Formale Methoden und Werkzeuge
Vorlesung
Wintersemester 2024–26

Johannes Waldmann, HTWK Leipzig

9. November 2026

Einleitung

Formale Methoden und Werkzeuge . . .

- . . . zur Spezifikation, Analyse (Verifikation), Synthese von Hard- und Softwaresystemen.
  - für (geplantes) System S (Bsp: CPU, Programm) *formales Modell* M angeben (Bsp: logische Schaltung, endl. Automat)
  - gewünschte System-Eigenschaft E angeben (Bsp: Boolesche Formel, reguläre Sprache)
  - Analyse: mit Werkzeug(unterstützung) feststellen, ob das Modell die Eigenschaft hat (Bsp: $M \Rightarrow E$, $M \subseteq E$)
  - Synthese (*Constraint-Programmierung*): Werkzeug konstruiert aus Constraint E ein passendes Modell M

Die Rolle der Abstraktion

- Schritt von System S zu Modell M ist *Abstraktion* (es werden Details ignoriert), das ist
  - teils *nützlich* (verbessert die Übersicht, Einsicht),
  - teils *notwendig* (nur wenn $M \subseteq E$ entscheidbar oder M aus E berechenbar, kann überhaupt ein Werkzeug zur Analyse bzw. Synthese implementiert werden)
- in den einfachen Beispielen (besonders am Anfang der VL) beginnen wir die Betrachtung bereits bei M (damit wir sehen, welche Theorien angewendet werden sollen)
- das Abstrahieren muß aber auch geübt werden, es ist aber eine anwendungsspezifische Kunst

Industrielle Anwendungen von FMW

- Verifikation von Schaltkreisen
  (*bevor* man diese tatsächlich produziert)
  $F =$ S-Implementierung$(x) \neq$ S-Spezifikation(x)
  wenn F unerfüllbar ($\neg \exists x.F$), dann ist Impl. korrekt
- Verifikation von Software durch *model checking*:
  Programmzustände abstrahieren durch Zustandsprädikate, Programmabläufe durch endliche Automaten.
  z. B. Thomas Ball et al. 2004: *Static Driver Verifier*
  `https://www.microsoft.com/en-us/research/project/slam/publications/`
  benutzt Constraint-Solver Z3 (Nikolaj Björner et al., 2007–) `https://github.com/Z3Prover/z3/wiki`

Industrielle Anwendungen von FMW

- automatische Analyse des Resourcenverbrauchs von Programmen
  - Termination (jede Rechnung hält)
  - Komplexität (. . . nach $O(n^2)$ Schritten)
- mittels *Bewertungen* von Programmzuständen:
  - W : Zustandsmenge $\to \mathbb{N}$
  - wenn $z_1 \to z_2$, dann $W(z_1) > W(z_2)$.
- Parameter der Bewertung werden durch Constraint-System beschrieben.
- vgl. Carsten Fuhs: *Automated Termination Analysis. . .*, Intl. School on Rewriting, 2022 `https://viam.science.tsu.ge/clas2022/isr/termination.html`

Anwendung: Polynom-Interpretationen

- Berechnungsmodell: Wortersetzung (≈ Turingmaschine)
- Programm: $ab \to ba$ (≈ Bubble-Sort)
  Beispiel-Rechnung: $abab \to baab \to baba \to bbaa$
- Bewertung W durch *Interpretation*: lineare Funktionen $f_a(x) = Px + Q, f_b(x) = Rx + S$ mit $P, Q, R, S \in \mathbb{N}$
  $W(abab) = f_a(f_b(f_a(f_b(0)))), \ldots$
- Interp. ist monoton: $x > y \Rightarrow f_a(x) > f_a(y) \wedge f_b(x) > f_b(y)$
- Interp. ist kompatibel mit Programm: $f_a(f_b(x) > f_b(f_a(x))$
- resultierendes Constraint-System für P, Q, R, S,
- Lösung mittels Z3

Constraint-Programmierung—Beispiel

```
(set-logic QF_NIA)(set-option :produce-models true)
(declare-fun P () Int) (declare-fun Q () Int)
(declare-fun R () Int) (declare-fun S () Int)
(assert (and (< 0 P) (<= 0 Q) (< 0 R) (<= 0 S)))
(assert (> (+ (* P S) Q) (+ (* R Q) S)))
(check-sat)(get-value (P Q R S))
```

- *Constraint-System*: eine prädikatenlogische Formel F
  $0 < P \wedge \cdots \wedge P \cdot S + Q > R \times Q + S$
- *Lösung*: Interpretation (Var.-Belegung), für die F wahr ist
- CP ist eine Form der *deklarativen* Programmierung.
- *Vorteil*: Benutzung von allgemeinen Suchverfahren (bereichs-, aber nicht anwendungsspezifisch).

Constraints in der Unterhaltungsmathematik

- Nikoli (1980–, „the first puzzle magazine in Japan.“) https://www.nikoli.co.jp/en/puzzles/
- Erich Friedman: *Math Magic* 1998–2020 https://erich-friedman.github.io/mathmagic/
- Simon Tatham's Portable Puzzle Collection https://www.chiark.greenend.org.uk/~sgtatham/puzzles/
- Angela und Otto Janko: http://www.janko.at/Raetsel/ ,
- Donald Knuth: *A Potpourri of Puzzles*, 2022, TAOCP, Band 4, Pre-Faszikel 9b, https://cs.stanford.edu/~knuth/taocp.html, https://cs.stanford.edu/~knuth/fasc9b.ps.gz

Wettbewerbe für Constraint-Solver

- für aussagenlogische Formeln: http://www.satcompetition.org/ (SAT = satisfiability)
- für prädikatenlogische Formeln https://smt-comp.github.io/ (SMT = satisfiability modulo theories) Theorien: $\mathbb{Z}$ mit $\leq$, Plus, Mal; $\mathbb{R}$ mit $\leq$, Plus; ...
- Termination und Komplexität https://www.termination-portal.org/wiki/Termination_Competition

Gliederung der Vorlesung

- Aussagenlogik
  - zur Modellierung (SAT-Kodierung) von Systemen mit endlichem Zustandsraum, begrenzter Schritt-Zahl (bounded model checking)
  - Werkzeuge: SAT-Kompiler (Tseitin-Transformation), SAT-Solver (Propagation, Backtracking, Lernen)
- eingeschränkte Prädikatenlogik
  - Unbekannte aus unendlichen Bereichen (Zahlen)
  - Werkzeuge: SMT-Solver (SAT modulo Theory)
- allgemeine Beweis-Systeme
  - zur Modellierung beliebiger (unbeschränkter) Systeme
  - Methode: Typisierung (Curry-Howard-Isomorphie, program = proof), Werkzeuge: interaktive Beweiser

Organisatorisches

- jede Woche 1 Vorlesung + 1 Übung
- Hausaufgaben (*Haus* bedeutet: zuhause bearbeiten, in der Übung diskutieren)
  - Aufgaben im Skript
  - Aufgaben in autotool

  Prüfungszulassung: regelmäßiges und erfolgreiches Bearbeiten von Hausaufgaben
- Klausur (2 h, keine Hilfsmittel).
- Quelltexte, Planung und Diskussion der Übungsaufgaben https://gitlab.dit.htwk-leipzig.de/johannes.waldmann/fmw-ws24 (Projekt-Mitgliedschaft beantragen, Zugang wird dann auf Mitglieder eingeschränkt)

Literatur

- Cerone, A., Roggenbach, M., Schlingloff, B.-H., Schneider, G., Shaikh, S.A.: *Teaching formal methods for software engineering - ten principles*, Informatica Didactica, Nr. 9, (2015). https://www.informaticadidactica.de/uploads/Artikel/Schlinghoff2015/Schlinghoff2015.pdf
- Uwe Schöning, Jacob Toran: *Das Erfüllbarkeitsproblem SAT* , Lehmanns (2012)
- Christel Baier, Joost Katoen: *Principles of Model Checking.* MIT Press, Cambridge (2008)
- Samuel Mimram: *Program = Proof*, https://www.lix.polytechnique.fr/Labo/Samuel.Mimram/teaching/INF551/course.pdf

Hausaufgaben

WS 25: 1b, 5, 4a, 2.

1. Pierluigi Crescenzi, Viggo Kann *A compendium of NP optimization problems* https://web.archive.org/web/20200227195925/http://www.nada.kth.se/~viggo/problemlist/compendium.html

   Beispiele:

(a) Minimum File Transfer Scheduling (node195).
(b) Minimum Dynamic Storage Allocation (node163).

   Erläutern Sie die Spezifikation an einem Beispiel. Geben Sie eine lösbare Instanz sowie dafür zwei Lösungen mit unterschiedlichem Maß an.

2. Aufgabe: formalisieren Sie *Math Magic* Februar 2007.

   Was ist dabei für Springer und König einfacher als für Dame, Läufer, Turm?

   (allgemein für Math Magic: offene Fragen https://erich-friedman.github.io/mathmagic/unsolved.html)

3. Aufgabe: formalisieren Sie Wolkenkratzer oder Towers (was ist der Unterschied?) https://www.janko.at/Raetsel/Wolkenkratzer https://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/towers.html

   - $B = \{0, 1 \ldots, n-1\}$, Unbekannte: $h_{x,y} \in B$ für $x, y \in B$
   - Constraint für eine Zeile x: (entspr. für Spalte)

$\bigvee_{p \in \text{Permutationen}(0,\ldots,n-1), p \text{kompatibel mit Vorgaben}} \bigwedge_{y \in \{0,\ldots,n-1\}} (h_{x,y} \; p(y))$

Bsp: $n = 4$, Vorgabe links 2, rechts 1, kompatibel sind $[0, 2, 1, 3], [2, 0, 1, 3], [2, 1, 0, 3], [2, 1, 0, 3]$.

- diese Formel wird exponentiell groß (wg. Anzahl Permutationen),
  Folge-Aufgabe: *geht das auch polynomiell*?
- Geben Sie eine Aufgabenstellung der Größe $w \times w$ an mit $4 \cdot w$ Vorgaben (d.h., *alle* Vorgaben), die mehr als eine Lösung hat.
  ... für $w = 4$, für größere w (einige, alle)
- (offene Frage?) Geben Sie eine eindeutig lösbare Instanz mit möglichst wenigen Vorgaben an. (Sind $w - 1$ Vorgaben für $w \times w$ immer möglich?)

4. Modellierung von Puzzles (aus Tatham-Collection)

(a) geben Sie ein Modell an für *Pegs* (Solitaire). Hinweise:
- Zustand = Boolesche Matrix,
- Schritt = Relation zwischen Matrizen,
- Lösung = Schrittfolge ($\Rightarrow$ Zustandsfolge). Wieviele Schritte?

(b) Modell für *Unruly*. (keine Zustandsfolge, sondern direkt die Lösung. Welches sind die Unbekannten, welches sind ihre Beziehungen, untereinander und zur Vorgabe)

(c) modellieren Sie *Untangle*

Vergleichen Sie mit den tatsächlichen Quelltexten
`https://git.tartarus.org/?p=simon/puzzles.git`

5. Constraint für monotone kompatible Bewertungsfunktion:
- lösen Sie mit Z3 (ist im Pool installiert, vgl. `https://www.imn.htwk-leipzig.de/~waldmann/etc/pool/`)
- eine *kleinste* Lösung finden (Summe von P, Q, R, S möglichst klein) — dafür Assert(s) hinzufügen.
- Abstieg der so gefundenen Bewertungsfunktion nachrechnen für $abab \to baab \to baba \to bbaa$
- gibt diese Bewertungsfunktion die maximale Schrittzahl genau wieder? (nein)
- Folge-Aufgabe: entspr. Constraint-System für Bewertungsfunktion für $ab \to bba$ aufstellen und lösen.

Erfüllbarkeit aussagenlogischer Formeln (SAT)

Aussagenlogik: Syntax

aussagenlogische Formel:
- elementar: Variable $v_1, \ldots$
- zusammengesetzt: durch Operatoren
  - einstellig: Negation
  - zweistellig: Implikation, Äquivalenz, Antivalenz,
  - mehrstellig möglich: Konjunktion, Disjunktion,

damit auch Quantifikation über endlichen Bereich E
$(\forall x \in E : F)$ ist (endliche!) Konjunktion $\bigwedge_{x \in E} F$

Aussagenlogik: Semantik
- Wertebereich $\mathbb{B} = \{0, 1\}$, Halbring $(\mathbb{B}, \vee, \wedge, 0, 1)$

  Übung: weitere Halbringe mit 2 Elementen?
- *Belegung* ist Abbildung $b : V \to \mathbb{B}$
- *Wert* einer Formel F unter Belegung b: $\mathrm{val}(F, b)$
- wenn $\mathrm{val}(F, b) = 1$, dann ist b ein *Modell* von F, Schreibweise: $b \models F$
- *Modellmenge* $\mathrm{Mod}(F) = \{b \mid b \models F\}$
- F *erfüllbar*, wenn $\mathrm{Mod}(F) \neq \emptyset$
- Modellmenge einer *Formelmenge*: $\mathrm{Mod}(M) = \{b \mid \forall F \in M : b \models F\}$

Formulierung von SAT-Problemen mit Ersatz
- Autoren: Edward Kmett et al., `https://hackage.haskell.org/package/ersatz`,
-
```
import Prelude hiding ((&&),(||),not )
import Ersatz
main = do
  ans <- solveWith minisat $ do
    p <- exists @Bit ; q <- exists @Bit
    assert $ (p || not q) && (not p || q)
    return [p,q]
  case ans of (Satisfied, Just res) -> print res
```
- Unbekannte erzeugen (`exists`), Formel konstruieren (`&&`,...), assertieren, lösen, Antwort benutzen
- zu Implementierung vgl. `https://www.imn.htwk-leipzig.de/~waldmann/etc/untutorial/ersatz/`

Benutzung von SAT-Solvern
- Eingabeformat: SAT-Problem in CNF:
  - Variable = positive natürliche Zahl
  - Literal = ganze Zahl ($\neq 0$, mit Vorzeichen)
  - Klausel = Zeile, abgeschlossen durch 0.
  - Formel = Header `p cnf <#Variablen> <#Klauseln>`, danach Klauseln
- Beispiel: die (konj. Normal-)Formel $(v_1 \vee \neg v_2) \wedge (\neg v_1 \vee v_2)$:
```
p cnf 2 2
1 -2 0
-1 2 0
```
- Löser (Bsp.): `minisat input.cnf output.text`
  Niklas Eén, Niklas Sörensson: *An Extensible SAT Solver*, 2003, `https://minisat.se/`,

SAT-Modellierung für das N-Damen-Problem

stelle möglichst viele Damen auf $N \times N$-Schachbrett, die sich nicht gegenseitig bedrohen.
- Unbekannte: $q_{x,y}$ für $(x, y) \in P = \{1, \ldots, N\}^2$

  mit Bedeutung: $q_{x,y} \iff$ Position (x, y) ist belegt
- Constraint: $\bigwedge_{a,b \in F, a \text{ bedroht } b} \neg q_a \vee \neg q_b$.
- „möglichst viele“ läßt sich hier vereinfachen zu:

  „in jeder Zeile genau eine“.

  vereinfachen zu: „. . . wenigstens eine.“

wir schreiben Programm,

N-Damen mit Ersatz
- Hilfsfunktionen für Boolesche Matrizen mit
```
import qualified Ersatz.Relation as R
```
-
```
queens n = do
 out <- solveWith (cryptominisat5Path "kissat") $ d
  b <- R.relation ((1,1),(n,n))
  assert $ flip all (rows b) $ \ r -> or r
  assert $ flip all (R.indices b) $ \ p ->
   flip all (R.indices b) $ \ q ->
    encode (reaches p q) ==> not (b R.! p && b R.! 
  return b
 case out of
  (Satisfied, Just b) -> do putStrLn $ R.table b
```
- (vollst. Quelltext: siehe Repo)

Zusammenfassung Ersatz (bisher)

- innerhalb von `solveWith` steht Befehlsfolge, Befehl ändert Zustand (Variablenzähler, Klausel-Ausgabe)
  - `exists @Bit :: MonadSAT s m => m Bit, R.relation _` konstruiert neue Variable(n)
  - `assert :: Bit -> m ()` übersetzt Formel in CNF, gibt Klauseln aus
- der Typ `Bit` beschreibt aussagenlogische Formeln (genauer: Schaltkreise), d.h., *symbolische* Repräsentation von (unbekannten) Wahrheitswerten
  Typ `Bool` beschreibt konkrete (bekannte) Wahrheitswerte
  booleschen Operatoren (`import Ersatz`) sind polymorph `not :: Boolean b => b -> b`, `instance Boolean Bool`, `instance Boolean Bit`

Modellierung durch SAT: Ramsey

gesucht ist Kanten-2-Färbung des K_5 ohne einfarbigen K_3.

- Aussagenvariablen $f_{i,j}$ = Kante (i, j) ist rot (sonst blau).
- Constraints:

$$\forall p : \forall q : \forall r : (p < q \wedge q < r) \Rightarrow ((f_{p,q} \vee f_{q,r} \vee f_{p,r}) \wedge \dots)$$

das ist ein Beispiel für ein Ramsey-Problem
(F. P. Ramsey, 1903–1930)
`http://www-groups.dcs.st-and.ac.uk/~history/Biographies/Ramsey.html`
diese sind schwer, z. B. ist bis heute unbekannt: gibt es eine Kanten-2-Färbung des K_{43} ohne einfarbigen K_5?
`http://www1.combinatorics.org/Surveys/ds1/sur.pdf`

SAT-Modell für Peg-Solitaire

- Spielzug: Stein überspringen und entfernen,
  Aufgabe: existiert Zugfolge von Initial (gegeben) zu Final (genau ein Stein übrig)
- Beispie: Initial ist volles Rechteck minus ein Stein
- (Wdhlg.) aussagenlog. Modell als Folge von Relationen $[B_0, B_1, \dots, B_n]$, für die gilt $\forall k : \mathrm{step}(B_k, B_{k+1})$.
- eine Realisierung: $\mathrm{move}(S, m, d, T)$ mit: S, T Brett, m Position (des übersprungenen Steins), d Richtung
  und $\mathrm{step}(S, T) = \bigvee_{m,d} \mathrm{move}(S, m, d, T)$
- Finalität von B_n ist einfach: das Zählen muß nicht SAT-kodiert werden—warum?

Hausaufgaben

WS 25: 2,3,4 (5, 7, 8 siehe Repo)

1. unterschiedliche Halbringe auf zwei Elementen?
2. für die Formel $S(b, h)$ (abhängig von Parametern $b, h \in \mathbb{N}$)

   Variablen: $v_{x,y}$ für $1 \le x \le b, 1 \le y \le h$

   Constraints:

   - für jedes x gilt: wenigstens einer von $v_{x,1}, v_{x,2}, \dots, v_{x,h}$ ist wahr
   - und für jedes y gilt: höchstens einer von $v_{1,y}, v_{2,y}, \dots, v_{b,y}$ ist wahr

   (a) unter welcher Bedingung an b, h ist $S(b, h)$ erfüllbar?
   Für den erfüllbaren Fall: geben Sie ein Modell an.

   Für den nicht erfüllbaren Fall: einen Beweis.
   (b) Erzeugen Sie (eine konjunktive Normalform für) $S(b, h)$ durch ein Programm (Sprache/Bibliothek beliebig) (b, h von der Kommandozeile, Ausgabe nach stdout)
   (c) Lösen Sie $S(b, h)$ durch minisat (kissat, Z3, ...), vergleichen Sie die Laufzeiten (auch im nicht erfüllbaren Fall).
3. Für $S(b, h)$ (vorige Aufgabe): Formel-Konstruktion, Löser-Aufruf mit Ersatz.
   - v vom Typ `Relation Int Int` (vgl. N Damen)
   - „für jedes x“: verwenden Sie `rows`
   - „für jedes y“: schreiben und verwenden Sie entsprechende Fkt. `columns`
   - „höchstens einer“: verwenden Sie „keine zwei“.

4. Für $a, b \ge 2$: die Ramsey-Zahl $R(a, b)$ ist die kleinste Zahl n, für die gilt: jede rot-blau-Kantenfärbung eines K_n enthält einen roten K_a oder einen blauen K_b.

   (Der Satz von Ramsey ist, daß es für jedes a, b tatsächlich solche n gibt.)

   (a) Beweisen Sie:
   i. $R(a, b) = R(b, a)$
   ii. $R(2, b) = b$
   iii. $R(a + 1, b + 1) \le R(a, b + 1) + R(a + 1, b)$
   (das liefert einen Beweis des Satzes von Ramsey)
   iv. wenn dabei beide Summanden rechts gerade Zahlen sind, dann $R(a + 1, b + 1) < \dots$
   (b) Bestimmen Sie damit obere Schranken für $R(3, 3), R(3, 4), R(4, 4)$ und vergleichen Sie mit den

   unteren Schranken durch SAT-Kodierung.
5. SAT-Kodierung für $R(a, b)$ mit Ersatz:

```
• main = ramsey 3 3 5

  ramsey a b n = do
    out <- solveWith (cryptominisat5Path "kissat") $
      r <- R.symmetric_relation ((1,1),(n,n))
      assert $ flip all (subs a [1 .. n]) $ \ c ->
        flip any (subs 2 c) $ \ [x,y] ->
           r R.! (x,y)
      assert $ flip all (subs b [1 .. n]) $ \ c ->
        flip any (subs 2 c) $ \ [x,y] ->
           not $ r R.! (x,y)
      return r
    ...
```

- Hilfsfunktion: verteilten Teilfolgen gegebener Länge:

  Beispiel: `subs 3 [1,2,3,4,5]`
  `= [[1,2,3],[1,2,4],[1,2,5],...,[3,4,5]]`
  (nicht notwendig in dieser Reihenfolge)

```
subs :: Int -> [a] -> [[a]]
subs 0 xs = [ [] ]  ;   subs k [] = []
subs k (x:xs) = map _  _  <> subs k xs
```

  mit `subs a [1 .. n]` zur Auswahl des K_a sowie `subs 2 c` zur Auswahl der Kanten.
- Diskutieren Sie Existenz (obere Schranke) und SAT-Kodierung für dreifarbigen Ramsey:
  $R(a, b, c) :=$ das kleinste n mit: jeder Kanten-3-Färbung des K_n enthält einen roten K_a oder einen grünen K_b oder einen blauen K_c.
  Ergänzen und beweisen: $R(a, b, c) \le R(a, R(b, c))$, anwenden für $R(3, 3, 3)$.

6. Modellierung als aussagenlogisches Constraint:

- Rösselsprung (= Hamiltonkreis)
- Norinori
  `https://nikoli.com/en/puzzles/norinori/`
- ABCEndView (oder ähnlich)
  `https://www.janko.at/Raetsel/AbcEndView/`

Vorgehen bei Modellierung:

- welches sind die Unbekannten, was ist deren Bedeutung?
  (Wie rekonstruiert man eine Lösung aus der Belegung, die der Solver liefert?)
- welches sind die Constraints?
  (wie stellt man sie in CNF dar? — falls nötig)

7. Unruly (S. Tatham Puzzles)

```
u1 = M.fromList -- eine Aufgabe (8 x 8)
  $ map (,False) [(1,7),(3,2),(5,1),(5,3),(5,4),(5,
  <> map (,True) [(1,4),(6,2),(6,3),(7,5),(8,4),(8,
unruly u = do
  let bnd = ((1,1),(8,8))
  out <- solveWith (cryptominisat5Path "kissat") $ (
    b <- R.relation bnd
    assert $ flip all (M.toList u) $ \ (k,v) ->
      b R.! k === encode v
    assert $ flip all (rows b <> columns b) $ \ rc
      balanced rc && mixed 3 rc
    ...
```

wie kann man feststellen, daß es genau eine Lösung gibt? (Solver nochmals aufrufen für modifizierte Formel.)

8. Peg (S. Tatham Puzzles)

```
peg b = do
  let bnd = ((1,1),(b,b))
      full = A.genArray bnd $ \ i -> True
      start :: A.Array (Int,Int) Bool
      start = full A.// [((1,2), False)]
  out <- solveWith (cryptominisat5Path "kissat") $ (
    boards <- replicateM (b*b - 1) $ R.relation bnd
    assert $ R.equals (head boards) (encode start)
    assert $ all step $ zip boards $ drop 1 boards
    return boards
  ...
type Board = R.Relation Int Int

step :: (Board, Board) -> Bit
move :: Board -> (Int,Int) -> (Int,Int) -> Board ->
```

Anwendg.: Bounded Model Checking

Begriff, Motivation

- *model checking*: feststellen, ob
  - ein *Modell* eines realen Hard- oder Softwaresystems (z.B. Zustandsübergangssystem f. nebenläufiges Programm)
  - eine *Spezifikation* erfüllt (z.B. gegenseitiger Ausschluß, Liveness, Fairness)
- *symbolic* model checking:
  symbolische Repräsentation von Zustandsfolgen
  im Unterschied zu tatsächlicher Ausführung (Simulation)
- *bounded*: für Folgen beschränkter Länge

Literatur, Software

- Armin Biere et al.: *Symbolic Model Checking without BDDs*, TACAS 1999, `http://fmv.jku.at/bmc/`
  Software damals: Übersetzung nach SAT, später: SMT (`QB_BV`), Solver: `http://fmv.jku.at/boolector/`
- Daniel Kroening und Ofer Strichman: *Decision Procedures, an algorithmic point of view*, Springer, 2008. `http://www.decision-procedures.org/`
  Software: `http://www.cprover.org/cbmc/`
- Nikolaj Bjørner et al.: *Program Verification as Satisfiability Modulo Theories*, SMT-Workshop 2012, `http://smt2012.loria.fr/`
  Softw.: `https://github.com/Z3Prover/z3/wiki`

BMC für Mutual Exclusion-Protokolle

System mit zwei (gleichartigen) Prozessen A, B:

```
A0: maybe goto A1
A1: if l goto A1 else goto A2
A2: l := 1; goto A3
A3: [critical;] goto A4
A4: l := 0; goto A0

B0: maybe goto B1
B1: if l goto B1 else goto B2
B2: l := 1; goto B3
B3: [critical;] goto B4
B4: l := 0; goto B0
```

Schließen sich `A3` und `B3` gegenseitig aus? (Nein.)

(nach: Donald E. Knuth: TAOCP, Vol. 4 Fasz. 6, S. 20ff)

Modell: Zustandsübergangssystem

Zustände:

- jeder Zustand besteht aus:
  - Inhalte der Speicherstellen (hier: $l \in \{0, 1\}$)
  - Programmzähler (PC) jedes Prozesses (hier: $A \in \{0 \dots 4\}, B \in \{0 \dots 4\}$)
- Initialzustand: $I = \{l = 0, A = 0, B = 0\}$
- Menge der Fehlerzustände: $F = \{A = 3, B = 3\}$

Übergangsrelation (nichtdeterministisch): für $P \in \{A, B\}$:

- P führt eine Aktion aus (schreibt Speicher, ändert PC)

Aussagenlog. Formel für $I \rightarrow^{\leq k} F$ angeben,
deren Erfüllbarkeit durch SAT- oder SMT-Solver bestimmen

One-Hot-Kodierung

- $p \in \{0 \dots 4\}$ symbolisch repräsentieren durch Folge $[p_0, \dots, p_4]$ von symbolischen Wahrheitswerten
  . . . von denen *genau einer* wahr ist
  Bsp: $p = 3$ kodiert durch $[0, 0, 0, 1, 0]$.
- das ist die *one hot*-Kodierung (eine Stelle ist *hot* = stromführend)
- eine Realisierung ist $(\bigvee_i p_i) \wedge \bigwedge_{i<j} (\neg p_i \vee \neg p_j)$
- es gibt andere Kodierungen für endliche Bereiche (z.B.: binär: benötigt weniger Unbekannte, aber evtl. größere Formeln)

Übung BMC

- Software: `https://git.imn.htwk-leipzig.de/waldmann/boumchak`
- überprüfe 1. gegenseitigen Ausschluß, 2. deadlock, 3. livelock (starvation) für weitere Systeme, z.B.
  E. W. Dijkstra, 1965:
  `https://www.cs.utexas.edu/~EWD/transcriptions/EWD01xx/EWD123.html#2.1.`
  G. L. Peterson, *Myths About the Mutual Exclusion Problem*, Information Processing Letters 12(3) 1981, 115–116

Anzahl-Constraints

Definition, Motivation

- $\text{Count}_{\leq k}(x_1, \ldots, x_n) := (\sum_i x_i) \leq k$.
- AMO (at most one): $= \text{Count}_{\leq 1}$, entspr. ALO, EXO
- Schubfach-Constraint (als Testfall, erfüllbar gdw. $B \leq H$)

  $\bigwedge_{1 \leq i \leq H} \text{AMO}(x_{ij} | 1 \leq j \leq B) \wedge \bigwedge_{1 \leq j \leq B} \text{ALO}(x_{ij} | 1 \leq i \leq H)$
- Schubfach für $B = H$: dann ist x Permutationsmatrix,

  repräsentiert Bijektion von $\{1, \ldots, B\}$ auf sich
- Anwend.: Rösselsprung, Hamiltonkreis in $G = (V, E)$

  Pfad p in G als Bijektion von Indexmenge $\{1, \ldots, |V|\}$ in Knotenmenge V mit $\bigwedge_i (p(i), p(i+1)) \in E$.

SAT-Kodierung eines Rösselsprungs

```
let n = height * width; places = [0 .. n-1]

let decode p = divMod p width
    edge p q =
        let (px,py) = decode p; (qx,qy) = decode q
        in  5 == (px-qx)^2 + (py-qy)^2
    rotate (x:xs) = xs <> [x]

a <- replicateM n $ replicateM n $ exists @Bit
assert $ all exactly_one a
assert $ all exactly_one $ transpose a
assert $ flip all (zip a $ rotate a) $ \ (e, f) ->
  flip all places $ \ p ->  e!!p  ==>
    flip any (filter (edge p) places) (\ q -> f!!q)
```


SAT-Kodierungen von AMO (I) (quad, lin)

- quadratisch: $\text{AMO}(x_1, \ldots, x_n) = \bigwedge \{\overline{x_i} \vee \overline{x_j} | 1 \leq i < j \leq n\}$

  $\binom{n}{2}$ Klauseln, keine zusätzlichen Variablen
- linear: mit Kodierung enc : $x \mapsto (x_e, x_z) = (x \geq 1, x \geq 2)$:

  $0 \mapsto (0,0), 1 \mapsto (1,0), 2, 3, \cdots \mapsto (1,1)$

  Addition: $(x_e, x_z) +_{\text{enc}} (y_e, y_z) = \ldots$

  so daß $\text{enc}(x+y) = \text{enc}(x) +_{\text{enc}} \text{enc}(y)$

  $\text{AMO}(x_1, \ldots, x_n) =$ `let` $(s_e, s_z) = \sum_{\text{enc}} \text{enc}(x_i)$ `in` ...

SAT-Kodierungen von AMO (II) - log

- $\text{AMO}(x) = \exists h : (x_i \Rightarrow (i = h))$

  h binär repräsentiert mit $\log n$ Bits.
- Bsp. $\text{AMO}(x_0, \ldots, x_3) = \exists h_1, h_0 :$

$$\begin{array}{ll} & (\overline{x}_0 \vee \overline{h}_1) \wedge (\overline{x}_0 \vee \overline{h}_0) \\ \wedge & (\overline{x}_1 \vee \overline{h}_1) \wedge (\overline{x}_1 \vee h_0) \\ \wedge & (\overline{x}_2 \vee h_1) \wedge (\overline{x}_2 \vee \overline{h}_0) \\ \wedge & (\overline{x}_3 \vee h_1) \wedge (\overline{x}_3 \vee h_0) \end{array}$$

- $n \log n$ Klauseln, $\log n$ zusätzliche Variablen
- die Hilfsvariablen h_0, h_1 sind keine Funktionen der Eingangsvariablen. (wenn alle x_i falsch, dann h_i beliebig)
- Ü: man kann eine Skolem-Funktion trotzdem einfach angeben

SAT-Kodierungen von AMO (III) - sqrt

- für $\text{AMO}(x)$: die x in einem Rechteck anordnen,

  $z_i := \bigvee_j x_{ij}$ (Zeile i), $s_j := \bigvee_i x_{ij}$ (Spalte j),

  dann $\text{AMO}(x) = \text{AMO}(z) \wedge \text{AMO}(s)$.

  Jingchao Chen: *A New SAT Encoding of the At-Most-One Constraint* 10th Workshop Constraint Modeling and Reformulation, 2010 `https://www.it.uu.se/research/group/astra/ModRef10/programme.html`
- Formelgröße $f(n) = \Theta(n) + 2f(\sqrt{n})$, mit $\Theta(\sqrt{n})$ Hilfsvar.

  Lineare Faktoren sind klein. Ü: wenn `assert (amo xs)`, kann man einige Klauseln weglassen. Welche?

Listen (Generatoren) zur Programmablaufsteuerung

- ```
  amo_quad bs = and $ do
    (b : cs) <- L.tails bs; c <- cs
    return $ not b || not c
  ```
- benutzt

  `L.tails [1,2,3] = [[1,2,3],[2,3],[3],[]]`
- erster (äußerer) Generator `(b : cs) <- L.tails bs`

  es werden der Reihen nach gebunden:

  `b=1,cs=[2,3]`, dann `b=2,cs=[3]`, dann `b=3,cs=[]`.
- zweiter (innerer) Generator `c <- cs`

  für `b=1, cs=[2,3]` werden der Reihe nach gebunden: `c=2`, dann `c=3`.

Matrizen als Listen von Listen

- ```
  x :: [[Bit]] <-
    replicateM h $ replicateM b $ exists @Bit
  ```
- `replicateM`: wiederholt Aktion, liefert Liste der Resultate

  `exists @Bit`: liefert eine neue Variable

  `replicateM b $ ...`: liefert b Variablen (= eine Zeile)

  `replicateM h $ ...`: liefert h Zeilen (= eine Matrix)
- `L.transpose` transponiert Matrix

  `L.transpose [[1,2,3],[4,5,6]] = [[1,4],[2,5],[3,6]]`

Quantifikation über Listen

- Anw.: `flip all x alo`, bedeutet $\forall e \in x : \text{alo}(e)$
- benutzt Funktionen

  ```
  flip :: (a -> b -> c) -> b -> a -> c
  all :: (Boolean b, Foldable t)
    => (a -> b) -> t a -> b
  ```

  dabei t ein Containertyp (über den man iterieren kann)

  das `flip` nur, damit Argumentreihenfolge wie bei Quantor bzw. Zählschleife (erst Bereich, dann Prädikat)
- eine mögliche Implementierung dafür ist

  ```
  all f x = and $ do e <- x ; return f e
  ```
- entsprechend `any` für endlichen Existenz-Quantor

Iterierte assoziative Verknüpfung

- Anw.: `foldMap enc bs`, realisiert $\sum_{b \in \mathtt{bs}} \mathtt{enc}(b)$
  wobei „Addition“ die durch `(<>)` definierte Verknüpfung ist (mit neutralem Element `mempty`)
- benutzt Funktion und Typklassen

```
foldMap :: (Foldable t, Monoid m)
  => (a -> m) -> t a -> m
class Semigroup a where
  (<>) :: a -> a -> a
class Semigroup a => Monoid a where
  mempty :: a  :  mappend :: a -> a -> a
```

- vgl. `https://www.imn.htwk-leipzig.de/~waldmann/etc/list/` (Lists that are really not there)

Aufgaben

WS26: 2, 10, 12

1. Modellierung Sudoku: Kodierung eines (ausgefüllten) Schemas durch $9 \times 9 \times 9$ unbekannte Bit mit $u_{x,y,z} \iff$ auf Koordinate (x, y) steht Zahl z.
   für welche Teilmengen gelten EXO-Constraints?
   Hinweis: für $9^2 + 3 \cdot 9^2$ Mengen, jede mit 9 Elementen.
2. geben Sie Beispiele aus Rätsel-Sammlungen von Nikoli, Janko, Tatham, bei deren SAT-Kodierung AMO- oder EXO-Constraints vorkommen sollten
3. Vergleiche Sie die *commander*-Kodierung für AMO von Klieber und Kwon, Int. Workshop Constraints in Formal Verification, 2007, mit Kodierungen aus dem Skript.
   a) auf dem Papier, b) praktisch: mit ersatz

implementieren, Formelgrößen messen, auch nach Vorverarbeitung durch minisat

4. für AMO-log: geben Sie eine Skolem-Funktion für $\forall x_0, \ldots \exists h_0, \ldots : \ldots$ an.
   d.h., die eine erfüllende Belegung der h_i bestimmt, falls AMO$(x_0, \ldots)$ wahr ist.
5. für AMO über 2^k Argumente: verwenden Sie sqrt-Kodierung für Rechteck mit Abmessungen $2 \times 2^{k-1}$, dann rekursiv über 2^{k-1}.
   Vergleichen Sie mit der log-Kodierung.
6. für AMO-lin: untersuchen Sie den Unterschied zwischen der Verwendung von `foldr` (von rechts) und `foldb` (balanciert)
   welche Maße der erzeugten Formel stimmen überein,

welche unterscheiden sich?

7. vergleichen Sie Formelgrößen und Solver-Laufzeiten für unterschiedliche AMO-Kodierungen für die Spalten in den unlösbaren Schubfach-Formeln $S(n+1, n)$.
8. für die AMO-Kodierungen linear und sqrt: wie kann man mit möglichst wenig Zusatz-Aufwand EXO erhalten?
9. ordnen Sie die EXO-Kodierung im Bounded Model Checker (boumchak) in die Systematik der VL ein.
10. Für den Rösselsprung (Code in `fmw-24/leap`)
    - gegebene EXO-Implementierung diskutieren, durch andere ersetzen, Formelgröße/Solverlaufzeit beobachten
    - zusätzliches `assert $ a !! 0 !! 0` diskutieren
    - andere Sprung-Distanzen (Giraffe usw., siehe unten)

- Kamel (1,3)-Sprung (auf nur schwarzen? oder weißen?) Feldern implementieren
- andere Kodierung für Hamiltonkreis: Neng-Fa Zhou: *In Pursuit of an Efficient SAT Encoding for the Hamiltonian Cycle Problem*, CP 2020, ModRef 2019, `https://www.sci.brooklyn.cuny.edu/~zhou/`
- Anwendung: George Jelliss: *Leapers at Large*, 2001, 2022 `http://www.mayhematics.com/t/pl.htm`
  Resultate zur Existenz von Hamilton-Pfaden (HP) und -Kreisen (HC) nachrechnen und ergänzen, Bsp.:
  – HP für Giraffe $(1, 4)$ auf 11× 11?
    (Nein—unsat nach 6 Stunden)
  – HP für Zebra $(2, 3)$ auf 13 × 13?
  – HP für Antilope $(3, 4)$ auf 20 × 20?

vgl. Donald Knuth: *Leaper Graphs*, 1994, `https://arxiv.org/abs/math/9411240`

1. Plan SAT-Kodierung Skyline (Towers, Wolkenkratzer)
   Jede Zahl (jedes Haus) wird one-hot-kodiert, dann

```
type Haus=[Bit];type Zeile=[Haus];type Stadt=[Zeile
```

- (ohne Vorgaben) welche AMO-Constraints gelten? Eine Matrix, die diese Constraints erfüllt, heißt *lateinisches Quadrat* (Anzahl der lateinischen Quadrate: `https://oeis.org/A002860`)
- (mit Vorgaben) Implementieren Sie die Berechnung der (von links) sichtbaren Dächer einer Zeile `sicht :: Zeile -> [Bit]`. Das Resultat soll der Bitvektor s sein mit $s_i \iff$ Dach der Höhe i ist sichtbar.
- für die anderen Blickrichtungen: die Funktion `sicht`

bleibt, die `Stadt` wird transformiert. Wie?

- Benutzen Sie eine SAT-Kodierung, um zu beweisen: jedes lateinische Quadrate der Seitenlänge 6 enthält wenigstens eine Zelle, die von keiner Seite sichtbar ist.

2. SAT-Kodierungen für das N-Damen-Problem:
   - naiv N^4
     für alle a aus Spielfeld: für alle b aus Spielfeld: sich a und b sehen, dann $\neg a \vee \neg b$
   - besser N^3
     für alle a aus Spielfeld: für alle b, die von a gesehen werden: ...
   - noch besser N^2
     AMO-Constraints (jeweils lineare Formelgröße) über geeigneten (linear vielen) Mengen

Symmetrien

Symmetriebrechungen (Def)

- Ansatz: zusätzliche Constraints $\rightarrow$ schnellere Lösung.
- viele Anwendungsaufgaben $\{x \mid x \in U, P(x)\}$ haben Symmetrien: $s : U \rightarrow U$ bijektiv mit $\forall x : P(x) \Rightarrow P(s(x))$
- Bsp: n Damen: die 8 Symmetrien des Quadrates
- eine Menge S von Symmetrien definiert eine Äquivalenzrelation $x \sim_S y \iff (x, y) \in (S \cup S^{-1})^*$
- eine *Brechung* einer Symmetriemenge S ist ein Constraint C, so daß für jedes x die Menge $\{y \mid x \sim_S y \wedge C(y)\}$ genau ein Element hat.
- Satz (Korrektheit,Vollständigkeit): $\emptyset \neq P \iff \emptyset \neq P \wedge C$

Symmetriebrechungen (Bsp)

- Rösselsprung:

  $\pi(1) = 1$ (bricht zyklische Verschiebung der Permutation)

  Ü: was bricht die Spiegelung?
- N Damen (und allgemeines Prinzip)

  definiere totale Ordnung $<$ auf U und

  $C(x) = \bigwedge_{s \in S^*} x \leq S(x)$

  z.B. lexikografische Ordnung auf Bitvektoren
- Ü: welche Symmetrien und Brechungen für
  - peacable coexisting armies of queens (A250000)
  - square free point set (A227133)

Symmetrievermutung

- für eine Gruppe S von Symmetrien von P verwende zusätzliches Constraint $C = \bigwedge_{s \in S} x = S(x)$
- Bsp: N Damen: Konfiguration ist drehsymmetrisch.
- aus $\mathrm{Mod}(C \wedge P) \subseteq \mathrm{Mod}(P)$

  Solver findet $b \in \mathrm{Mod}(C \wedge P)$ eventuell schneller
- aus $\emptyset = \mathrm{Mod}(C \wedge P)$ folgt nicht $\emptyset = \mathrm{Mod}(P)$
- M. Heule: Avoiding Triples in Arithmetic Progression. Journal of Combinatorics 8(3): 391-422. `https://www.cs.cmu.edu/~mheule/publications/JOC_08_03_A01.pdf` (Abschnitt 4. Symmetry and Satisfiability)

Übung

1. Symmetriebrechung für verallg. Rösselsprung:
   (a) Rotation der Zeit $\pi \mapsto \pi'$ mit $\pi'(t) = \pi(t + \Delta)$ gebrochen durch $\pi(0) = (0, 0)$ (Zeit 0: Pos. oben links)
   (b) Spiegelung der Zeit: $\pi'(t) = \pi(-t)$ kann gebrochen werden durch $\pi(1) < \pi(-1)$ (beweisen, implementieren)
2. Symmetrievermutung für verallg. Rösselsprung: Bsp: Drehung um 180 Grad und Zeitverschiebung um halbe Periode (bei Quadrat: 90/viertel)
3. Welche Symmetrien? Bsp. $N = 2, Q = 3$. `https://erich-friedman.github.io/mathmagic/0207.html` Lösen Sie die offenen Fälle (Bachelorarbeit, SAT-Kodierung von Anzahl-Constraints)

SAT: Normalformen, Transformationen

Normalformen (DNF, CNF)

Definitionen:

- Variable: $v_1, \ldots$ Literal: v oder $\neg v$
- DNF-Klausel: Konjunktion von Literalen
- DNF-Formel: Disjunktion von DNF-Klauseln
- CNF-Klausel: Disjunktion von Literalen
- CNF-Formel: Konjunktion von CNF-Klauseln

Disjunktion als Implikation: diese Formeln sind äquivalent:

- $(x_1 \wedge \ldots \wedge x_m) \rightarrow (y_1 \vee \ldots \vee y_n)$
- $(\neg x_1 \vee \ldots \vee \neg x_m \vee y_1 \vee \ldots \vee y_n)$

Äquivalenzen

- Def: Formeln F und G heißen *äquivalent*, wenn $\mathrm{Mod}(F) = \mathrm{Mod}(G)$.
- Satz: zu jeder Formel F existiert äquivalente Formel G in DNF.
- Satz: zu jeder Formel F existiert äquivalente Formel G' in CNF.
- aber . . . wie groß sind diese Normalformen?

Erfüllbarkeits-Äquivalenz

- Def: F und G *erfüllbarkeitsäquivalent*, wenn $\mathrm{Mod}(F) \neq \emptyset \iff \mathrm{Mod}(G) \neq \emptyset$.
- Satz: es gibt einen Polynomialzeit-Algorithmus, der zu jeder Formel F eine erfüllbarkeitsäquivalente CNF-Formel G berechnet.
- (Zeit $\geq$ Platz, also auch $|G| = \mathrm{Poly}(|F|)$)
- Beweis (folgt): Tseitin-Transformation
- Vor-Überlegung: warum gibt es keine vergleichbare Aussage für DNF?

Tseitin-Transformation

- Grigori Tseitin, *On the Complexity of Derivation in propositional calculus*, 1966
- Spezifikation:
  - Geg.: F, ges,: erfüllbarkeitsäquivalentes G in CNF.
  - wir verschärfen das zu: $\mathrm{Var}(F) \subseteq \mathrm{Var}(G)$ und $\forall b : b \models F \iff \exists b' : b \subseteq b' \wedge b' \models G$.
- Plan:
  - für jeden nicht-Blatt-Teilbaum T des Syntaxbaumes von F eine zusätzliche Variable n_T einführen,
  - so daß $\forall b' \in \mathrm{Mod}(G) : \mathrm{val}(n_T, b') = \mathrm{val}(T, b)$.
- Realisierung:
  - (Bsp.) $T = L \vee R$, dann $n_T \leftrightarrow (n_L \vee n_R)$ als CNF
  - für jeden der $|F|$ Knoten: ≤ 8 Klauseln mit 3 Literalen

Tseitin-Transf. für Schaltkreise

- beschriebenes Verfahren funktioniert ebenso für Schaltkreise (azyklische gerichtete Graphen, mit Booleschen Operatoren markiert)
- Schaltkreis entsteht aus Baum durch Identifikation (sharing) von Knoten
- für jeden Knoten eine neue Variable angelegen und deren Wert lokal durch eine CNF bestimmen
- Ersatz realisiert *observable sharing* durch Adress-Vergleich von Syntaxbaumknoten (des Typs `Bit`)

```
-- (xor a b) wird nur einmal T-transformiert:
let {s = (xor a  b)} in (c || s  ) && (not c || not
```


Visualisierung von Schaltkreisen

- mit `https://git.imn.htwk-leipzig.de/waldmann/ersatz-viz`
- Beispiel

```
import qualified Ersatz.Bit.Display as D

D.display $ do
   a <- exists ; b <- exists; c <- exists
   return $ let {s = (xor a  b)}
            in (c || s  ) && (not c || not  s)
```

- zu zeichnende Formel steht nach `return`

Tseitin-Transformation in Ersatz (Bsp)

- so ausprobieren:

```
ghci> runSAT' $ do
  x <- exists; y <- exists @Bit; assert (x === y)
((),SAT 4 ((1) & (-4 | -3 | -2) & (-4 | 2 | 3)
   & (-2 | 3 | 4) & (-3 | 2 | 4) & (-4)) mempty)
```

- zu transformierende Formel steht nach `assert` (nicht: nach `return`, wie bei `display`)

  Tseitin-Transformation nach jedem `assert` (ohne `assert`: keine Ausgabe einer CNF)
- Aufgabe: damit observable sharing bestätigen
- Aufgabe: CNF zu `assert (x /== (y /== z))` (XOR)

Abmessungen von Schaltkreisen

- (die Frage ist *nicht*, ob AMD Epyc in SP5-Sockel paßt)
- Def. Schaltkreis (circuit): gerichteter kreisfreier markierter Graph, als symbolische Repräsent. einer Booleschen Fkt.
- Def. Abmessung (complexity):
  - Def. Größe (size): Anzahl der Knoten
    - bei Hardware: Material-Aufwand
    - für SAT-Kodierung: (nach Tseitin-T.) Größe der CNF
  - Def. Tiefe (depth): Länge eines längsten Pfades
    - bei (paralleler!) Hardware: Rechenzeit
    - bei SAT-Kodierung: Länge von Abhängigkeitsketten von (Hilfs)variablen, beeinflußt Solver-Laufzeit
- Bsp: XOR N-stellig: $O(N)$ Größe, $O(\log(N))$ Tiefe

Abmessungen von Schaltkreisen: wozu?

- SAT-Kodierung: man möchte immer die (Teil)Formel, für die der Solver am schnellsten die Erfüllbarkeit entscheiden kann
- das läßt sich sehr schwer vorhersagen, abhängig von
  - Lösungsverfahren auf Teilformeln
  - nicht-lokale Kombination von Teil-Lösungen
- betrachten stattdessen die Größe der Eingabe (CNF, Größen sind: Anzahl Variablen, Anzahl Klauseln)
  in vielen Publikationen so durchgeführt (siehe AMO)
- . . . stattdessen Größe eines Schaltkreises
  - mit beschränktem Eingangsgrad (z. B. 2)
  - unbeschränkt nur f. Disjunktion, Konjunktion

Aufgaben zu Tseitin-Transformation

1. für diese Formeln:
   - $(x_1 \leftrightarrow x_2) \leftrightarrow (x_3 \leftrightarrow x_4)$
   - Halb-Adder (2 Eingänge x, y, 2 Ausgänge r, c)
     $(r \leftrightarrow (\neg(x \leftrightarrow y))) \wedge (c \leftrightarrow (x \wedge y))$
   - Full-Adder (3 Eingänge, 2 Ausgänge)

   jeweils:
   - führe die Tseitin-Transformation durch
   - gibt es kleinere erfüllbarkeitsäquivalente CNF? (deren Modelle Erweiterungen der Original-Modelle sind)
2. `data Bit` hat weitere Konstruktoren (Xor, Mux).
   Wo werden diese benutzt?
   Helfen sie tatsächlich bei der Erzeugung kleiner CNFs?

SAT-Solver

Spezifikation

- Eingabe: eine Formel in CNF
  [[1],[-4,-3,11],[-11,3],[-11,4],[-11],[3,4,13],[-13,-3],[-13,-4],[-2,12,13],[-12,2],[-13,-12],[-12],[-7,-6,14],[-14,6],[-14,7],[-14],[6,7,16],[-16,-6],[-16,-7],[-5,15,16],[-15,5],[-16,-15],[-15],[-10,-9,17],[-17,9],[-17,10],[-17],[9,10,19],[-19,-9],[-19,-10],[-8,18,19],[-18,8],[-19,-18],[-18],[2,5,8,20],[-20,-2],[-20,-5],[-20,-8],[-20],[3,6,9,21],[-21,-3],[-21,-6],[-21,-9],[-21],[4,7,10,22],[-22,-4],[-22,-7],[-22,-10],[-22]]
- Ausgabe:
  - eine erfüllende Belegung
  - *oder* ein Beweis für Nichterfüllbarkeit

Implementierung eines naiven SAT-Solvers

- benutzt die Schnittstelle aus `ersatz`

```
minisat :: Solver SAT IO
type Solver s m
  = s -> m (Result, IntMap Bool)
data Result
  = Unsolved  | Unsatisfied | Satisfied
class DIMACS t where
  dimacsNumVariables :: t -> Int
  dimacsClauses :: t -> Seq IntSet
data SAT; instancs DIMACS Sat
```


Lösungsverfahren

- vollständige Suche (alle Belegungen, vollst. Binärbaum)
- unvollständige Suche (einige Belegungen)
  - evolutionär (Genotyp $=$ Belegung)
  - lokale Suche (Selman, Kautz, Cohen 1993: Walksat)
- verbesserte vollständige Suche (Erkennen und Abschneiden sinnloser Teilbäume)
  DPLL (Davis, Putnam, Logeman, Loveland 1960/61)
- weitere Verbesserungen durch
  - Lernen (und Vergessen!) von zusätzlichen Klauseln
  - Vorverarbeitung zum Entfernen von Variablen
  - Parallelisierung
    (Kommunikation mehrerer Suchverfahren)

Evolutionäre Algorithmen für SAT

- Genotyp: Bitfolge $[x_1, \ldots, x_n]$ fester Länge
- Phänotyp: Belegung $b = \{(v_1, x_1), \ldots, (v_n, x_n)\}$
- Fitness: z. B. Anzahl der von b erfüllten Klauseln
- Operatoren:
  - Mutation: einige Bits ändern
  - Kreuzung: one/two-point crossover?
    Problem: starke Abhängigkeit von Variablenreihenfolge

Lokale Suche (GSat, Walksat)

Bart Selman, Cornell University,
Henry Kautz, University of Washington
`https://web.archive.org/web/20070311005144/http://www.cs.rochester.edu/u/kautz/walksat/`
Algorithmus:

- beginne mit zufälliger Belegung
- wiederhole: ändere das Bit, das die Fitness am stärksten erhöht

Problem: lokale Optima — Lösung: Mutationen.

DPLL

Davis, Putnam (1960), Logeman, Loveland (1962),
`http://dx.doi.org/10.1145/321033.321034`
`http://dx.doi.org/10.1145/368273.368557`
Zustand $=$ partielle Belegung

- *Decide*: eine Variable belegen
- *Propagate*: alle Schlußfolgerungen ziehen
  Beispiel: Klausel $x_1 \vee x_3$, partielle Belegung $x_1 = 0$,
  Folgerung: $x_3 = 1$
- bei *Konflikt* (widersprüchliche Folgerungen)
  - (DPLL original) Backtrack (zu letztem Decide)
  - (DPLL mit CDCL) Backjump (zu früherem Decide)

DPLL-Begriffe

für partielle Belegung b (Bsp: $\{(x_1, 1), (x_3, 0)\}$): Klausel c ist

- *erfüllt*, falls $\exists l \in c : b(l) = 1$, Bsp: $(\neg x_1 \vee x_2 \vee \neg x_3)$
- *Konflikt*, falls $\forall l \in c : b(l) = 0$, Bsp: $(\neg x_1 \vee x_3)$
- *unit*, falls $\exists l \in c : b(l) = \bot \wedge \forall l' \in (c \setminus \{l\}) : b(l') = 0$,
  Bsp: $(\neg x_1 \vee \neg x_2 \vee x_3)$. Dabei ist $l = \neg x_2$ das Unit-Literal.
- *offen*, sonst. Bsp: $(x_2 \vee x_3 \vee x_4)$.

Eigenschaften: für CNF F und partielle Belegung b:

- wenn $\exists c \in F : c$ ist Konflikt für b, dann $\neg\exists b' \supseteq b$ mit $b' \models F$
  (d.h., die Suche kann dort abgebrochen werden)
- wenn $\exists c \in F$: c ist Unit für b mit Literal l, dann
  $\forall b' \supseteq b : b' \models F \Rightarrow b'(l) = 1$
  (d.h., l kann ohne Suche belegt werden)

DPLL-Algorithmus

Eingabe: CNF F,
Ausgabe: Belegung b mit $b \models F$ oder UNSAT.
$\mathrm{DPLL}(b)$ (verwendet Keller für Entscheidungspunkte):

- (success) falls $b \models F$, dann halt (SAT), Ausgabe b.
- (backtrack) falls F eine b-Konfliktklausel enthält, dann:
  - falls Keller leer, dann halt (UNSAT)
  - sonst $v := pop()$ und $\mathrm{DPLL}(b_{<v} \cup \{(v, 1)\})$.
    dabei ist $b_{<v}$ die Belegung *vor* decide(v)
- (propagate) falls F eine b-Unitklausel c mit Unit-Literal l enthält: $\mathrm{DPLL}(b \cup \{(\text{variable}(l), \text{polarity}(l))\})$.
- (decide) sonst wähle $v \notin \mathrm{dom}\, b$, push$(v)$, und $\mathrm{DPLL}(b \cup \{(v, 0)\})$.

DPLL: Eigenschaften

- Termination: DPLL hält auf jeder Eingabe
- Korrektheit: wenn DPLL mit SAT hält, dann $b \models F$.
- Vollständigkeit: wenn DPLL: UNSAT, dann $\neg\exists b : b \models F$

wird bewiesen durch Invariante

- $\forall b' : b' \in \mathrm{Mod}(F) \Rightarrow b \leq_{\text{lex}} b'$
  (wenn DPLL derzeit b betrachtet, und wenn F ein Modell b' besitzt, dann ist b' unterhalb oder rechts von b)
- dabei bedeutet: $b \leq_{\text{lex}} b'$:
  $b \subseteq b'$ oder $\exists v : b(v) = 0 \wedge (b_{<v} \cup \{(v, 1)\}) \subseteq b'$

Satz (Ü): für alle endlichen V: $<_{\text{lex}}$ ist eine wohlfundierte Relation auf der Menge der partiellen V-Belegungen:

DPLL-Beispiel

```
[[2,3],[3,5],[-3,-4],[2,-3,-4]
,[-3,4],[1,-2,-4,-5],[1,-2,4,-5]]
```

decide belegt immer die kleinste freie Variable, immer zunächst negativ

DPLL-Beispiel (Lösung)

```
[[2,3],[3,5],[-3,-4],[2,-3,-4]
,[-3,4],[1,-2,-4,-5],[1,-2,4,-5]]

[Dec (-1),Dec (-2),Prop 3,Prop (-4),Back
,Dec 2,Dec (-3),Prop 5,Prop (-4),Back
,Dec 3,Prop (-4),Back,Back,Back
,Dec 1,Dec (-2),Prop 3,Prop (-4),Back
,Dec 2,Dec (-3),Prop 5]
```


DPLL: Implement., Heuristik, Ergänzungen

- Grundlage ist effiziente Impl. von UP und Konflikt-Erkennung
- Methoden:
  - Wahl der nächsten Entscheidungsvariablen (kommt am häufigsten in aktuellen Konflikten vor)
  - Lernen von Konflikt-Klauseln (erlaubt Backjump)
  - Vorverarbeitung (Variablen und Klauseln eliminieren)
- alles vorbildlich implementiert und dokumentiert in Minisat `http://minisat.se/` (Niklas Een, Niklas Sorenson) (seit ca. 2005 sehr starker Solver)
  später übernimmt diese Rolle: Cadical, Kissat `https://fmv.jku.at/kissat/` (Armin Biere)

Aufgaben

1. Geben Sie eine erfüllbare CNF an, für die monotone lokale Suche

   ```
   improve n cnf b0 = ...
         let b1 = S.insert (negate l) $ S.delete l b0
         if badness cnf b1 <= badness cnf b0
           then improve n cnf b1
           else improve n cnf b0
   ```

   nicht funktioniert: es gibt eine Belegung b_0, von der aus *keine* zulässige Schrittfolge zu einer erfüllenden Belegung führt. Hinweis: z.B., weil es überhaupt keine erlaubten Schritte gibt.
   Wie behandeln gsat/walksat diesen Fall?
2. wenden Sie den SAT-Solver mit lokaler Suche auf

realistische CNFs an, z.B. aus Kodierung Rösselsprung.

3. wie werden in minisat (kissat, ...) Einheits- und Konfliktklauseln erkannt? (Hinweis: two watched literals)
4. unit propagation (UP) implementieren:
   - Einheitsklauseln erkennen:
     ```
     type Literal = Int
     units :: CNF -> [Literal]
     units [[1,2],[-3],[3,4]] = [-3]
     ```
   - ein Literal belegen:
     ```
     assign :: Literal -> CNF -> CNF
     assign (-3) [[1,2],[-3],[3,4]] = [[1,2],[4]]
     ```

   Wo stehen die entsprechenden Funktionen im Quelltext der Autotool-Aufgabe zu DPLL?