SAT-Solving und Anwendungen

Software Verifikation mit Bounded Model Checking

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

Universität Tübingen

15. Mai 2012

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Fahrplan für heute

1. **Einführung Software-Verifikation**
   - Was ist Software Verifikation?
   - Warum Software Verifikation?
2. **Ein formales Modell für Software**
   - Wie kann Software formalisiert werden?
   - Wie kann man Verifikationsbedingungen und Gegenbeispiele formalisieren?
3. **Bit-Vektoren**
   - Was sind Bit-Vektoren?
   - Wie kann man Bit-Vektor Gleichungen entscheiden?
4. **Bounded Model Checking (BMC)**
   - Wie funktioniert BMC?
   - Beispiele für BMC?
   - Und in der echten Welt?

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

Allerdings:

- Es gibt viele universelle Spezifikationen: keine Null-Pointer Dereferenzierung, kein Array-Index-Überlauf, nur positive Werte in *unsigned* Variablen ...
- Compiler-Fehler durch Verifikation des Zwischencodes finden.

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, was sie soll (Semantik)

Softwareverifikation — 3

Beispiel (Compilerchecks vs. Verifikation)

```
// Soll das Maximum von drei Zahlen berechnen
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 aussagenlogische Formel
  - Verifiziert mit SAT-Solving
  - Dicht am Quellcode und an der hardwareseitigen Implementierung, 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
- ...

Motivation

Warum Software-Verifikation?

Immer mehr Software...

- ... in sicherheitskritischen Bereichen, in denen ein Fehler Millionen (z.B. Kreditkartenfirmen, Banken,...) oder auch Menschenleben (Bremssysteme, Flugsteuerung,...) kosten kann
- ..., die nicht einfach mit Updates versehen werden kann (z.B. Motorensteuerung, Waschmaschine, Satellit,...)

Die Korrektheit von Software ist ein Thema seit es Programme gibt:

- *Checking a Large Routine*, A.M. Turing, 1949
- *Assigning Meanings to Programs*, R. W. Floyd, 1967
- *An Axiomatic Basis for Computer Programming*, C. A. R. Hoare, 1969

Standpunkt:

- Programme sind eine Reihe von Instruktionen, die die Werte einer gegebenen Menge an Variablen verändern
- Diese Variablen werden z.B. als ganze oder reelle Zahlen modelliert

Wie sieht es in der Realität aus?

Realität

Die meisten hardware-nahen Programmiersprachen haben vordefinierte Datentypen mit Bit-Vektor Arithmetik, d.h.

- beschränkter Wertebereich, je nach Anzahl allokierter Bits
- Überläufe bei Überschreiten des Wertebereichs sind möglich

Auf Hardware-Ebene vor allem bitweise Operationen, z.B. Shift, Und, Oder,...

Beispiel (Absolutbetrag)

Mathematisch

$$\text{abs}(a) = \begin{cases} a & \text{falls } a \geq 0 \\ -a & \text{sonst} \end{cases}$$

Programmatisch

```
short abs(short a) {
  if (a >= 0)
    return a;
  else
    return -a;
}
```

Mathematisch korrekt, programmatisch gibt es jedoch bei -32768 ein Problem.

Was wäre also besser?

Besser...

Hardwarenahe Modellierung von Programmen mit Bit-Vektoren und dadurch genauere Übereinstimmung von Modell und Realität

- weniger *False Positives*: Es werden weniger Fehler gemeldet, die zwar mathematisch ein Problem sind, jedoch nicht programmatisch
- weniger *False Negatives*: Es werden weniger Fehler übersehen, die zwar mathematisch korrekt sind, jedoch programmatisch ein Problem sind

Bit-Vektoren:

- Vektor (typischerweise der Länge 16, 32 oder 64) von Bits
- können mit Aussagenvariablen modelliert werden
- Bit-Vektor Operationen können mit Booleschen Funktionen modelliert werden
- Verifikations- bzw. Fehlerproblem kann als SAT Instanz kodiert werden

Ist das Fehlerproblem unerfüllbar, so gab es keinen Fehler im Programm, ist das Fehlerproblem erfüllbar, so gab es einen Fehler im Programm und die erfüllende Belegung der SAT Instanz gibt einen möglichen Pfad zu diesem Fehler an.

Fahrplan für heute: Wo stehen wir?

1. **Einführung Software-Verifikation ✓**
   - SW-Verifikation: Korrektheit eines Programms mit mathematischen Methoden beweisen
   - Motivation: Immer mehr sicherheitskritische Software
2. **Ein formales Modell für Software**
   - Wie kann Software formalisiert werden?
   - Wie kann man Verifikationsbedingungen und Gegenbeispiele formalisieren?
3. **Bit-Vektoren**
   - Was sind Bit-Vektoren?
   - Wie kann man Bit-Vektor Gleichungen entscheiden?
4. **Bounded Model Checking (BMC)**
   - Wie funktioniert BMC?
   - Beispiele für BMC?
   - Und in der echten Welt?

Ein formales Softwaremodell — 1

Üblicherweise wird Software als Zustandsübergangssystem (*Transition System*) modelliert.

Transition System

Ein Transition System ist ein Drei-Tupel $M = (S, S_0, R)$, mit

- S einer Menge an Zuständen,
- $S_0 \subseteq S$ einer Menge an Startzuständen und
- $R \subseteq S \times S$ einer Übergangsrelation.

Ein Zustand s fasst folgende Informationen zusammen:

- den momentanen Stand der Programmausführung (*program location*)
- die aktuellen Werte der lokalen und globalen Variablen

Program Locations

- $\mathcal{L}$: endliche Menge aller Program Locations
- $s.\ell$: Program Location im Zustand $s \in S$
- ℓ_0: Program Location beim Programmeintritt (Main Methode)

Ein formales Softwaremodell — 2

Programmvariablen

- Menge V von Variablen über endlichem Wertebereich D
- V möglicherweise unbeschränkt wenn dynamische Allokation von Objekten erlaubt ist
- $s.v$: Wert der Variable $v \in V$ im Zustand $s \in S$

Funktionsaufrufe

- Modellierung mit Hilfe eines unbeschränkten Funktionsaufruf-Stacks $\Gamma : \mathbb{N} \to D \dot{\cup} \mathcal{L}$
- Auf dem Stack werden Program Locations sowie lokale Variablen gespeichert
- $s.\Gamma$: Stack im Zustand $s \in S$

Zusammenfassend: S besteht aus drei Komponenten:

$$S = \mathcal{L} \times \underbrace{(V \to D)}_{\text{Variablen}} \times \underbrace{(\mathbb{N} \to D \dot{\cup} \mathcal{L})}_{\text{Stack}}$$

Ein formales Softwaremodell — 3

Bemerkungen

- Es gibt nur einen Programmzähler, d.h. nur einen Thread
- S_0 weist dem Programm den initialen Ausführungsstand ℓ_0 zu
- Initialisierung von globalen Variablen muss innerhalb des Programms geschehen

Programminstruktionen werden mit Übergangsrelation modelliert

Übergangsrelation R

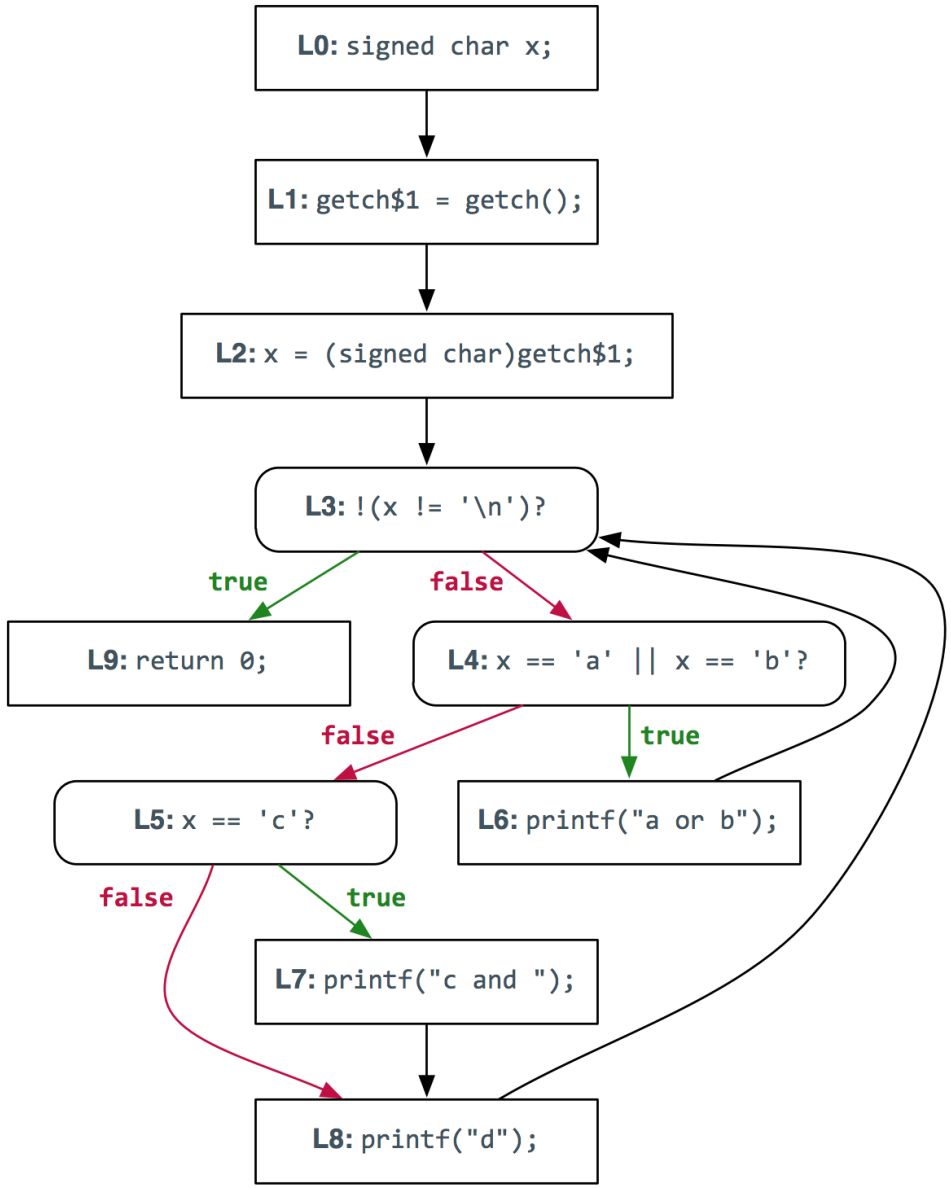
Für zwei Zustände $\langle s, s' \rangle \in S \times S$ gilt $R(s, s')$ genau dann, wenn es einen Übergang im Programm von s nach s' gibt.

- Sowohl Programmfluss als auch Veränderungen der Daten werden mit R beschrieben
- R wird partitioniert in ein R_l für jede Program Location $l \in \mathcal{L}$:

$$R(s, s') \Leftrightarrow \bigwedge_{l \in \mathcal{L}} (s.\ell = l \rightarrow R_l(s, s'))$$

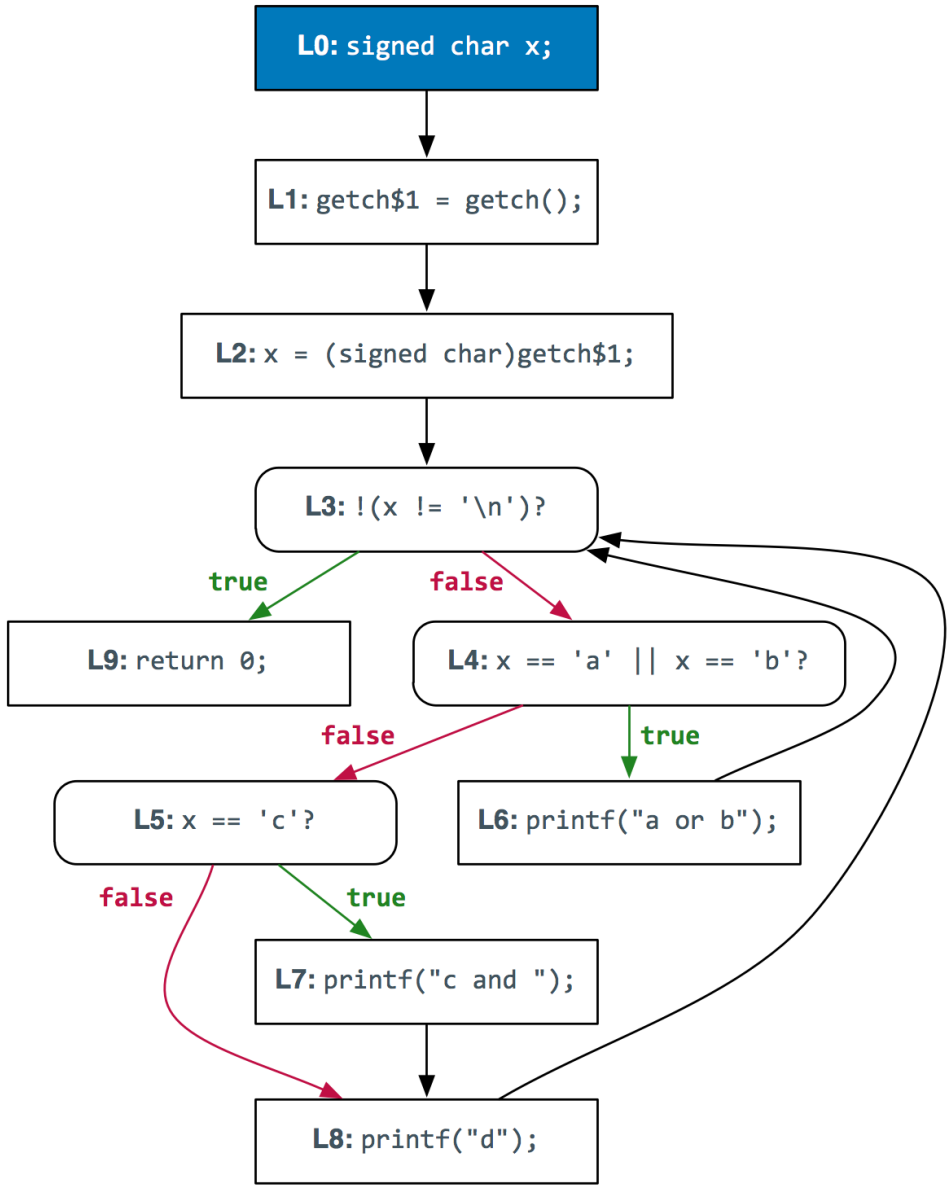
Ein formales Softwaremodell — Beispiel

```
int main( void ) {
  char x;
  x = getch();
  while (x!='\n') {
    switch(x) {
      case 'a':
      case 'b':
        printf("a or b");
        break;
      case 'c':
        printf("c and ");
        /* fall-through */
      default:
        printf("d");
        break;
    }
  }
  return 0;
}
```



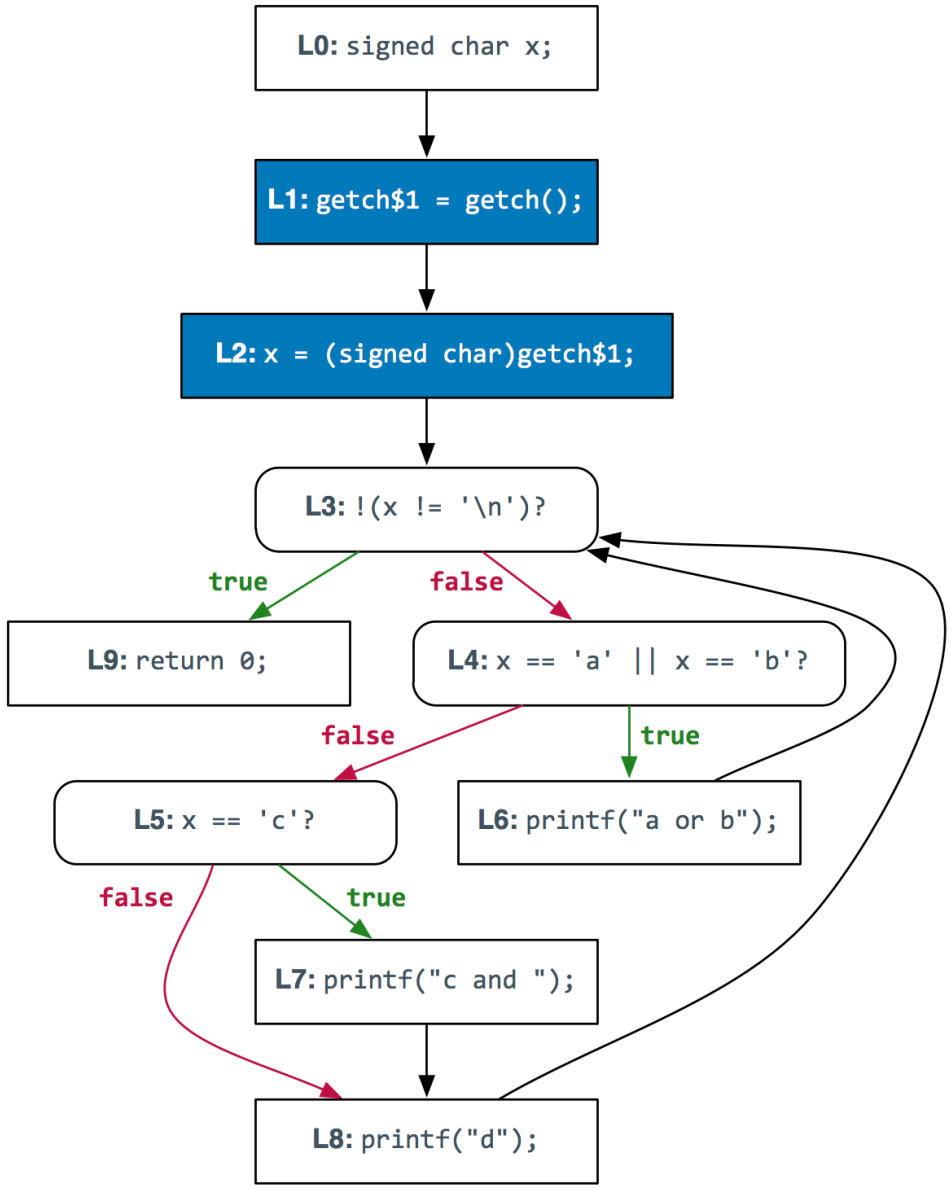
Ein formales Softwaremodell — Beispiel

```
int main( void ) {
  char x;
  x = getch();
  while (x!='\n') {
    switch(x) {
      case 'a':
      case 'b':
        printf("a or b");
        break;
      case 'c':
        printf("c and ");
        /* fall-through */
      default:
        printf("d");
        break;
    }
  }
  return 0;
}
```



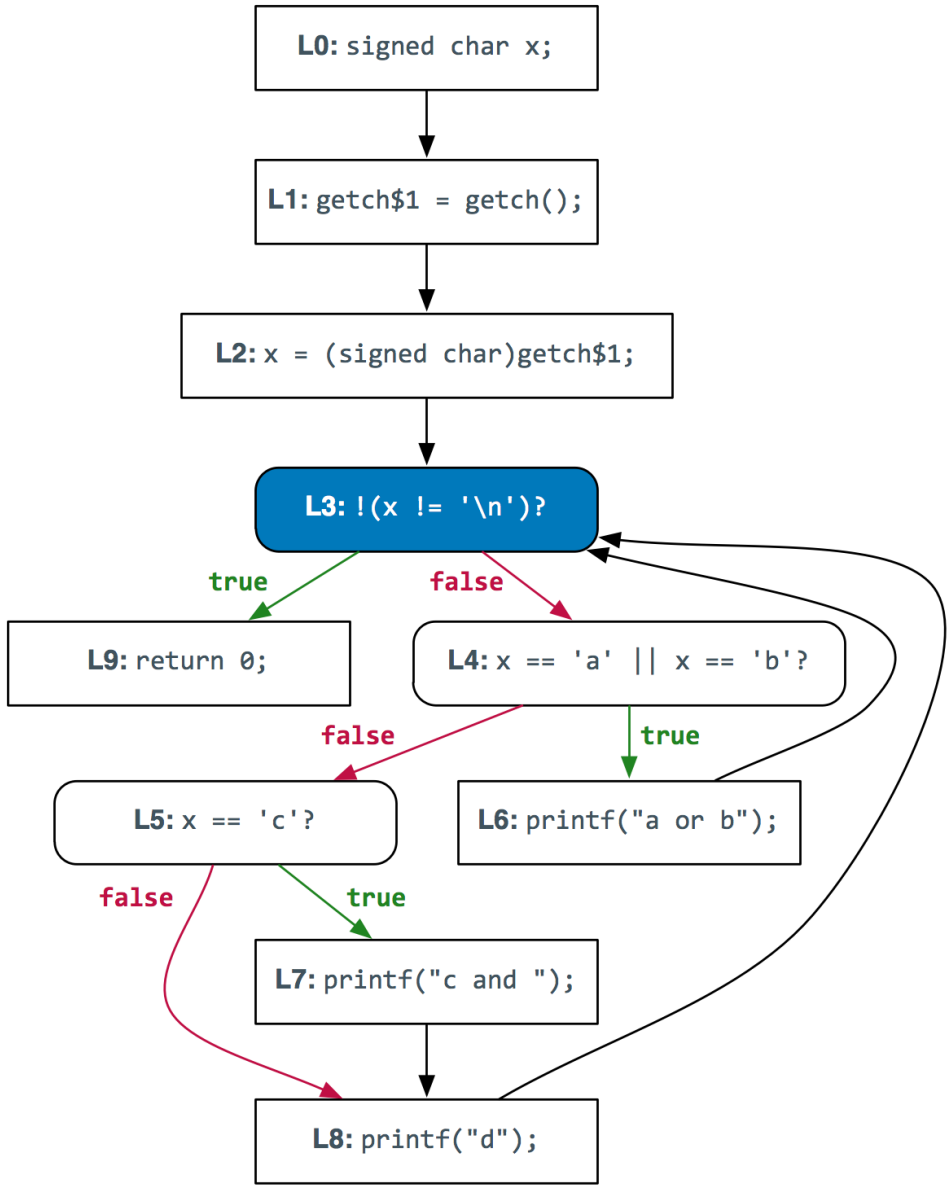
Ein formales Softwaremodell — Beispiel

```
int main( void ) {
  char x;
  x = getch();
  while (x!='\n') {
    switch(x) {
      case 'a':
      case 'b':
        printf("a or b");
        break;
      case 'c':
        printf("c and ");
        /* fall-through */
      default:
        printf("d");
        break;
    }
  }
  return 0;
}
```



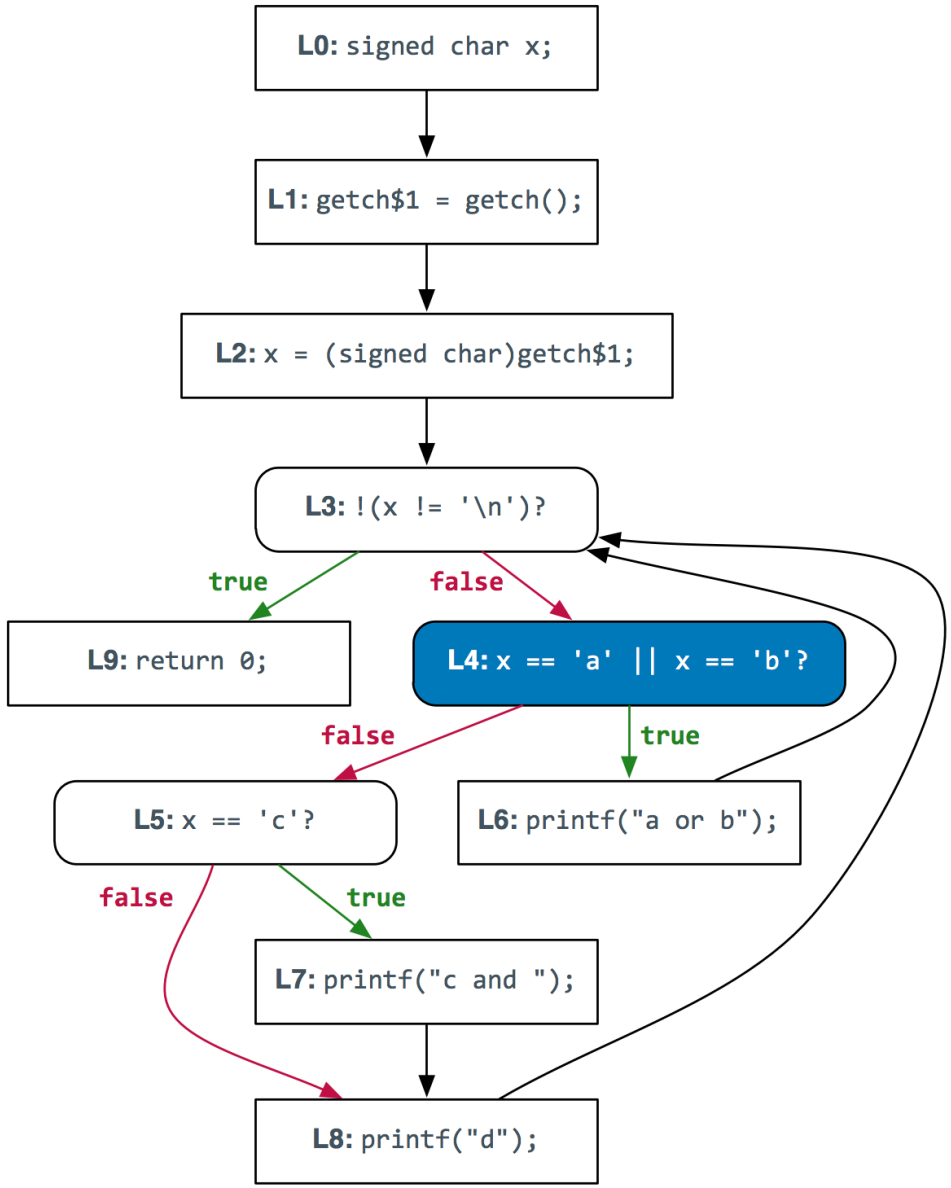
Ein formales Softwaremodell — Beispiel

```
int main( void ) {
  char x;
  x = getch();
  while (x!='\n') {
    switch(x) {
      case 'a':
      case 'b':
        printf("a or b");
        break;
      case 'c':
        printf("c and ");
        /* fall-through */
      default:
        printf("d");
        break;
    }
  }
  return 0;
}
```



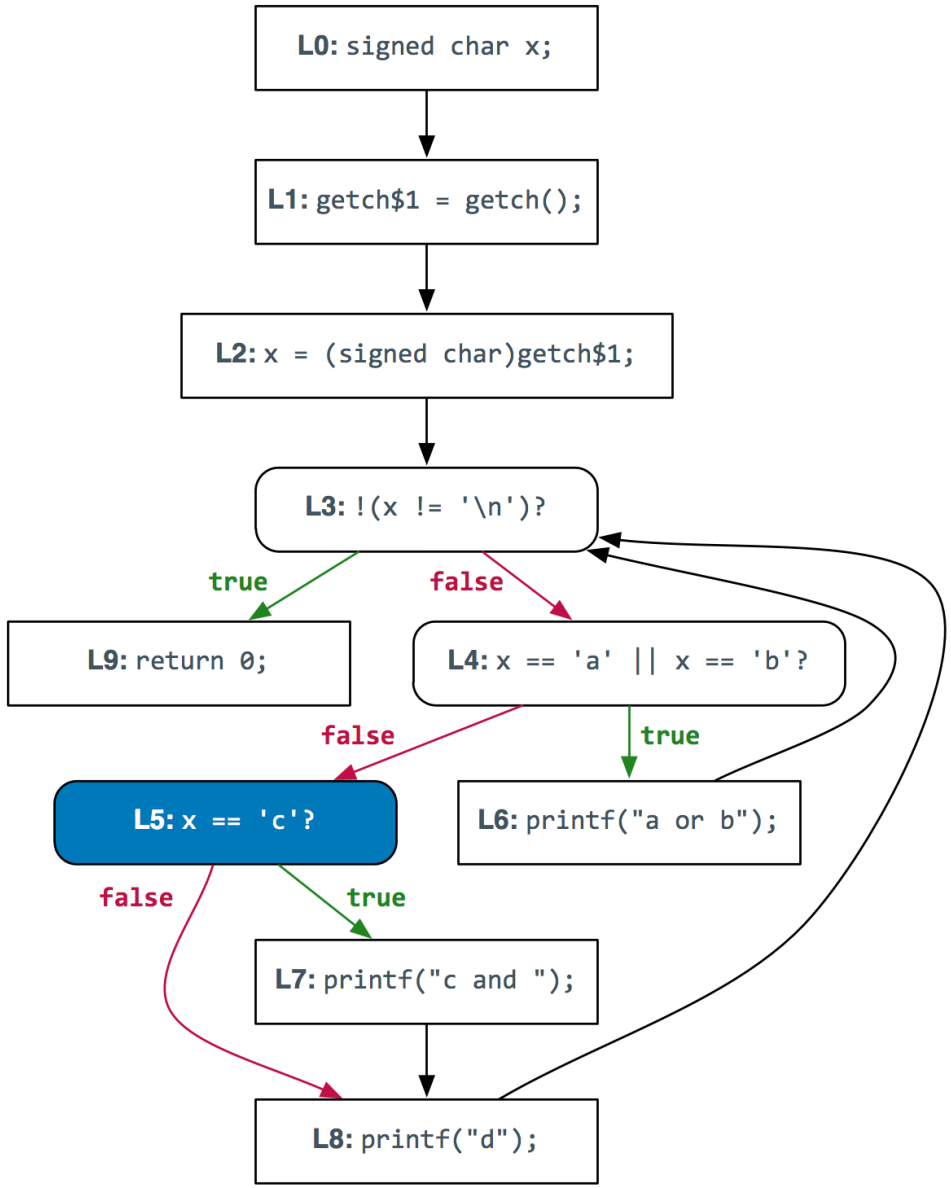
Ein formales Softwaremodell — Beispiel

```
int main( void ) {
  char x;
  x = getch();
  while (x!='\n') {
    switch(x) {
      case 'a':
      case 'b':
        printf("a or b");
        break;
      case 'c':
        printf("c and ");
        /* fall-through */
      default:
        printf("d");
        break;
    }
  }
  return 0;
}
```



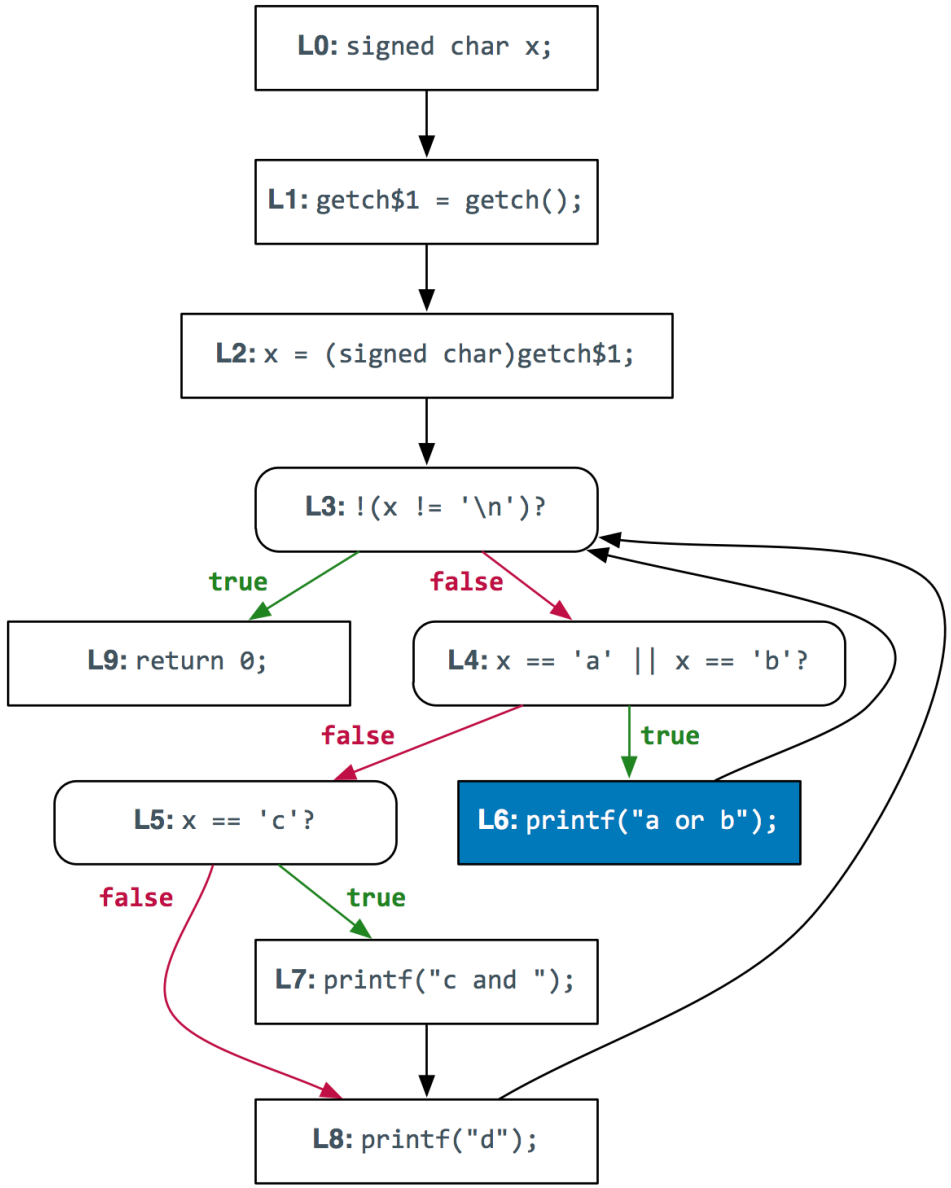
Ein formales Softwaremodell — Beispiel

```
int main( void ) {
  char x;
  x = getch();
  while (x!='\n') {
    switch(x) {
      case 'a':
      case 'b':
        printf("a or b");
        break;
      case 'c':
        printf("c and ");
        /* fall-through */
      default:
        printf("d");
        break;
    }
  }
  return 0;
}
```



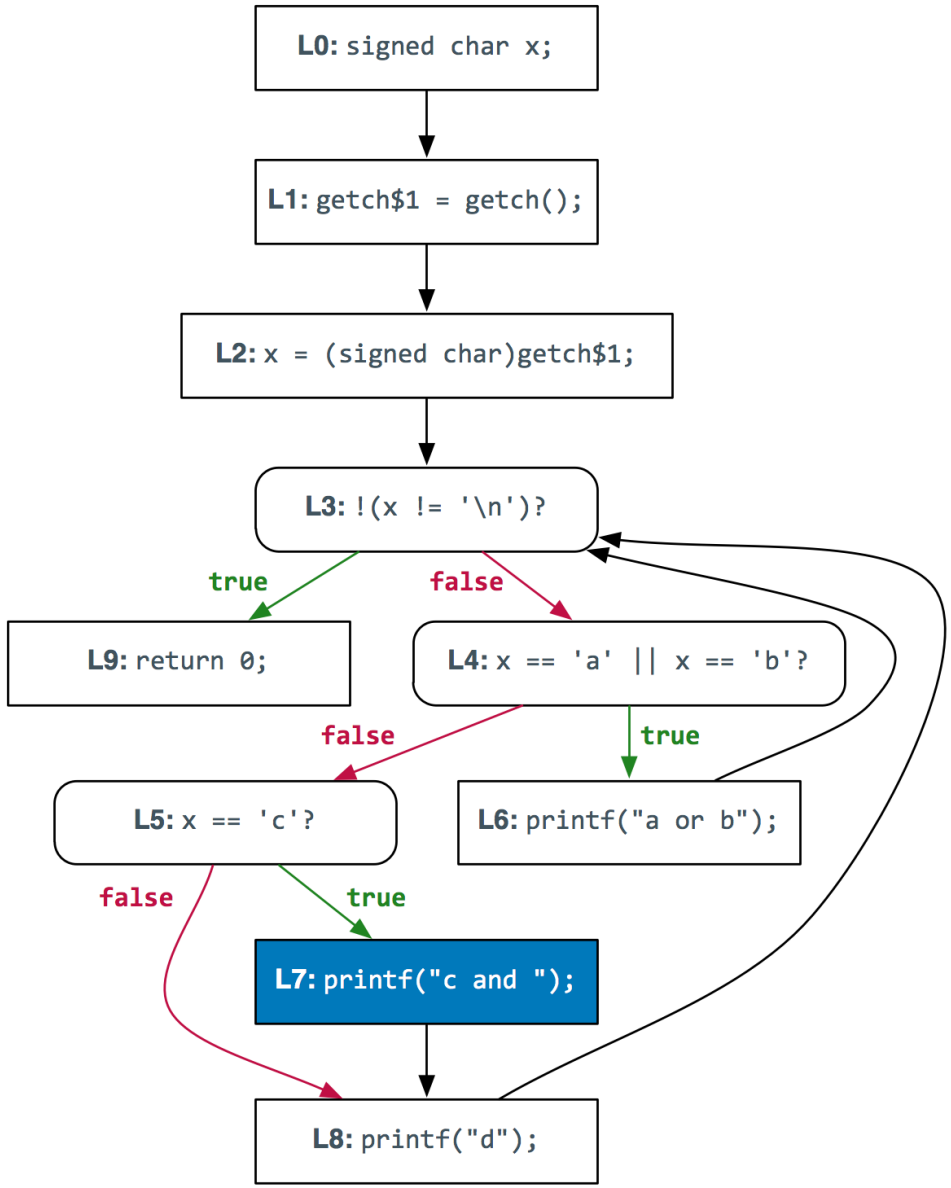
Ein formales Softwaremodell — Beispiel

```
int main( void ) {
  char x;
  x = getch();
  while (x!='\n') {
    switch(x) {
      case 'a':
      case 'b':
        printf("a or b");
        break;
      case 'c':
        printf("c and ");
        /* fall-through */
      default:
        printf("d");
        break;
    }
  }
  return 0;
}
```



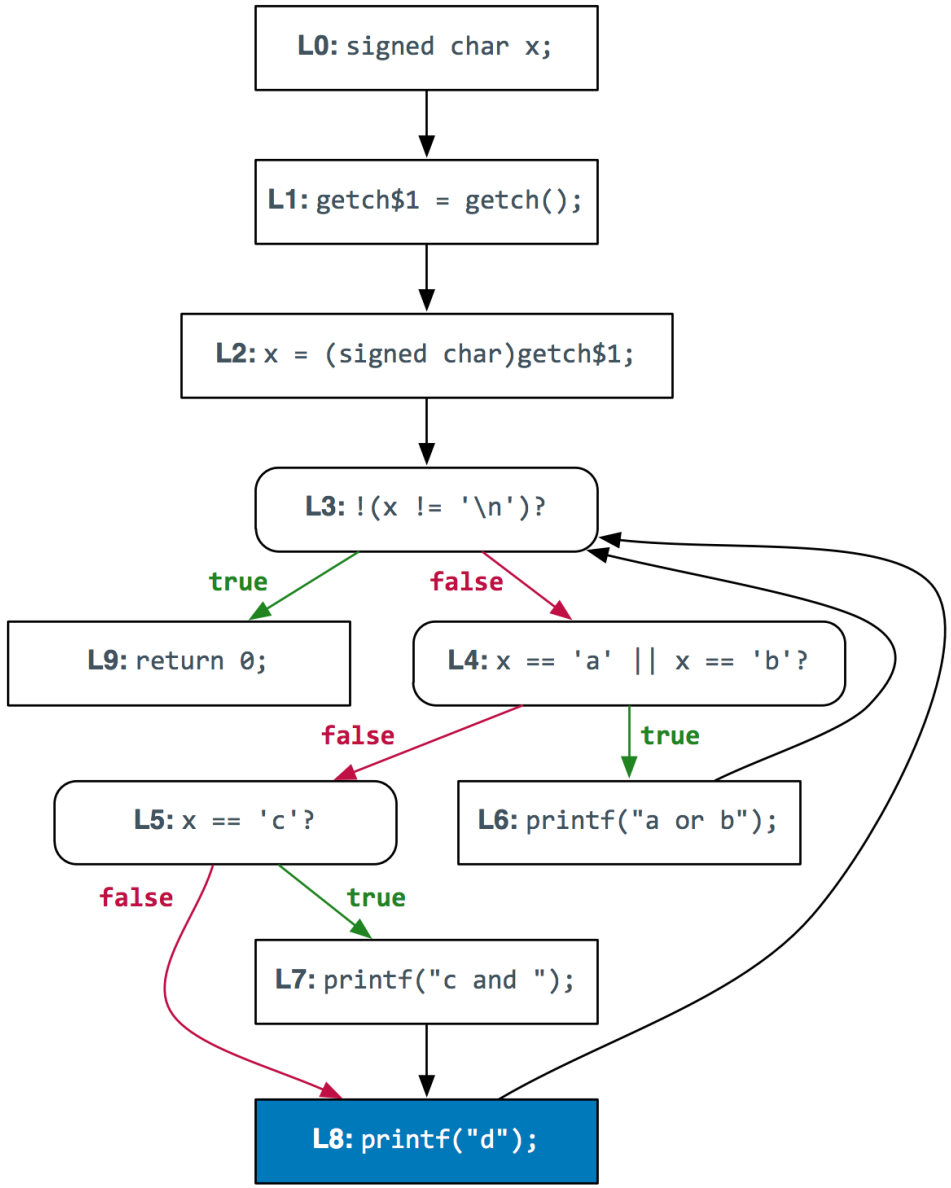
Ein formales Softwaremodell — Beispiel

```
int main( void ) {
  char x;
  x = getch();
  while (x!='\n') {
    switch(x) {
      case 'a':
      case 'b':
        printf("a or b");
        break;
      case 'c':
        printf"c and ");
        /* fall-through */
      default:
        printf("d");
        break;
    }
  }
  return 0;
}
```



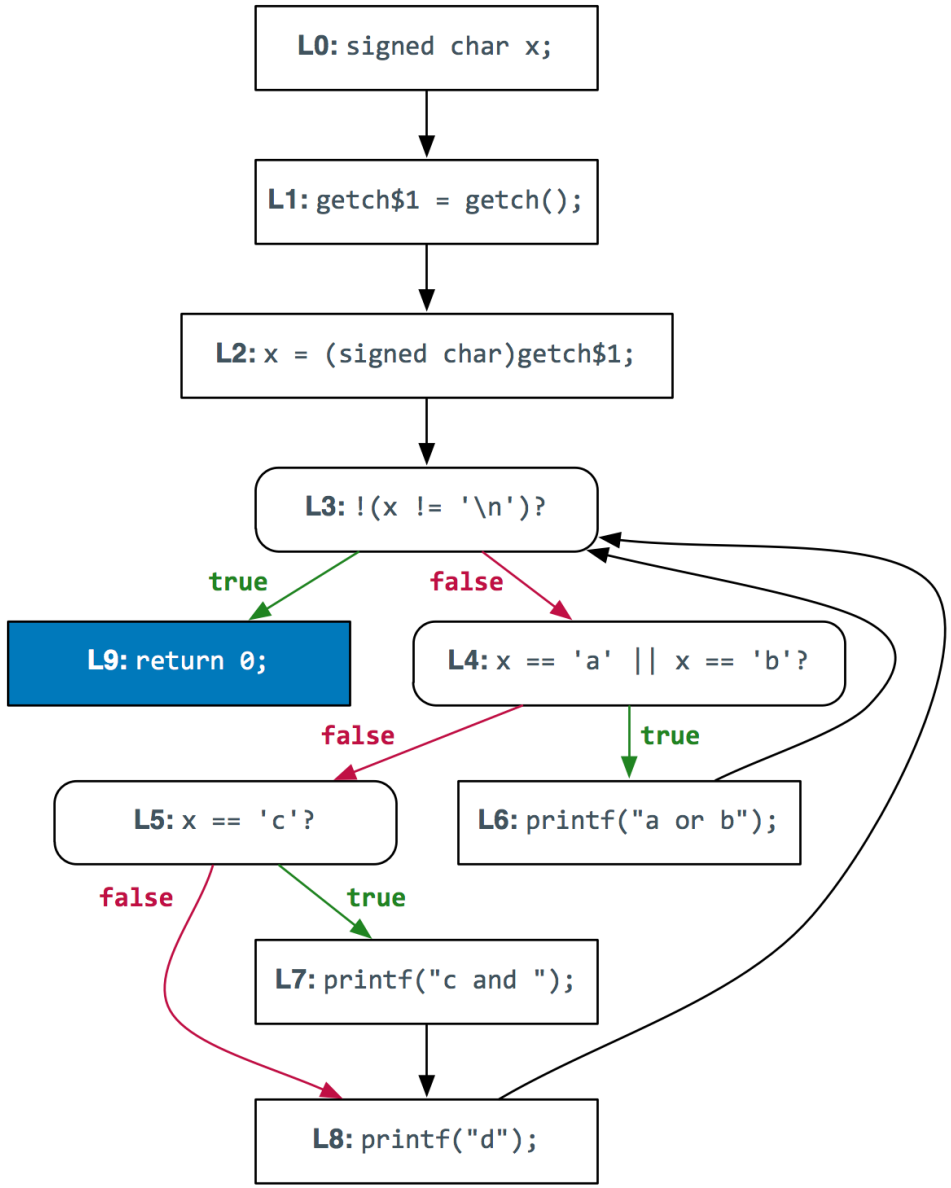
Ein formales Softwaremodell — Beispiel

```
int main( void ) {
  char x;
  x = getch();
  while (x!='\n') {
    switch(x) {
      case 'a':
      case 'b':
        printf("a or b");
        break;
      case 'c':
        printf("c and ");
        /* fall-through */
      default:
        printf("d");
        break;
    }
  }
  return 0;
}
```

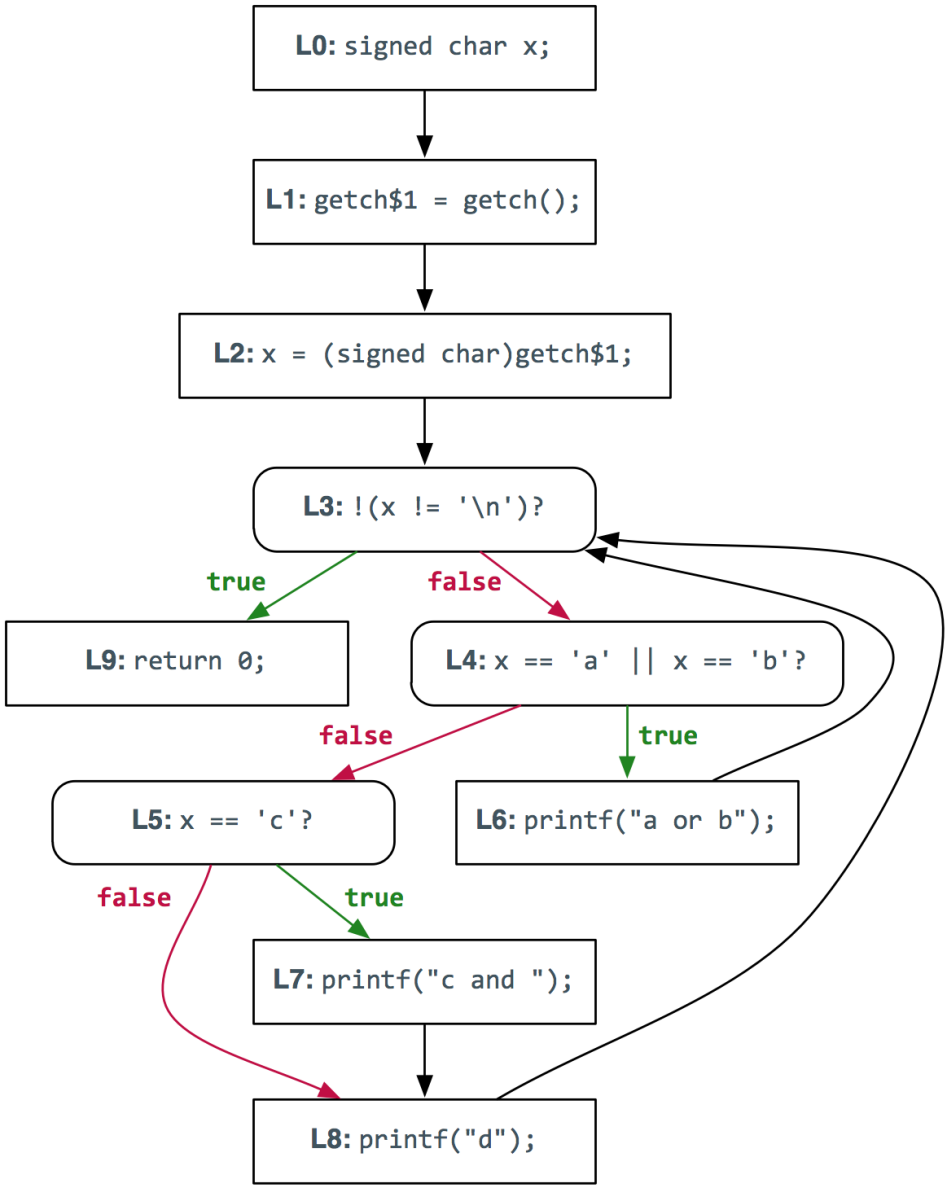


Ein formales Softwaremodell — Beispiel

```
int main( void ) {
  char x;
  x = getch();
  while (x!='\n') {
    switch(x) {
      case 'a':
      case 'b':
        printf("a or b");
        break;
      case 'c':
        printf("c and ");
        /* fall-through */
      default:
        printf("d");
        break;
    }
  }
  return 0;
}
```



Ein formales Softwaremodell — Kodierungen



Übergangsrelation an L_2:

$$R_{L2}(s, s') \iff s'.\ell = L3$$
$$\wedge\ \forall v \neq x : s'.v = s.v$$
$$\wedge\ s'.x = TC(s.\texttt{getch\$1})$$
$$\wedge\ s'.\Gamma = s.\Gamma$$

- `TC` bildet `integer` auf `char` ab (modulo 256 auf den meisten Architekturen)

Übergangsrelation an L_3:

$$R_{L3}(s, s') \iff$$
$$s'.\ell = \begin{cases} L9 & \text{if } s.x =' \backslash n' \\ L4 & \text{otherwise} \end{cases}$$
$$\wedge\ \forall v \in V : s'.v = s.v$$
$$\wedge\ s'.\Gamma = s.\Gamma$$

Gegenbeispiel

Notationen

Wir schreiben

- $s \to t$ für $R(s,t)$ und
- $s \rightsquigarrow t$ für $R^*(s,t)$ (reflexiv-transitive Hülle).

Was uns interessiert? Können bestimmte „schlechte“ Programmzustände (*error locations*) erreicht werden? *(Reachability Problem)*

Gegenbeispiel (*Counterexample*)

- $\mathcal{L}_E \subset \mathcal{L}$: Menge der Error Locations
- $s \in S$ mit $s.\ell \in \mathcal{L}_E$: Fehlerhafter Zustand (error state)

Ein Gegenbeispiel erhält man durch Lösung eines Erreichbarkeitsproblems als Sequenz von Zuständen $s_0, s_1, \ldots, s_n$ mit $s_i \in S$, $s_0 \in S_0$, $s_n.\ell \in \mathcal{L}_E$ und $s_j \to s_{j+1}$ für $0 \leq j < n$.

Fahrplan für heute: Wo stehen wir?

1. **Einführung Software-Verifikation ✓**
   - SW-Verifikation: Korrektheit eines Programms mit mathematischen Methoden beweisen
   - Motivation: Immer mehr sicherheitskritische Software
2. **Ein formales Modell für Software ✓**
   - Formalisierung eines Programms als Transition System
   - Zustand speichert: Stand des Program Counter, Variablenbelegungen, Ausführungsstack
   - Gegenbeispiel: Pfad zu einem Fehler
3. **Bit-Vektoren**
   - Was sind Bit-Vektoren?
   - Wie kann man Bit-Vektor Gleichungen entscheiden?
4. **Bounded Model Checking (BMC)**
   - Wie funktioniert BMC?
   - Beispiele für BMC?
   - Und in der echten Welt?

Bit-Vektoren — 1

Bit Vektoren

Ein Bit-Vektor der Länge n ist eine Sequenz von n Booleschen Variablen. $b[i]$ notiert das i-te Bit von Vektor b (von links, beginnend mit 0).

Beispiel

- $\langle 00001111 \rangle$, Bit-Vektor der Länge 8, $b[4] = 1$

Grammatik für gültige BV Formeln

$$
\begin{array}{rcl}
\text{formula} & : & \text{formula} \wedge \text{formula} \mid \neg\text{formula} \mid (\text{formula}) \mid \text{atom} \\
\text{atom} & : & \text{term rel term} \mid \text{Boolean-Identifier} \\
\text{rel} & : & < \mid = \\
\text{term} & : & \text{term op term} \mid \text{identifier} \mid \sim \text{term} \mid \text{constant} \mid \\
 & & \text{atom?term} : \text{term} \\
\text{op} & : & + \mid - \mid \cdot \mid / \mid << \mid >> \mid \& \mid \mid \mid \oplus \mid \circ
\end{array}
$$

Entscheidungsverfahren für Bit-Vektoren

Durch schnelle SAT-Solver begünstigt: **Bit Blasting**

Bit Blasting

Eingabeformel φ in Bit-Vektor Arithmetik, Ausgabe: Erfüllbarkeitsäquivalente Formel in Aussagenlogik φ_f

- $A(\varphi)$ die Menge aller Atome in φ

1. φ_{sk}: Originalformel, in der jedes Atom $a \in A(\varphi)$ durch eine neue Boolesche Variable $\mu(a)$ ersetzt ist (Aussagenlogisches Skelett), $\varphi_f = \varphi_{sk}$
2. Jeder Bit-Vektor Term t wird durch einen neuen Vektor $\mu(t)$ von Booleschen Variablen ersetzt
3. Constraints für $\mu(a)$ und $\mu(t)$ werden je nach Art des Atoms oder des zu Grunde liegenden Operators generiert. Z.B. für $t = a|b$ mit jeweils n Bits:

$$c_t = \bigwedge_{i=0}^{n-1} ((a[i] \vee b[i]) \Leftrightarrow \mu(t)[i]$$

4. Füge für jedes t c_t zu φ_f hinzu

Bit Blasting — Beispiel

Beispiel

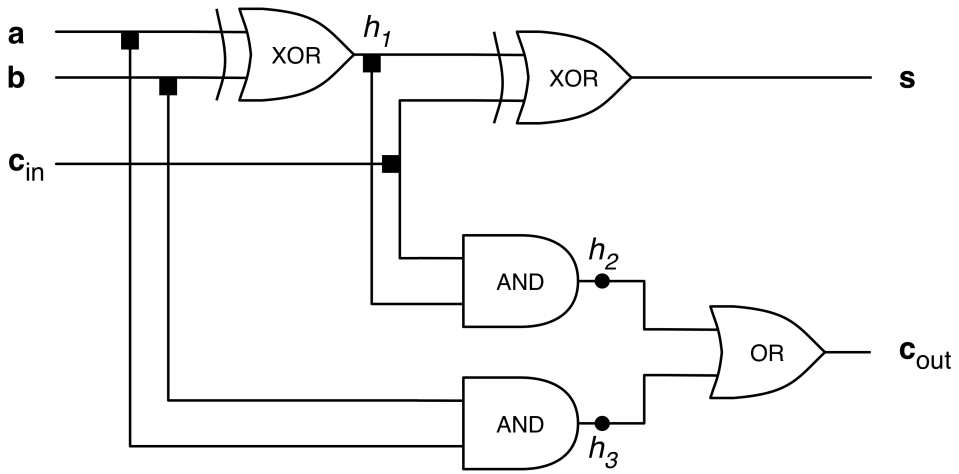
- 32 Bit
- Zeile im Programm: `x = a & b; y = x;`
- Bit-Vektor Formel: $(x = a \;\&\; b) \wedge (y = x)$

Bit-Blasting:

1. $\varphi_{sk} = \alpha \wedge \beta$
2. $c_1 = \bigwedge_{i=0}^{31}(a[i] \wedge b[i]) \Leftrightarrow \gamma[i]$ (Constraint für $a \;\&\; b$)
3. $c_2 = \bigwedge_{i=0}^{31} x[i] \Leftrightarrow \gamma[i]$ (Constraint für $x = a \;\&\; b$)
4. $c_3 = \bigwedge_{i=0}^{31} y[i] \Leftrightarrow x[i]$ (Constraint für $y = x$)
5. $\varphi_f = \varphi_{sk} \wedge c_1 \wedge c_2 \wedge c_3$

Bit-Blasting — Arithmetische Operationen

Arithmetische Operationen werden mit Kodierungen von Schaltkreisen repräsentiert



Beispiel

Addition von zwei Bit-Vektoren $\langle a_0 a_1 a_2 a_3 \rangle$ und $\langle b_0 b_1 b_2 b_3 \rangle$ ist ein neuer Bit-Vektor $\langle s_0 s_1 s_2 s_3 \rangle$ mit

- $s_0 \Leftrightarrow (a_0 \oplus b_0) \oplus c_1$
- $s_1 \Leftrightarrow (a_1 \oplus b_1) \oplus c_2$, $c_1 \Leftrightarrow (a_1 \wedge b_1) \vee (c_2 \wedge (a_1 \oplus b_1))$
- $s_2 \Leftrightarrow (a_2 \oplus b_2) \oplus c_3$, $c_2 \Leftrightarrow (a_2 \wedge b_2) \vee (c_3 \wedge (a_2 \oplus b_2))$
- $s_3 \Leftrightarrow a_3 \oplus b_3$, $c_3 \Leftrightarrow a_3 \wedge b_3$

Fahrplan für heute: Wo stehen wir?

1. **Einführung Software Verifikation ✓**
   - SW-Verifikation: Korrektheit eines Programms mit mathematischen Methoden beweisen
   - Motivation: Immer mehr sicherheitskritische Software
2. **Ein formales Modell für Software ✓**
   - Formalisierung eines Programms als Transition System
   - Zustand speichert: Stand des Program Counter, Variablenbelegungen, Ausführungsstack
   - Gegenbeispiel: Pfad zu einem Fehler
3. **Bit-Vektoren ✓**
   - Bit-Vektor: Sequenz von Booleschen Variablen
   - Entscheiden von Bit-Vektor Gleichungen mit Bit-Blasting und SAT-Solvern
4. **Bounded Model Checking (BMC)**
   - Wie funktioniert BMC?
   - Beispiele für BMC?
   - Und in der echten Welt?

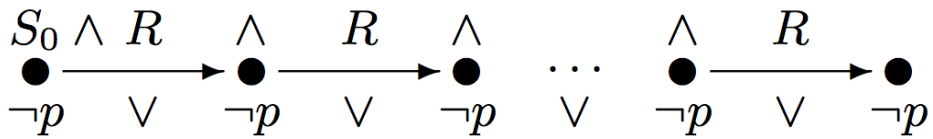
Bounded Model Checking

- **Idee:** Wickle das zu verifizierende Design k-mal ab und teste die Verifikationseigenschaft dabei
- Wenn die Formel erfüllbar ist, gibt es ein Gegenbeispiel der Länge k

Bounded Model Checking (BMC)

- Übergangsrelation R
- Menge der Anfangszustände S_0
- zu verifizierende Eigenschaft p

1. Kopiere die Übergangsrelation k mal
2. Benenne die Variablen in jeder Kopie so um, dass der Folgezustand von Schritt n der aktuelle Zustand von Schritt $n+1$ ist
3. Der aktuelle Zustand des ersten Schritts muss in S_0 sein
4. An einem der Zustände muss $\neg p$ gelten



Abwickeln des gesamten Programmes

- Betrachte gesamtes Programm als eine Übergangsrelation (keine Partitionen)
- Repräsentiere den Program Counter ℓ als Bit-Vektor mit Länge $\lceil \log_2 |\mathcal{L}| \rceil$
- *Optimierungsmöglichkeit: Führe Instruktionen, die einen Block bilden zu einer Program Location zusammen*

Die resultierende Formel

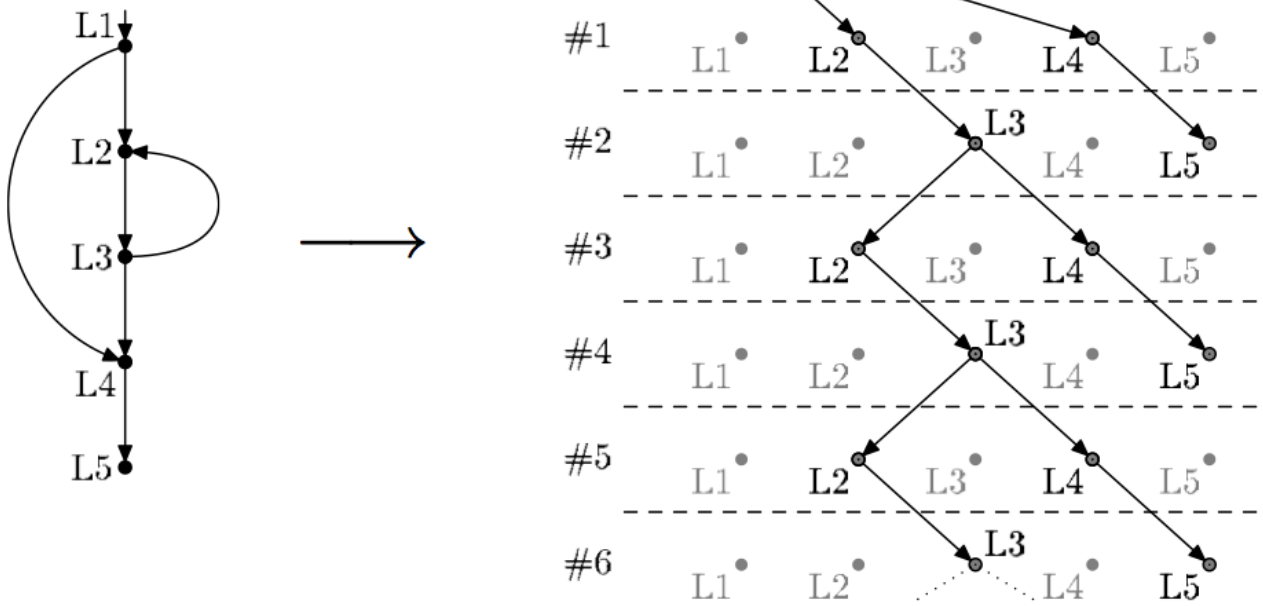
- s_i Zustand im Schritt i der Abwicklung

$$\varphi = \underbrace{S_0(s_0)}_{s_0 \text{ ist Startzustand}} \wedge \underbrace{\bigwedge_{i=0}^{k-1} R(s_i, s_{i+1})}_{\text{Es gibt einen Übergang}} \wedge \underbrace{\bigvee_{i=0}^{k} \bigvee_{l \in \mathcal{L}_E} s_i.\ell = l}_{\text{Ein Fehlerzustand wird erreicht}}$$

- φ erfüllbar: Gegenbeispiel kann konstruiert werden (aus $s_0, \ldots, s_k$)
- φ unerfüllbar: Es gibt kein Gegenbeispiel, das in k oder weniger Schritten erreichbar ist

Optimierungsmöglichkeiten

- **Problem:** k-faches Abwickeln des gesamten Programms führt schnell zu sehr großen Formeln
- **Lösungsidee:** Kopiere nicht gesamt R k-mal sondern führe eine Analyse des Kontrollflussgraphen durch

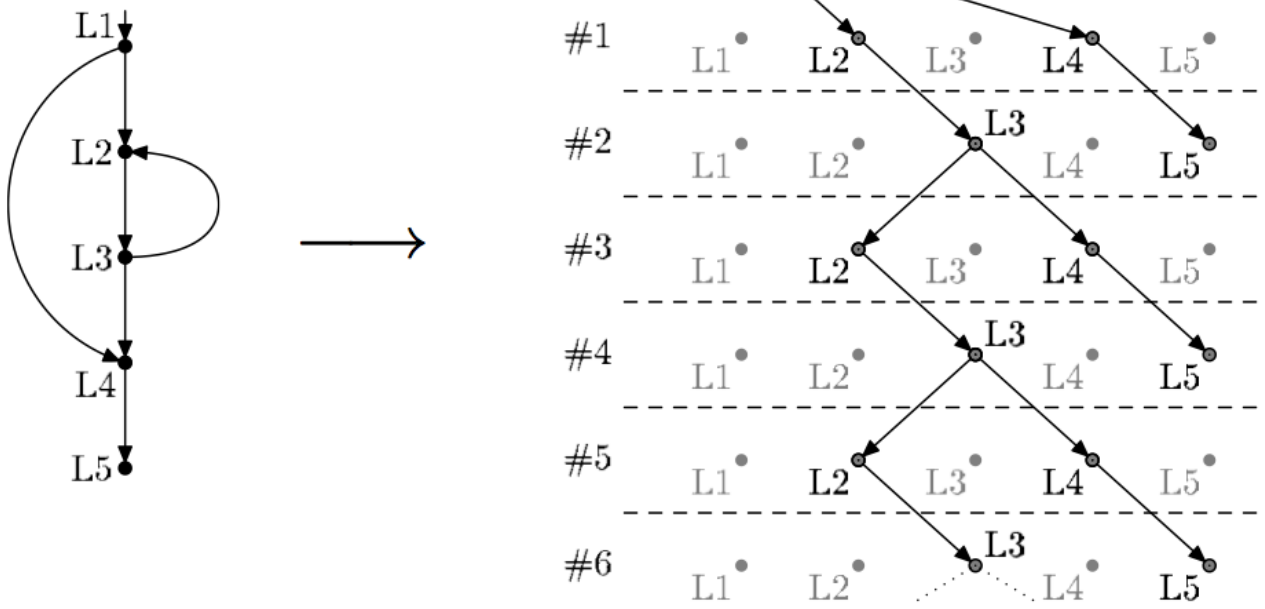


Einzelnes Abwickeln von Schleifen — 1

Aber: Es wird immer noch zu viel kopiert:

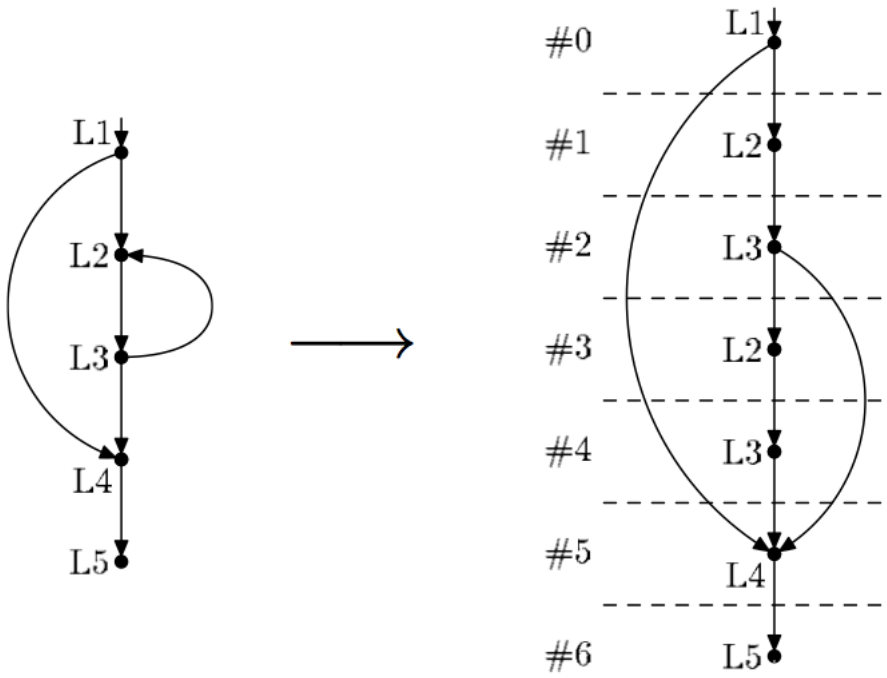
- Z.B. Kann $L4$ in einem Programmablauf nur einmal erreicht werden, jedoch haben wir $k/2$ Kopien davon

Lösungsidee: Kopiere nicht das gesamte Programm k mal, sondern wickle nur jede Schleife k mal ab



Einzelnes Abwickeln von Schleifen — 2

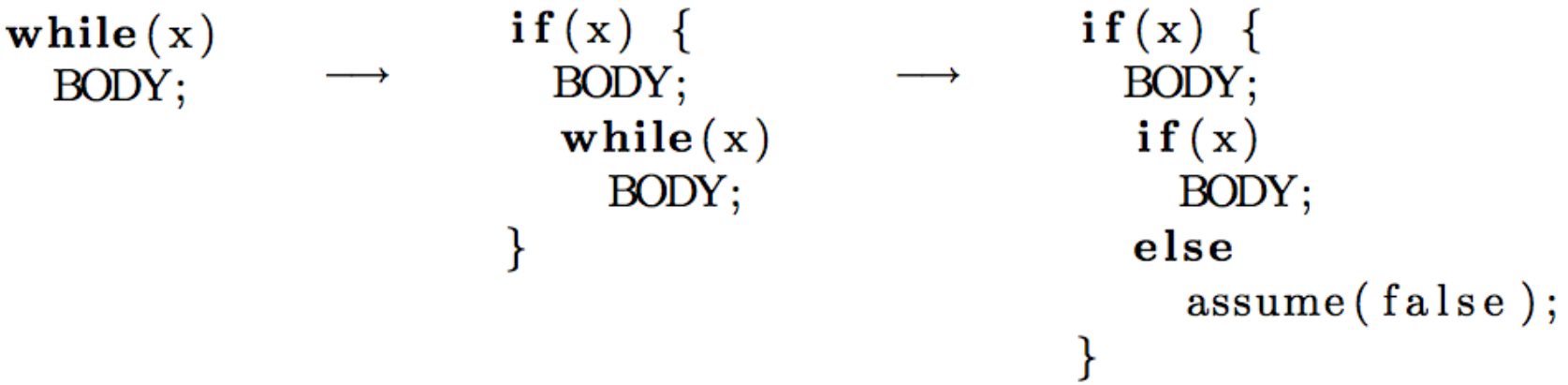
Abwicklung der Schleife mit $k = 2$



- Für jede Schleife kann eine eigene Grenze k_i verwaltet werden
- Auch geschachtelte Schleifen können verarbeitet werden

Einzelnes Abwickeln von Schleifen — 3

- R wird nun nicht mehr einfach kopiert
- Der originale Kontrollflussgraph des Programms wird verändert
- Das k-fache Abwickeln einer Schleife bedeutet k-faches Kopieren des Schleifenkörpers
- Jede Abwicklung wird mit einer `if` Abfrage umgeben falls die Schleife frühzeitig terminiert



- `assume(false)` wird benutzt, um festzustellen, dass die Grenze k zu klein war (und nicht fälschlicherweise kein Gegenbeispiel ausgegeben wird)

Vom Programm zur Formel — 1

Wie gehen wir mit anderen Programmkonstrukten um?

Schritt 1

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

Wie bereits gesehen:

- Ersetze `while` durch `if`
- Kopiere Schleife maximal k mal (Bound k) hintereinander
- Prüfe Bound durch nachfolgende *unwinding assertion*

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 expandieren (Aufruf durch Funktionsinhalt ersetzen)
- Rekursive Funktionen k mal abwickeln + Assertion, dass Rekursionstiefe nicht tiefer als k sein kann
- `return` Statements ersetzen durch `goto` zum Ende der Funktion

Beispiel (Function unwinding)

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

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

```
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

Problem: Auflösung der zeitlichen Reihenfolge

- Anweisungssequenz `x=0; x=1;` kann nicht in die Formel $(x = 0) \wedge (x = 1)$ übersetzt werden.
- Was im Programm nacheinander geschieht, das geschieht in der Formel gleichzeitig

Schritt 4

Programm wird in Single Static Assignment Form gebracht

- x_i bezeichnet den i-ten Wertezustand der Variable x.
- Sequenz `x=0; x=1;` wird zur Formel $(x_i = 0) \wedge (x_{i+1} = 1)$

Single Static Assignment Form — 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 dort Vereinfachungen

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

Single Static Assignment Form — 2

Übersetzen von if-then-else in SSA

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

Übersetzung in SSA:

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

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

Übersetzung in SSA:

$y_1 = 3$

$y_2 = (x == 1)\ ?\ 5 : y_1$

Single Static Assignment Form — 3

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: Bit-Blasting und Formel $C \wedge \neg P$ in CNF konvertieren

Das komplette Beispiel — 1

Beispiel (C 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)
```

Das komplette 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 $\neg y_0[0] \wedge y_0[1] \wedge \neg y_0[2] \wedge ... \wedge \neg y_0[31]$
- $y_1 = 5$ wird zu $y_1[0] \wedge \neg y_1[1] \wedge y_1[2] \wedge \neg y_1[3] \wedge ... \wedge \neg y_1[31]$
- $x_0 < 4$ 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 $y_2[31] \vee (\neg y_2[31] \wedge ... \wedge \neg y_2[2])$

Das komplette Beispiel — 3

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

Codiere $C \wedge \neg P$

Beispiel (Endformel)

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

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

Implementierungen von BMC

CBMC

Bounded Model Checker für C Programme[a]

- emuliert eine große Anzahl an Architekturen
- Little-Endian und Big-Endian Speichermodelle
- Wickelt Schleifen wie hier beschrieben ab
- Benutzt Bit-Blasting und MiniSAT um die Formel zu entscheiden
- Kann Formeln im Dimacs Format ausgeben
- Unterstützt auch andere Ausgabeformate (z.B. SMT)
- Unterstützt (in Ansätzen) auch C++, SpecC und SystemC

[a] `http://www.cprover.org/cbmc/`

FYI:

- An unserem Arbeitsbereich wird viel mit CBMC gearbeitet
- Erfolgreiche Diplomarbeiten im Bereich Linux Treiberverifikation oder Verifikation von hardwarenahem C++ Code

Fahrplan für heute: Das war's!

1. **Einführung Software Verifikation ✓**
   - SW-Verifikation: Korrektheit eines Programms mit mathematischen Methoden beweisen
   - Motivation: Immer mehr sicherheitskritische Software
2. **Ein formales Modell für Software ✓**
   - Formalisierung eines Programms als Transition System
   - Zustand speichert: Stand des Program Counter, Variablenbelegungen, Ausführungsstack
   - Gegenbeispiel: Pfad zu einem Fehler
3. **Bit-Vektoren ✓**
   - Bit-Vektor: Sequenz von Booleschen Variablen
   - Entscheiden von Bit-Vektor Gleichungen mit Bit-Blasting und SAT-Solvern
4. **Bounded Model Checking (BMC) ✓**
   - BMC: k-faches Abwickeln von Schleifen und Rekursion, SSA, Bit-Blasting, SAT-Solver
   - CBMC für C Programme