SAT-Solving und Anwendungen

SAT-Solving in der Praxis: Softwareverifikation mit Bounded Model Checking

Prof. Dr. Wolfgang Küchlin
Dipl. Inform. Christoph Zengler

Universität Tübingen

13. Mai 2009

Softwareverifikation - 1

Definition (Softwareverifikation)

Beweis der Korrektheit eines Programms mit mathematischen Methoden.

Einschränkungen:

- Es kann nur verifiziert werden, was vorher spezifiziert wurde, d.h. es muss eine formale Beschreibung vorliegen, was eine bestimmte Methode / ein bestimmter Algorithmus tun soll
- Bei falscher Spezifikation bringt auch die Verifikation nichts
- Verifikation auf Sourcecode Ebene: Selbst ein korrektes Programm kann z.B. durch das Linken inkorrekter externer Libraries, einen fehlerhaften Compiler oder einen Fehler im Betriebssystem nicht korrekt ausgeführt werden

Softwareverifikation - 2

Unterschiede zu den Überprüfungen, die eine IDE oder ein Compiler vornimmt:

- **Syntaxchecks:** Kann ein Programm überhaupt compiliert werden (genügt es der Grammatik einer bestimmten Programmiersprache)
- **Typchecks:** Stimmen die Typen im Programm überein (erwartet eine Methode ein Objekt der Klasse "Student", kann kein Objekt der Klasse "Stuhl" übergeben werden)
- Sind alle Felder, Methoden, etc. vorhanden und eindeutig
- Aber keine Überprüfung, ob ein Programm / eine Methode auch das tut, für was sie geschrieben wurde

Softwareverifikation - 3

Beispiel (Compilerchecks vs. Verifikation)

```
// Soll das Maximum von drei Zahlen berechnen
public int max3(int a, int b, int c) {
    if (a > b) {
        return a;
    } else {
        return c;
    }
}
```

- Compiliert problemlos (kein Syntax- oder Typfehler)
- Ist semantisch falsch bezüglich folgender Spezifikation

$$max3(a, b, c) \geq a \wedge max3(a, b, c) \geq b \wedge max3(a, b, c) \geq c$$

- Dann soll Verifikation einen Fehler melden (und einen Pfad zu einem Fehler ausgeben, z.B. $a = 2, b = 5, c = 3$)

Softwareverifikation - 4

Viele verschiedene Verfahren zur Softwareverifikation (hier nur einige:)

- Software Bounded Model Checking (SBMC)
  - übersetzt Code in Formel der Booleschen Algebra
  - Verifiziert mit SAT-Solving
  - dicht am Quellcode, weitgehend automatisiert
  - *bounded*: setzt (beliebige aber feste) Grenze für Schleifen / Rekursionen
- Theorembeweiser
  - Manuelle Spezifikation mit (höherer) Logik (1. Stufe, 2. Stufe)
  - Deduktion von Nachbedingungen
  - Ausdrucksstark, bedarf menschlicher Begleitung
- Hoare-Logik mit Theorembeweiser
- Konventionelle Statische Codeanalyse
- ...

(Software) Bounded Model Checking (SBMC) - 1

Idee

Übersetze Code + Verifikationsbedingung in Aussagenlogische Formel
Suche Gegenbeispiel mit modernem SAT-Solver

Bemerkung

Begrenzung der Ausführungspfade liefert endlich große Formel
Wickle Schleifen und rekursive Funktionsaufrufe bis zu einer Grenze (Bound) k ab.
Das reicht, um viele Fehler zu finden

Mögliche Ausgaben:

- Programm entspricht der Spezifikation
- Programm entspricht der Spezifikation nicht
- Grenze k ist zu klein (dann muss k größer gewählt werden und das Verfahren wiederholt werden)

(Software) Bounded Model Checking (SBMC) - 2

Erkenntnis für Praxis

(Automatisch) Fehler finden ist wichtiger als (manuelle) Verifikation

- Debugging dauernd – Verifikation nur einmal
- Software Entwickler sind keine Logiker
- Verifikation ist sowieso unmöglich (mangels Spezifikation der Funktion / des Moduls / des Programms / des Gesamtsystems))

Symbolic debugging

Debugging durch symbolische Programm-Analyse zur Compilezeit

- Neue Methode des statischen Debugging
- Findet echte Fehler in echtem Code (Linux Device Drivers, Automotive Software, zOS, ...)
- ziemlich automatisch (dank moderner SAT-Solver Technologie)
- bessere SAT-Solver $\rightarrow$ stärkeres SBMC auf größeren Programmen

Vom Programm zur Formel - 1

Schritt 1

- Präprozessor auf dem Code ausführen (z.B. `#define` entpacken)
- `break` und `continue` durch passende `goto`-Statements ersetzen
- `for`-Schleifen und `do-while`-Schleifen durch äquivalente `while` Schleifen ersetzen

Beispiel (for-Schleife)

```
for (int i=0; i<3; i++) {
    a[i] = i*i;
}
```

wird konvertiert zu:

```
int i=0;
while (i < 3) {
    a[i] = i*i;
    i++;
}
```

Vom Programm zur Formel - 2

Schritt 2

Schleifen werden "abgewickelt"

- Ersetze `while` durch `if`
- Kopiere Schleife maximal k mal (Bound k) hintereinander
- Prüfe Bound durch nachfolgende *unwinding assertion*
  (negierte Schleifenbedingung - sollte die Assertion false sein, so brauchen wir zur Verifikation eine höhere Bound k)

Beispiel (Loop unwinding mit Bound $k = 2$)

```
int i = 0;
while (i < 3) {
    a[i] = i*i;
    i++;
}
```

```
int i = 0;
if (i < 3) { // Kopie 1
    a[i] = i*i;
    i++;
}
if (i < 3) { // Kopie 2
    a[i] = i*i;
    i++;
}
assert !(i < 3) // unwinding assertion
```

Vom Programm zur Formel - 3

Schritt 3

Funktionsaufrufe werden ähnlich den `while`-Schleifen abgewickelt

- Funktionsaufrufe werden expandiert (Funktionsaufruf durch Funktionsinhalt ersetzt)
- Rekursive Funktionen werden k mal abgewickelt
- Assertion, dass Rekursionstiefe nicht tiefer als k sein kann
- `return` Statements werden durch Variablenzuweisung und `goto` zum Ende der Funktion ersetzt.

Beispiel (Function unwinding)

```
private int max(int a, int b) {
    if (a > b)
        return a;
    else
        return b;
}

public main() {
    int max12 = max(1,2);
}
```

```
public main() {
    int temp_val;
    if (1 > 2)
        temp_val = 1;
    else
        temp_val = 2;
    int max12 = temp_val;
}
```

Vom Programm zur Formel - 4

Programm besteht nun nur noch aus:

- (möglicherweise verschachtelten) `if-then-else` Blöcken
- Zuweisungen
- Assertions
- `goto`-Statements mit zugehörigen labels

Schritt 4

Programm wird in Single Static Assignment Form gebracht

Bounded Model Checking - SSA 1

Single Static Assignment Form

Spezielle Darstellungsform von Programmen: Jede Variable wird genau einmal zugewiesen. Vorhandene Variablen werden in verschiedene Versionen aufgespalten.

- Kommt aus dem Compilerdesign
- Erlaubt Vereinfachungen für den Compiler

Beispiel (Vereinfachungen)

```
int x = 1;
x = 2;
int y = x;
```

- Erste Zuweisung ist nicht nötig

Bouned Model Checking - SSA 2

Beispiel (Vereinfachungen - Fortsetzung)

```
int x = 1;
x = 2;
int y = x;
```

Konvertierung in SSA:

$x_1 = 1$

$x_2 = 2$

$y_1 = x_2$

Offensichtlich (auch für den Compiler):

- x_1 wird nie verwendet (kommt nie auf einer rechten Seite vor)

$\rightarrow$ Erste Zuweisung überflüssig

Bounded Model Checking - SSA 3

Übersetzen von if-then-else in SSA

```
if (x == 1) {
    y = 5;
} else {
    y = 42;
}
```

Übersetzung in SSA:
$y_1 = (x == 1)\ ?\ 5 : 42$

Übersetzen von if-then in SSA

```
y = 3;
if (x == 1) {
    y = 5;
}
```

Übersetzung in SSA:
$y_1 = 3$
$y_2 = (x == 1)\ ?\ 5 : y_1$

SSA - Beispiel

Beispiel (Programm in SSA bringen)

`x = x + y;`	`x_1 = x_0 + y_0;`
`if (x != 1)`	
`  x = 2;`	`x_2 = 2;`
`else`	
`  x++;`	`x_3 = x_1 + 1;`
	`x_4 = (x_1 != 1) ? x_2 : x_3;`
`assert (x <= 3);`	`assert (x_4 <= 3);`

Beispiel (SSA zu Bitvektorgleichung)

$C := x_1 = x_0 + y_0 \wedge$
$x_2 = 2 \wedge$
$x_3 = x_1 + 1 \wedge$
$x_4 = (x_1 != 1) ? x_2 : x_3$
$P := x_4 \leq 3$

letzter Schritt: Bitvektoren expandieren und Formel $C \wedge \neg P$ in CNF konvertieren

Komplettes Beispiel - 1

Beispiel (Java Programm)

```
x = 3;
if (x < 4) {
    y = 2;
} else {
    y = 5;
}
assert (y < 4)
```

Beispiel (Programm in SSA)

```
x_0 = 3;
y_0 = 2;
y_1 = 5;
y_2 = (x_0 < 4) ? y_0 : y_1;
assert (y_2 < 4)
```

Komplettes Beispiel - 2

Beispiel (SSA in Bitvektorgleichung)

$C := x_0 = 3 \wedge$
$y_0 = 2 \wedge$
$y_1 = 5 \wedge$
$y_2 = (x_0 < 4)?y_0 : y_1$
$P := y_2 < 4$

Expandieren der Bitvektoren für eine 32-Bit Architektur

- Jede Variable x wird expandiert zu 32 Variablen $x[31]...x[0]$
- $x_0 = 3$ wird zu $x_0[0] \wedge x_0[1] \wedge \neg x_0[2] \wedge ... \wedge \neg x_0[31]$
- $y_0 = 2$ wird zu $\neg y_0[0] \wedge y_0[1] \wedge \neg y_0[2] \wedge ... \wedge \neg y_0[31]$
- $y_0 = 5$ wird zu $y_0[0] \wedge \neg y_0[1] \wedge y_0[2] \wedge \neg y_0[3] \wedge ... \wedge \neg y_0[31]$
- $x_0 < 4$ wird zu $x_0[31] \vee (\neg x_0[31] \wedge ... \wedge \neg x_0[2])$
- $a = b?c : d$ wird zu $(b \wedge (a = c)) \vee (\neg b \wedge (a = d))$
- $y_2 = (x_0 < 4)?y_0 : y_1$ wird zu
  $((x_0[31] \vee (\neg x_0[31] \wedge ... \wedge \neg x_0[2])) \wedge \neg y_2[0] \wedge y_2[1] \wedge \neg y_2[2] \wedge ... \wedge \neg y_2[31]) \vee$
  $(\neg(x_0[31] \vee (\neg x_0[31] \wedge ... \wedge \neg x_0[2])) \wedge y_2[0] \wedge \neg y_2[1] \wedge y_2[2] \wedge \neg y_2[3] \wedge ... \wedge \neg y_2[31])$
- $y_2 < 4$ wird zu $y_2[31] \vee (\neg y_2[31] \wedge ... \wedge \neg y_2[2])$

Komplettes Beispiel - 3

- $x_0 = 3$ wird zu $x_0[0] \wedge x_0[1] \wedge \neg x_0[2] \wedge ... \wedge \neg x_0[31]$
- $y_2 = (x_0 < 4)?y_0 : y_1$ wird zu
  $((x_0[31] \vee (\neg x_0[31] \wedge ... \wedge \neg x_0[2])) \wedge \neg y_2[0] \wedge y_2[1] \wedge \neg y_2[2] \wedge ... \wedge \neg y_2[31]) \vee$
  $(\neg(x_0[31] \vee (\neg x_0[31] \wedge ... \wedge \neg x_0[2])) \wedge y_2[0] \wedge \neg y_2[1] \wedge y_2[2] \wedge \neg y_2[3] \wedge ... \wedge \neg y_2[31])$
- $y_2 < 4$ wird zu $y_2[31] \vee (\neg y_2[31] \wedge ... \wedge \neg y_2[2])$

Codiere $C \wedge \neg P$

Beispiel (Endformel)

$F := (x_0[0] \wedge x_0[1] \wedge \neg x_0[2] \wedge ... \wedge \neg x_0[31]) \wedge$
$((x_0[31] \vee (\neg x_0[31] \wedge ... \wedge \neg x_0[2])) \wedge \neg y_2[0] \wedge y_2[1] \wedge \neg y_2[2] \wedge ... \wedge \neg y_2[31]) \vee$
$(\neg(x_0[31] \vee (\neg x_0[31] \wedge ... \wedge \neg x_0[2])) \wedge y_2[0] \wedge \neg y_2[1] \wedge y_2[2] \wedge \neg y_2[3] \wedge ... \wedge \neg y_2[31])) \wedge$
$\neg(y_2[31] \vee (\neg y_2[31] \wedge ... \wedge \neg y_2[2]))$

- Ist F erfüllbar, so gibt es eine Belegung, so dass das Programm C gilt, und gleichzeitig die Assertion P verletzt ist (wegen $\neg P$), d.h. die Verifikation schlägt fehl
- Die Belegung stellt auch gleichzeitig einen Pfad zu einem Fehler dar
- Ist F unerfüllbar, so gibt es keinen Pfad zu einer Verletzung der Assertion, und daher ist die Verifikation erfolgreich