Anwendungen von Logik

SMT-Solver (Satisfiability modulo theories)

1. Beispiel: Verwendung eines SMT Solvers in Java
2. SMT Grundlagen
3. Beispiel: Programm-Verifikation
4. Grundlagen der Programm-Verifikation:
   Von Programmen zu Logik

SMT-Solver

Definition 1.1 (Satisfiability modulo theories)

Seien $T_1, \ldots, T_n$ Theorien erster Stufe mit jeweils unterschiedlichen Funktions- und Prädikats-Symbolen und A eine Formel über eine Signatur mit den Symbolen aus den Theorien.
Dann ist die Frage, ob $T_1 \cup \cdots \cup T_n \cup \{A\}$ erfüllbar ist, das Problem der **Erfüllbarkeit modulo Theorien** $T_1, \ldots, T_n$.

Definition 1.2 (SMT-Solver)

Ein SMT-Solver für Theorien $T_1, \ldots, T_n$ ist ein Programm, welches eine Formel A nimmt und prüft, ob diese zusammen mit den eingebauten Theorien $T_1, \ldots, T_n$ erfüllbar ist.
Der SMT-Solver muss nicht für jede Eingabe ein Ergebnis liefern. Das Problem ist im Allgemeinen unentscheidbar.

Z3 SMT-Solver

Z3 ist ein SMT-Solver.
Entwickelt von Microsoft Research (Leonardo de Moura, Nikolaj Bjørner).
Unterstützte Theorien:

- Arrays
- Bit-Vectors
- Integer Arithmetic
- Real Arithmetic
- mixed Integer Real Arithmetic
- Integer Difference Logic
- Rational Difference Logic
- ...

Z3 verwendet eine Logik mit Typen (Sorten).

Beispiel

Theorie: Ganze Zahlen mit üblichen Operatoren

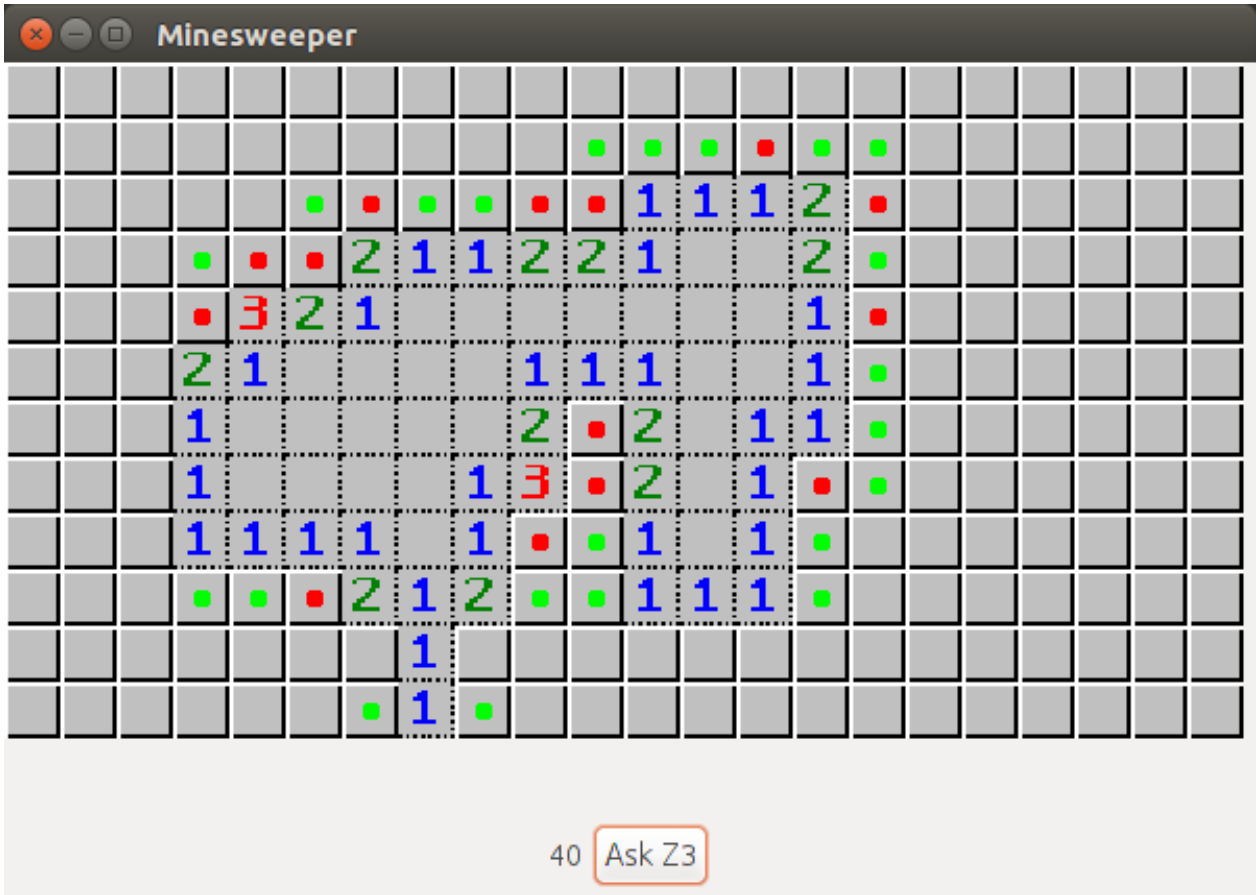
$$A \equiv x > 5 \wedge \Big(\neg(x > 3) \vee y > 2\Big) \wedge \neg(x + y > 8)$$

Z3 als Java-Bibliothek

$$A \equiv x > 5 \wedge \Big(\neg(x > 3) \vee y > 2\Big) \wedge \neg(x + y > 8)$$

```
Context c = new Context(cfg);
IntExpr x = c.mkIntConst("x");
IntExpr y = c.mkIntConst("y");
// x > 5
BoolExpr p1 = c.mkGt(x, c.mkInt(5));
// x > 3
BoolExpr p2 = c.mkGt(x, c.mkInt(3));
// y > 2
BoolExpr p3 = c.mkGt(y, c.mkInt(2));
// x+y > 8
BoolExpr p4 = c.mkGt(c.mkAdd(x, y), c.mkInt(8));
// p1 && (!p2 || p3) && !p4
BoolExpr and = c.mkAnd(
        p1,
        c.mkOr(c.mkNot(p2), p3),
        c.mkNot(p4));
Solver s = c.mkSolver();
s.add(and);
Status res = s.check();
```

Z3 Anwendung: Minesweeper KI



- Ziel: Alle sicheren Felder aufdecken.
- Verloren, wenn Miene aufgedeckt.
- Für aufgedeckte Felder: Anzeige, wie viele Mienen in den Nachbarfeldern sind.

Minesweeper KI

Zwei Funktionssymbole:

- $m(x, y)$: Anzahl der Mienen auf einem Feld.
- $s(x, y)$: Anzahl der Mienen in den Nachbarfeldern.

Spielregeln als logische Formeln:

1. Auf jedem Feld ist entweder eine oder keine Miene.
$$\forall x.\forall y.\ m(x, y) = 0 \vee m(x, y) = 1$$

2. Außerhalb des Spielfeldes sind keine Mienen.
(Beispiel: 12 Zeilen, 22 Spalten)
$$\forall x.\forall y.\ (x < 0 \vee x \geq 22 \vee y < 0 \vee y \geq 12) \rightarrow m(x, y) = 0$$

3. Zusammenhang zwischen m und s.
$$\begin{aligned}\forall x.\forall y.\ s(x, y) = {} & m(x-1, y-1) + m(x, y-1) + m(x+1, y-1) \\ & + m(x-1, y) \qquad\qquad + m(x+1, y) \\ & + m(x-1, y+1) + m(x, y+1) + m(x+1, y+1)\end{aligned}$$

Minesweeper KI

	1	
	1	
		?

Frage: Ist Zelle unten rechts sicher?

$$\Gamma,\ m(1,0)=0,\ s(1,0)=1,\ m(1,1)=0,\ s(1,1)=1 \overset{?}{\models} m(2,2)=0$$

Wobei Γ die Spielregeln der vorherigen Folie sind.

Äquivalent:
Ist
$$\Gamma \cup \Big\{ m(1,0)=0,\ s(1,0)=1,\ m(1,1)=0,\ s(1,1)=1,\ \neg\big(m(2,2)=0\big) \Big\}$$
unerfüllbar?

Diese Frage kann der SMT-Solver Z3 beantworten.

Implementierung

Java-Implementierung:
https://github.com/peterzeller/java-simple-mine-sweeper

Basierend auf Minesweeper-Implementierung von Syohei Yoshida:
https://github.com/syohex/java-simple-mine-sweeper.

Python Implementierung und weitere Anwendungen von Z3 (in Python):
https://yurichev.com/writings/SAT_SMT_draft-EN.pdf
https://github.com/dennis714/SAT_SMT_article/blob/master/SMT/minesweeper/minesweeper_solver.py

SMT Grundlagen: T-Conflict

Ein Solver für eine Theorie stellt die Funktion **T-Conflict** bereit.

Eingabe: Konjunktion von Literalen:
$L_1 \wedge \cdots \wedge L_n$

Ausgabe: Erfüllbar oder unerfüllbar? (oder "unknown")
Falls unerfüllbar: unerfüllbare Teilmenge (möglichst klein)

Beispiel:
$x > 0 \ \wedge \ x + y > 10 \ \wedge \ 2 * x < y \ \wedge \ y \leq 6$

Nicht erfüllbar, es muss gelten:
$\neg(x + y > 10) \ \vee \ \neg(2 * x < y) \ \vee \ \neg(y \leq 6)$

Kombination mit Davis-Putnam-Algorithmen

$$x > 5 \wedge \Big(\neg(x > 3) \vee y > 2\Big) \wedge \neg(x + y > 8)$$

$$p_1 \quad \wedge \ (\quad \neg p_2 \quad \vee \quad p_3 \ ) \ \wedge \quad \neg p_4$$

$$\Big| \ p_1 := 1$$

$$(\neg p_2 \vee p_3) \wedge \neg p_4$$

$$\Big| \ p_2 := 0$$

$$\neg p_4$$

$$\Big| \ p_4 := 0$$

$$1$$

Anfrage an Theory-Solver: $p_1 \wedge \neg p_2 \wedge \neg p_4$ erfüllbar?
Antwort: Nein, es muss $\neg p_1 \vee p_2$ gelten.

Kombination mit Davis-Putnam-Algorithmen (Iteration 2)

$$x > 5 \wedge \Big(\neg(x > 3) \vee y > 2\Big) \wedge \neg(x + y > 8)$$

$$p_1 \wedge (\neg p_2 \vee p_3) \wedge \neg p_4 \wedge (\neg p_1 \vee p_2)$$

$$\Big| \; p_1 := 1$$

$$(\neg p_2 \vee p_3) \wedge \neg p_4 \wedge p_2$$

$$\Big| \; p_2 := 1$$

$$p_3 \wedge \neg p_4$$

$$\Big| \; p_3 := 1$$

$$\neg p_4$$

$$\Big| \; p_4 := 0$$

$$1$$

Theorie-Lemma
wird hinzugefügt.

Anfrage an Theory-Solver:
$p_1 \wedge p_2 \wedge p_3 \wedge \neg p_4$ erfüllbar?

Antwort: Nein, es muss gelten:
$\neg p_1 \vee \neg p_3 \vee p_4$

Kombination mit Davis-Putnam-Algorithmen (Iteration 3)

$$x > 5 \wedge \Big(\neg(x > 3) \vee y > 2\Big) \wedge \neg(x + y > 8)$$

$$p_1 \wedge (\neg p_2 \vee p_3) \wedge \neg p_4 \wedge (\neg p_1 \vee p_2) \wedge (\neg p_1 \vee \neg p_3 \vee p_4)$$

$$\Big| \; p_1 := 1$$

$$(\neg p_2 \vee p_3) \wedge \neg p_4 \wedge p_2 \wedge (\neg p_3 \vee p_4)$$

$$\Big| \; p_2 := 1$$

$$p_3 \wedge \neg p_4 \wedge (\neg p_3 \vee p_4)$$

$$\Big| \; p_3 := 1$$

$$\neg p_4 \vee p_4$$

$$\Big| \; p_4 := 0$$

$$0$$

Formel unerfüllbar!

Verfahren

Für eine Formel ohne Quantoren in KNF Form:

- Betrachte Literale und verwende aussagenlogischen Erfüllbarkeits-Check.
- Wenn Formel aussagenlogisch unerfüllbar, dann fertig.
  Falls erfüllende Belegung gefunden, Anfrage an Theorie-Solver.
- Falls erfüllbar in Theorie: Formel erfüllbar.
  Andernfalls füge Theorie-Lemma von Theorie-Solver zur Formel hinzu.

Terminierung:

- Menge der Literale ist endlich
  $\Rightarrow$ Menge der erzeugten Klauseln endlich.
  $\Rightarrow$ Wenn Theorie-Solver immer terminiert, dann terminiert das Verfahren.

Weiterführende Mechanismen

Für einen SMT-Solver sind in der Praxis noch weitere Mechanismen notwendig.

- Davis–Putnam–Logemann–Loveland (DPLL) Algorithmus (Weiterentwicklung des Davis-Putnam-Verfahrens aus der Vorlesung)
- Früheres Aufrufen des Theorie-Solvers.
- T-Propagation: Wenn bereits Bewertung für Literale $L_1 \dots L_n$ gewählt wurde und Literal L in der Formel vorkommt und $T \cup \{L_1, \dots, L_n\} \models L$, dann kann L gewählt werden (kein Anwenden der Split-Regel nötig).
- Formeln umschreiben, sodass ein Literal jeweils nur die Signatur einer Theorie verwendet. (Verknüpfung durch Gleichheits-Einschränkungen)
- ...

Quantoren

- $\exists$-Quantoren durch Skolemisierung eliminieren.
- $\forall$-Quantoren *intelligent* instantiieren.
  - Heuristiken:
    Beispiel: Für Formel $\forall x.\ A$, wähle Term t, so dass $A\{x/t\}$ viele bereits bekannte Terme enthält.
  - Hilfe von Benutzer durch Angabe von *Triggern*.
  - Superposition Calculus:
    Verallgemeinerung der prädikatenlogischen Resolution zu Formeln mit Gleichheit

Zusammenfassung: SMT-Solver

- Kombination von Logik mit praktischen Algorithmen für bestimmte Theorien.
- Intern: Verwendet Weiterentwicklung von Davis-Putnam und Resolution.
- Können als allgemeiner Algorithmus verwendet werden um Probleme zu lösen.
  Insbesondere: Probleme aus NP.
- Überwiegende Anwendung: Programmanalyse, Programmverifikation

Programm-Verifikation

- Vor dem Ausführen eines Programms ($\Rightarrow$ statisch)
- Prüfen, ob ein Programm bestimmte Eigenschaften hat.
  Beispielsweise:
  - Das Programm wirft keine `Nullpointer`-Exceptions
  - Das Programm terminiert für alle Eingaben.
  - Das Programm verbraucht nie mehr als 100MB Arbeitsspeicher.
  - Die Methode `m` verhält sich für alle gültigen Eingaben wie spezifiziert.
- Tool-Support notwendig
  (von Hand zu aufwendig und zu unzuverlässig)

Beispiel: Dafny

Dafny ist eine Programmiersprache mit Unterstützung für Spezifikation.

Der **Dafny Static-Program-Verifier** ist ein Tool um **funktionale Korrektheits-Eigenschaften** eines Dafny-Programms zu verifizieren.

Entwickelt von Microsoft Research (Rustan Leino).

Dafny Beispiel: Minimum von 3 Zahlen

```
1  method min(x: int, y: int, z: int) returns (r: int)
2  ensures r <= x;
3  ensures r <= y;
4  ensures r <= z;
5  {
6      if (x < y && x < z) {
7        r := x;
8      } else if (y < x && y < z) {
9        r := y;
10     } else {
11       r := z;
12     }
13 }
```

```
$ Dafny.exe min.dfy
min.dfy(10,11): Error BP5003: A postcondition might not hold on this
    return path.
min.dfy(2,10): Related location: This is the postcondition that might
    not hold.
```

http://rise4fun.com/Dafny/y7D5

Dafny Beispiel: Minimum aus Array

```
1  method min(ar: array<int>) returns (min: int)
2    requires ar != null;
3    requires ar.Length > 0;
4    ensures forall i :: 0 <= i < ar.Length ==> min <= ar[i];
5    ensures exists i :: 0 <= i < ar.Length && ar[i] == min;
6  {
7      min := ar[0];
8      var pos := 1;
9      while (pos < ar.Length)
10       decreases ar.Length - pos;
11       invariant pos <= ar.Length;
12       invariant forall i :: 0 <= i < pos ==> min <= ar[i];
13       invariant exists i :: 0 <= i < pos && ar[i] == min;
14     {
15         if (ar[pos] < min) {
16             min := ar[pos];
17         }
18         pos := pos + 1;
19     }
20 }
```

http://rise4fun.com/Dafny/s7BI

Begriffsklärung: Vor-, Nachbedingung

Prozedureigenschaften lassen sich durch Vor- und Nachbedingungen beschreiben:

- Die **Vorbedingung** formuliert Anforderungen an den Vorzustand; wenn die Vorbedingung gilt, muss die Prozedur ohne Fehler terminieren.
- Die **Nachbedingung** formuliert die Eigenschaften des Nachzustands
  - $\rightarrow$ in Abhängigkeit vom Vorzustand (z.B. Parameterwerte);
  - $\rightarrow$ unter der Voraussetzung, dass beim Aufruf die Vorbedingung gilt.

Begriffsklärung: Prozedurspezifikation

Eine **Prozedurspezifikation** besteht aus:

- einer Vorbedingung: **`requires`** `<Ausdruck>`
- einer Variablenliste: **`modifies`** `<Liste von Variablen>`
- einer Nachbedingung: **`ensures`** `<Ausdruck>`

Eine Prozedur darf nur die globalen Variablen und referenzierten Objekte / Arrays verändern, die in der Variablenliste aufgeführt sind.

Partielle Korrektheit

Definition 1.3 (Partielle Korrektheit)

Wenn c eine Anweisung mit Vorbedingung P und Nachbedingung Q ist, dann nennen wir c **partiell korrekt**, wenn für alle Programm-Zustände s_1 mit $P(s_1)$ gilt: Wenn die Ausführungen von Anweisung c startend im Zustand s_1 in einem Zustand s_2 enden kann, dann gilt $Q(s_1, s_2)$.

Formal:

$$\forall s_1, s_2.\ exec(c, s_1, s_2) \land P(s_1) \rightarrow Q(s_2)$$

Wobei $exec(c, s_1, s_2)$ heißt, dass das Ausführen von Anweisung c in Zustand s_1 in Zustand s_2 enden kann.

Totale Korrektheit

Definition 1.4 (Totale Korrektheit)

Wenn c eine Anweisung mit Vorbedingung P und Nachbedingung Q ist, dann nennen wir c **(total) korrekt**, wenn für alle Programm-Zustände s_1 mit $P(s_1)$ gilt: Wenn die Anweisung c im Zustand s_1 ausgeführt wird, dann terminiert die Ausführung in einem Zustand s_2 für den gilt $Q(s_1, s_2)$.

Formal:

$$\forall s_1, s_2.\ P(s_1) \rightarrow (\mathit{terminates}(c, s_1) \land \forall s_2.\ \mathit{exec}(c, s_1, s_2) \rightarrow Q(s_2))$$

Wobei $\mathit{exec}(c, s_1, s_2)$ wie zuvor definiert ist und $\mathit{terminates}(c, s_1)$ aussagt, dass die Ausführung von c im Zustand s_1 immer terminiert.

Von Programm und Spezifikation zu logischer Formel

Idee: Schwächste Vorbedingung (engl.: weakest precondition) berechnen: Berechne aus Nachbedingung und Anweisung, was als Vorbedingung notwendig ist.

Prüfe dann, ob die spezifizierte Vorbedingung die tatsächliche impliziert.
$\Rightarrow$ SMT-Solver verwenden (Dafny verwendet Z3)

Notation: Für eine Anweisung c und Nachbedingung Q ist $wp(c, Q)$ die schwächste Vorbedingung, so dass c korrekt ist.

wp: Zuweisungen

Zuweisung:
$wp(x := E, Q) = Q\{x/E\}$

Beispiele:
$wp(x := 2 * y, (x > 0 \wedge x < 10)) = (2 * y > 0 \wedge 2 * y < 10)$
$wp(x := x + 1, x > 0) = x + 1 > 0$

Einige Aspekte wurden hier ignoriert:
Bei der Substitution muss eventuell der Ausdruck E in die Logik übersetzt werden.
Außerdem haben Ausdrücke eventuell auch Vorbedingungen, wie zum Beispiel, dass der Index bei einem Array-Zugriff im gültigen Bereich ist.

wp: Sequenz von Anweisungen

$wp(c_1; c_2, Q) = wp(c_1, wp(c_2, Q))$

Beispiel:

$wp(t := a; a := b; b := t, a = b_{old} \wedge b = a_{old})$
$= wp(t := a, wp(a := b; b := t, a = b_{old} \wedge b = a_{old}))$
$= wp(t := a, wp(a := b, wp(b := t, a = b_{old} \wedge b = a_{old})))$
$= wp(t := a, wp(a := b, a = b_{old} \wedge t = a_{old}))$
$= wp(t := a, b = b_{old} \wedge t = a_{old})$
$= (b = b_{old} \wedge a = a_{old})$

Notation: Assertions für Zwischenschritte

```
1  class C {
2      var a: int;
3      var b: int;
4
5      method swap()
6        modifies this;
7        ensures a == old(b) && b == old(a);
8      {
9          var t: int;
10
11         assert b == old(b) && a == old(a);
12         t := a;
13         assert b == old(b) && t == old(a);
14         a := b;
15         assert a == old(b) && t == old(a);
16         b := t;
17         assert a == old(b) && b == old(a);
18     }
19 }
```

http://rise4fun.com/Dafny/dIILF

Dafny Beispiel: Binärsuche

```
1  method binarysearch(x: int, ar: array<int>) returns (result: int)
2    requires sorted(ar);
3    ensures result >= 0 ==> result < ar.Length && ar[result] == x;
4    ensures (exists i: int :: i>= 0 && i<ar.Length && ar[i] == x)
5          <==> (result >= 0);
6  {
7      var start := 0;
8      var end := ar.Length - 1;
9      while (end >= start) {
10         var mid := start + (end - start)/2;
11         if (x < ar[mid]) {
12             end := mid - 1;
13         } else if (x > ar[mid]) {
14             start := mid + 1;
15         } else {
16             result := mid;
17             return;
18         }
19     }
20     result := -1;
21 }
```

Dafny Beispiel: Binärsuche

Zur Verifikation in Dafny werden Annotationen benötigt:

```
...
while (end >= start)
  decreases 1 + end - start;
  invariant start >= 0;
  invariant end < ar.Length;
  invariant end >= start-1;
  invariant forall i: int :: i>= 0 && i<ar.Length && ar[i] == x
                  ==> start <= i && i <= end;
{
  ...
```

- Schleifen-Invariante (**invariant**)
- Maß für Terminierung (**decreases**)

http://rise4fun.com/Dafny/Re4T

Seminar WS 17/18

Themen:

- SMT solvers
- Generating verification conditions
- The Dafny and Boogie verification tools
- Property based testing
- Concolic testing

Mehr Informationen:
`https://softech.cs.uni-kl.de/homepage/de/teaching/WS17/seminar/`

Material

Leonardo Mendonça de Moura, Nikolaj Bjørner:
Satisfiability Modulo Theories: Introduction and Applications.

Z3 SMT solver
`https://github.com/Z3Prover/z3`

Dafny Verfification tool
`https://github.com/Microsoft/dafny`

Klausur

- Abschlussklausur: 11. September
- Lernraum: 28. August bis 8. September
  Teilweise mit Ansprechpartner (Zeiten werden noch angekündigt)
- Altklausuren von Prof. Meyer und Prof. Madlener im KAI
- Gleiche Regeln wie bei Zwischenklausur:
  - Ein DIN A4 Blatt mit eigenen, handschriftlichen Notizen (beidseitig) erlaubt