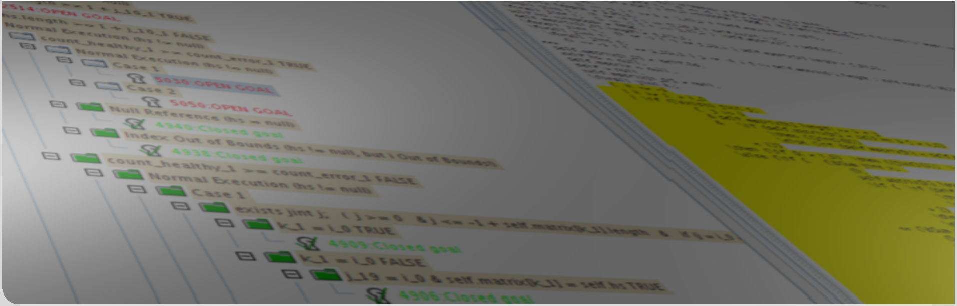
Kurz-Einführung in Logik & Programmverifikation

PSE-Thema *Analyse formaler Eigenschaften von Wahlverfahren*

Prof. Bernhard Beckert, Sarah Grebing, Michael Kirsten | Karlsruhe, 10. November 2016

INSTITUT FÜR THEORETISCHE INFORMATIK – ANWENDUNGSORIENTIERTE FORMALE VERIFIKATION



Aussagen- und Prädikatenlogik

Propositional Logic: Syntax

Special symbols

$(\quad) \quad \neg \quad \wedge \quad \vee \quad \rightarrow \quad \leftrightarrow$

Signature

Propositional variables $\Sigma = \{p_0, p_1, \ldots\}$

Formulas

- **The propositional variables** $p \in \Sigma$ **are formulas**
- **If** A, B **are formulas, then**

$$\neg A \qquad (A \wedge B) \qquad (A \vee B) \qquad (A \rightarrow B) \qquad (A \leftrightarrow B)$$

are formulas

Propositional Logic: Semantics

Interpretation

Function $I : \Sigma \to \{\underline{\textbf{true}}, \underline{\textbf{false}}\}$

Valuation

Extension of interpretation to formulas as follows:

$$\textbf{val}_I(p) \quad = \quad I(p)$$

$$\textbf{val}_I(\neg p) \quad = \quad \begin{cases} \underline{\textbf{true}} & \textbf{if } I(p) = \underline{\textbf{false}} \\ \underline{\textbf{false}} & \textbf{if } I(p) = \underline{\textbf{true}} \end{cases}$$

B. Beckert: Formal Verification of Software – p.5

Propositional Logic: Semantics

$$\mathbf{val}_I(\alpha) \quad = \quad \begin{cases} \underline{\mathbf{true}} & \text{if } \mathbf{val}_I(\alpha_1) = \underline{\mathbf{true}} \\ & \text{and } \mathbf{val}_I(\alpha_2) = \underline{\mathbf{true}} \\ \underline{\mathbf{false}} & \text{if } \mathbf{val}_I(\alpha_1) = \underline{\mathbf{false}} \\ & \text{or } \mathbf{val}_I(\alpha_2) = \underline{\mathbf{false}} \end{cases}$$

$$\mathbf{val}_I(\beta) \quad = \quad \begin{cases} \underline{\mathbf{true}} & \text{if } \mathbf{val}_I(\beta_1) = \underline{\mathbf{true}} \\ & \text{or } \mathbf{val}_I(\beta_2) = \underline{\mathbf{true}} \\ \underline{\mathbf{false}} & \text{if } \mathbf{val}_I(\beta_1) = \underline{\mathbf{false}} \\ & \text{and } \mathbf{val}_I(\beta_2) = \underline{\mathbf{false}} \end{cases}$$

B. Beckert: Formal Verification of Software – p.6

Propositional Logic: Semantics

$$\textbf{val}_I(A \leftrightarrow B) \quad = \quad \begin{cases} \underline{\textbf{true}} & \text{if } \textbf{val}_I(A) = \textbf{val}_I(B) \\ \underline{\textbf{false}} & \text{if } \textbf{val}_I(A) \neq \textbf{val}_I(B) \end{cases}$$

B. Beckert: Formal Verification of Software – p.7

Predicate Logic: Syntax

Additional special symbols

"," "$\forall$" "$\exists$"

Object variables

Var $= \{x_0, x_1, \ldots\}$

Signature

Triple $\Sigma = \langle F_\Sigma, P_\Sigma, \alpha_\Sigma \rangle$ **consisting of**

- **set** F_Σ **of functions symbols**
- **set** P_Σ **of predicate symbols**
- **function** $\alpha_\Sigma : F_\Sigma \cup P_\Sigma \to \mathbb{N}$
  assigning aritys to function and predicate symbols

B. Beckert: Formal Verification of Software – p.8

Predicate Logic: Syntax

Terms

- **variables $x \in$ Var are terms**
- **if $f \in F_\Sigma$, $\alpha_\Sigma(f) = n$, and $t_1, \ldots, t_n$ terms, then $f(t_1, \ldots, t_n)$ is a term**

Atoms

If $p \in P_\Sigma$, $\alpha_\Sigma(p) = n$, and $t_1, \ldots, t_n$ terms, then $p(t_1, \ldots, t_n)$ is an atom

Predicate Logic: Syntax

Formulas

- **Atoms are formulas**
- **If A, B are formulas, $x \in$ Var, then**

$$\neg A, \quad (A \land B), \quad (A \lor B), \quad (A \to B), \quad (A \leftrightarrow B), \quad \forall x A, \quad \exists x A$$

are formulas

Literals

If A is an atom, then A and $\neg A$ are literals

B. Beckert: Formal Verification of Software – **p.10**

First-order Logic

Propositional logic

Assumes that the world contains facts

First-order logic

Assumes that the world contains

- **Objects**
  people, houses, numbers, theories, Donald Duck, colors, centuries, ...
- **Relations**
  red, round, prime, multistoried, ...
  brother of, bigger than, part of, has color, occurred after, owns, ...
- **Functions**
  $+$, middle of, father of, one more than, beginning of, ...

Syntax of First-order Logic: Basic Elements

Symbols

Constants	$KingJohn,\ 2,\ Karlsruhe,\ C, \ldots$
Predicates	$Brother,\ >,\ =,\ \ldots$
Functions	$Sqrt,\ LeftLegOf,\ \ldots$
Variables	$x,\ y,\ a,\ b,\ \ldots$
Connectives	$\wedge \quad \vee \quad \neg \quad \Rightarrow \quad \Leftrightarrow$
Quantifiers	$\forall \quad \exists$

Note

The equality predicate is always in the vocabulary
It is written in infix notation

Syntax of First-order Logic: Atomic Sentences

Atomic sentence

$$predicate\,(\,term_1, \ldots, term_n\,)$$

or

$$term_1 = term_2$$

Term

$$function\,(\,term_1, \ldots, term_n\,)$$

or

constant

or

variable

Syntax of First-order Logic: Atomic Sentences

Example

$$\mathit{Brother}\,(\,\mathit{KingJohn},\ \mathit{RichardTheLionheart}\,)$$

Syntax of First-order Logic: Atomic Sentences

Example

$$\underbrace{\mathit{Brother}}_{\text{predicate}}\,(\,\underbrace{\mathit{KingJohn}}_{\text{constant}},\ \underbrace{\mathit{RichardTheLionheart}}_{\text{constant}}\,)$$

Syntax of First-order Logic: Atomic Sentences

Example

$$\underbrace{\underbrace{\textit{Brother}}_{\text{predicate}}\ (\ \underbrace{\underbrace{\textit{KingJohn}}_{\text{constant}}}_{\text{term}},\ \underbrace{\underbrace{\textit{RichardTheLionheart}}_{\text{constant}}}_{\text{term}}\ )}$$

Syntax of First-order Logic: Atomic Sentences

Example

$$\underbrace{\underbrace{\textit{Brother}}_{\text{predicate}} \; (\; \underbrace{\underbrace{\textit{KingJohn}}_{\text{constant}}}_{\text{term}}, \; \underbrace{\underbrace{\textit{RichardTheLionheart}}_{\text{constant}}}_{\text{term}} \;)}_{\text{atomic sentence}}$$

Syntax of First-order Logic: Complex Sentences

Built from atomic sentences using connectives

$$\neg S \qquad S_1 \land S_2 \qquad S_1 \lor S_2 \qquad S_1 \Rightarrow S_2 \qquad S_1 \Leftrightarrow S_2$$

(as in propositional logic)

Syntax of First-order Logic: Complex Sentences

Built from atomic sentences using connectives

$$\neg S \qquad S_1 \wedge S_2 \qquad S_1 \vee S_2 \qquad S_1 \Rightarrow S_2 \qquad S_1 \Leftrightarrow S_2$$

(as in propositional logic)

Example

$$Sibling(\,KingJohn, Richard\,) \quad \Rightarrow \quad Sibling(\,Richard, KingJohn\,)$$

Syntax of First-order Logic: Complex Sentences

Built from atomic sentences using connectives

$$\neg S \qquad S_1 \wedge S_2 \qquad S_1 \vee S_2 \qquad S_1 \Rightarrow S_2 \qquad S_1 \Leftrightarrow S_2$$

(as in propositional logic)

Example

$$\underbrace{Sibling}_{\text{predicate}}(\underbrace{KingJohn}_{\text{term}}, \underbrace{Richard}_{\text{term}}) \quad \Rightarrow \quad \underbrace{Sibling}_{\text{predicate}}(\underbrace{Richard}_{\text{term}}, \underbrace{KingJohn}_{\text{term}})$$

Syntax of First-order Logic: Complex Sentences

Built from atomic sentences using connectives

$$\neg S \qquad S_1 \land S_2 \qquad S_1 \lor S_2 \qquad S_1 \Rightarrow S_2 \qquad S_1 \Leftrightarrow S_2$$

(as in propositional logic)

Example

$$\underbrace{\underbrace{Sibling}_{\text{predicate}}(\underbrace{KingJohn}_{\text{term}}, \underbrace{Richard}_{\text{term}})}_{\text{atomic sentence}} \Rightarrow \underbrace{\underbrace{Sibling}_{\text{predicate}}(\underbrace{Richard}_{\text{term}}, \underbrace{KingJohn}_{\text{term}})}_{\text{atomic sentence}}$$

Syntax of First-order Logic: Complex Sentences

Built from atomic sentences using connectives

$$\neg S \qquad S_1 \wedge S_2 \qquad S_1 \vee S_2 \qquad S_1 \Rightarrow S_2 \qquad S_1 \Leftrightarrow S_2$$

(as in propositional logic)

Example

$$\underbrace{\underbrace{\underbrace{Sibling}_{\text{predicate}}(\underbrace{KingJohn}_{\text{term}}, \underbrace{Richard}_{\text{term}})}_{\text{atomic sentence}} \Rightarrow \underbrace{\underbrace{Sibling}_{\text{predicate}}(\underbrace{Richard}_{\text{term}}, \underbrace{KingJohn}_{\text{term}})}_{\text{atomic sentence}}}_{\text{complex sentence}}$$

Semantics in First-order Logic

Models of first-order logic

Sentences are true or false with respect to models, which consist of

- a domain (also called universe)
- an interpretation

Domain

A non-empty (finite or infinite) set of arbitrary elements

Interpretation

Assigns to each

- constant symbol: a domain element
- predicate symbol: a relation on the domain (of appropriate arity)
- function symbol: a function on the domain (of appropriate arity)

Semantics in First-order Logic

Definition

An atomic sentence

$$predicate\,(\ term_1,\ \ldots,\ term_n\ )$$

is true in a certain model (that consists of a domain and an interpretation)
iff
the domain elements that are the interpretations of $term_1, \ldots, term_n$ are in the relation that is the interpretation of *predicate*

The truth value of a complex sentence in a model
is computed from the truth-values of its atomic sub-sentences
in the same way as in propositional logic

Universal Quantification: Syntax

Syntax

$\forall$ *variables sentence*

Example

"Everyone studying in Karlsruhe is smart:

$$\forall \underbrace{\; x \;}_{\text{variables}} \underbrace{(StudiesAt(x, Karlsruhe) \Rightarrow Smart(x))}_{\text{sentence}}$$

Universal Quantification: Semantics

Semantics

$\forall x P$ is true in a model
iff
for all domain elements d in the model:
P is true in the model when x is interpreted by d

Intuition

$\forall x\, P$ is roughly equivalent to the conjunction of all instances of P

Example $\forall\, x\ StudiesAt(x, Karlsruhe) \Rightarrow Smart(x)$ equivalent to:

$$
\begin{array}{ll}
 & StudiesAt(KingJohn, Karlsruhe) \Rightarrow Smart(KingJohn) \\
\wedge & StudiesAt(Richard, Karlsruhe) \Rightarrow Smart(Richard) \\
\wedge & StudiesAt(Karlsruhe, Karlsruhe) \Rightarrow Smart(Karlsruhe) \\
\wedge & \ldots
\end{array}
$$

Existential Quantification: Syntax

Syntax

$\exists$ *variables sentence*

Example

"Someone studying in Karlsruhe is smart:

$$\exists \underbrace{x}_{\text{variables}} \underbrace{(StudiesAt(x, Karlsruhe) \land Smart(x))}_{\text{sentence}}$$

Existential Quantification: Semantics

Semantics

$\exists x P$ is true in a model

iff

there is a domain element d in the model such that:

P is true in the model when x is interpreted by d

Intuition

$\exists x\, P$ is roughly equivalent to the disjunction of all instances of P

Example $\exists\, x\ StudiesAt(x, Karlsruhe) \wedge Smart(x)$ equivalent to:

$$
\begin{array}{ll}
 & StudiesAt(KingJohn, Karlsruhe) \wedge Smart(KingJohn) \\
\vee & StudiesAt(Richard, Karlsruhe) \wedge Smart(Richard) \\
\vee & StudiesAt(Karlsruhe, Karlsruhe) \wedge Smart(Karlsruhe) \\
\vee & \ldots
\end{array}
$$

Equality

Semantics

$term_1 = term_2$ is true under a given interpretation
if and only if
$term_1$ and $term_2$ have the same interpretation

Examples

$1 = 2$ and $\forall x \times (Sqrt(x), Sqrt(x)) = x$ are satisfiable
$2 = 2$ is valid

Properties of First-order Logic

Important notions

- validity
- satisfiability
- unsatisfiablity
- entailment

are defined for first-order logic in the same way as for propositional logic

Programmverifikation

Spezifikation von Programmeigenschaften

Hoare-Tripel (Syntax)

$$\{P\}\ \mathtt{S}\ \{Q\}$$

- P, Q: Formeln der Prädikatenlogik; beschreiben Mengen von Programmzuständen
- $\mathtt{S}$: Programm der while-Sprache

Hoare-Tripel (intuitive Semantik)

Wird $\mathtt{S}$ in einem Zustand gestartet, in dem P gilt, so gilt nach dessen Ausführung Q, *falls* $\mathtt{S}$ terminiert. (partielle Korrektheit).
Totale Korrektheit entspricht partieller Korrektheit + Terminierung.

Hoare-Tripel: Beispiele

$$\{x \doteq 0\} \;\; \mathsf{x = x + 1} \;\; \{x > 0\}$$

$$\{i \doteq a \;\wedge\; j \doteq b\} \; \mathsf{k = i; i = j; j = k} \; \{i \doteq b \;\wedge\; j \doteq a\}$$

Hoare-Tripel: Beispiele

$$\{x \doteq 0\} \;\; \mathsf{x = x + 1} \;\; \{x > 0\}$$

$$\{i \doteq a \;\wedge\; j \doteq b\} \; \mathsf{k = i;\, i = j;\, j = k} \; \{i \doteq b \;\wedge\; j \doteq a\}$$

Deduktive Programm-Verifikation: Bsp.

Zu zeigen:

$$\{i \doteq a \ \wedge \ j \doteq b\} \ \mathsf{k = i; i = j; j = k} \ \{i \doteq b \ \wedge \ j \doteq a\}$$

Beweis durch Hoare-Kalkül:

$$\dfrac{\dfrac{\dfrac{\{P \wedge i \doteq i\} \quad \{P \wedge i \doteq i\}}{\{P \wedge i \doteq i\} \ \mathsf{k := i} \ \{P \wedge k \doteq i\}}}{\vdots}}{\{P\} \ \mathsf{k := i} \ \{P \wedge k \doteq a\}} \qquad \dfrac{\dfrac{\quad}{\vdots}}{\{P \wedge k \doteq a\} \ \mathsf{i := j; j := k} \ \{Q\}}$$

$$\overline{\{P\} \ \mathsf{k := i; i := j; j := k} \ \{Q\}}$$

Weakest Precondition

Verstärkung der Vorbedingung

Für Formeln P, P': wenn $P' \rightarrow P$ und $\{P\}\ S\ \{Q\}$ gültig, dann ist auch $\{P'\}\ S\ \{Q\}$ gültig.

Beispiel

Aus

$$\underbrace{\{x \geq 0\}}_{P}\ \underbrace{\mathrm{x} = \mathrm{x} + 1}_{S}\ \underbrace{\{x > 0\}}_{Q}$$

folgt

$$\underbrace{\{x \doteq 0\}}_{P'}\ \mathrm{x} = \mathrm{x} + 1\ \{x > 0\}$$

Die Formel P ist für dieses Beispiel auch die *schwächste Vorbedingung* für S bzgl. Q (im Folgenden geschrieben: $P = \mathrm{wp}(S, Q)$).

Weakest Precondition

Verstärkung der Vorbedingung

Für Formeln P, P': wenn $P' \to P$ und $\{P\}\ S\ \{Q\}$ gültig, dann ist auch $\{P'\}\ S\ \{Q\}$ gültig.

Beispiel

Aus

$$\{\underbrace{x \geq 0}_{P}\}\ \underbrace{\mathrm{x = x + 1}}_{S}\ \{\underbrace{x > 0}_{Q}\}$$

folgt

$$\{\underbrace{x \doteq 0}_{P'}\}\ \mathrm{x = x + 1}\ \{x > 0\}$$

Die Formel P ist für dieses Beispiel auch die *schwächste Vorbedingung* für S bzgl. Q (im Folgenden geschrieben: $P = \mathrm{wp}(S, Q)$).

Zusammenhang wp und Hoare-Logik

$$\{P\}\ S\ \{Q\} \quad \text{gültig}$$

gdw.

$$P \to \mathrm{wp}(S, Q)$$

Berechnen der Weakest Precondition

- wp("skip", P) $=$ P
- wp("S; Prog", P) $=$ wp("S", wp("Prog", P))
- wp("x = e", P) $=$ $P[x/e]$
- wp("assert Q", P) $=$ $P \wedge Q$
- wp("assume Q", P) $=$ $Q \rightarrow P$
- wp("if (B) S1; else S2", P) $=$
  $(B \rightarrow$ wp("S1", P) $) \wedge (\neg B \rightarrow$ wp("S2", P))

Berechnen der Weakest Precondition

- wp("skip", P) $=$ P
- wp("S; Prog", P) $=$ wp("S", wp("Prog", P))
- wp("x = e", P) $=$ $P[x/e]$
- wp("assert Q", P) $=$ $P \wedge Q$
- wp("assume Q", P) $=$ $Q \rightarrow P$
- wp("if (B) S1; else S2", P) $=$
  $(B \rightarrow$ wp("S1", P) $) \wedge (\neg B \rightarrow$ wp("S2", P))

Berechnen der Weakest Precondition

Karlsruher Institut für Technologie

- wp("skip", P) $=$ P
- wp("S; Prog", P) $=$ wp("S", wp("Prog", P))
- wp("x = e", P) $=$ $P[x/e]$
- wp("assert Q", P) $=$ $P \wedge Q$
- wp("assume Q", P) $=$ $Q \rightarrow P$
- wp("if (B) S1; else S2", P) $=$
  $(B \rightarrow \mathrm{wp}(\text{"S1"}, P)\ ) \wedge (\neg B \rightarrow \mathrm{wp}(\text{"S2"}, P))$

Berechnen der Weakest Precondition

- `wp( "skip",` P`)` $=$ P
- `wp( "S; Prog",` P`)` $=$ `wp("S", wp("Prog",` P`))`
- `wp( "x = e",` P`)` $=$ $P[x/e]$
- `wp( "assert Q",` P`)` $=$ $P \wedge Q$
- `wp( "assume Q",` P`)` $=$ $Q \rightarrow P$
- `wp( "if (B) S1; else S2",` P`)` $=$
  $(B \rightarrow \mathrm{wp}(\text{"S1"}, P)) \wedge (\neg B \rightarrow \mathrm{wp}(\text{"S2"}, P))$

Berechnen der Weakest Precondition

- `wp( "skip", `P`)` $=$ P
- `wp( "S; Prog", `P`)` $=$ `wp("S", wp("Prog", `P`))`
- `wp( "x = e", `P`)` $=$ $P[x/e]$
- `wp( "assert Q", `P`)` $=$ $P \land Q$
- `wp( "assume Q", `P`)` $=$ $Q \rightarrow P$
- `wp( "if (B) S1; else S2", `P`)` $=$
  $(B \rightarrow \mathrm{wp}(\text{"S1"}, P)) \land (\neg B \rightarrow \mathrm{wp}(\text{"S2"}, P))$

Berechnen der Weakest Precondition

- wp("skip", P) $=$ P
- wp("S; Prog", P) $=$ wp("S", wp("Prog", P))
- wp("x = e", P) $=$ $P[x/e]$
- wp("assert Q", P) $=$ $P \wedge Q$
- wp("assume Q", P) $=$ $Q \rightarrow P$
- wp("if (B) S1; else S2", P) $=$
  $(B \rightarrow \mathrm{wp}(\text{"S1"}, P)) \wedge (\neg B \rightarrow \mathrm{wp}(\text{"S2"}, P))$

Berechnen der Weakest Precondition

- wp("skip", P) $=$ P
- wp("S; Prog", P) $=$ wp("S", wp("Prog", P))
- wp("x = e", P) $=$ $P[x/e]$
- wp("assert Q", P) $=$ $P \wedge Q$
- wp("assume Q", P) $=$ $Q \rightarrow P$
- wp("if (B) S1; else S2", P) $=$ $(B \rightarrow$ wp("S1", P) $) \wedge (\neg B \rightarrow$ wp("S2", P))

Berechnen der Weakest Precondition

- `wp( "skip", ` P`)` $=$ P
- `wp( "S; Prog", ` P`)` $=$ `wp("S", wp("Prog", ` P`))`
- `wp( "x = e", ` P`)` $=$ $P[x/e]$
- `wp( "assert Q", ` P`)` $=$ $P \wedge Q$
- `wp( "assume Q", ` P`)` $=$ $Q \rightarrow P$
- `wp( "if (B) S1; else S2", ` P`)` $=$
  $(B \rightarrow$ `wp("S1",` P`)` $) \wedge (\neg B \rightarrow$ `wp("S2",` P`))`

Berechnen der Weakest Precondition

- wp("skip", P) $=$ P
- wp("S; Prog", P) $=$ wp("S", wp("Prog", P))
- wp("x = e", P) $=$ $P[x/e]$
- wp("assert Q", P) $=$ $P \wedge Q$
- wp("assume Q", P) $=$ $Q \rightarrow P$
- wp("if (B) S1; else S2", P) $=$
  $(B \rightarrow$ wp("S1", P) $) \wedge (\neg B \rightarrow$ wp("S2", P))

Berechnen der Weakest Precondition

- $\texttt{wp("skip"}, \ P) \ = \ P$
- $\texttt{wp("S; Prog"}, \ P) \ = \ \texttt{wp("S"}, \ \texttt{wp("Prog"}, \ P))$
- $\texttt{wp("x = e"}, \ P) \ = \ P[x/e]$
- $\texttt{wp("assert Q"}, \ P) \ = \ P \wedge Q$
- $\texttt{wp("assume Q"}, \ P) \ = \ Q \rightarrow P$
- $\texttt{wp("if (B) S1; else S2"}, \ P) \ =$
  $(B \rightarrow \mathrm{wp}(\texttt{"S1"}, P)) \wedge (\neg B \rightarrow \mathrm{wp}(\texttt{"S2"}, P))$

Programme mit Schleifen

Vor Berechnung der WP eines Programms mit Schleifen werden alle Schleifen k-mal ausgerollt.

Für $k = 1$:

```
assume(x >= 0);
i = 0;
y = 0;
while (i < x)
{
  y = y + 2 * i + 1;
  i = i + 1;

}
assert(y == x*x);
```

Programme mit Schleifen

Vor Berechnung der WP eines Programms mit Schleifen werden alle Schleifen k-mal ausgerollt.

Für $k = 1$:

```
assume(x >= 0);
i = 0;
y = 0;
if (i < x)
{
  y = y + 2 * i + 1;
  i = i + 1;
  if (i < x) assume(false)
}
assert(y == x*x);
```

Programme mit Schleifen

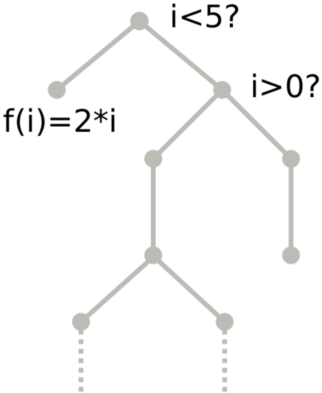
Für $k = 2$:

```
assume(x >= 0);
i = 0;
y = 0;
if (i < x)
{
  y = y + 2 * i + 1;
  i = i + 1;
  if (i < x) {
    y = y + 2 * i + 1;
    i = i + 1;
    if (i < x) assume(false)
  }
}
assert(y == x*x);
```

Verschiedene Nachweis-Methoden

```
int f(int i) {
  if (i < 5)
    return 2*i
  else {

    while (i > 0)
      { ... }
  }
  ...
}
```



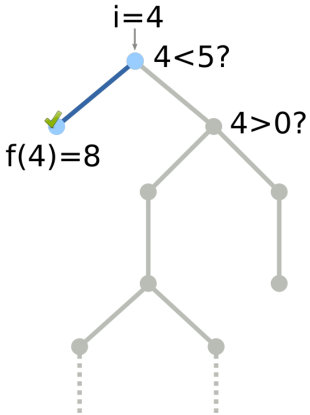
Nachweisverfahren

- Testen
- Bounded Model Checking
- Deduktive Verifikation

Verschiedene Nachweis-Methoden

```
int f(int i) {
  if (i < 5)
    return 2*i
  else {

    while (i > 0)
      { ... }
  }
  ...
}
```



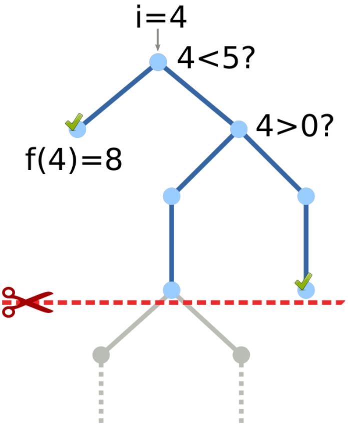
Nachweisverfahren

- Testen
- Bounded Model Checking
- Deduktive Verifikation

Verschiedene Nachweis-Methoden

```
int f(int i) {
  if (i < 5)
    return 2*i
  else {

    while (i > 0)
      { ... }
  }
  ...
}
```



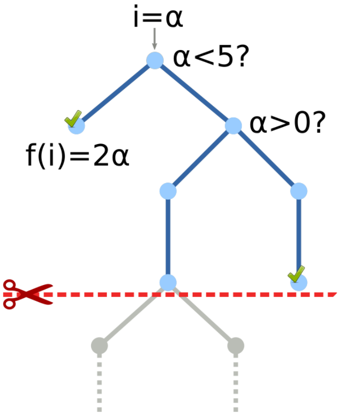
Nachweisverfahren

- Testen
- Bounded Model Checking
- Deduktive Verifikation

Verschiedene Nachweis-Methoden

```
int f(int i) {
  if (i < 5)
    return 2*i
  else {

    while (i > 0)
      { ... }
  }
  ...
}
```

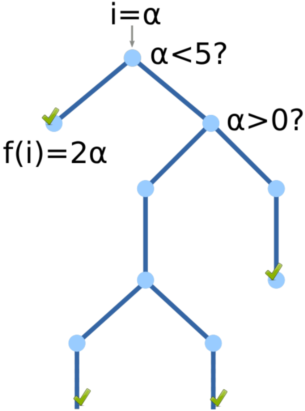


Nachweisverfahren

- Testen
- Bounded Model Checking
- Deduktive Verifikation

Verschiedene Nachweis-Methoden

```
int f(int i) {
  if (i < 5)
    return 2*i
  else {
    //invariant i%2 = 0
    while (i > 0)
      { ... }
  }
  ...
}
```

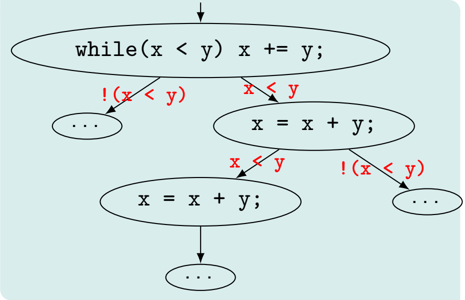


Nachweisverfahren

- Testen
- Bounded Model Checking
- Deduktive Verifikation

(Software) Bounded Model Checking

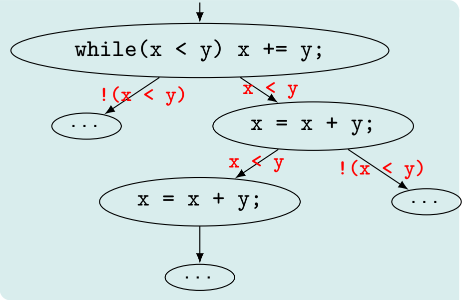
Software Bounded Model Checking (SBMC)



- Static program analysis using symbolic execution
- Exhaustive check by unwinding the control flow graph
- Bounded in number of loop unwindings and recursions

- SBMC converts program into logical equations, sent to SAT solver
- Special „unwinding assertion“ claims added to check whether longer program paths may be possible
- Checks whether specified assertions can be violated

Software Bounded Model Checking (SBMC)



- Static program analysis using symbolic execution
- Exhaustive check by unwinding the control flow graph
- Bounded in number of loop unwindings and recursions

- SBMC converts program into logical equations, sent to SAT solver
- Special „unwinding assertion“ claims added to check whether longer program paths may be possible
- Checks whether specified assertions can be violated

Specifying & Verifying Properties in SBMC

Specification

- Properties specified using `assume` and `assert` statements
- A program `Prog` is **correct** if

$$\mathtt{Prog} \wedge \bigwedge \mathtt{assume} \Rightarrow \bigwedge \mathtt{assert}$$

  is valid.
- `Prog` is automatically generated logical encoding of the program

Verification

- Checking properties for programs generally undecidable
- SBMC analyses only program runs up to **bounded** length
- Property checking becomes decidable by logical encoding
- Can be decided using SAT- or SMT-solver

Specifying & Verifying Properties in SBMC

Specification

- Properties specified using `assume` and `assert` statements
- A program `Prog` is **correct** if

$$\texttt{Prog} \land \bigwedge \texttt{assume} \Rightarrow \bigwedge \texttt{assert}$$

  is valid.
- `Prog` is automatically generated logical encoding of the program

Verification

- Checking properties for programs generally undecidable
- SBMC analyses only program runs up to **bounded** length
- Property checking becomes decidable by logical encoding
- Can be decided using SAT- or SMT-solver

CBMC

CBMC: Ein Beispiel zum Ausprobieren

```c
#include <stdlib.h>
#include <stdint.h>
#include <assert.h>

int nondet_uint();
#define LENGTH 2
#define assume(x) __CPROVER_assume(x)

int main(int argc, char *argv[]) {
    unsigned int a[LENGTH];
    for (unsigned int i = 0; i < LENGTH; i++) {
        a[i] = nondet_uint();
        assume (0 < a[i]);
    }
    assert (0 < a[0] + a[1]);
    return 0;
}
```

CBMC: Ein Beispiel zum Ausprobieren

```
#include <stdlib.h>
#include <stdint.h>
#include <assert.h>

int nondet_uint();
#define LENGTH 2
#define assume(x) __CPROVER_assume(x)

int main(int argc, char *argv[]) {
    unsigned int a[LENGTH];
    for (unsigned int i = 0; i < LENGTH; i++) {
        a[i] = nondet_uint();
        assume (0 < a[i]);
    }
    assert (0 < a[0] + a[1]);
    return 0;
```

- Analysieren Sie das Programm mit CBMC
- Probieren Sie auch die Option `--trace`
- Konstanten (`LENGTH`) können übergeben werden (`-D LENGTH=2`)
- Manchmal muss man mittels `--unwind` eine direkte Schranke geben

CBMC: Ein Beispiel zum Ausprobieren

```c
#include <stdlib.h>
#include <stdint.h>
#include <assert.h>

int nondet_uint();
#define LENGTH 2
#define assume(x) __CPROVER_assume(x)

int main(int argc, char *argv[]) {
    unsigned int a[LENGTH];
    for (unsigned int i = 0; i < LENGTH; i++) {
        a[i] = nondet_uint();
        assume (0 < a[i] && a[i] < 3);
    }
    assert (0 < a[0] + a[1]);
    return 0;
```

- Analysieren Sie das Programm mit CBMC
- Probieren Sie auch die Option `--trace`
- Konstanten (`LENGTH`) können übergeben werden (`-D LENGTH=2`)
- Manchmal muss man mittels `--unwind` eine direkte Schranke geben

CBMC: Und jetzt Wahlverfahren ...

```
unsigned int voting(unsigned int votes[V]) { .. }
void anonymity(unsigned int votes1[V],
               unsigned int votes2[V],
               unsigned int c, unsigned int d,
               unsigned int v, unsigned int w) {
  assume (0<c && c<=C); assume (0<d && d<=C);
  assume (0<=v && v<V); assume (0<=w && w<V);
  for (unsigned int i = 0; i < V; i++) {
    assume (0 < votes1[i] && votes1[i] <= C);
    assume (0 < votes2[i] && votes2[i] <= C);
    if (i != v && i != w)
      assume (votes1[i] == votes2[i]);
  }
  assume (votes1[v] == c && votes1[w] == d);
  assume (votes2[v] == d && votes2[w] == c);
  unsigned int elect1, elect2 = ..;
  assert (elect1 == elect2);
}
```

CBMC: ... mit `main`-Methode

```c
int main(int argc, char *argv[]) {
    unsigned int v1[V], v2[V];
    for (unsigned int i = 0; i < V; i++) {
        v1[i] = nondet_uint();
        v2[i] = nondet_uint();
    }
    unsigned int cand1  = nondet_uint();
    unsigned int cand2  = nondet_uint();
    unsigned int voter1 = nondet_uint();
    unsigned int voter2 = nondet_uint();

    anonymity(v1,v2,cand1,cand2,voter1,voter2);
    return 0;
}
```

Vielen Dank

für die Aufmerksamkeit!

Gibt es Fragen?

Vielen Dank

für die Aufmerksamkeit!

Gibt es Fragen?