge Compilation Format (Arbeit steckt im Aufbauen der Datenstruktur)
- Datenstruktur DAG
  - Knoten sind Variablen
  - Blätter sind die Terminale $\boxed{0}$ (`false`) und $\boxed{1}$ (`true`)
  - Kanten sind Belegung der Variable mit `true` oder `false`

Beispiel (BDD)

Formel: $(x_1 \leftrightarrow x_2) \vee x_3$, Variablenordnung $x_1 < x_2 < x_3$



ROBDD

- Normalerweise betrachten wir immer ROBDD:
  - reduziert: keine gleichen Teilbäume im BDD, bei keinem Knoten ist der rechte gleich dem linken Teilbaum
  - geordnet: auf allen Pfaden gilt die gleiche Variablenreihenfolge
- Kanonische Normalform für aussagenlogische Formeln
  - $\Rightarrow$ Tautologie entspricht BDD $\boxed{1}$
  - $\Rightarrow$ Kontradiktion entspricht BDD $\boxed{0}$
  - $\Rightarrow$ Erfüllbare Formel entspricht BDD ungleich $\boxed{0}$
- Graphalgorithmen können benutzt werden
  - Finden einer erfüllenden Belegung $\Rightarrow$ ein Pfad zu $\boxed{1}$
  - Auflisten aller erfüllenden Belegungen $\Rightarrow$ alle Pfade zu $\boxed{1}$
  - ...

Ab jetzt: BDD = ROBDD

Algorithmen auf ROBDD

Werden hier nur überblicksweise behandelt, Details siehe in ABG.

- $\mathcal{B}(\varphi)$: ROBDD, dass die aussagenlogische Formel φ repräsentiert
- Jeder Knoten hat eine positive (high) und eine negative (low) Kante

Drei Algorithmen sind von Interesse

1. `apply`$(\mathcal{B}_1, \mathcal{B}_2, \circ)$: Kombiniert die beiden ROBDD $\mathcal{B}_1$ und $\mathcal{B}_2$ mit dem Booleschen Operator $\circ$, d.h

$$\mathcal{B}(\varphi_1 \circ \varphi_2) = \texttt{apply}(\mathcal{B}(\varphi_1), \mathcal{B}(\varphi_2), \circ)$$

2. `restrict`$(\mathcal{B}, v, b)$: Berechnet das ROBDD, bei dem die Variable v mit $b \in \{\bot, \top\}$ belegt ist (Restriktion), d.h.

$$\mathcal{B}(\varphi[b/v]) = \texttt{restrict}(\mathcal{B}(\varphi), v, b)$$

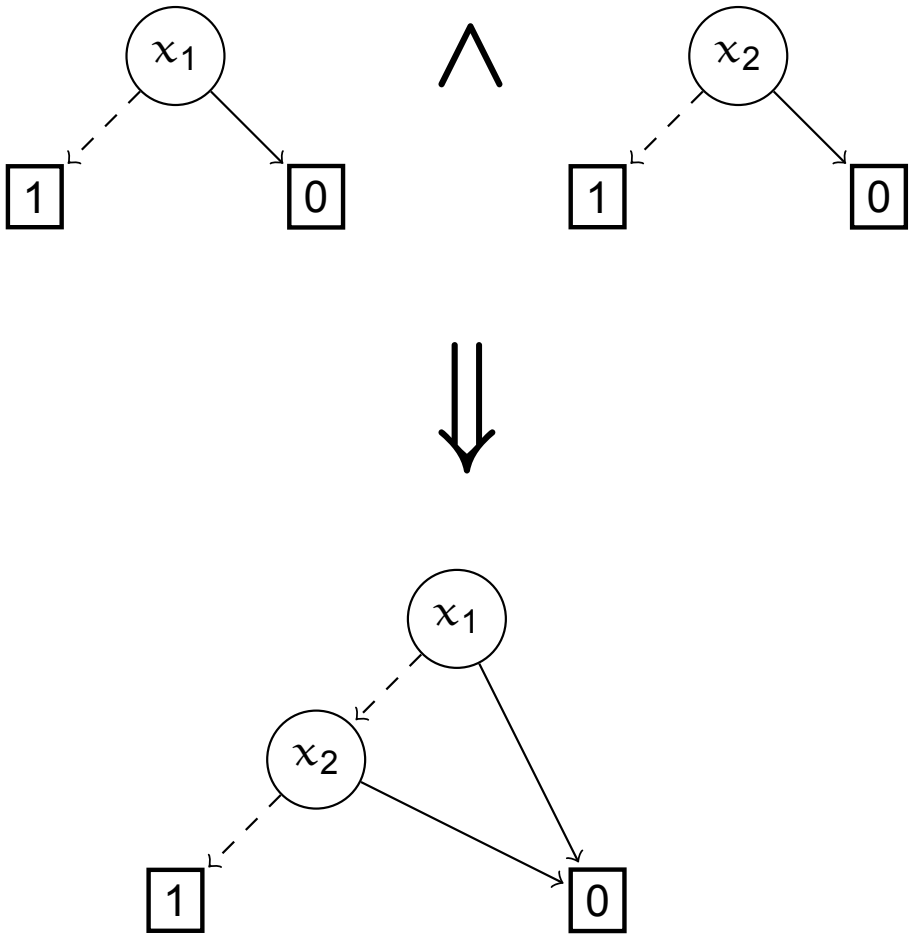
3. `exists`$(\mathcal{B}, v)$: Berechnet eine Projektion des ROBDD auf alle Variablen außer v, also eine existenzielle QE von v, d.h.

$$\mathcal{B}(\varphi[\top/v] \vee \varphi[\bot/v]) = \texttt{exists}(\mathcal{B}(\varphi), v)$$

Algorithmen auf ROBDD

Der `apply` Algorithmus

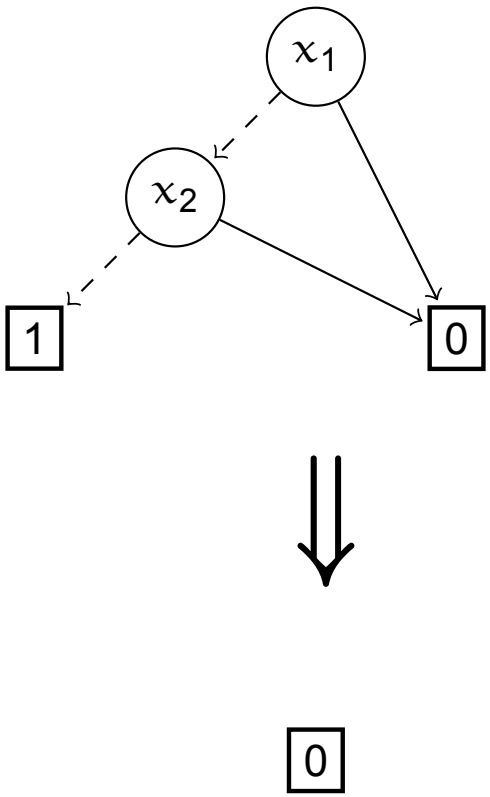
$$\mathcal{B}(\neg x_1 \wedge \neg x_2) = \texttt{apply}(\mathcal{B}(\neg x_1), \mathcal{B}(\neg x_2), \wedge)$$



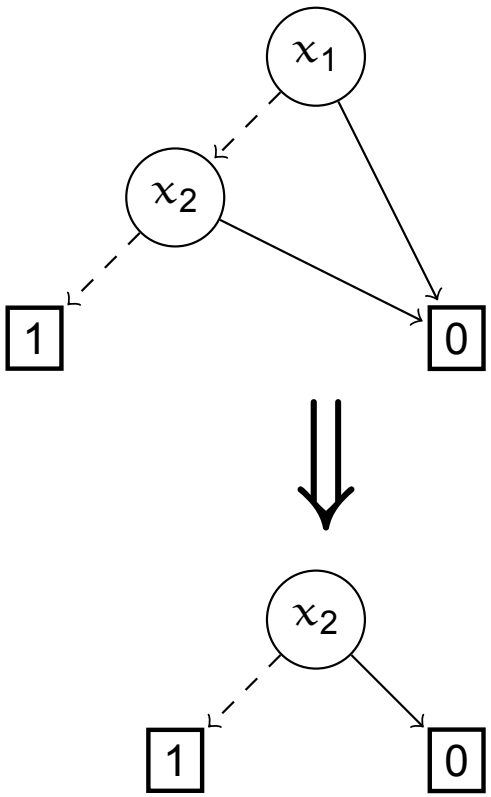
Algorithmen auf ROBDD

Der `restrict` Algorithmus

$$\mathcal{B}((\neg x_1 \wedge \neg x_2)[\top/x_1]) =$$
$$\texttt{restrict}(\mathcal{B}(\neg x_1 \wedge \neg x_2), x_1, \top)$$



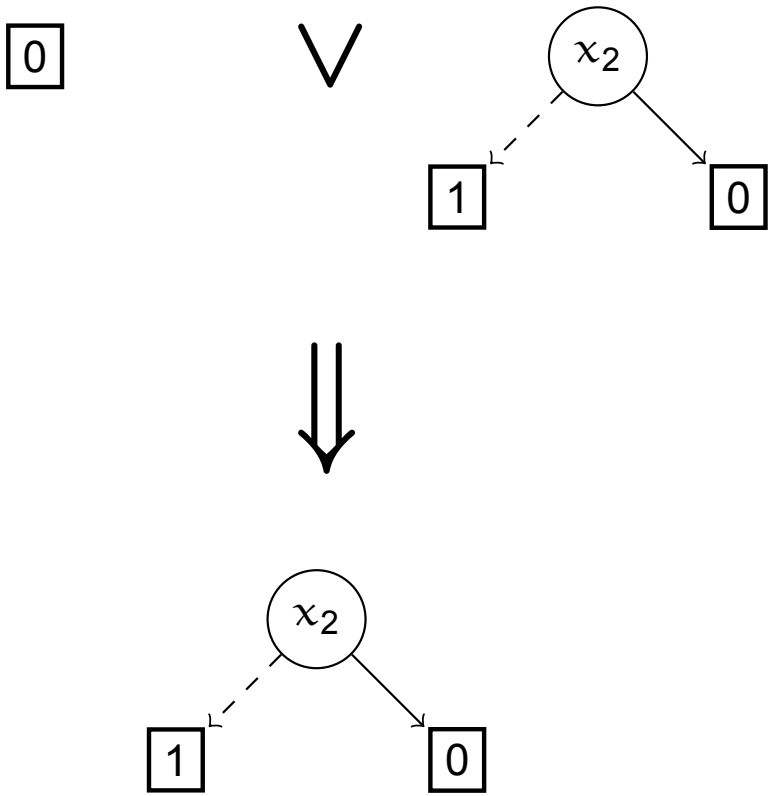
$$\mathcal{B}((\neg x_1 \wedge \neg x_2)[\bot/x_1]) =$$
$$\texttt{restrict}(\mathcal{B}(\neg x_1 \wedge \neg x_2), x_1, \bot)$$



Algorithmen auf ROBDD

Der `exists` Algorithmus

$$\mathcal{B}((\neg x_1 \wedge \neg x_2)[\top/x_1] \vee (\neg x_1 \wedge \neg x_2)[\bot/x_1]) = \texttt{exists}(\mathcal{B}(\neg x_1 \wedge \neg x_2), x_1)$$



Erweiterung: Elimination mehrerer Variablen $X = \{x_0, \ldots, x_n\}$ mit

$$\texttt{exists}(\mathcal{B}, X)$$

Grundidee des Algorithmus

1. Konstruiere ein BDD, das die Menge aller Zustände der Kripke-Struktur beschreibt
2. Konstruiere ein BDD, das die Zustandsübergänge beschreibt
3. Kombiniere die beiden BDD zu einem BDD, dass die gesamte Kripke-Struktur beschreibt
4. Führe den Algorithmus `modelCheckCTL` nun statt auf der Kripke-Struktur auf dem BDD aus

Definition (Mengenoperationen)

Beschreiben die beiden Formeln φ_1 und φ_2 jeweils die Zustandsmengen S_1 und S_2, so können wir die Mengenoperationen wie folgt realisieren

- $S_1 \cup S_2$ entspricht $\varphi_1 \vee \varphi_2$
- $S_1 \cap S_2$ entspricht $\varphi_1 \wedge \varphi_2$
- $S_1 - S_2$ entspricht $\varphi_1 \wedge \neg\varphi_2$

Darstellung von Zustandsmengen

Übersetzen eines einzelnen Zustandes s

Konjunktion aller Label, die an dem Zustand vorkommen (positive Literale) und die nicht vorkommen (negative Literale):

$$\mathcal{T}_S(s) = \bigwedge_{l \in \mathcal{L}} \lambda(l) \text{ mit } \lambda(l) = l, \text{ falls } l \in L(s), \text{ sonst } \neg l$$

Übersetzen einer Zustandsmenge

Für eine Zustandsmenge $\{s_0, \ldots, s_n\}$: Disjunktion über alle $\mathcal{T}_S(s_i)$:

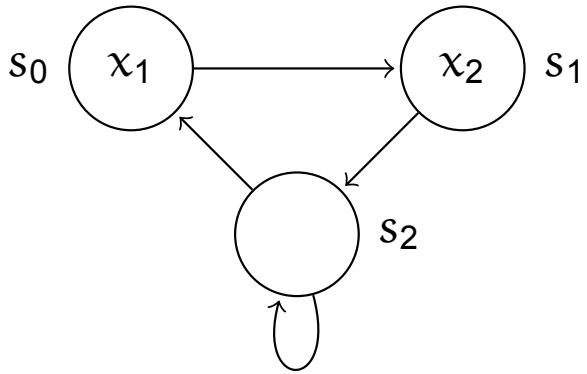
$$\mathcal{T}_S(\{s_0, \ldots, s_n\}) = \bigvee_{i \in \{0,\ldots,n\}} \mathcal{T}_S(s_i)$$

Hinweis

Zustände werden eindeutig über ihre Labelmengen identifiziert. Gibt es mehrere Zustände mit gleichen Beschriftungen, so müssen diese durch das Einführen von Hilfsvariablen unterschieden werden.

Darstellung von Zustandsmengen

Beispiel



Beispiel (Zustandsmengen als Boolesche Formel)

Menge der Zustände M	$\mathcal{T}_S(M)$
$\varnothing$	$\bot$
$\{s_0\}$	$x_1 \wedge \neg x_2$
$\{s_1\}$	$\neg x_1 \wedge x_2$
$\{s_2\}$	$\neg x_1 \wedge \neg x_2$
$\{s_0, s_1\}$	$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$
$\{s_0, s_2\}$	$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge \neg x_2)$
$\{s_1, s_2\}$	$(\neg x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_2)$
S	$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_2)$

Darstellung von Zustandsübergängen

- Führe eine Kopie x' jeder Variable x ein
- x ist die Belegung der Variable *vor* dem Übergang
- x' ist die Belegung der Variable *nach* dem Übergang
- Wir bezeichnen die Kopie eines Zustandes s mit s'

Übersetzen eines Zustandsübergangs

Der Übergang $s_0 \to s_1$ wird dann dargestellt als:

$$\mathcal{T}_T(s_0 \to s_1) = \mathcal{T}_S(s_0) \wedge \mathcal{T}_S(s_1')$$

Übersetzen einer Menge an Zustandsübergängen

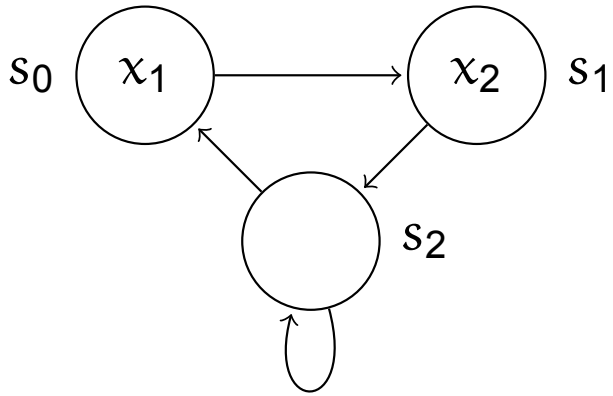
Für Zustandsübergänge $t_0, \ldots, t_n$ ist die Übersetzung

$$\mathcal{T}_T(\{t_0, \ldots, t_n\}) = \bigvee_{i \in \{0,\ldots,n\}} \mathcal{T}_T(t_i)$$

Übersetzung alle Zustandsübergänge: $\mathcal{T}_T(\longrightarrow)$

Darstellung von Zustandsübergängen

Beispiel



Beispiel (Zustandsübergänge als Formel)

$(x_1 \wedge \neg x_2 \wedge \neg x_1' \wedge x_2') \vee$	$s_0 \rightarrow s_1$
$(\neg x_1 \wedge x_2 \wedge \neg x_1' \wedge \neg x_2') \vee$	$s_1 \rightarrow s_2$
$(\neg x_1 \wedge \neg x_2 \wedge \neg x_1' \wedge \neg x_2') \vee$	$s_2 \rightarrow s_2$
$(\neg x_1 \wedge \neg x_2 \wedge x_1' \wedge \neg x_2')$	$s_2 \rightarrow s_0$

Implementierung der Urbilder

Das Universelle Urbild $\texttt{pre}_\forall$

$$\texttt{pre}_\forall(Y) = S - \texttt{pre}_\exists(S - Y)$$

Das Existenzielle Urbild $\texttt{pre}_\exists$

Berechnung von $\texttt{pre}_\exists(Y)$

- Wir suchen das Urbild, d.h. Vorgänger in der Übergangsrelation

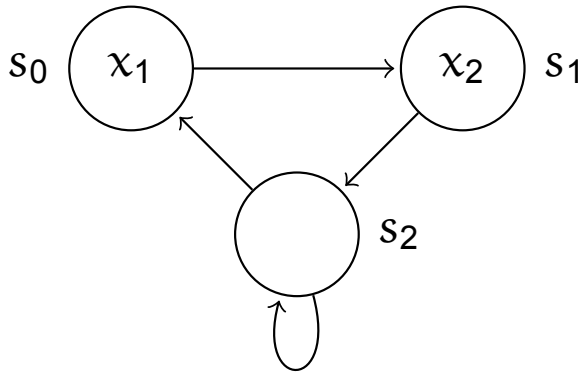
$\Rightarrow$ die Variablen in Y werden durch ihre Kopien ersetzt (Y')

- Bilde die Formel für $\mathcal{T}_T(\longrightarrow) \wedge \mathcal{T}_S(Y')$
- Eliminiere alle Kopien der Variablen aus der Formel

$$\texttt{exists}(\texttt{apply}(\mathcal{B}(\mathcal{T}_T(\longrightarrow)), \mathcal{B}(\mathcal{T}_S(Y')), \wedge), \text{vars}(Y'))$$

Implementierung der Urbilder

Beispiel



Beispiel (Berechnung von $\mathrm{pre}_\exists(\{s_2\})$)

$$\begin{aligned}\mathcal{T}_T(\longrightarrow) \wedge \mathcal{T}_S(Y') &= [(x_1 \wedge \neg x_2 \wedge \neg x_1' \wedge x_2') \vee (\neg x_1 \wedge x_2 \wedge \neg x_1' \wedge \neg x_2') \vee \\ &\quad (\neg x_1 \wedge \neg x_2 \wedge \neg x_1' \wedge \neg x_2') \vee (\neg x_1 \wedge \neg x_2 \wedge x_1' \wedge \neg x_2')] \wedge \\ &\quad \neg x_1' \wedge \neg x_2' \\ &= [(\neg x_1 \wedge x_2 \wedge \neg x_1' \wedge \neg x_2') \vee (\neg x_1 \wedge \neg x_2 \wedge \neg x_1' \wedge \neg x_2')] \wedge \\ &\quad \neg x_1' \wedge \neg x_2'\end{aligned}$$

- Elimination von x_1': $[(\neg x_1 \wedge x_2 \wedge \neg x_2') \vee (\neg x_1 \wedge \neg x_2 \wedge \neg x_2')] \wedge \neg x_2'$
- Elimination von x_2': $(\neg x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_2)$ (*beschreibt* $\{s_1, s_2\}$)

Der Algorithmus

Wir wissen, wie wir Zustände, $\cup, \cap, -, \texttt{pre}_\forall$ und $\texttt{pre}_\exists$ mit BDD implementieren können, d.h. unveränderter Algorithmus

Algorithmus: $\texttt{modelCheckCTL}(\mathcal{M}, \psi)$

Eingabe: Kripke-Struktur $\mathcal{M}$ und CTL Formel ψ
Ausgabe: Menge an Zuständen $\subseteq S$, an denen ψ gilt

1. Reduziere ψ auf die Operatoren $\bot$, $\neg$, $\wedge$ $\mathbf{AF}$, $\mathbf{EU}$ und $\mathbf{EX}$
2. $\texttt{modelCheckCTL}(\mathcal{M}, \psi) = \psi$ *match*

$$
\begin{aligned}
&\bot && \rightsquigarrow \varnothing \\
\mid\ &v \in \mathcal{V} && \rightsquigarrow \{s \in S \mid v \in L(s)\} \\
\mid\ &\neg\varphi && \rightsquigarrow S - \texttt{modelCheckCTL}(\mathcal{M}, \varphi) \\
\mid\ &\varphi_1 \wedge \varphi_2 && \rightsquigarrow \texttt{modelCheckCTL}(\mathcal{M}, \varphi_1) \cap \texttt{modelCheckCTL}(\mathcal{M}, \varphi_2) \\
\mid\ &\mathbf{EX}\,\varphi && \rightsquigarrow \texttt{mc}_{\mathbf{EX}}(\mathcal{M}, \varphi) \\
\mid\ &\mathbf{AF}\,\varphi && \rightsquigarrow \texttt{mc}_{\mathbf{AF}}(\mathcal{M}, \varphi) \\
\mid\ &\mathbf{E}[\varphi_1\,\mathbf{U}\,\varphi_2] && \rightsquigarrow \texttt{mc}_{\mathbf{EU}}(\mathcal{M}, \varphi_1, \varphi_2)
\end{aligned}
$$

Komplexität

Komplexität wichtiger BDD Operationen

- Äquivalenztest auf zwei BDD (Fixpunkt finden): $\mathrm{O}(|\mathcal{B}|)$
- Reduzieren eines BDD: $\mathrm{O}(|\mathcal{B}| \cdot \log |\mathcal{B}|)$
- `apply`$(\mathcal{B}_1, \mathcal{B}_2, \circ)$: $\mathrm{O}(|\mathcal{B}_1| \cdot |\mathcal{B}_2|)$
- `restrict`$(\mathcal{B}, v, \mathrm{b})$: $\mathrm{O}(|\mathcal{B}| \cdot \log |\mathcal{B}|)$
- `exists`$(\mathcal{B}, X)$: NP-vollständig

Existentielle Quantorenelimination auf Booleschen Formeln

Aktuelle Forschung am Arbeitsbereich:

- DNNF kompilieren & projizieren
- Projizierte Model Enumeration
- Portfolio Ansatz

Idee des Bounded Model Checking

Motivation:

- Erfolg von SAT Solvern auf Model Checking übertragen
- Anleihen an Planungsproblemen

Wir wissen:

- LTL Model Checking arbeitet auf Pfaden
- Implizite Allquantifizierung über alle Pfade
- Finde einen Pfad, der die Bedingung verletzt

Neue Idee:

- Untersuche nur Pfade der Länge n
- Sukzessive Erhöhung der Länge (Bound)
- vgl. Abstraction Refinement Loop

Grundlagen

Definition (Pfad)

Ein Pfad in einem Modell $\mathcal{M} = (S, \longrightarrow, L)$ ist eine unendliche Folge von Zuständen $s_0, s_1, s_2, \ldots$ in S, so dass für jedes $i \geq 0$ $s_i \longrightarrow s_{i+1}$.

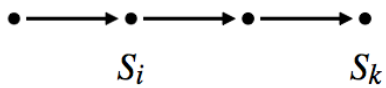
- π^i: Suffix des Pfades, beginnend bei dem i-ten Zustand

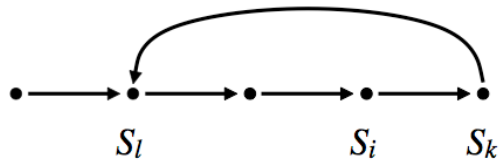
Unendliche Pfade

- Auch bei endlichem Präfix kann ein Pfad unendlich lang sein

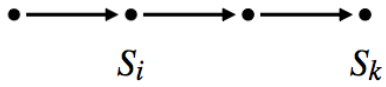
$\Rightarrow$ Existenz von Rücksprüngen (Back Loops)

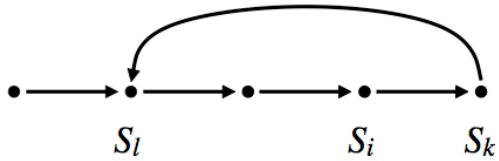
- Existieren keine Rücksprünge, so kann über das Verhalten des Pfades nach dem Präfix nichts gesagt werden





Schleifen





Definition (Schleifen)

Für $l \leq k$ ist ein Pfad $\pi = s_0, s_1, \ldots$ eine (k, l)-Schleife, wenn

- $s_k \rightarrow s_l$
- $\pi = s_0, s_1, \ldots, s_{l-1}, (s_l, \ldots, s_k)^*$

π ist eine k-Schleife, wenn es ein $k \geq l \geq 0$ gibt, für dass π eine (k, l)-Schleife ist

Idee: Bounded Semantics

- Wir betrachten nur endliche Präfixe von Pfaden
- d.h. nur die ersten $k + 1$ Zustände $(s_0, \ldots, s_k)$
- Werte LTL Formeln entlang eines solchen begrenzten Pfad aus
- Ist der Pfad eine k-Schleife, so ändert sich nichts an der Semantik

Bounded Semantics mit Schleifen

Definition (Bounded Semantics mit Schleife)

Sei

- $\mathcal{M}$ eine Kripke-Struktur, $k \geq 0$ und
- π eine k-Schleife.

Dann ist eine LTL Formel φ gültig entlang des Pfades π mit Grenze k ($\pi \models_k \varphi$), wenn $\texttt{holds}(\mathcal{M}, \pi, \varphi) = \texttt{true}$.

☜ Problem!

In der üblichen (unbounded) Semantik von LTL Model Checking gilt:

- $\mathbf{F}\, p$ ist auf einem Pfad gültig, wenn wir ein Suffix finden, dass mit p beginnt

Aber

- In der Bounded Semantics von LTL Model Checking hat der Zustand s_k keinen Nachfolger mehr

Bounded Semantics ohne Schleifen

Algorithmus: $\texttt{holdsBound}(\mathcal{M}, \pi, k, i, \psi)$

Eingabe: Modell $\mathcal{M} = (S, \longrightarrow, L)$, Pfad $\pi = s_0 \longrightarrow \ldots$, der keine k-Schleife ist, Bound k, aktuelle Position im Pfad i, LTL Formel ψ
Ausgabe: Evaluation von ψ auf dem Pfad π mit k und i

$\texttt{holdsBound}(\mathcal{M}, \pi, k, i, \psi) = \psi$ *match*

$\top \rightsquigarrow$ true

$\mid\ \nu \in \mathcal{V} \rightsquigarrow$ *if* $\nu \in L(s_i)$ *then* true *else* false

$\mid\ \neg\varphi \rightsquigarrow$ *if* $\texttt{holdsBound}(\mathcal{M}, \pi, k, i, \varphi)$ *then* false *else* true

$\mid\ \varphi_1 \wedge \varphi_2 \rightsquigarrow \texttt{holdsBound}(\mathcal{M}, \pi, k, i, \varphi_1)$ *and* $\texttt{holdsBound}(\mathcal{M}, \pi, k, i, \varphi_2)$

$\mid\ \mathbf{G}\,\varphi \rightsquigarrow$ false

$\mid\ \mathbf{F}\,\varphi \rightsquigarrow$ *exists* $i \le j \le k$: $\texttt{holdsBound}(\mathcal{M}, \pi, k, j, \varphi)$

$\mid\ \mathbf{X}\,\varphi \rightsquigarrow i < k$ *and* $\texttt{holdsBound}(\mathcal{M}, \pi, k, i+1, \varphi)$

$\mid\ \varphi_1\,\mathbf{U}\,\varphi_2 \rightsquigarrow$ *exists* $i \le j \le k$: $\texttt{holdsBound}(\mathcal{M}, \pi, k, j, \varphi_2)$ *and*
for all $i \le n < j$: $\texttt{holdsBound}(\mathcal{M}, \pi, k, n, \varphi_1)$

$\mid\ \varphi_1\,\mathbf{R}\,\varphi_2 \rightsquigarrow$ *exists* $i \le j \le k$: $\texttt{holdsBound}(\mathcal{M}, \pi, k, j, \varphi_1)$ *and*
for all $i \le n \le j$: $\texttt{holdsBound}(\mathcal{M}, \pi, k, n, \varphi_2)$

- Wenn π eine k-Schleife ist, so wird einfach $\texttt{holds}(\mathcal{M}, \pi^i, \psi)$ aufgerufen

Das Bounded Model Checking Problem

Definition (Bounded Semantic ohne Schleife)

Sei $k \geq 0$ und π ein Pfad, der keine k-Schleife ist. Dann ist eine LTL Formel φ gültig entlang π mit Bound k ($\pi \models_k \varphi$), wenn $\texttt{holdsBound}(\mathcal{M}, \pi, k, 0, \varphi)$.

- Üblicherweise suchen wir ein Gegenbeispiel, d.h. *existiert ein Pfad, der die Bedingung verletzt*

Lemma

Für eine LTL Formel φ und einen Pfad π gilt $\pi \models_k \varphi$ impliziert $\pi \models \varphi$

Lemma

Existiert ein Pfad π, für den $\pi \models \varphi$ gilt, so existiert auch ein Pfad π', so dass für ein $k \geq 0$ gilt $\pi' \models_k \varphi$.

Theorem (Bounded Model Checking)

Ein Pfad (Gegenbeispiel) π mit $\pi \models \varphi$ existiert gdw. ein $k \geq 0$ existiert mit $\pi \models_k \varphi$, d.h. $\texttt{holdsBound}(\mathcal{M}, \pi, k, 0, \varphi)$.

Grundidee

- Generiere eine Aussagenlogische Formel aus den drei Teilen
  1. Kripke-Struktur $\mathcal{M}$
  2. Bound k
  3. LTL Formel φ

  $\texttt{bmc2sat}(\mathcal{M}, k, \varphi)$
- Diese Formel codiert Constraints an einen Pfad $\pi = s_0 \rightarrow \ldots$, der ein Zeuge für φ ist
- s_i steht für Zustand s im Zeitschritt i
- D.h. ist $\texttt{bmc2sat}(\mathcal{M}, k, \varphi)$ erfüllbar, codiert diese erfüllende Belegung ein **Gegenbeispiel** für die Verifikationsbedingung.
- Ist $\texttt{bmc2sat}(\mathcal{M}, k, \varphi)$ unerfüllbar, so ist die Verifikationsbedingung für alle Pfade der Länge k **wahr**.

Bestandteile von $\texttt{bmc2sat}(\mathcal{M}, k, \varphi)$

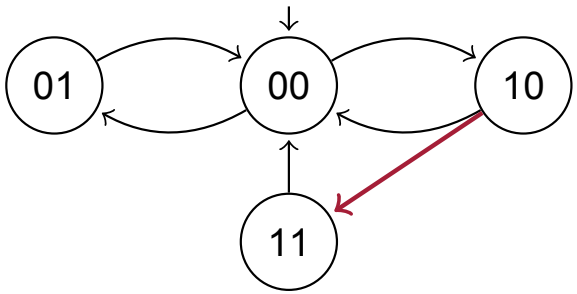
1. `trans2sat`: Constraints, dass $\pi = s_0, \ldots, s_k$ ein gültiger Pfad ist
2. `loopCondition`: Schleifenbedingung; ist wahr, wenn π eine Schleife enthält
3. `formLoop`, `formNoLoop`: Constraints, dass π die Formel φ erfüllen muss (mit und ohne Schleife)

Kopieren der Übergangsrelation

- Die Übergangsfunktion muss bei einer Bound von k auch k mal kopiert werden

$\Rightarrow$ Es ist sinnvoll sich eine *generische* Version der Übergangsrelation zu speichern, die dann nur noch mit den entsprechenden Zeitschritten parametrisiert wird

- Realisierung wie $\mathcal{T}_T$
- Bezeichnung $T(s, s')$



- Zustand: $s[1]s[0]$
- $T(s, s') = (\neg s[1] \wedge (s'[0] \leftrightarrow \neg s'[0])) \vee (\neg s[0] \wedge (s'[1] \leftrightarrow \neg s'[1])) \vee (s[0] \wedge s[1] \wedge \neg s'[0] \wedge \neg s'[1]) \vee (s[1] \wedge \neg s[0] \wedge s'[0] \wedge s'[1])$

Übersetzung der Übergangsrelation

Algorithmus: $\texttt{trans2sat}(\mathcal{M}, k)$

Eingabe: Kripke-Struktur $\mathcal{M} = (S, \longrightarrow, L)$, Bound k
Ausgabe: Aussagenlogische Formel, die $\longrightarrow$ beschreibt

$$\mathcal{T}_S(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$$

Beispiel (Übersetzen der Übergangsrelation)

- $\texttt{trans2sat}(\mathcal{M}, 0) = \mathcal{T}_S(s_0)$
- $\texttt{trans2sat}(\mathcal{M}, 1) = \mathcal{T}_S(s_0) \wedge T(s_0, s_1)$
- $\texttt{trans2sat}(\mathcal{M}, 2) = \mathcal{T}_S(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2)$
- $\texttt{trans2sat}(\mathcal{M}, 3) = \mathcal{T}_S(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge T(s_2, s_3)$

Übersetzung der Schleifenbedingung

- Übersetzung der Formel hängt davon ab, ob der Pfad π eine k-Schleife ist oder nicht.
- Codierung der Schleifenbedingung: Gibt es einen Rücksprung von s_k zu einen früheren Zustand

Algorithmus: $\texttt{loopCondition}(\mathcal{M}, k)$

Eingabe: Kripke-Struktur $\mathcal{M} = (S, \longrightarrow, L)$, Bound k
Ausgabe: Aussagenlogische Formel, die `true` ist, gdw. es einen Rücksprung vom Zustand s_k gibt

$$\bigvee_{l=0}^{k} T(s_k, s_l)$$

Beispiel (Übersetzen der Schleifenbedingung)

- $\texttt{loopCondition}(\mathcal{M}, 0) = T(s_0, s_0)$
- $\texttt{loopCondition}(\mathcal{M}, 1) = T(s_1, s_0) \vee \mathcal{T}_T(s_1, s_1)$
- $\texttt{loopCondition}(\mathcal{M}, 2) = T(s_2, s_0) \vee T(s_2, s_1) \vee T(s_2, s_2)$

Übersetzung von Formeln (Mit Schleifen)

Definition (Nachfolger in einer Schleife)

Seien $l, i \leq k$. Dann ist $\mathtt{succ}(i)$ für ein i in einer (k, l)-Schleife definiert als

$$\mathtt{succ}(i) = \begin{cases} i+1 & \text{für } i < k \\ l & \text{für } i = k \end{cases}$$

Algorithmus: $\mathtt{formLoop}(\psi, k, l, i)$

Eingabe: LTL Formel ψ, Bound k, Nr. des Rücksprung Zustands l, Nr. des aktuellen Zustands i

Ausgabe: Aussagenlogische Formel, die sicherstellt, dass ein Pfad π, der eine (k, l)-Schleife ist, ψ erfüllt

---

$\mathtt{formLoop}(\psi, k, l, i) = \psi$ *match*

$\quad \top \rightsquigarrow \top$

$\quad \mid v \in \mathcal{V} \rightsquigarrow v(s_i)$ *(Indizierung von v mit dem Zeitschritt i)*

$\quad \mid \neg\varphi \rightsquigarrow \neg\mathtt{formLoop}(\varphi, k, l, i)$

$\quad \mid \varphi_1 \wedge \varphi_2 \rightsquigarrow \mathtt{formLoop}(\varphi_1, k, l, i) \wedge \mathtt{formLoop}(\varphi_2, k, l, i)$

$\quad \mid \ldots$

Übersetzung von Formeln (Mit Schleifen)

Algorithmus: $\texttt{formLoop}(\psi, k, l, i)$

$$
\begin{aligned}
&| \ \ldots \\
&| \ \mathbf{G}\,\varphi \ \rightsquigarrow\ \texttt{formLoop}(\varphi, k, l, i) \wedge \texttt{formLoop}(\mathbf{G}\,\varphi, k, l, \texttt{succ}(i)) \\
&| \ \mathbf{F}\,\varphi \ \rightsquigarrow\ \texttt{formLoop}(\varphi, k, l, i) \vee \texttt{formLoop}(\mathbf{F}\,\varphi, k, l, \texttt{succ}(i)) \\
&| \ \mathbf{X}\,\varphi \ \rightsquigarrow\ \texttt{formLoop}(\varphi, k, l, \texttt{succ}(i)) \\
&| \ \varphi_1\,\mathbf{U}\,\varphi_2 \ \rightsquigarrow\ \texttt{formLoop}(\varphi_2, k, l, i) \vee \\
&\qquad\qquad (\texttt{formLoop}(\varphi_1, k, l, i) \wedge \texttt{formLoop}(\varphi_1\,\mathbf{U}\,\varphi_2, k, l, \texttt{succ}(i))) \\
&| \ \varphi_1\,\mathbf{R}\,\varphi_2 \ \rightsquigarrow\ \texttt{formLoop}(\varphi_2, k, l, i) \wedge \\
&\qquad\qquad (\texttt{formLoop}(\varphi_1, k, l, i) \vee \texttt{formLoop}(\varphi_1\,\mathbf{R}\,\varphi_2, k, l, \texttt{succ}(i)))
\end{aligned}
$$

- Speichert man die Subterme eindeutig (z.B. durch das Einführen neuer Variablen für Subterme), so ist die entstehende Formel linear in der Größe von ψ und k

Übersetzung von Formeln (Ohne Schleifen)

Algorithmus: $\texttt{formNoLoop}(\psi, k, i)$

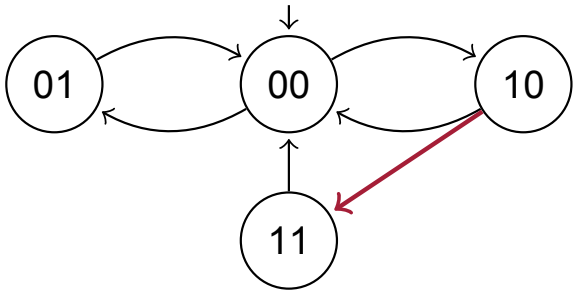
Eingabe: LTL Formel ψ, Bound k, Nr. des aktuellen Zustands i
Ausgabe: Aussagenlogische Formel, die sicherstellt, dass ein Pfad π, der **keine** k-Schleife ist, ψ erfüllt

---

if $i > k$ *then return* $\bot$ *else* ψ *match*

$$
\begin{aligned}
&\top \rightsquigarrow \top \\
&\mid \nu \in \mathcal{V} \rightsquigarrow \nu(s_i) \text{ (Indizierung von } \nu \text{ mit dem Zeitschritt } i) \\
&\mid \neg\varphi \rightsquigarrow \neg\texttt{formNoLoop}(\varphi, k, i) \\
&\mid \varphi_1 \wedge \varphi_2 \rightsquigarrow \texttt{formNoLoop}(\varphi_1, k, i) \wedge \texttt{formNoLoop}(\varphi_2, k, i) \\
&\mid \mathbf{G}\,\varphi \rightsquigarrow \texttt{formNoLoop}(\varphi, k, i) \wedge \texttt{formNoLoop}(\mathbf{G}\,\varphi, k, i+1) \\
&\mid \mathbf{F}\,\varphi \rightsquigarrow \texttt{formNoLoop}(\varphi, k, i) \vee \texttt{formNoLoop}(\mathbf{F}\,\varphi, k, i+1) \\
&\mid \mathbf{X}\,\varphi \rightsquigarrow \texttt{formNoLoop}(\varphi, k, i+1) \\
&\mid \varphi_1\,\mathbf{U}\,\varphi_2 \rightsquigarrow \texttt{formNoLoop}(\varphi_2, k, i) \vee \\
&\qquad (\texttt{formNoLoop}(\varphi_1, k, i) \wedge \texttt{formNoLoop}(\varphi_1\,\mathbf{U}\,\varphi_2, k, i+1)) \\
&\mid \varphi_1\,\mathbf{R}\,\varphi_2 \rightsquigarrow \texttt{formNoLoop}(\varphi_2, k, i) \wedge \\
&\qquad (\texttt{formNoLoop}(\varphi_1, k, i) \vee \texttt{formNoLoop}(\varphi_1\,\mathbf{R}\,\varphi_2, k, i+1))
\end{aligned}
$$

Übersetzung von Formeln (Beispiele)



Wir möchten verifizieren, dass wir nie in den Zustand 11 kommen, d.h. $\mathbf{G}\,\neg(s[1] \wedge s[0])$, d.h. wir suchen einen Pfad für $\varphi = \mathbf{F}(s[1] \wedge s[0])$

Beispiel (Ohne Schleifen)

- $\texttt{formNoLoop}(\varphi, 2, 0) = (s_0[1] \wedge s_0[0]) \vee \texttt{formNoLoop}(\varphi, 2, 1)$
- $\texttt{formNoLoop}(\varphi, 2, 1) = (s_1[1] \wedge s_1[0]) \vee \texttt{formNoLoop}(\varphi, 2, 2)$
- $\texttt{formNoLoop}(\varphi, 2, 2) = (s_2[1] \wedge s_2[0]) \vee \texttt{formNoLoop}(\varphi, 2, 3)$
- $\texttt{formNoLoop}(\varphi, 2, 3) = \bot$

$\Rightarrow \texttt{formNoLoop}(\varphi, 2, 0) = (s_0[1] \wedge s_0[0]) \vee (s_1[1] \wedge s_1[0]) \vee (s_2[1] \wedge s_2[0])$

- Übersetzung für Version mit Schleifen identisch

Übersetzung des gesamten Systems

Algorithmus: $\texttt{bmc2sat}(\mathcal{M}, k, \varphi)$

Eingabe: Kripke-Struktur $\mathcal{M}$, Bound k, LTL Formel φ
Ausgabe: Aussagenlogische Formel, die `true` ist, wenn eine Pfad π mit Länge k existiert, der φ erfüllt

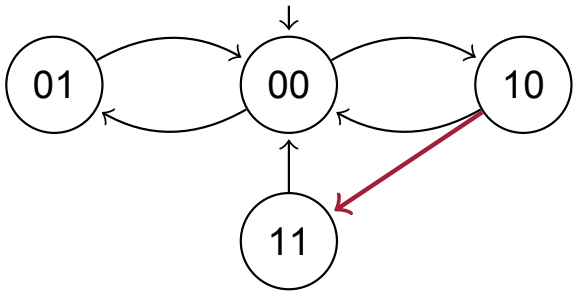
---

$$\texttt{trans2sat}(\mathcal{M}, k) \wedge$$
$$[(\neg\texttt{loopCondition}(\mathcal{M}, k) \wedge \texttt{formNoLoop}(\varphi, k, 0)) \vee$$
$$\bigvee_{l=0}^{k} (T(s_k, s_l) \wedge \texttt{formLoop}(\varphi, k, l, 0))]$$

Erklärung

- $\texttt{trans2sat}(\mathcal{M}, k)$: k-faches Abwickeln der Übergangsrelation
- $\neg\texttt{loopCondition}(\mathcal{M}, k) \wedge \texttt{formNoLoop}(\varphi, k, 0)$: Der Pfad ist keine k-Schleife, also benutze die Codierung der Formel ohne Schleife
- $\bigvee_{l=0}^{k}(T(s_k, s_l) \wedge \texttt{formLoop}(\varphi, k, l, 0))$: Für ein (oder mehrere) l ist der Pfad eine (k, l)-Schleife; für dieses l muss dann auch die Codierung der Formel mit Schleife gelten

Finales Beispiel mit Bound $k = 2$



Suche einen Pfad, auf dem $\mathbf{F}(s[0] \wedge s[1])$ gilt

- $T(s, s') = (\neg s[1] \wedge (s'[0] \leftrightarrow \neg s'[0])) \vee (\neg s[0] \wedge (s'[1] \leftrightarrow \neg s'[1])) \vee (s[0] \wedge s[1] \wedge \neg s'[0] \wedge \neg s'[1]) \vee (s[1] \wedge \neg s[0] \wedge s'[0] \wedge s'[1])$
- $\texttt{trans2sat}(\mathcal{M}, 2) = \mathcal{T}_S(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2)$
- $\texttt{loopCondition}(\mathcal{M}, 2) = T(s_2, s_0) \vee T(s_2, s_1) \vee T(s_2, s_2)$
- $\texttt{formNoLoop}(\varphi, 2, 0) = (s_0[1] \wedge s_0[0]) \vee (s_1[1] \wedge s_1[0]) \vee (s_2[1] \wedge s_2[0])$
- $\texttt{formLoop}(\varphi, 2, 2, 0) = (s_0[1] \wedge s_0[0]) \vee (s_1[1] \wedge s_1[0]) \vee (s_2[1] \wedge s_2[0])$
- $\texttt{bmc2sat}(\mathcal{M}, k, \varphi) =$ $\texttt{trans2sat}(\mathcal{M}, k) \wedge [(\neg \texttt{loopCondition}(\mathcal{M}, k) \wedge \texttt{formNoLoop}(\varphi, k, 0)) \vee \bigvee_{l=0}^{k} (T(s_k, s_l) \wedge \texttt{formLoop}(\varphi, k, l, 0))]$

$\texttt{bmc2sat}(\mathcal{M}, k, \varphi)$ hat eine erfüllende Belegung:

$$\{\neg s_0[1], \neg s_0[0], s_1[1], \neg s_1[0], s_2[1], s_2[0]\}$$

Anmerkungen

Vollständigkeit

- Bounded Model Checking ist offensichtlich nicht vollständig
- Es gibt Techniken, für bestimmte Fälle eine maximale Grenze k zu bestimmen (z.B. Programmverifikation: obere Grenze einer Schleife ist bekannt)
- Prädestiniert zum Suchen von Fehlern, nicht zum Beweisen der Abwesenheit von Fehlern

Erweiterungen

- Das Benutzen von QBF-Solvern anstelle von SAT-Solvern erspart das k-fache Kopieren der Übergangsrelation
- Es gibt auch SMT-basierte Bounded Model Checker

Implementierungen

- Implementierung für das Software Bounded Model Checking: `cbmc` zum Verifizieren von C-Programmen

Literaturtipps

Literaturhinweis

- *M. Huth & M. Ryan.* **Logic in Computer Science** **Section 6.3.** Cambridge University Press, 2004.
- *E. M. Clarke, O. Grumberg & D. A. Peled.* **Model Checking.** The MIT Press, 1999.

- *A. Biere et al.* **Bounded Model Checking.** Highly Dependable Software, Academic Press, 2003.
- *A. Armando, J. Mantovani & L. Platania.* **Bounded Model Checking of Software using SMT Solvers Instead of SAT Solvers.** International Journal on Software Tools for Technology Transfer, 2009.
- *N. Dershowitz, Z. Hanna & J. Katz.* **Bounded Model Checking with QBF.** Proceedings of the SAT 2005.

Web Links

- `http://nusmv.fbk.eu/` — Symbolic Model Checker NuSMV
- `http://www.cprover.org/cbmc/` — Software Bounded Model Checker für C Programme