Formale Spezifikation und Verifikation

Wintersemester 2024

Prof. Dr. Gidon Ernst
gidon.ernst@lmu.de

Software and Computational Systems Lab
Ludwig-Maximilians-Universität München, Germany

November 29, 2024

SMT: Integration von Theorien

SAT/SMT

Satisfiability von Aussagenlogischen Formeln ϕ:
Gibt es eine Belegung s der Propositionen, sodass $s \models \phi$?

Theorie $=$ mathematischer Datentyp $+$ Operationen

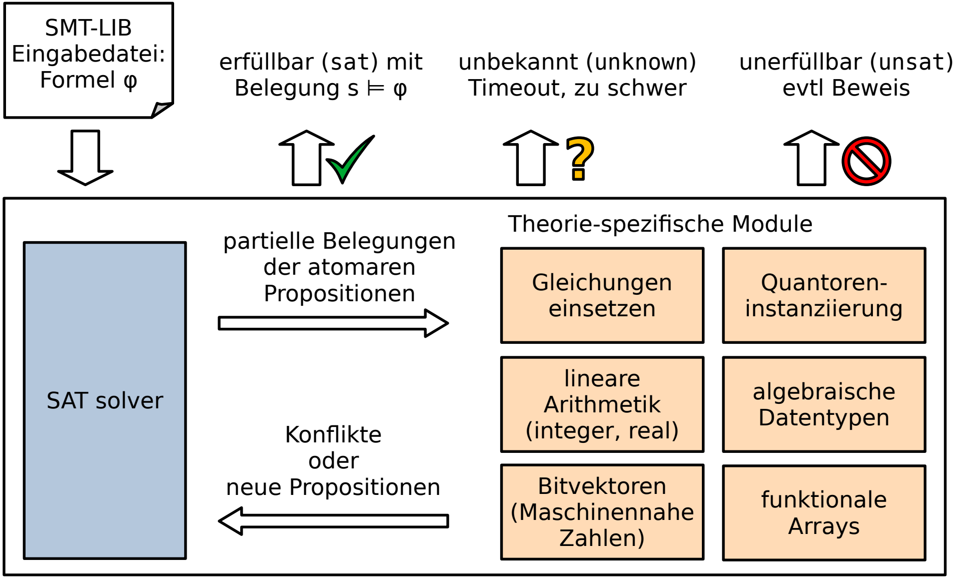
SMT $=$ Satisfiability modulo Theory (SAT $+$ Theorien)
Gibt es eine Belegung s aller Variablen, sodass $s \models \phi$?

Austauschformat: SMT-LIB `http://smtlib.cs.uiowa.edu/index.shtml`

Anwendungen, z.B. wenn Belegung $s =$ Programmzustand

- Erfüllbarkeit $=$ Erreichbarkeit von (Fehler-)Zuständen
- Allgemeingültigkeit von Implikationen $P \Rightarrow Q$ (Hoare: Konsequenzregel)

Satisfiability Modulo Theory



Theorie: Lineare Arithmetik der reellen Zahlen (LRA)

Erweiterung der Aussagenlogik um

- Variablen $x : \mathbb{R}$ und Konstanten $k \in \mathbb{R}$
- Terme $\quad t ::= x \mid k \mid -t \mid t_1 + t_2 \mid t_1 \cdot t_2$
- *lineare* Arithmetik: nur Multiplikation mit Konstanten $k \cdot t$
- Propositionen $\quad A ::= \cdots \mid t_1 = t_2 \mid t_1 < t_2 \mid t_1 \leq t_2$

Beispielproblem: lineare Gleichungssysteme

$$
\begin{aligned}
2x + 3y - z &= 0 \\
-x - y + 4 &= 0 \\
5x + y + 2z &= 0
\end{aligned}
$$

Lösungsansatz: Auflösen nach Variablen und Einsetzen
Hier genau eine Lösung: $s = \{x \mapsto -14, y \mapsto 18, z \mapsto 26\}$

Klassische Algorithmen: Gauss (nur $=$), Fourier-Motzkin, Simplex (auch $<, \leq$)

Theorie: Lineare Arithmetik der *ganzen* Zahlen (LIA)

Erweiterung der Aussagenlogik um

- Variablen $x : \mathbb{Z}$ und Konstanten $k \in \mathbb{Z}$
- Terme $t ::= x \mid k \mid -t \mid t_1 + t_2 \mid k \cdot t$
- Propositionen $A ::= \cdots \mid t_1 = t_2 \mid t_1 < t_2 \mid t_1 \leq t_2$

Schwieriger als über den reellen Zahlen, da man sicherstellen muss, dass das Ergebnis ganzzahlig ist, z.B.: $2x + 2y = 7$ hat keine ganzzahligen Lösungen

Verwandte Problemstellung “Integer Linear Programming”: konvexe Optimierung

Klassischer Algorithmus: Omega-Test

Demo: LIA `min` (SMT-LIB, Python API)

Integration von SAT und (mehreren) Theorien

Ansatz von Nelson/Oppen

- Propositionen A aus Sicht des SAT-Solvers atomar
- Theorie-Solver kennt Definition von A, z.B. $A \equiv (x = 7)$
- Theorie-Solver hat Zugriff auf partielle Belegung der Propositionen

Integration von SAT und (mehreren) Theorien

Ansatz von Nelson/Oppen

- Propositionen A aus Sicht des SAT-Solvers atomar
- Theorie-Solver kennt Definition von A, z.B. $A \equiv (x = 7)$
- Theorie-Solver hat Zugriff auf partielle Belegung der Propositionen

Sobald eine Proposition A mit true oder false belegt wird

- Theorie-Solver prüft, ob das noch konsistent ist, z.B.
  - ✗ $A \equiv (x = 7)$ und $B \equiv (x < 4)$
  - ✓ $A \equiv (x = 7)$ und $\neg B \equiv (x \geq 4)$

Integration von SAT und (mehreren) Theorien

Ansatz von Nelson/Oppen

- Propositionen A aus Sicht des SAT-Solvers atomar
- Theorie-Solver kennt Definition von A, z.B. $A \equiv (x = 7)$
- Theorie-Solver hat Zugriff auf partielle Belegung der Propositionen

Sobald eine Proposition A mit $true$ oder $false$ belegt wird

- Theorie-Solver prüft, ob das noch konsistent ist, z.B.
  - ✗ $A \equiv (x = 7)$ und $B \equiv (x < 4)$
  - ✓ $A \equiv (x = 7)$ und $\neg B \equiv (x \geq 4)$
- Hinzunahme neuer Klauseln falls Widerspruch ✗, z.B.
  - $A \implies \neg B$ bzw in KNF $(\neg A \vee \neg B)$
  - In der Zukunft "sieht" der SAT-Solver den Konflikt dann direkt (relevant falls die Situation noch einmal auftritt)

Beispiel: Satisfiability Modulo Theory

Ist die Formel erfüllbar?

$(x \geq y) \land (x < y \lor x = y)$

Abstraktion in Aussagenlogik

- $A \land (B \lor C)$ für
- $A \equiv (x \geq y) \quad B \equiv (x < y) \quad C \equiv (x = y)$

Beispiel: Satisfiability Modulo Theory

Ist die Formel erfüllbar?

$(x \geq y) \wedge (x < y \vee x = y)$

Abstraktion in Aussagenlogik

- $A \wedge (B \vee C)$ für
- $A \equiv (x \geq y) \quad B \equiv (x < y) \quad C \equiv (x = y)$

Suche nach Belegungen (KNF $\rightsquigarrow$ DPLL)

- DPLL: $A =$ `true` (UP) Theorie: $x \geq y$ ✓

Beispiel: Satisfiability Modulo Theory

Ist die Formel erfüllbar?

$(x \geq y) \wedge (x < y \vee x = y)$

Abstraktion in Aussagenlogik

- $A \wedge (B \vee C)$ für
- $A \equiv (x \geq y) \quad B \equiv (x < y) \quad C \equiv (x = y)$

Suche nach Belegungen (KNF $\rightsquigarrow$ DPLL)

- DPLL: $A = \texttt{true}$ (UP) Theorie: $x \geq y$ ✓
- DPLL: $B = \texttt{true}$ (PLE) Theorie: $x \geq y$, $x < y$ ✗

Beispiel: Satisfiability Modulo Theory

Ist die Formel erfüllbar?
$(x \geq y) \land (x < y \lor x = y)$
Abstraktion in Aussagenlogik

- $A \land (B \lor C)$ für
- $A \equiv (x \geq y) \quad B \equiv (x < y) \quad C \equiv (x = y)$

Suche nach Belegungen (KNF $\rightsquigarrow$ DPLL)

- DPLL: $A = \texttt{true}$ (UP) Theorie: $x \geq y$ ✓
- DPLL: $B = \texttt{true}$ (PLE) Theorie: $x \geq y$, $x < y$ ✗
- Neue Klausel $(\neg A \lor \neg B)$ (mit $A = \texttt{true}$ äquivalent zu $(\neg B)$

Beispiel: Satisfiability Modulo Theory

Ist die Formel erfüllbar?
$(x \geq y) \land (x < y \lor x = y)$
Abstraktion in Aussagenlogik

- $A \land (B \lor C)$ für
- $A \equiv (x \geq y) \quad B \equiv (x < y) \quad C \equiv (x = y)$

Suche nach Belegungen (KNF $\rightsquigarrow$ DPLL)

- DPLL: $A = \texttt{true}$ (UP) Theorie: $x \geq y$ ✓
- DPLL: $B = \texttt{true}$ (PLE) Theorie: $x \geq y$, $x < y$ ✗
- Neue Klausel $(\neg A \lor \neg B)$ (mit $A = \texttt{true}$ äquivalent zu $(\neg B)$
- Revidieren von $B = \texttt{true}$
- $B = \texttt{false}$ (UP aus neuer Klausel) Theorie: $x \geq y$, $\neg(x < y)$ ✓

Beispiel: Satisfiability Modulo Theory

Ist die Formel erfüllbar?
$(x \geq y) \wedge (x < y \vee x = y)$
Abstraktion in Aussagenlogik

- $A \wedge (B \vee C)$ für
- $A \equiv (x \geq y) \quad B \equiv (x < y) \quad C \equiv (x = y)$

Suche nach Belegungen (KNF $\rightsquigarrow$ DPLL)

- DPLL: $A = \texttt{true}$ (UP) Theorie: $x \geq y$ ✓
- DPLL: $B = \texttt{true}$ (PLE) Theorie: $x \geq y$, $x < y$ ✗
- Neue Klausel $(\neg A \vee \neg B)$ (mit $A = \texttt{true}$ äquivalent zu $(\neg B)$
- Revidieren von $B = \texttt{true}$
- $B = \texttt{false}$ (UP aus neuer Klausel) Theorie: $x \geq y$, $\neg(x < y)$ ✓
- $C = \texttt{true}$ (UP/PLE) Theorie: $x \geq y$, $\neg(x < y), x = y$ ✓

Beispiel: Satisfiability Modulo Theory

Ist die Formel erfüllbar?

$(x \geq y) \wedge (x < y \vee x = y)$

Abstraktion in Aussagenlogik

- $A \wedge (B \vee C)$ für
- $A \equiv (x \geq y) \quad B \equiv (x < y) \quad C \equiv (x = y)$

Suche nach Belegungen (KNF $\rightsquigarrow$ DPLL)

- DPLL: $A =$ `true` (UP) Theorie: $x \geq y$ ✓
- DPLL: $B =$ `true` (PLE) Theorie: $x \geq y$, $x < y$ ✗
- Neue Klausel $(\neg A \vee \neg B)$ (mit $A =$ `true` äquivalent zu $(\neg B)$
- Revidieren von $B =$ `true`
- $B =$ `false` (UP aus neuer Klausel) Theorie: $x \geq y$, $\neg(x < y)$ ✓
- $C =$ `true` (UP/PLE) Theorie: $x \geq y$, $\neg(x < y), x = y$ ✓

Erfüllbar

Theorie: Bitvektoren (BV)

Direkte Unterstützung von Zahlen in Maschinendarstellung

- Zweierkomplement
- Beschränkter Wertebereich (z.B. 32 Bit)
- Bit-Präzise Operationen: Addition mit Überlauf, Shifts, Bit-Xor, ...
- Analog zu Java/C: `int`/`long`, teilweise auch `float`/`double`

Theorie: Bitvektoren (BV)

Direkte Unterstützung von Zahlen in Maschinendarstellung

- Zweierkomplement
- Beschränkter Wertebereich (z.B. 32 Bit)
- Bit-Präzise Operationen: Addition mit Überlauf, Shifts, Bit-Xor, ...
- Analog zu Java/C: `int`/`long`, teilweise auch `float`/`double`

Nützlich für die Analyse von systemnahen C-Code oder Assembly, z.B.
https://graphics.stanford.edu/~seander/bithacks.html

```
r = y ^ ((x ^ y) & -(x < y)); // min(x, y)
```

(Don't try this at home: `gcc`/`clang` können das besser)

Demo: Bitvektoren `min` (SMT-LIB, Python API)

Theorie: Uninterpretierte Funktionen (UF)

- Variablen — $x\colon S$ für abstrakte Sorten S
- Konstanten — $c\colon S$
- Funktionen — $f\colon S_1 \times \cdots \times S_n \to S$
- Prädikate — $p\colon S_1 \times \cdots \times S_n$
- Terme — $t ::= x \mid f(t_1, \ldots, t_n) \mid \mathtt{ite}(\phi, t_1, t_2)$
- Propositionen — $A ::= \cdots \mid t_1 = t_2 \mid p(t_1, \ldots, t_n)$

Theorie: Uninterpretierte Funktionen (UF)

- Variablen $x\colon S$ für abstrakte Sorten S
- Konstanten $c\colon S$
- Funktionen $f\colon S_1 \times \cdots \times S_n \to S$
- Prädikate $p\colon S_1 \times \cdots \times S_n$
- Terme $t ::= x \mid f(t_1, \ldots, t_n) \mid \texttt{ite}(\phi, t_1, t_2)$
- Propositionen $A ::= \cdots \mid t_1 = t_2 \mid p(t_1, \ldots, t_n)$

Gültige Schlussregeln

- Gleichheit $=$ ist Äquivalenzrelation (reflexiv, symmetrisch, transitiv)
- Einsetzen von Gleichungen (Kongruenzregel), z.B.
  $x = y$ impliziert dass $f(x) = f(y)$ und $p(x) \iff p(y)$

Theorie: Uninterpretierte Funktionen (UF)

- Variablen $x\colon S$ für abstrakte Sorten S
- Konstanten $c\colon S$
- Funktionen $f\colon S_1 \times \cdots \times S_n \to S$
- Prädikate $p\colon S_1 \times \cdots \times S_n$
- Terme $t ::= x \mid f(t_1, \ldots, t_n) \mid \texttt{ite}(\phi, t_1, t_2)$
- Propositionen $A ::= \cdots \mid t_1 = t_2 \mid p(t_1, \ldots, t_n)$

Gültige Schlussregeln

- Gleichheit $=$ ist Äquivalenzrelation (reflexiv, symmetrisch, transitiv)
- Einsetzen von Gleichungen (Kongruenzregel), z.B.
  $x = y$ impliziert dass $f(x) = f(y)$ und $p(x) \iff p(y)$

- Oft verwendet man UF in Verbindung mit quantifizierten Axiomen
- Klassischer Algorithmus: Fast Congruence Closure [Nelson & Oppen 1980]

Beispiele: Congruence Closure

$$f(a,b) = a \Longrightarrow f(f(a,b),b) = a$$

$$f(f(f(a))) = a \land f(f(f(f(f(a))))) = a \Longrightarrow f(a) = a$$

`z3 UF/sledgehammer/Hoare/uf.615090.smt2`

Theorie: Quantoren (Q)

- Formeln: $\phi ::= \cdots \mid \forall\, x.\ \phi \mid \exists\, x.\ \phi$

Theorie: Quantoren (Q)

- Formeln: $\phi ::= \cdots \mid \forall\ x.\ \phi \mid \exists\ x.\ \phi$

Quantoreninstanziierung:

- $(\forall\ x.\ \phi) \implies \phi[x \mapsto t]$ (für beliebigen Term t)
- $\phi[x \mapsto t] \implies (\exists\ x.\ \phi)$

Theorie: Quantoren (Q)

- Formeln: $\phi ::= \cdots \mid \forall\ x.\ \phi \mid \exists\ x.\ \phi$

Quantoreninstanziierung:

- $(\forall\ x.\ \phi) \implies \phi[x \mapsto t]$ (für beliebigen Term t)
- $\phi[x \mapsto t] \implies (\exists\ x.\ \phi)$

Anwendung: Axiomatisierung von Datenstrukturen

- Mengenoperationen: $\forall\ e, s_1, s_2.\ e \in (s_1 \cup s_2) \iff (e \in s_1 \vee e \in s_2)$
- Operationen auf Sequenzen (siehe Dafny/OOP)

Wie wähle ich Instanzen t?

- E-Matching von *syntaktischen Mustern* (“Trigger”, z.B. $e \in (s_1 \cup s_2)$)
- Konstruktion von Modellen der axiomatisierten Funktionen

Beispiel: Beweise mit Quantoren

$$(\exists y_1.\forall x_1.P(x_1, y_1)) \Longrightarrow (\forall x_2.\exists y_2.P(x_2, y_2))$$

Beispiel: Axiomatisierung von Mengen

Einschub: Semantik von SMT-LIB Formeln

Wann gilt $s \models \phi$?

Ansatz: Reduktion anhand der *Grammatik* von Formeln auf *natürliche Sprache*

Einsetzen und Belegungen

Syntaktisch Einsetzen: $\phi[x \mapsto t]$

Ersetze jedes Vorkommen von Variable x in ϕ durch Term t

Ändern der Belegung: $s' = s[x \mapsto v]$

Dann $s'(y) = \begin{cases} v & \text{if } x = y \\ s(y) & \text{sonst} \end{cases}$

Der Wert für x wird überschrieben: $s'(x) = v$,
die Werte aller anderen Variablen y bleiben gleich: $s'(y) = s(y)$

Substitutionslemma

$s \models \phi[x \mapsto t] \iff s[x \mapsto [\![t]\!]_s] \models \phi$

Einsetzen von t für x entspicht der Annahme, dass x schon den Wert $[\![t]\!]_s$ von t in s hat

Beispiel: Substitutionslemma

Begründung der Quantorenregeln

Definition: $s \models \forall x.\ \phi \iff$ for all v we have $s[x \mapsto v] \models \phi$

Ziel: $s \models \big((\forall x.\ \phi) \Rightarrow \phi[x \mapsto t]\big)$

Korollar: $(\forall x.\ \phi) \iff \big((\forall x.\ \phi) \land \phi[x \mapsto t]\big)$

Beweis: Wenn $A \Rightarrow B$ dann $A \iff (A \land B)$

Theorie: Funktionale Arrays (A)

- Variablen $a\colon \texttt{Array}\langle s, s' \rangle$ für Sorten s (Domain) und s' (Range)
- Terme $t ::= a[k] \mid a[k := v]$ (falls $k\colon s$ und $v\colon s'$)
- Propositionen $A ::= \cdots \mid a_1 = a_2$

Theorie: Funktionale Arrays (A)

- Variablen $a\colon \texttt{Array}\langle s, s'\rangle$ für Sorten s (Domain) und s' (Range)
- Terme $t ::= a[k] \mid a[k := v]$ (falls $k\colon s$ und $v\colon s'$)
- Propositionen $A ::= \cdots \mid a_1 = a_2$

Gültige Schlussregeln

- $a[k := v][k'] = \texttt{ite}(k = k', v, a[k']) = \begin{cases} v & k = k' \\ a[k'] & k \neq k' \end{cases}$
- Extensionalität $(\forall k.\ a[k] = b[k]) \implies a = b$

Extensionalität ist knifflig

- Heuristik: wann anwenden?
- Unterstützung für Quantoren notwendig

Theorie: Funktionale Arrays (A)

Beispiele für gültige Formeln

- Lesen einer Modifikation:
  $a[k := v][k] = v$
- Zwei unabhängige Modifikationen kommutieren:
  $k_1 \neq k_2 \implies a[k_1 := v_1][k_2 := v_2] = a[k_2 := v_2][k_1 := v_1]$

Beispiel: Spezifikationen mit Quantoren

- ein Element ist in einem Array enthalten
- ein Element ist das Maximum in einem Array; in einer Menge

Beispiel: Modellierung von Speicher als Arrays

```
class Box { int x; }
Box p = new Box();
Box q = new Box();
```

Beispiel: Modellierung von Speicher als Arrays

```
class Box { int x; }
Box p = new Box();
Box q = new Box();




p.x = 7;




q.x = 8;




assert p.x + 1 == q.x;
```

Theorie: Algebraische Datentypen

Funktionale Datenstrukturen

- Tupel, Listen, Bäume, Enumerationen, ...
- unveränderlich (analog zu Arrays)
- Funktionen typischerweise rekursiv $\rightarrow$ Induktion (schwierig!)

Beispiel: Algebraische Datentypen

Was Sie können und wissen sollten

- DPLL Algorithmus mit Theorie-Integration ($\mathbb{Z}$, $\mathbb{R}$) auf Papier durchführen (analog zu den Beispielen)
- Wie helfen neue Klauseln der algorithmischen Suche nach Belegungen?
- Welche Problemstellungen lassen sich mit modernen SMT-Solvern lösen?
- Einfache Formalisierungen mit Quantoren/Arrays (z.B. `sorted`)

License