Strukturelles Testen

Software-Praktikum

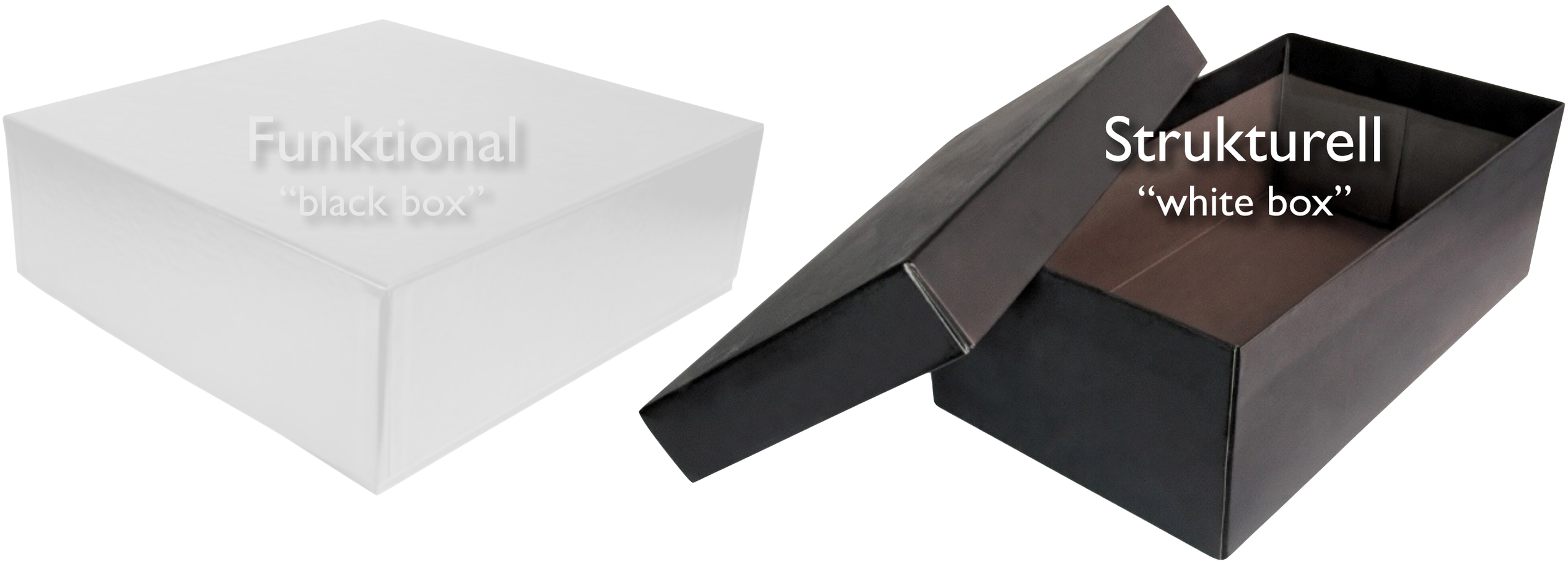
Andreas Zeller • Saarland University

Test-Taktiken

- Tests basieren auf *Spezifikation*
- Test deckt soviel *spezifiziertes* Verhalten wie möglich ab

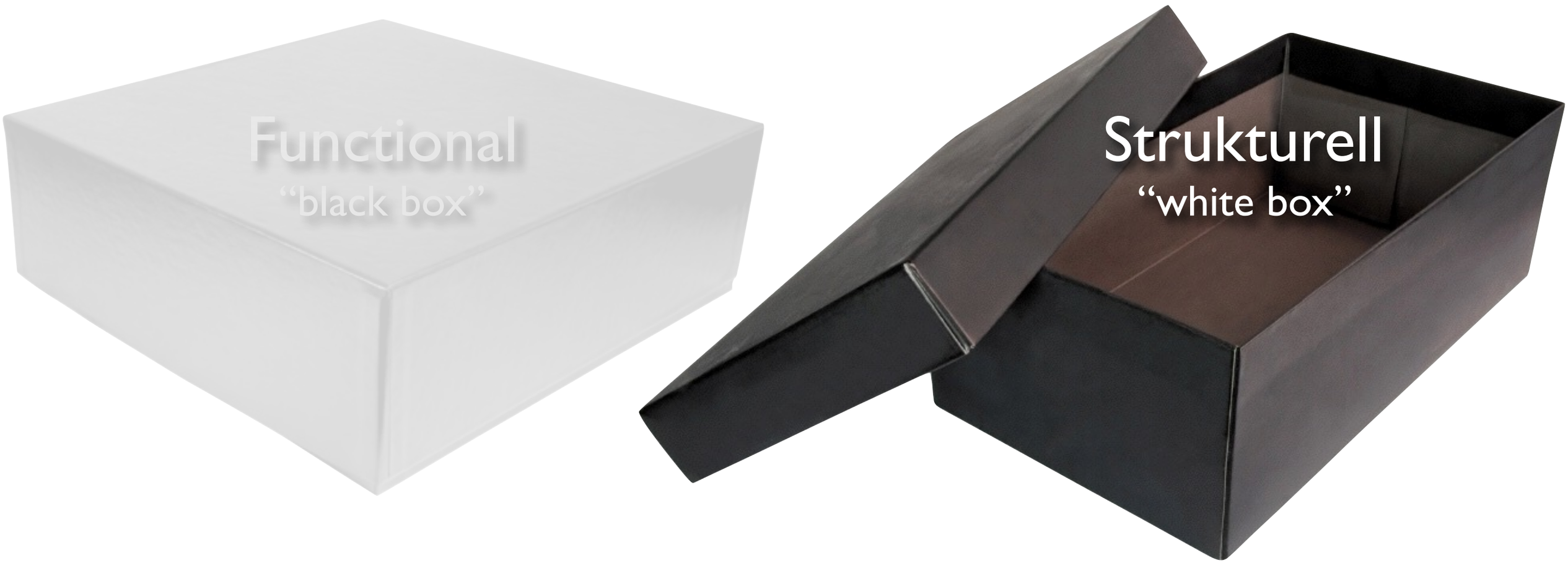
- Tests basieren auf *Code*
- Test deckt soviel *implementiertes* Verhalten wie möglich ab

Warum Strukturell?



- Wenn ein Teil des Programms *nie* ausgeführt wird, bleiben Defekte dort unentdeckt
  "Teile" = Anweisungen, Module, Bedingungen…
- Attraktiv, da *automatisierbar*

Warum Strukturell?



- Ergänzt funktionale Tests
  Erst funktionale Test ausführen, dann nicht abgedeckten Code testen
- Kann *Details* abdecken, die in der abstrakten Spezifikation fehlen

Herausforderung

```
class Roots {
    // Löse ax² + bx + c = 0
    public roots(double a, double b, double c)
    { … }

    // Ergebnis: Werte für x
    double root_one, root_two;
}
```

- Welche Werte für *a*, *b*, *c* sollten wir testen?

  Wenn a, b, c 32-bit integers sind, haben wir $(2^{32})^3 \approx 10^{28}$ legal inputs
  Mit 1.000.000.000.000 Tests/s brauchen wir immer noch 2.5 Mrd. Jahre

Der Code

```
// Löse ax² + bx + c = 0
public roots(double a, double b, double c)
{
    double q = b * b - 4 * a * c;
    if (q > 0 && a ≠ 0) {
        // Code für zwei Lösungen
    }

    else if (q == 0) {
        // Code für eine Lösung
    }

    else {
        // Code für keine Lösungen
    }
}
```

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Teste diesen Fall

und diesen

und diesen!

The Test Cases

```
// Löse ax² + bx + c = 0
public roots(double a, double b, double c)
{
    double q = b * b - 4 * a * c;
    if (q > 0 && a ≠ 0) {
        // Code für zwei Lösungen
    }

    else if (q == 0) {
        // Code für eine Lösung
    }

    else {
        // Code für keine Lösungen
    }
}
```

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

(a, b, c) = (3, 4, 1)

(a, b, c) = (0, 0, 1)

(a, b, c) = (3, 2, 1)

A Defect

```
// Löse ax² + bx + c = 0
public roots(double a, double b, double c)
{
    double q = b * b - 4 * a * c;
    if (q > 0 && a ≠ 0) {
        // Code für zwei Lösungen
    }

    else if (q == 0) {
        x = (-b) / (2 * a);
    }

    else {
        // Code für keine Lösungen
    }
}
```

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

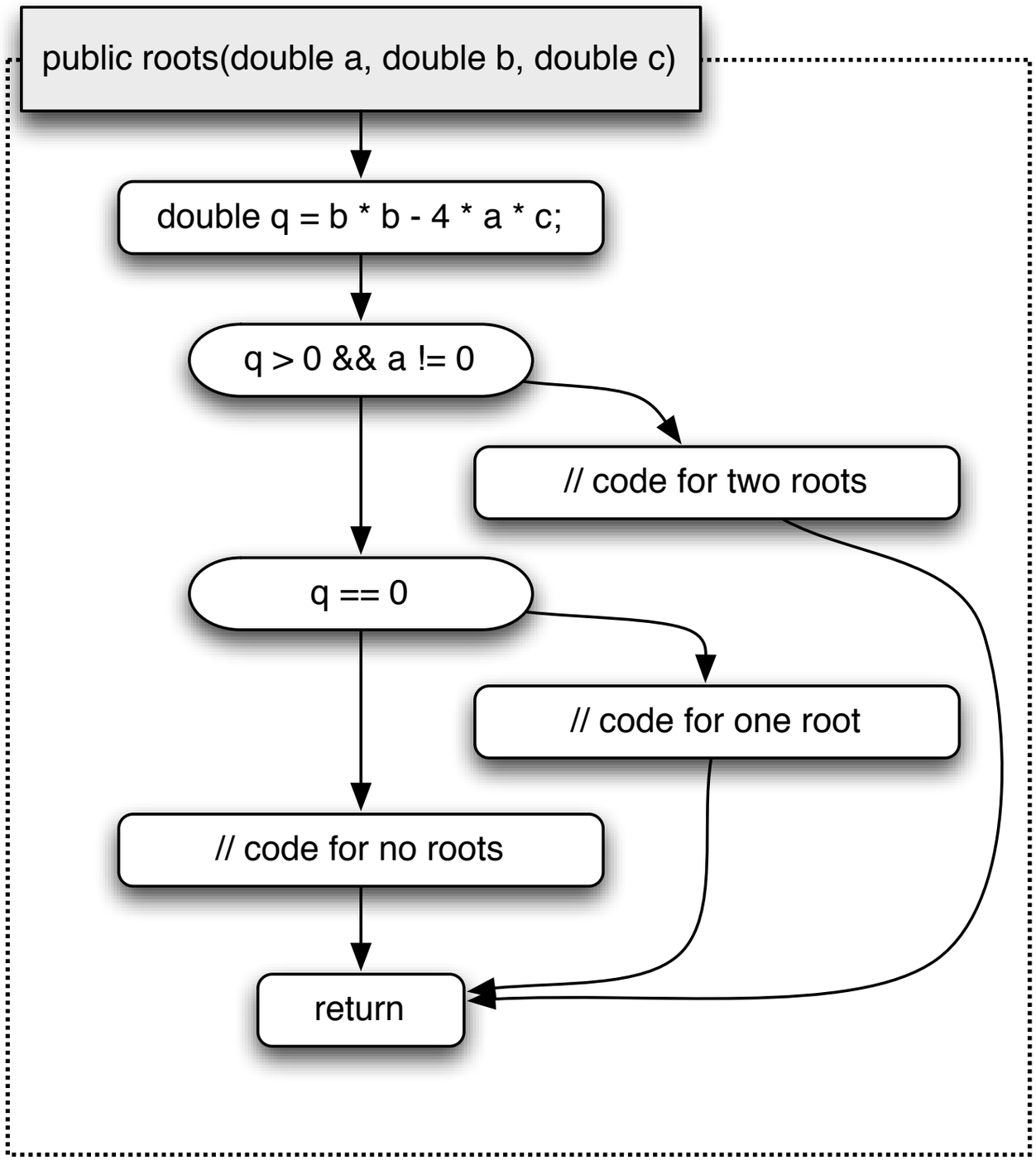
code muss a = 0 berücksichtigen

(a, b, c) = (0, 0, 1)

Struktur ausdrücken

```
// Löse ax² + bx + c = 0
public roots(double a, double b, double c)
{
    double q = b * b - 4 * a * c;
    if (q > 0 && a ≠ 0) {
        // Code für zwei Lösungen
    }

    else if (q == 0) {
        x = (-b) / (2 * a);
    }

    else {
        // Code für keine Lösungen
    }
}
```

Kontrollflussgraph

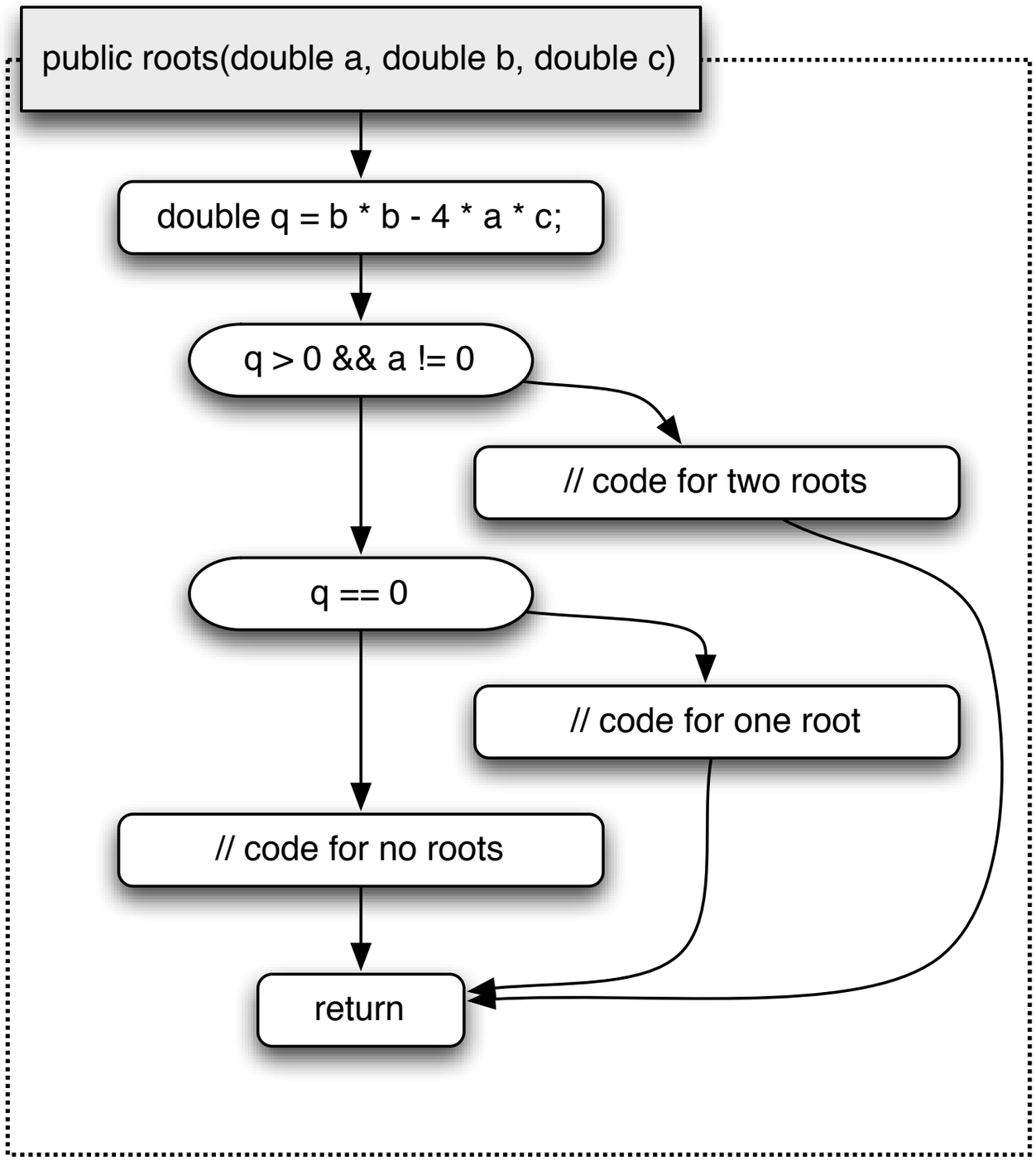


- Ein *Kontrollflussgraph* (CFG) drückt mögliche Ausführungspfade eines Programms aus

- *Knoten* sind *Basic Blocks* – Anweisungsfolgen mit 1 Eingang und 1 Ausgang

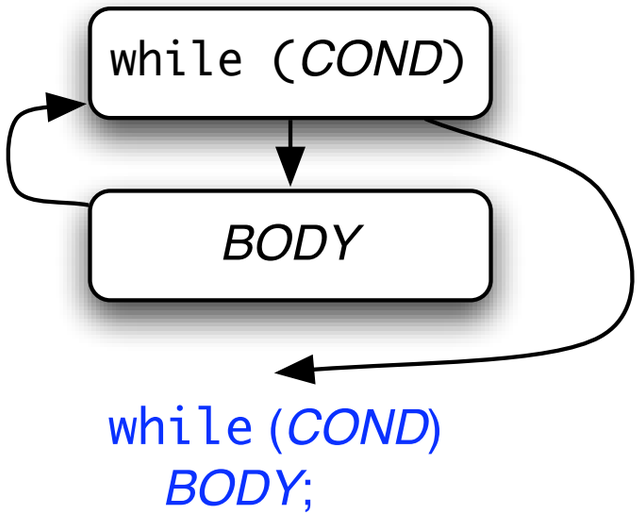
- *Kanten* stellen *Kontrollfluss* dar – die Möglichkeit, dass ein Basic Block nach einem anderen ausgeführt wird

Strukturelles Testen

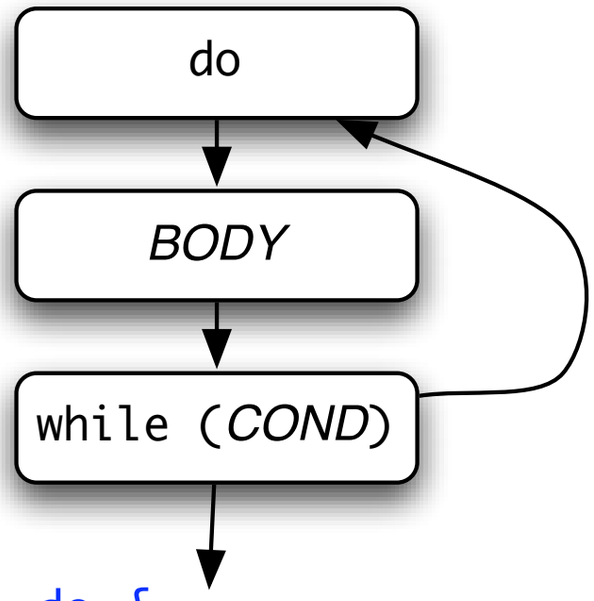


- Aus dem CFG lassen sich *Testkriterien* ableiten
- Je mehr Teile abgedeckt (ausgeführt) werden, um so höher die Chance eines Tests, einen Fehler zu finden
- "Teile" können sein: Knoten, Kanten, Pfade, Bedingungen…

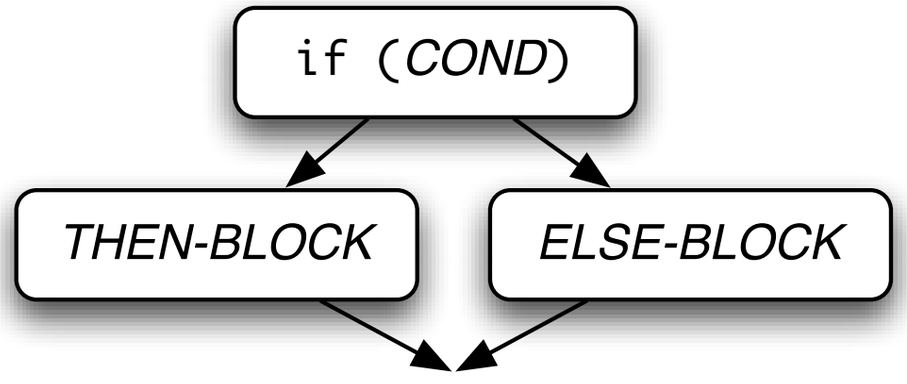
Kontrollfluss-Muster



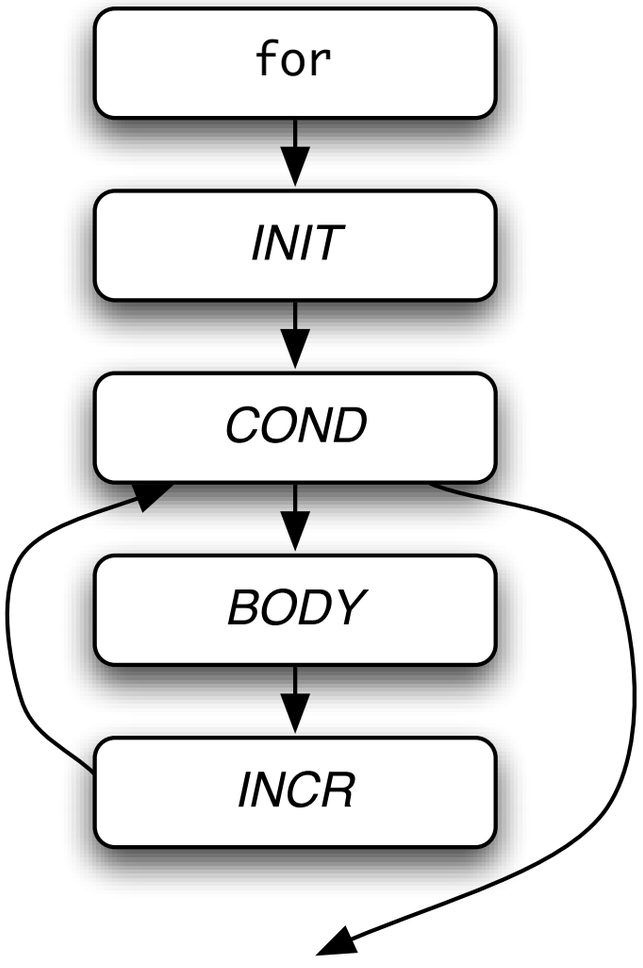

```
while (COND)
  BODY;
```




```
do {
    BODY
}while (COND);
```




```
if (COND)
  THEN-BLOCK;
else
  ELSE-BLOCK;
```




```
for (INIT; COND; INCR)
  BODY;
```

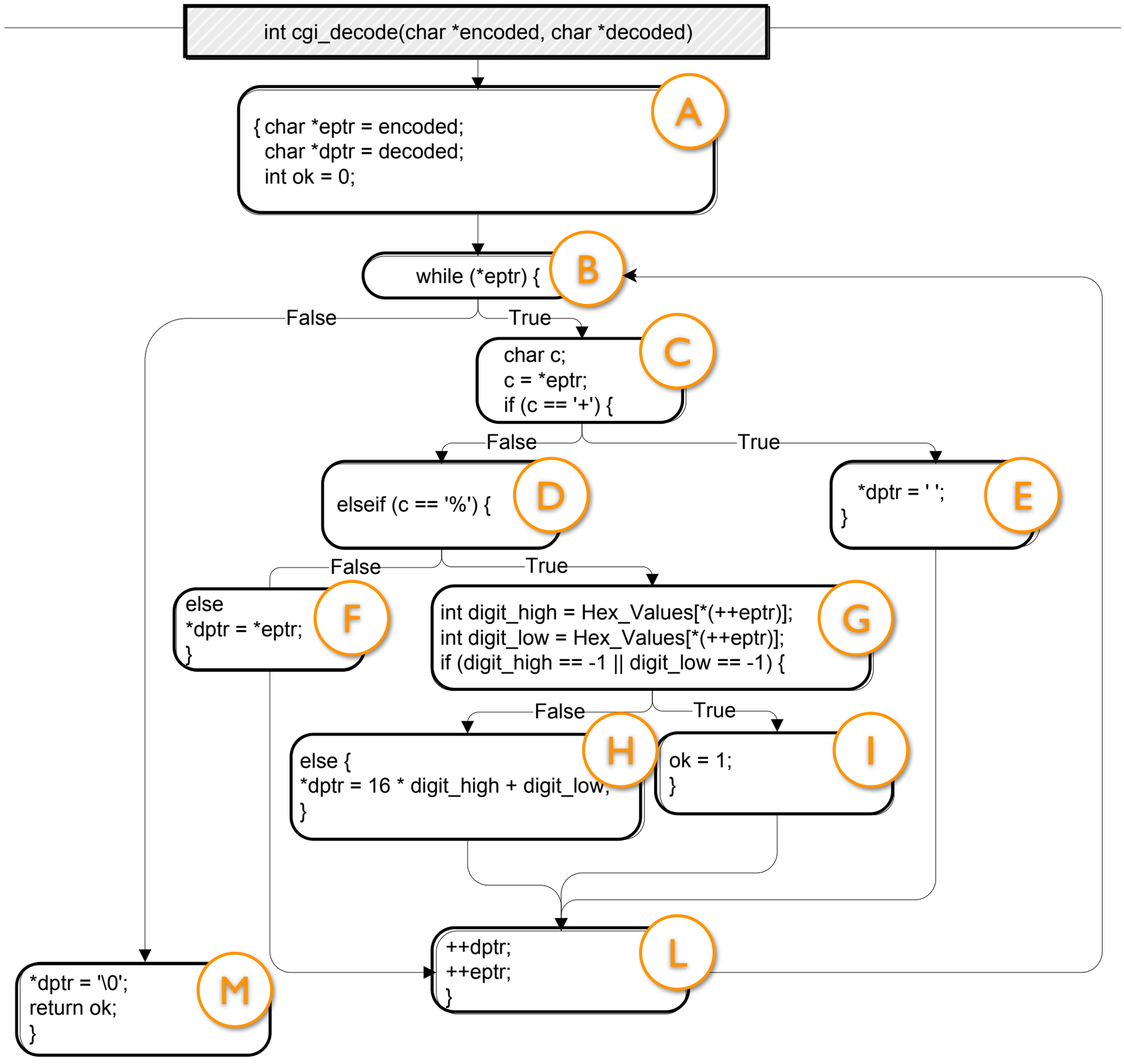
cgi_decode

```
/**
  * @title cgi_decode
  * @desc
  * Translate a string from the CGI encoding to plain ascii text
  * '+' becomes space, %xx becomes byte with hex value xx,
  * other alphanumeric characters map to themselves
  *
  * returns 0 for success, positive for erroneous input
  * 1 = bad hexadecimal digit
  */

int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;
```

A

```
while (*eptr)  /* loop to end of string ('\0' character) */
{
    char c;
    c = *eptr;
    if (c == '+') {  /* '+' maps to blank */
        *dptr = ' ';
    } else if (c == '%') { /* '%xx' is hex for char xx */
        int digit_high = Hex_Values[*(++eptr)];
        int digit_low  = Hex_Values[*(++eptr)];
        if (digit_high == -1 || digit_low == -1)
            ok = 1; /* Bad return code */
        else
            *dptr = 16 * digit_high + digit_low;
    } else { /* All other characters map to themselves */
        *dptr = *eptr;
    }
    ++dptr; ++eptr;
}

*dptr = '\0';   /* Null terminator for string */
return ok;
}
```

B C E D G I H F L M


int cgi_decode(char *encoded, char *decoded)
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
A
while (*eptr) {
B
False
True
char c;
c = *eptr;
if (c == '+') {
C
False
True
elseif (c == '%') {
D
*dptr = ' ';
}
E
False
True
else
*dptr = *eptr;
}
F
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
G
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
H
ok = 1;
}
I
++dptr;
++eptr;
}
L
*dptr = '\0';
return ok;
}
M

int cgi_decode(char *encoded, char *decoded)

"test"

```
{ char *eptr = encoded;
  char *dptr = decoded;
  int ok = 0;
```

```
while (*eptr) {
```

False

True

```
char c;
c = *eptr;
if (c == '+') {
```

False

True

```
elseif (c == '%') {
```

```
  *dptr = ' ';
}
```

E

False

True

```
else
*dptr = *eptr;
}
```

```
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
```

G

False

True

```
else {
*dptr = 16 * digit_high + digit_low;
}
```

H

```
ok = 1;
}
```

I

```
++dptr;
++eptr;
}
```

```
*dptr = '\0';
return ok;
}
```

int cgi_decode(char *encoded, char *decoded)

{ char *eptr = encoded;
  char *dptr = decoded;
  int ok = 0;

“test”

“a+b”

while (*eptr) {

False

True

char c;
c = *eptr;
if (c == '+') {

False

True

elseif (c == '%') {

*dptr = ' ';
}

False

True

else
*dptr = *eptr;
}

int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {

G

False

True

else {
*dptr = 16 * digit_high + digit_low;
}

H

ok = 1;
}

I

++dptr;
++eptr;
}

*dptr = '\0';
return ok;
}

int cgi_decode(char *encoded, char *decoded)

{ char *eptr = encoded;
  char *dptr = decoded;
  int ok = 0;

while (*eptr) {

False

True

char c;
c = *eptr;
if (c == '+') {

False

True

elseif (c == '%') {

*dptr = ' ';
}

False

True

else
*dptr = *eptr;
}

int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {

False

True

else {
*dptr = 16 * digit_high + digit_low;
}

ok = 1;
}

++dptr;
++eptr;
}

*dptr = '\0';
return ok;
}

“test”

“a+b”

“%3d”

int cgi_decode(char *encoded, char *decoded)

{ char *eptr = encoded;
  char *dptr = decoded;
  int ok = 0;

while (*eptr) {

False

True

char c;
c = *eptr;
if (c == '+') {

False

True

elseif (c == '%') {

*dptr = ' ';
}

False

True

else
*dptr = *eptr;
}

int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {

False

True

else {
*dptr = 16 * digit_high + digit_low;
}

ok = 1;
}

++dptr;
++eptr;
}

*dptr = '\0';
return ok;
}

“test”

“a+b”

“%3d”

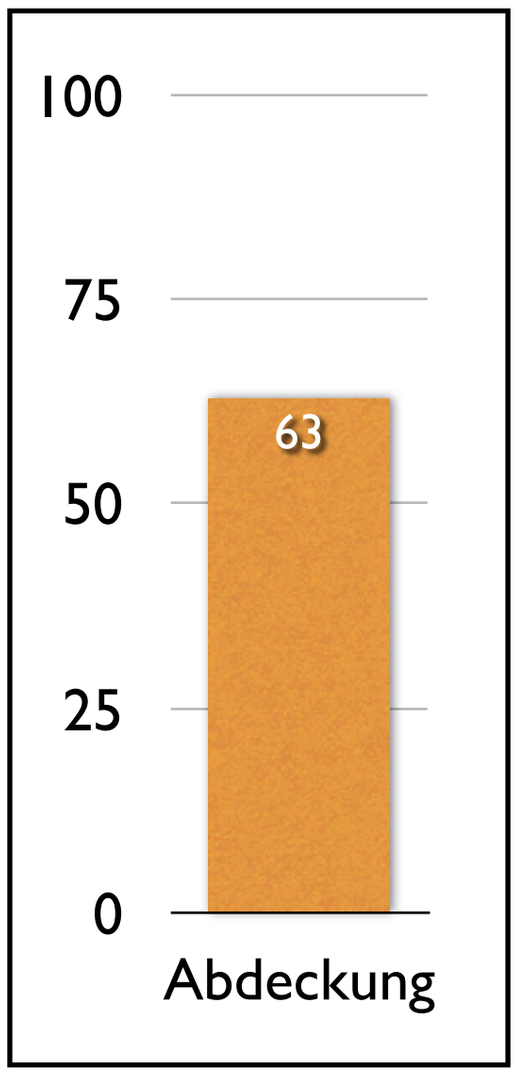
“%g”

Testkriterien

- Wie wissen wir, ob ein Test "gut genug" ist?
- Ein Testkriterium ist ein Prädikat, das für ein Paar $\langle Programm, Testsuite \rangle$ erfüllt oder nicht erfüllt ist
- Gewöhnlich als Regel ausgedrückt – z.B. "alle Anweisungen müssen abgedeckt sein"

Anweisungsabdeckung

- Testkriterium: Jede Anweisung (oder Knoten im CFG) muss wenigstens 1x ausgeführt werden
- Hintergrund: Ein Defekt in einer Anweisung kann nur gefunden werden, wenn die Anweisung ausgeführt wird
- Abdeckung: $\frac{\text{\# ausgeführte Anweisungen}}{\text{\# Anweisungen}}$



int cgi_decode(char *encoded, char *decoded)

“test”

{ char *eptr = encoded;
  char *dptr = decoded;
  int ok = 0;

while (*eptr) {

False

True

char c;
c = *eptr;
if (c == '+') {

False

True

elseif (c == '%') {

*dptr = ' ';
}

E

False

True

else
*dptr = *eptr;
}

int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {

G

False

True

else {
*dptr = 16 * digit_high + digit_low;
}

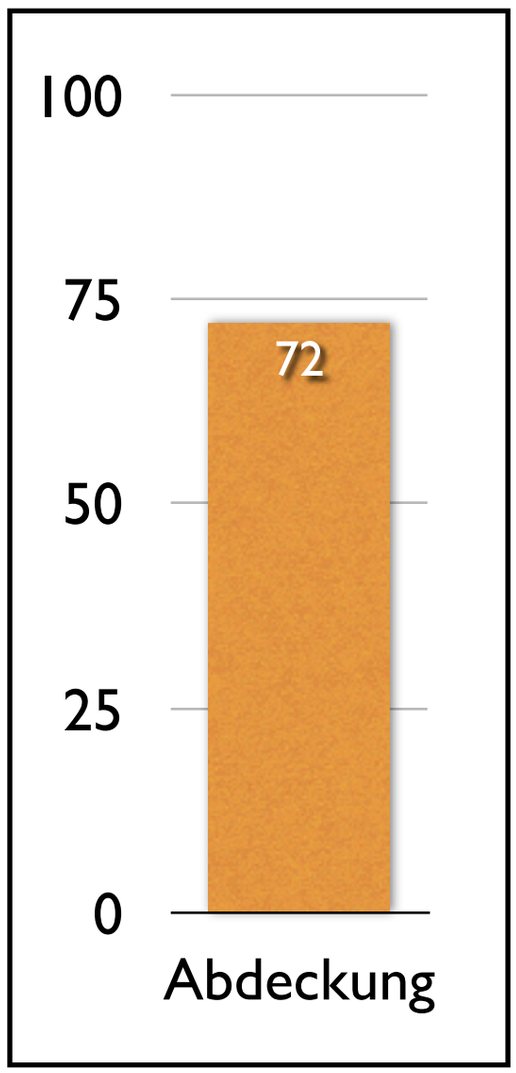
H

ok = 1;
}

I

++dptr;
++eptr;
}

*dptr = '\0';
return ok;
}



int cgi_decode(char *encoded, char *decoded)

“test”

“a+b”

{ char *eptr = encoded;
  char *dptr = decoded;
  int ok = 0;

while (*eptr) {

False

True

char c;
c = *eptr;
if (c == '+') {

False

True

elseif (c == '%') {

*dptr = ' ';
}

False

True

else
*dptr = *eptr;
}

int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {

G

False

True

else {
*dptr = 16 * digit_high + digit_low;
}

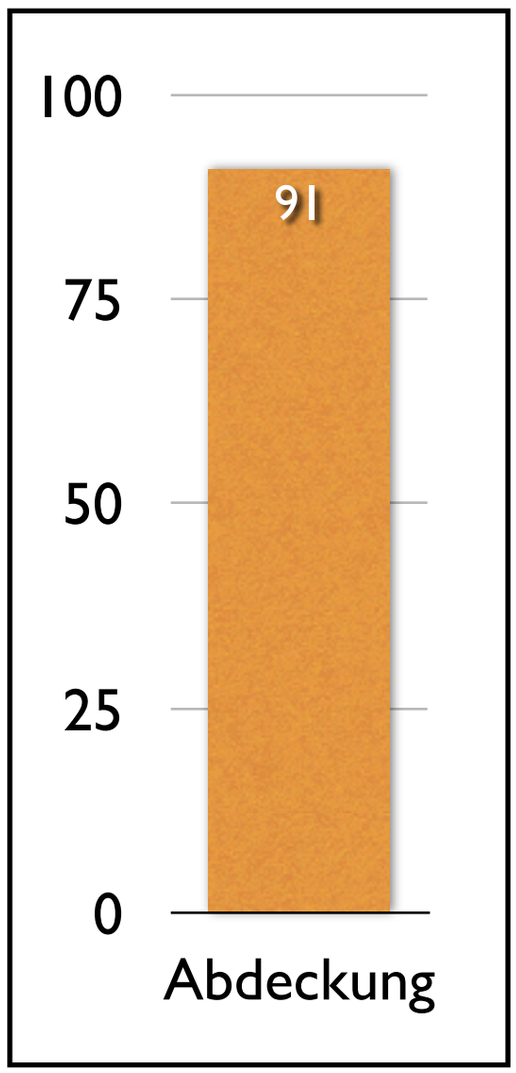
H

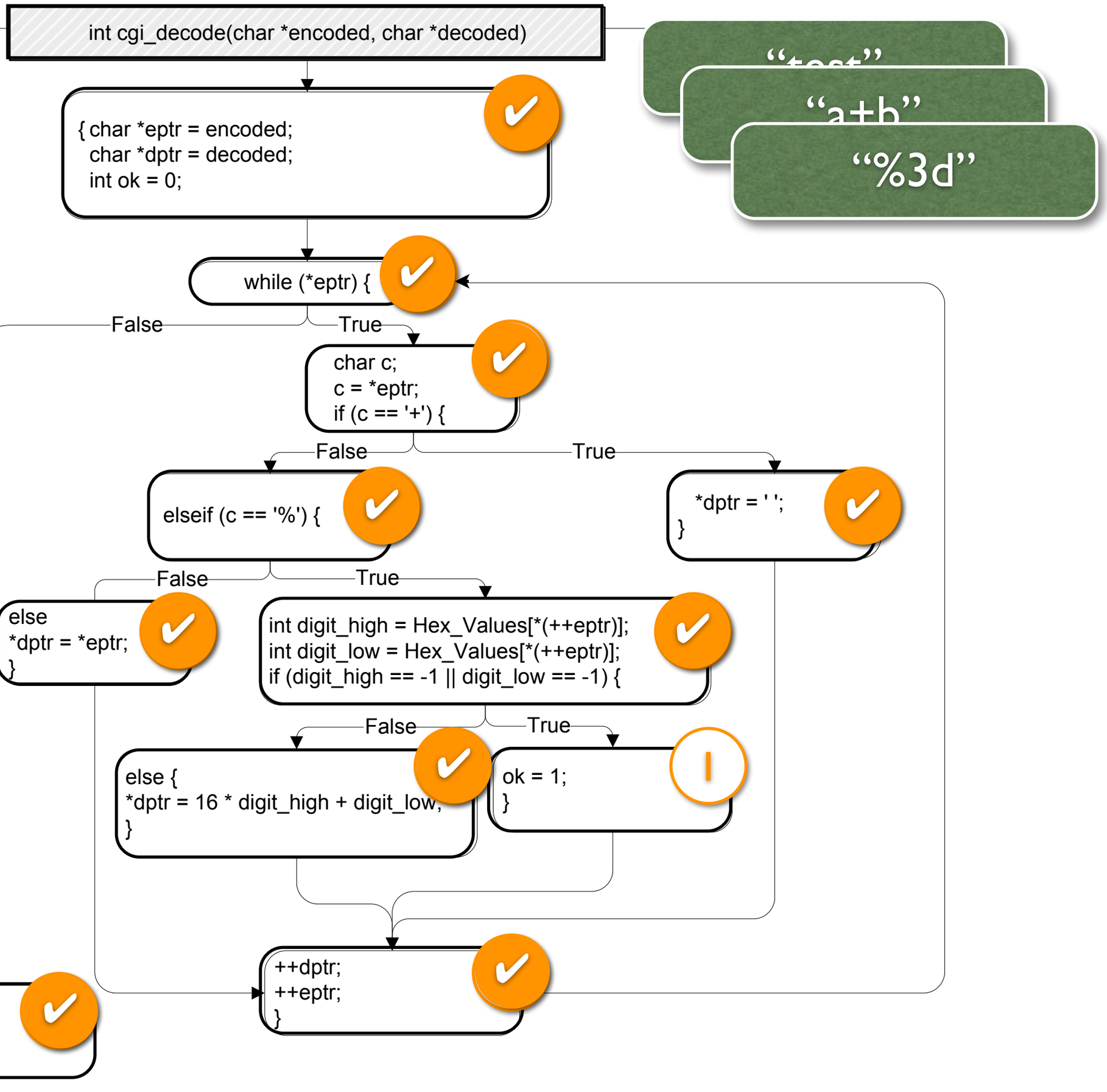
ok = 1;
}

I

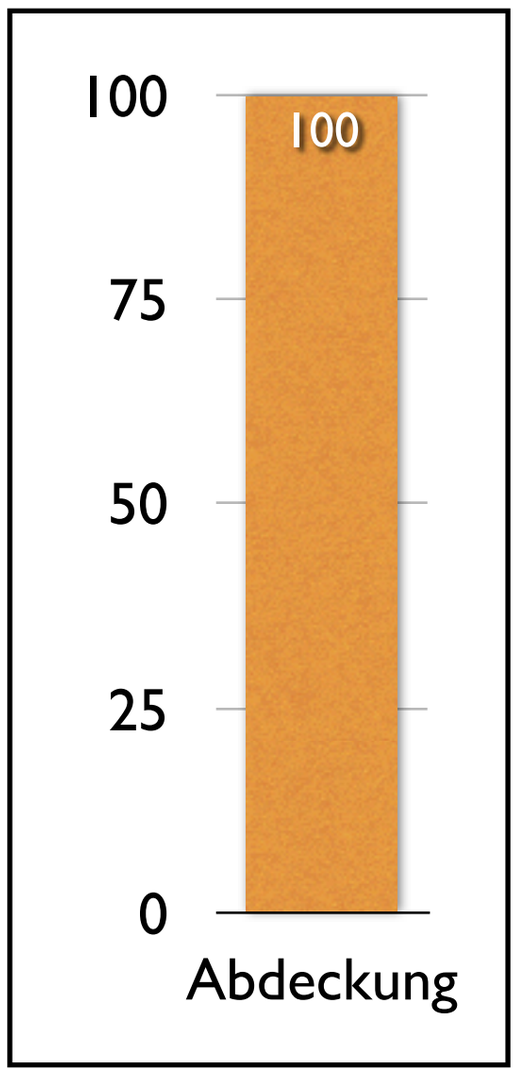
++dptr;
++eptr;
}

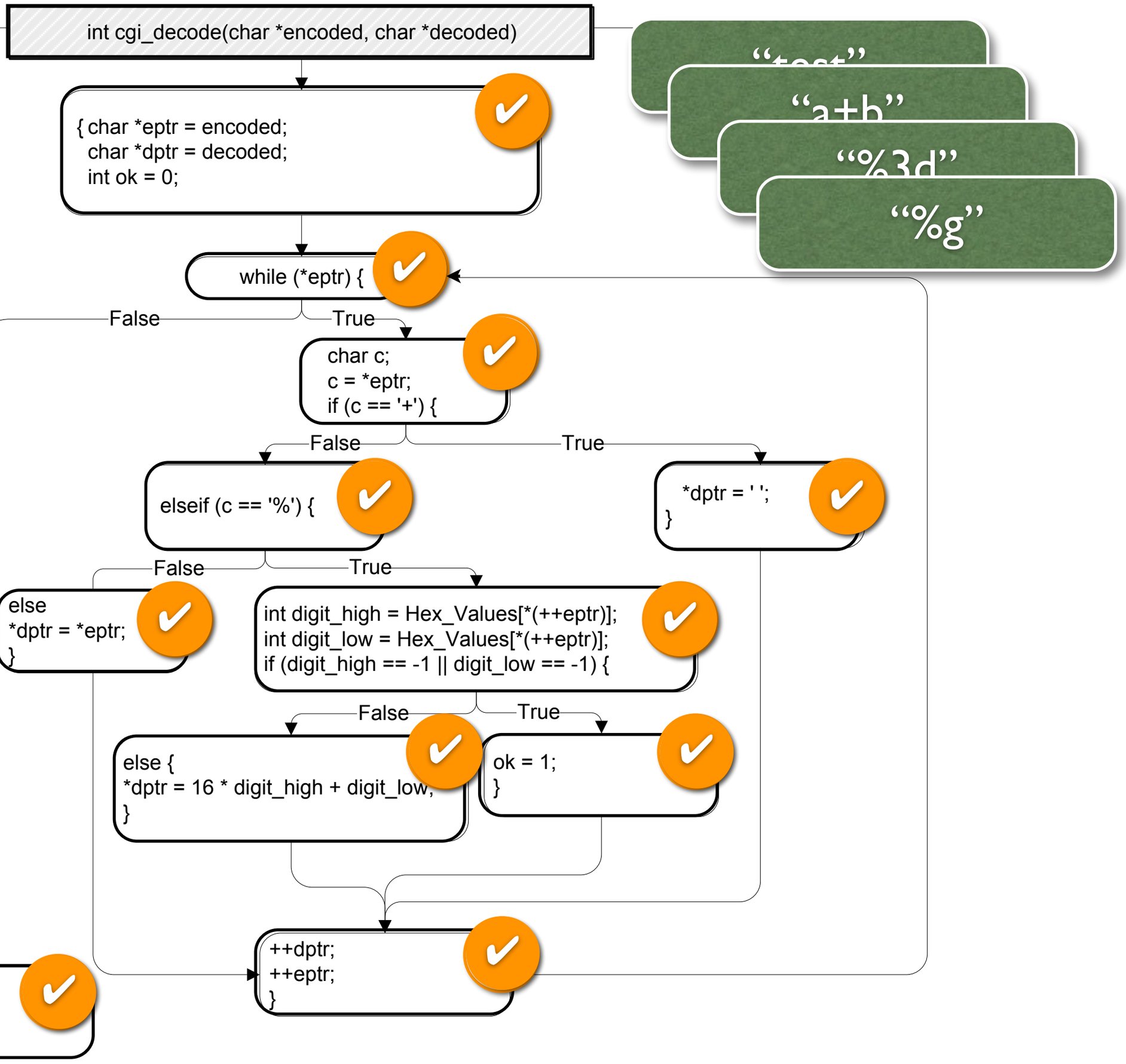
*dptr = '\0';
return ok;
}


100
75
50
25
0
91
Abdeckung


int cgi_decode(char *encoded, char *decoded)
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
while (*eptr) {
False
True
char c;
c = *eptr;
if (c == '+') {
False
True
elseif (c == '%') {
*dptr = ' ';
}
False
True
else
*dptr = *eptr;
}
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
ok = 1;
}
++dptr;
++eptr;
}
*dptr = '\0';
return ok;
}

“test”
“a+b”
“%3d”


100
75
50
25
0
100
Abdeckung


int cgi_decode(char *encoded, char *decoded)
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
while (*eptr) {
False
True
char c;
c = *eptr;
if (c == '+') {
False
True
elseif (c == '%') {
*dptr = ' ';
}
False
True
else
*dptr = *eptr;
}
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
ok = 1;
}
++dptr;
++eptr;
}
*dptr = '\0';
return ok;
}

“test”
“a+b”
“%3d”
“%g”

Abdeckung berechnen

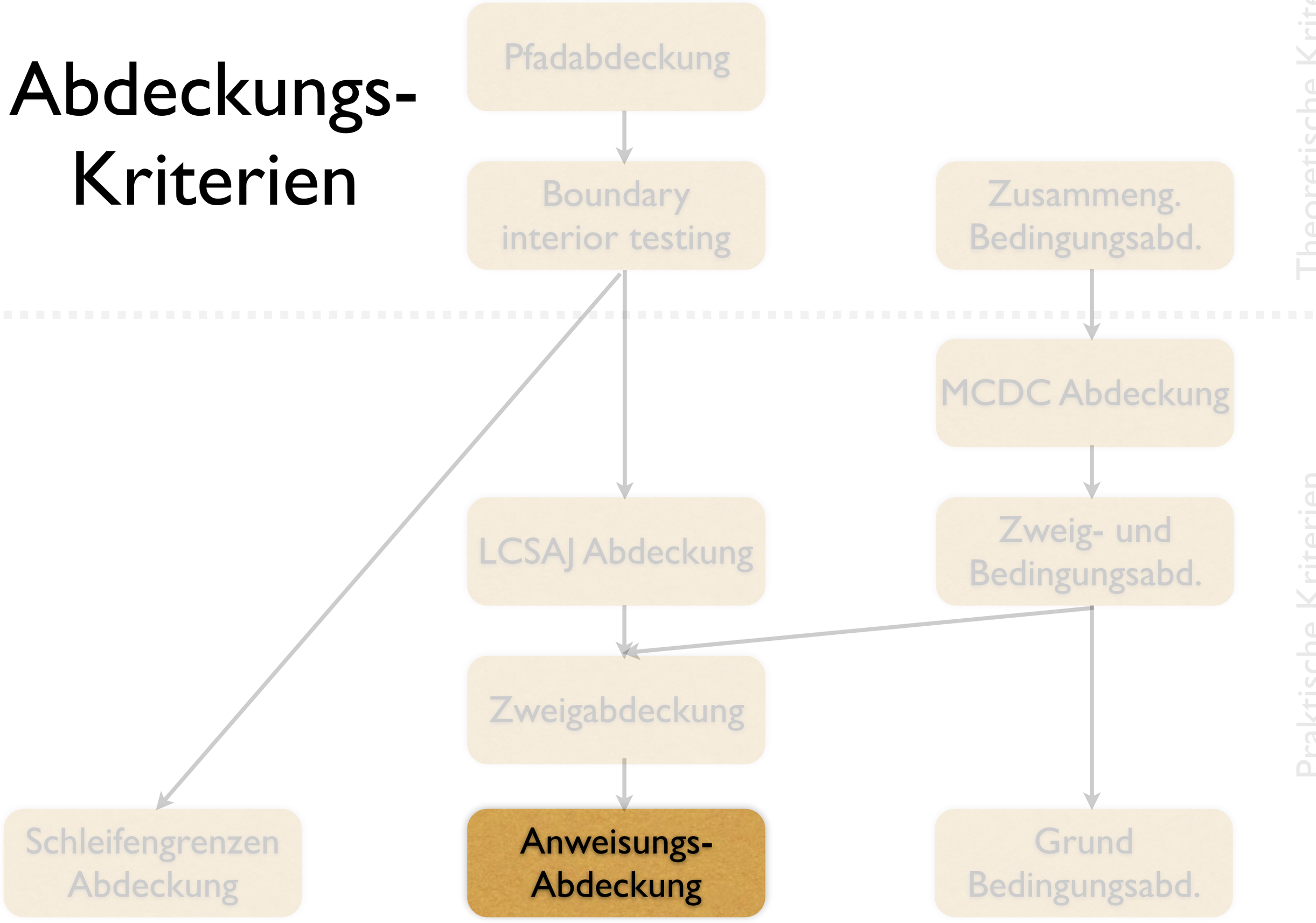
- Die Abdeckung wird automatisch berechnet, während das Programm läuft
- Benötigt *Instrumentierung* zur Übersetzungszeit
  Mit GCC etwa: –ftest-coverage –fprofile-arcs
- Nach Ausführung prüft ein *Abdeckungswerkzeug* die Ergebnisse
  Mit GCC etwa: "gcov *source-file*" erzeugt .gcov-Dateien mit Abdeckung

Pippin: cgi_encode — less — 80×24

```
 4:   18:    int ok = 0;
 -:   19:
38:   20:    while (*eptr)  /* loop to end of string ('\0' character) */
 -:   21:    {
 -:   22:        char c;
30:   23:        c = *eptr;
30:   24:        if (c == '+') {  /* '+' maps to blank */
 1:   25:            *dptr = ' ';
29:   26:        } else if (c == '%') { /* '%xx' is hex for char xx */
 3:   27:            int digit_high = Hex_Values[*(++eptr)];
 3:   28:            int digit_low  = Hex_Values[*(++eptr)];
 5:   29:            if (digit_high == -1 || digit_low == -1)
 2:   30:                ok = 1; /* Bad return code */
 -:   31:            else
 1:   32:                *dptr = 16 * digit_high + digit_low;
 -:   33:        } else { /* All other characters map to themselves */
26:   34:           *dptr = *eptr;
 -:   35:        }
30:   36:        ++dptr; ++eptr;
 -:   37:    }
 4:   38:    *dptr = '\0';   /* Null terminator for string */
 4:   39:    return ok;
 -:   40:}
(END)
```

Demo

Abdeckungs-Kriterien

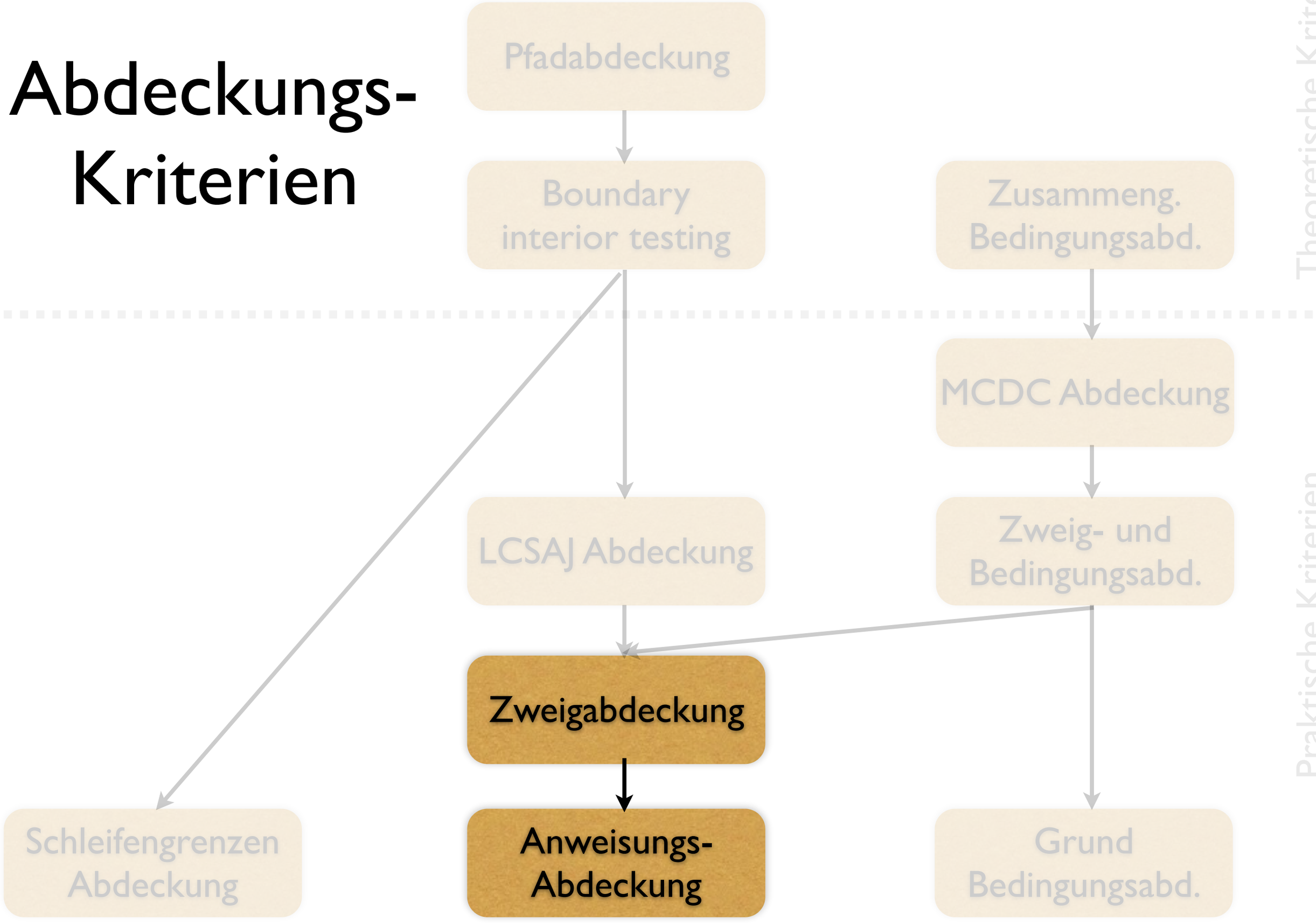


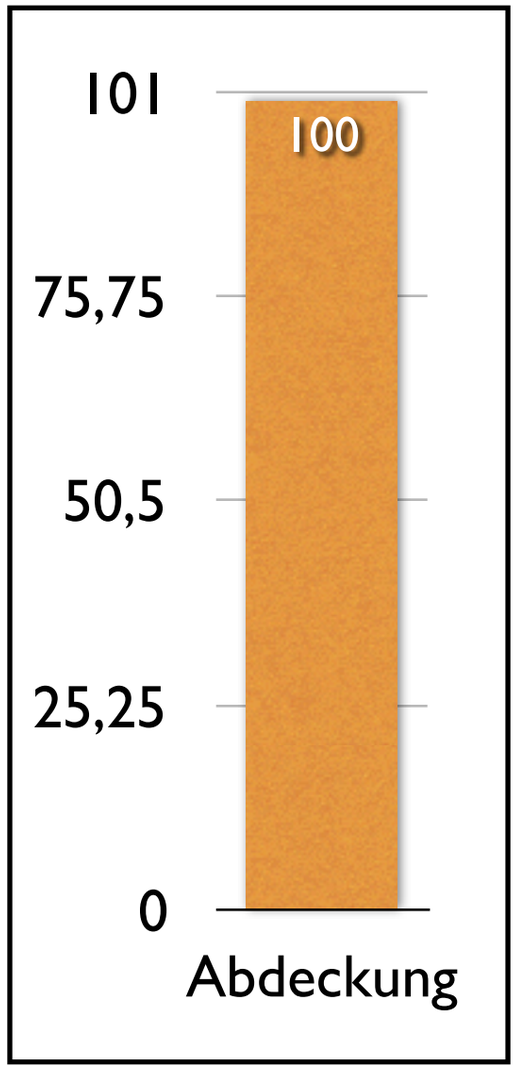
Abdeckungs-Kriterien

umfasst

Theoretische Kriterien

Praktische Kriterien





int cgi_decode(char *encoded, char *decoded)

{ char *eptr = encoded;
  char *dptr = decoded;
  int ok = 0;

while (*eptr) {

False

True

char c;
c = *eptr;
if (c == '+') {

False

True

elseif (c == '%') {

*dptr = ' ';
}

False

True

else
*dptr = *eptr;
}

F

int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {

False

True

else {
*dptr = 16 * digit_high + digit_low;
}

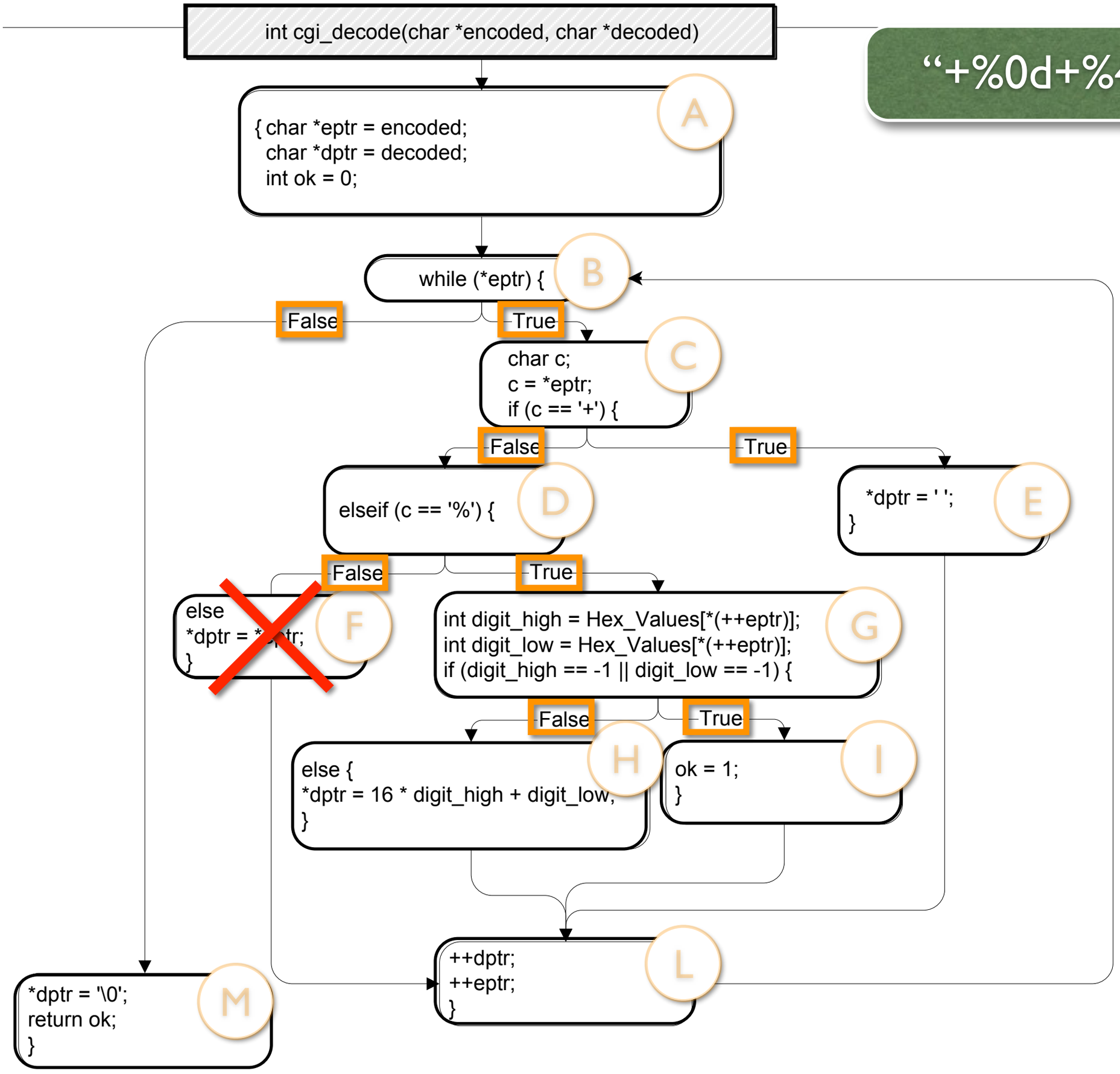
ok = 1;
}

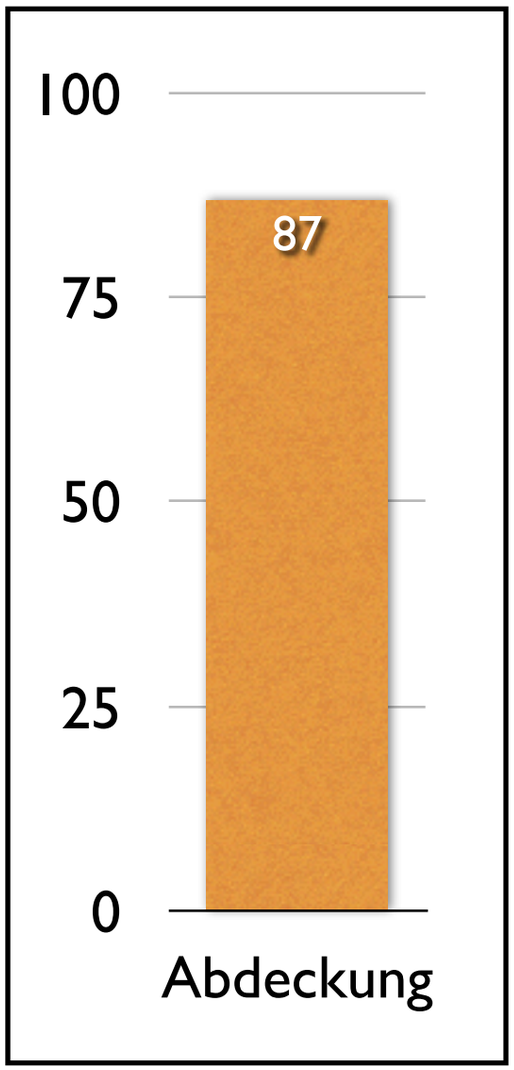
++dptr;
++eptr;
}

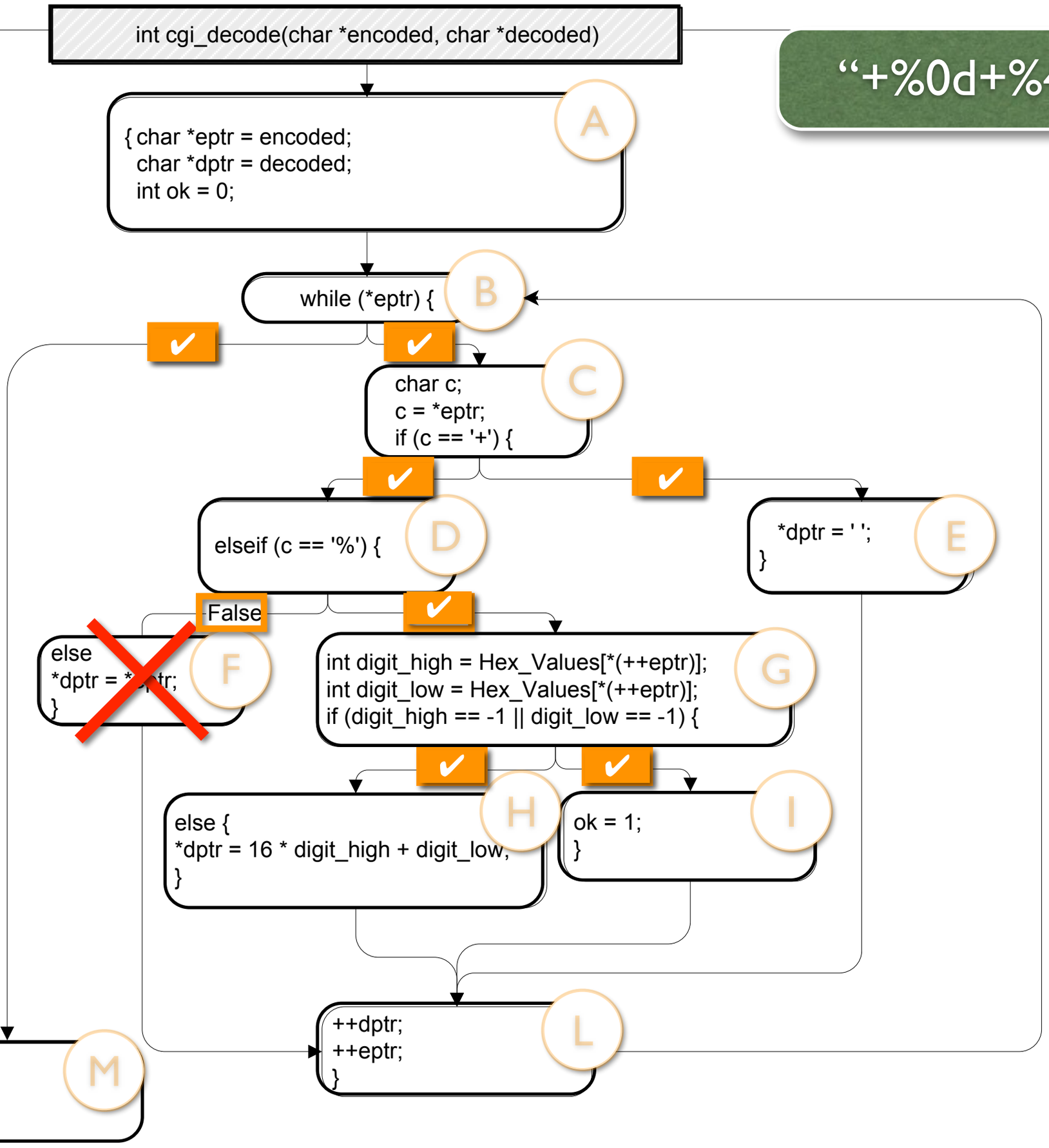
*dptr = '\0';
return ok;
}

“+%0d+%4j”

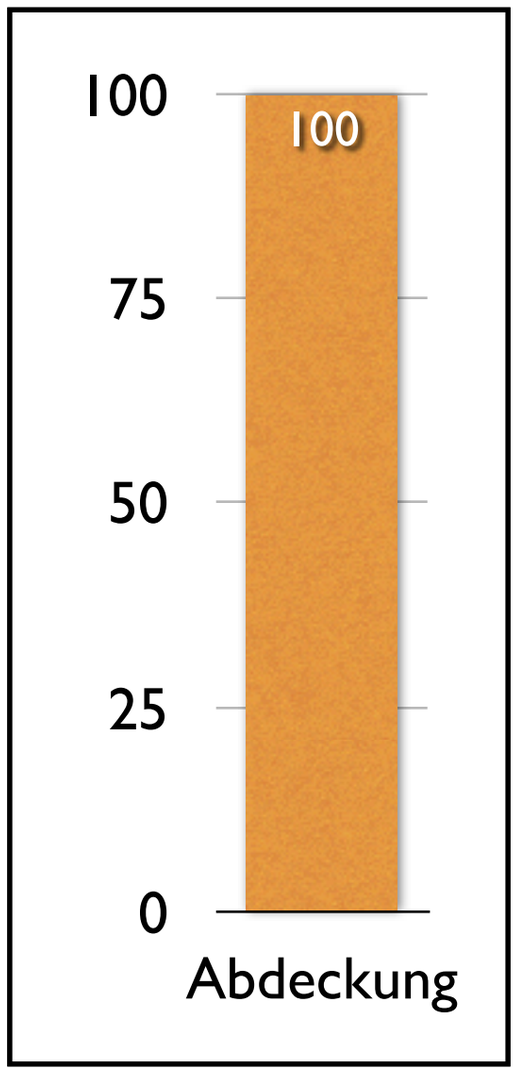
“+%0d+%4j”


int cgi_decode(char *encoded, char *decoded)
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
A
while (*eptr) {
B
False
True
char c;
c = *eptr;
if (c == '+') {
C
False
True
elseif (c == '%') {
D
*dptr = ' ';
}
E
False
True
else
*dptr = *eptr;
}
F
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
G
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
H
ok = 1;
}
I
++dptr;
++eptr;
}
L
*dptr = '\0';
return ok;
}
M


100
75
50
25
0
87
Abdeckung


int cgi_decode(char *encoded, char *decoded)
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
A
while (*eptr) {
B
char c;
c = *eptr;
if (c == '+') {
C
elseif (c == '%') {
D
*dptr = ' ';
}
E
False
else
*dptr = *eptr;
}
F
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
G
else {
*dptr = 16 * digit_high + digit_low;
}
H
ok = 1;
}
I
++dptr;
++eptr;
}
L
*dptr = '\0';
return ok;
}
M

"+%0d+%4j"




```
int cgi_decode(char *encoded, char *decoded)
{ char *eptr = encoded;                                   A
  char *dptr = decoded;
  int ok = 0;
while (*eptr) {                                           B
char c;                                                   C
c = *eptr;
if (c == '+') {
elseif (c == '%') {                                       D
else                                                      F
*dptr = *eptr;
}
int digit_high = Hex_Values[*(++eptr)];                   G
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
else {                                                    H
*dptr = 16 * digit_high + digit_low;
}
ok = 1;                                                   I
}
*dptr = ' ';                                              E
}
++dptr;                                                   L
++eptr;
}
*dptr = '\0';                                             M
return ok;
}
```

“+%0d+%4j”

“abc”

Zweigabdeckung

- Testkriterium: Jeder Zweig im CFG muss mindestens 1x ausgeführt werden
- Abdeckung: $\frac{\#\text{ ausgeführte Verzweigungen}}{\#\text{Verzweigungen}}$
- Umfasst Anweisungsabdeckung
  da bei Erreichen aller Zweige auch alle Knoten erreicht werden
- In Industrie am häufigsten genutzt

Bedingungsabdeckung

- Betrachten Sie den Fehler
  ```
  (digit_high == 1 || digit_low == -1)
   // sollte -1 sein
  ```
- Zweigabdeckung wird erreicht durch Ändern von nur `digit_low`
  Der fehlerhafte Teilausdruck muss nie geprüft werden
- Fehlerhafte Teilbedingung wird nie getestet
  obwohl wir beide Zweige abgedeckt haben

Bedingungsabdeckung

- Idee: in zusammengesetzten Bedingungen auch *Teilbedingungen* prüfen
  = beide Teilbedingungen in `digit_high == 1 || digit_low == -1`

Bedingungsabdeckung

- Testkriterium: jede Grundbedingung muss wenigstens 1x ausgewertet werden
- Abdeckung:

  $$\frac{\#\text{Wahrheitswerte der Grundbedingungen}}{2 * \#\text{ Grundbedingungen}}$$

- Beispiel: "test+%9k%k9"

  100% Abdeckung aller Grundbedingungen

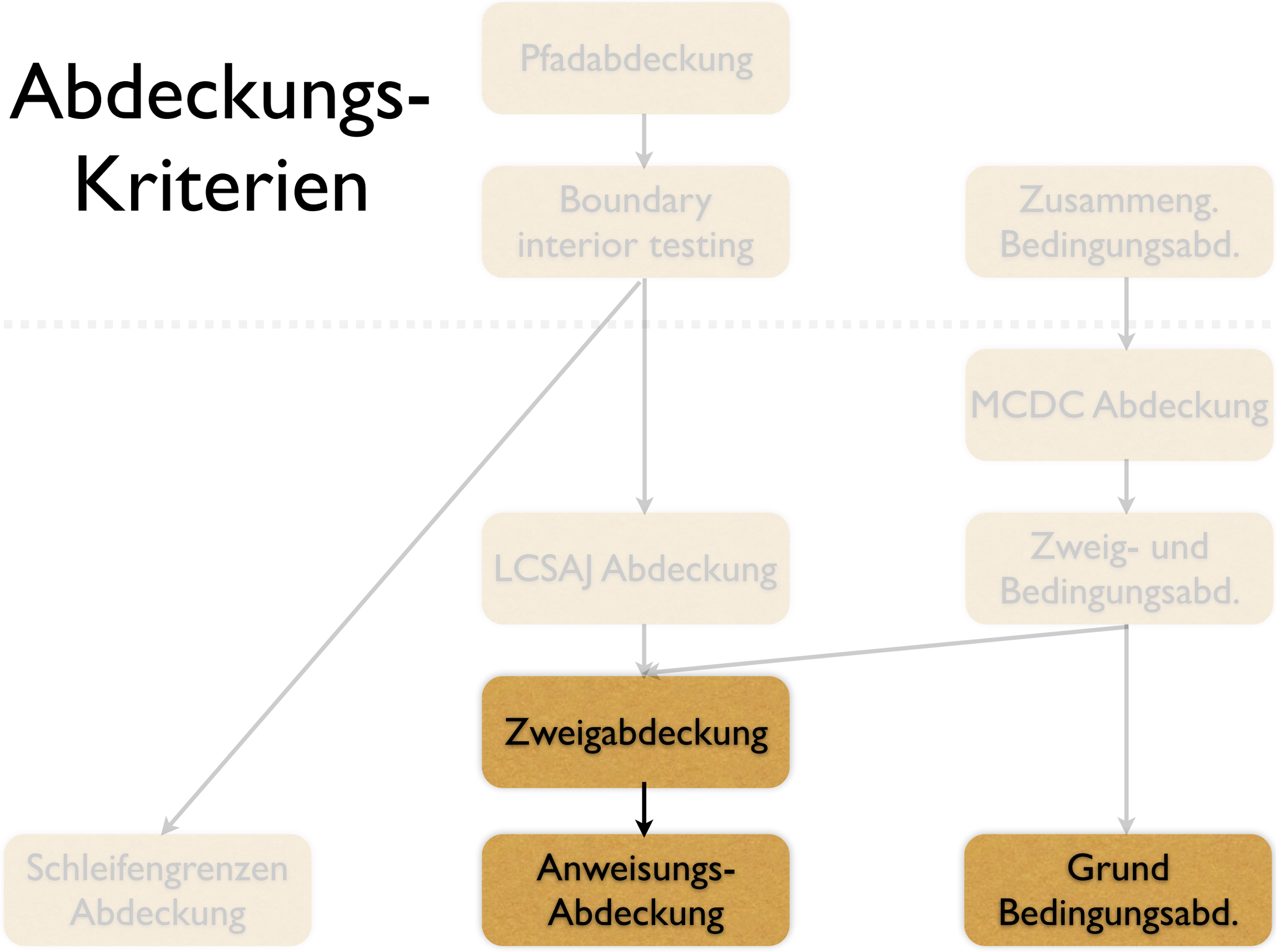
  aber nur 87% Zweigabdeckung

Abdeckungs-Kriterien

umfasst

Theoretische Kriterien

Praktische Kriterien



Abdeckungs-Kriterien

umfasst

Pfadabdeckung

Boundary interior testing

Zusammeng. Bedingungsabd.

MCDC Abdeckung

LCSAJ Abdeckung

Zweig- und Bedingungsabd.

Zweigabdeckung

Schleifengrenzen Abdeckung

Anweisungs-Abdeckung

Grund Bedingungsabd.

Theoretische Kriterien

Praktische Kriterien

Abdeckungs-Kriterien

umfasst

Pfadabdeckung

Boundary interior testing

Zusammeng. Bedingungsabd.

MCDC Abdeckung

LCSAJ Abdeckung

Zweig- und Bedingungsabd.

Zweigabdeckung

Schleifengrenzen Abdeckung

Anweisungs-Abdeckung

Grund Bedingungsabd.

Theoretische Kriterien

Praktische Kriterien

Zusammengesetzte Bedingungen

- Betrachte (((a ∨ b) ∧ c) ∨ d) ∧ e)
- Wir brauchen 13 Tests, um alle Kombinationen abzudecken
- Im allgemeinen Fall: *kombinatorische Explosion*

Test Case	a	b	c	d	e
(1)	**True**	–	**True**	–	**True**
(2)	**False**	**True**	**True**	–	**True**
(3)	**True**	–	**False**	**True**	**True**
(4)	**False**	**True**	**False**	**True**	**True**
(5)	**False**	**False**	–	**True**	**True**
(6)	**True**	–	**True**	–	**False**
(7)	**False**	**True**	**True**	–	**False**
(8)	**True**	–	**False**	**True**	**False**
(9)	**False**	**True**	**False**	**True**	**False**
(10)	**False**	**False**	–	**True**	**False**
(11)	**True**	–	**False**	**False**	–
(12)	**False**	**True**	**False**	**False**	–
(13)	**False**	**False**	–	**False**	–

Abdeckungs-Kriterien

umfasst

Theoretische Kriterien

Praktische Kriterien

Pfadabdeckung

Boundary interior testing

LCSAJ Abdeckung

Schleifengrenzen Abdeckung

Zusammeng. Bedingungsabd.

MCDC Abdeckung

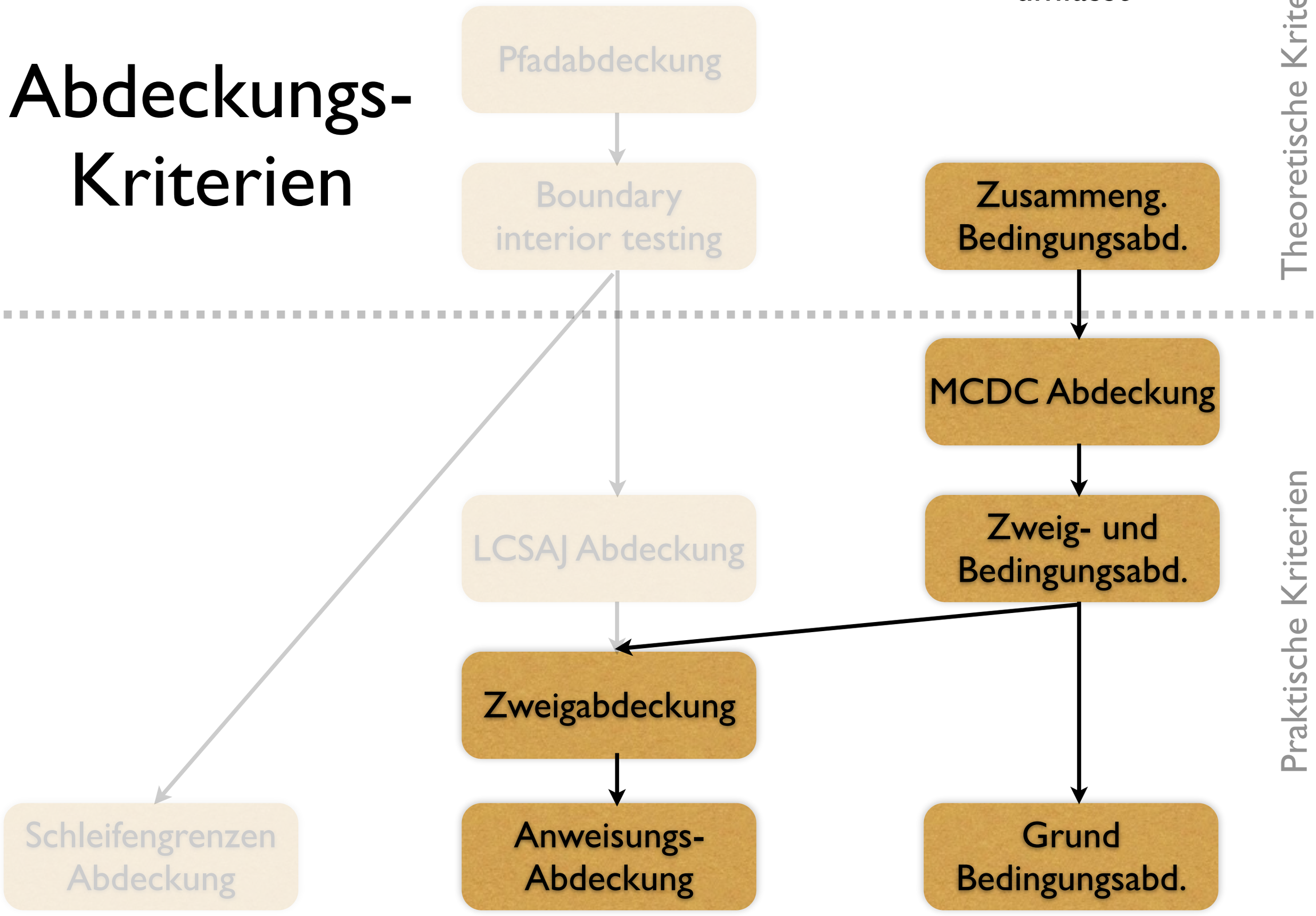
Zweig- und Bedingungsabd.

Zweigabdeckung

Anweisungs-Abdeckung

Grund Bedingungsabd.

Abdeckungs-Kriterien



MCDC Abdeckung

Modified Condition Decision Coverage

- Idee: Nur *wichtige Kombinationen* testen, um exponentielles Wachstum zu vermeiden
- Eine Kombination ist "wichtig", wenn wir für jede Grundbedingung zeigen, wie sie unabhängig die Entscheidung beeinflusst

MCDC Abdeckung

Modified Condition Decision Coverage

- Für jede Grundbedingung *C* brauchen wir zwei Testfälle T_1 and T_2
- Die Werte aller *ausgewerteten* Grundbedingungen außer *C* bleiben gleich
- Die zusammengesetzte Bedingung als Ganzes wird *True* bei T_1 und *False* bei T_2
- Guter Kompromiss zwischen Strenge und Testgröße (und daher häufig verwendet)

MCDC Abdeckung

Modified Condition Decision Abdeckung

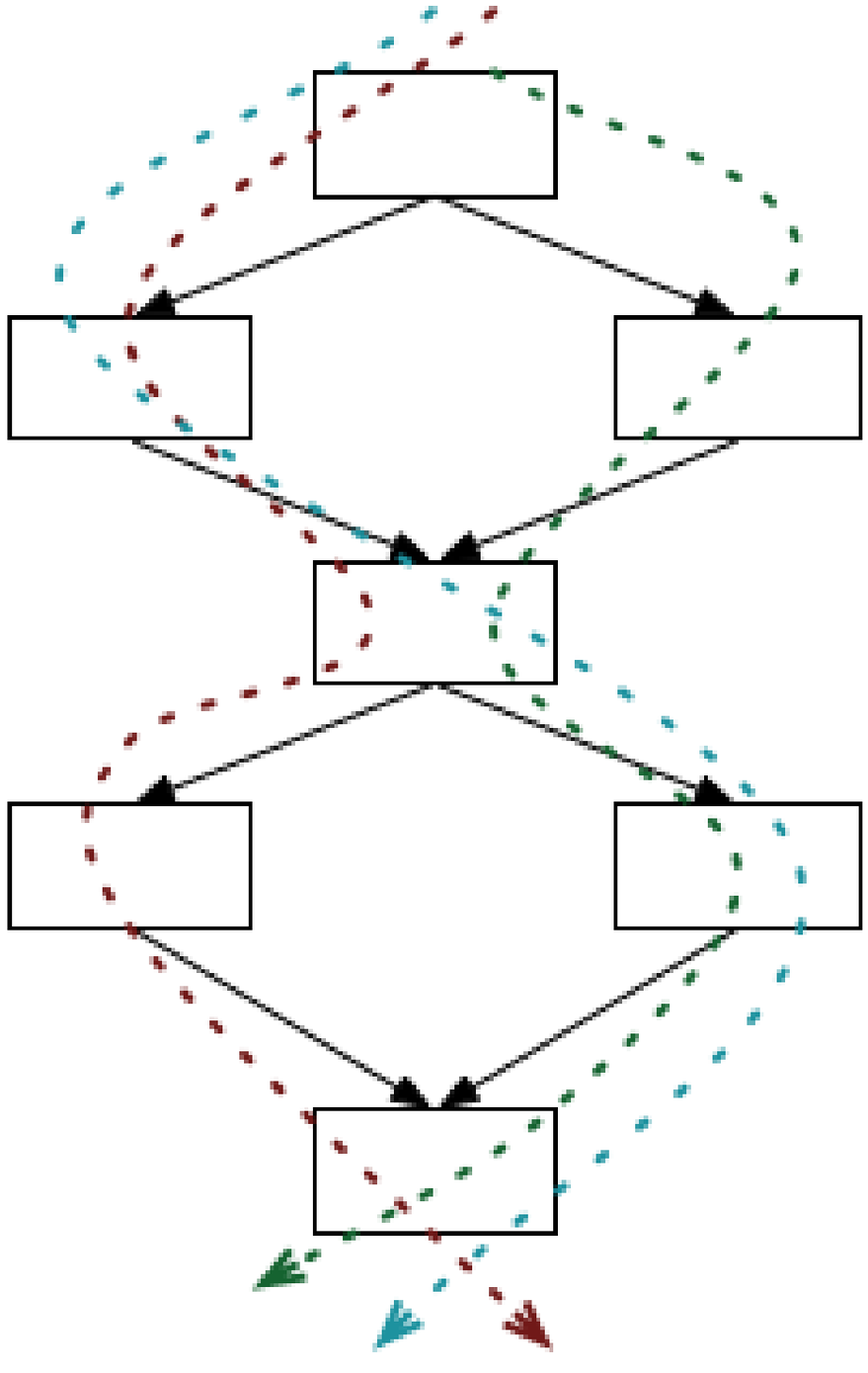
- Betrachte (((a ∨ b) ∧ c) ∨ d) ∧ e)
- Sechs Tests genügen zur MCDC-Abdeckung

	a	b	c	d	e	Decision
(1)	**True**	–	**True**	–	**True**	**True**
(2)	**False**	**True**	**True**	–	**True**	**True**
(3)	**True**	–	**False**	**True**	**True**	**True**
(6)	**True**	–	**True**	–	**False**	**False**
(11)	**True**	–	**False**	**False**	–	**False**
(13)	**False**	**False**	–	**False**	–	**False**

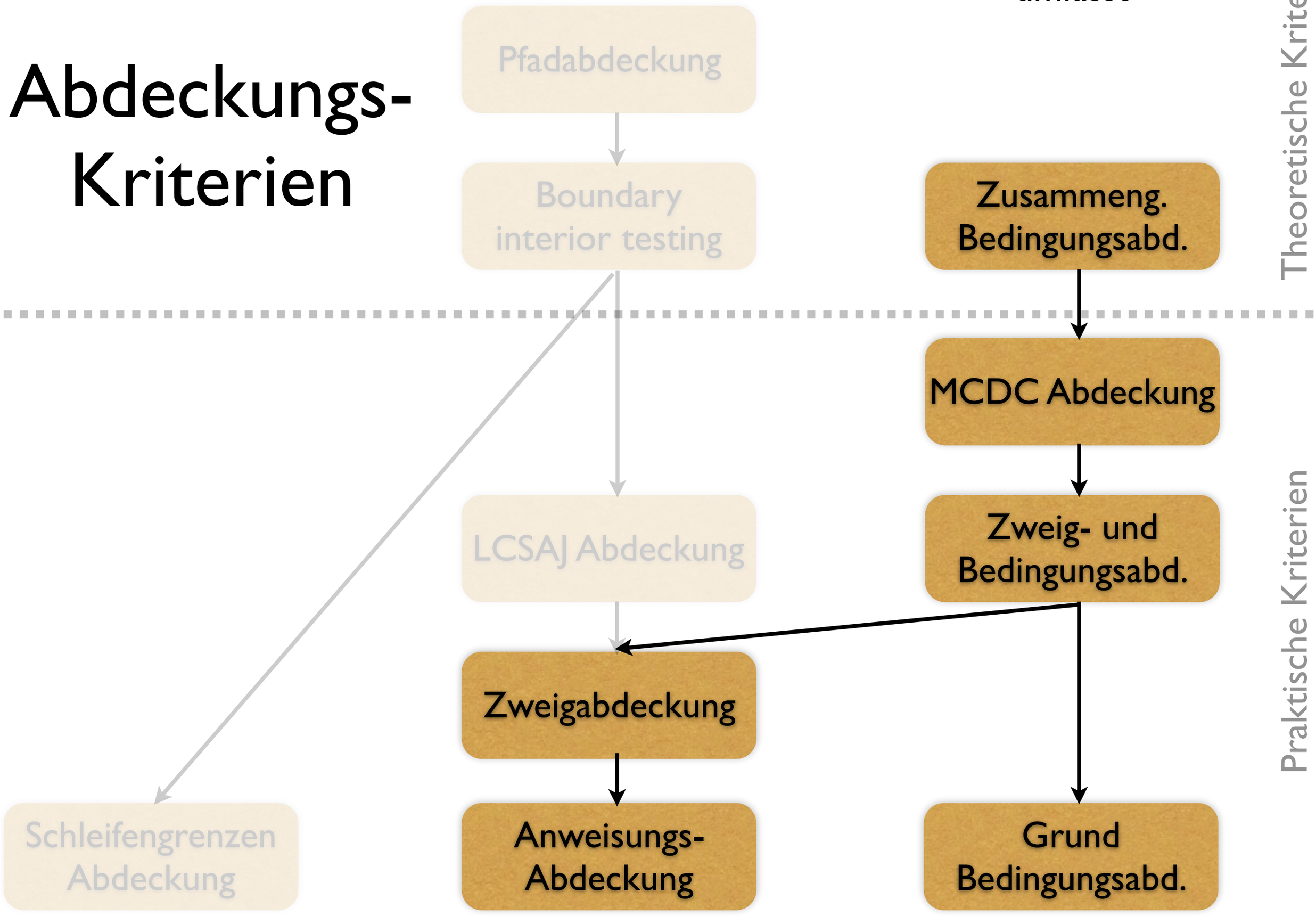
Pfadabdeckung

über einzelne Verzweigungen hinaus

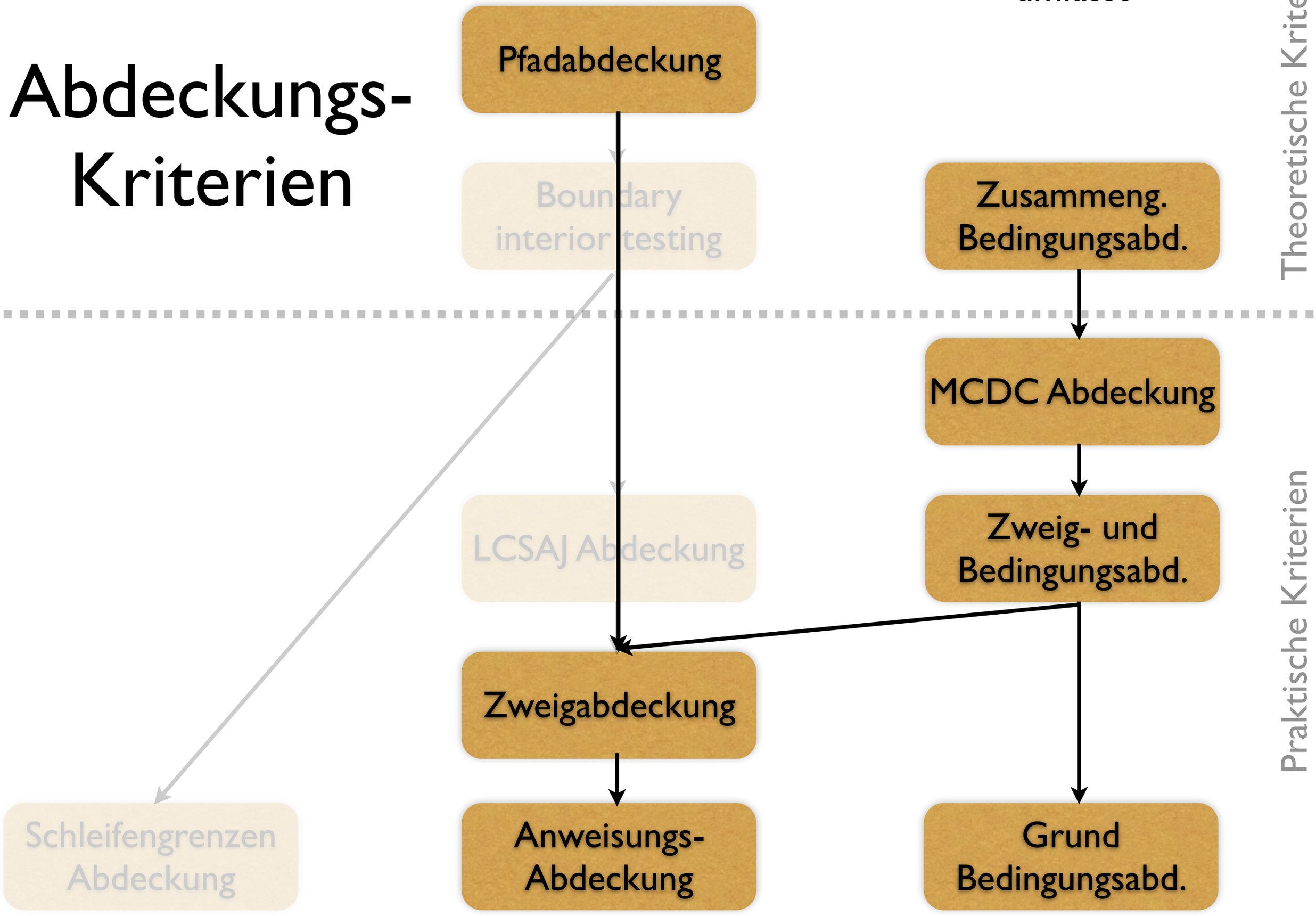
- Grundidee: *Folgen von Verzweigungen* im Kontrollfluss abdecken
- Weit mehr Pfade als Verzweigungen
  impliziert Kompromisse



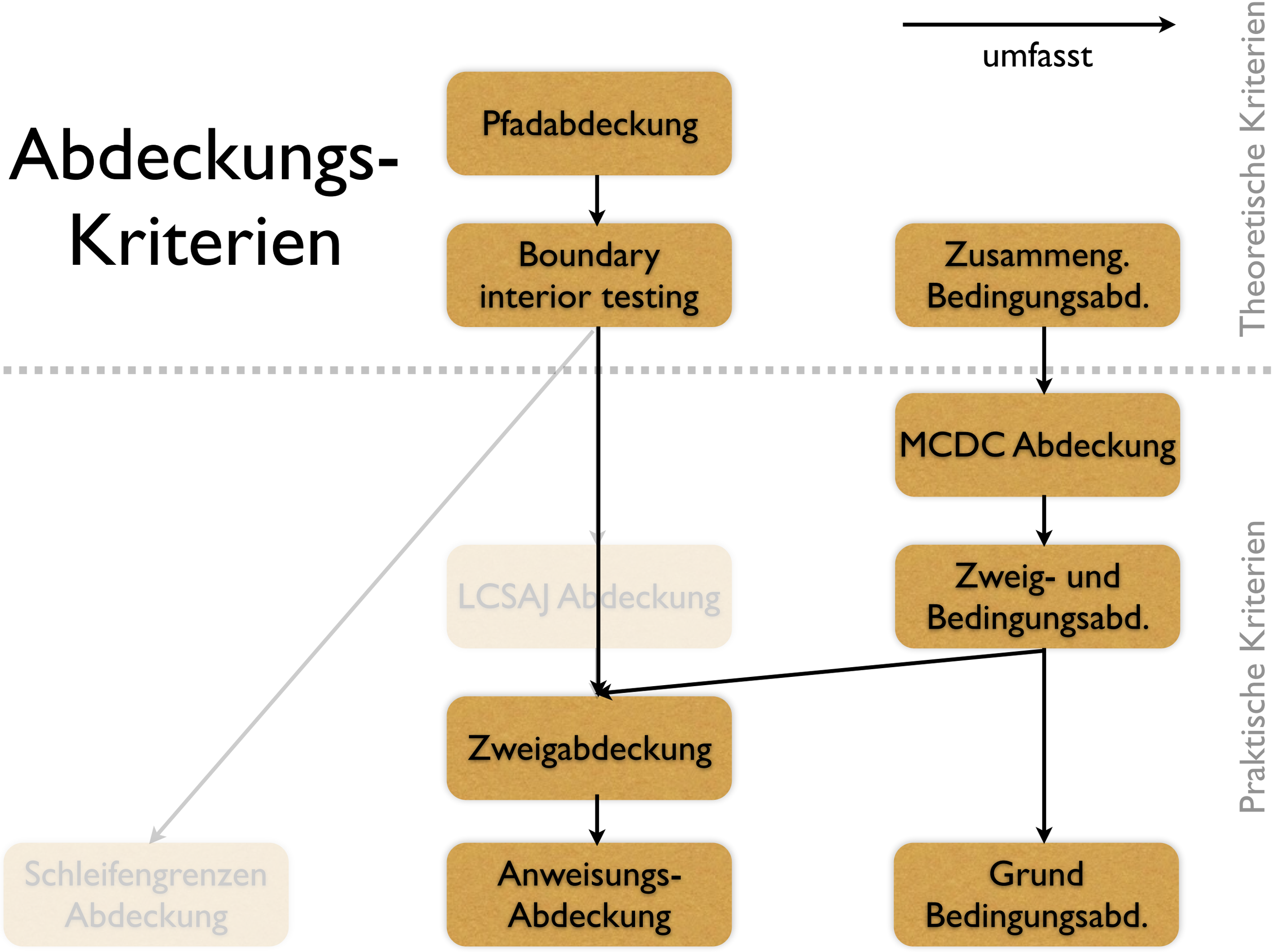
Abdeckungs-Kriterien



Abdeckungs-Kriterien



Abdeckungs-Kriterien



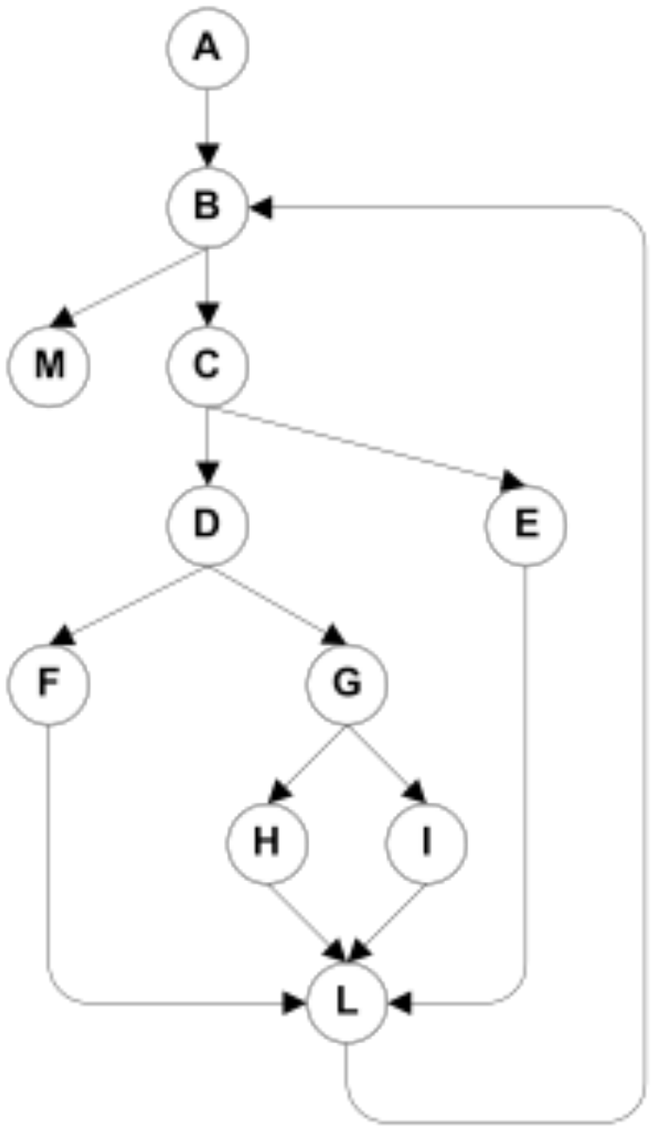
Boundary Interior Abdeckung

für cgi_decode

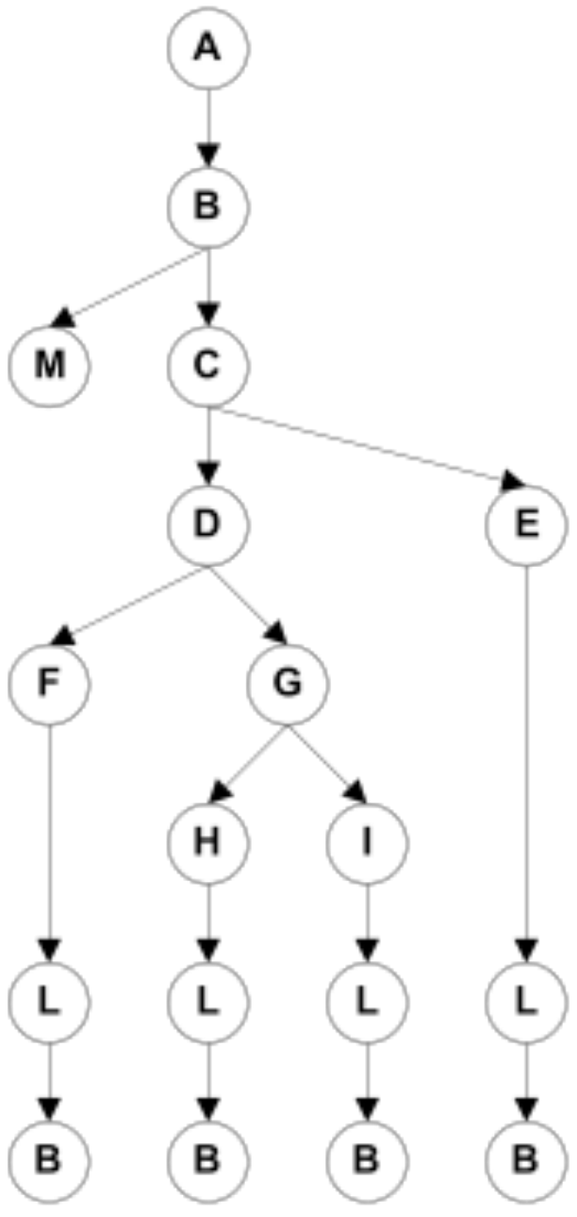
- Boundary Interior betrachtet alle Pfade als gleich, die sich nur im wiederholten Schleifendurchlauf unterscheiden
- In anderen Worten: Wir folgen jedem Pfad im CFG, bis wir einen Knoten zum zweiten Mal sehen

Boundary Interior Abdeckung

für cgi_decode

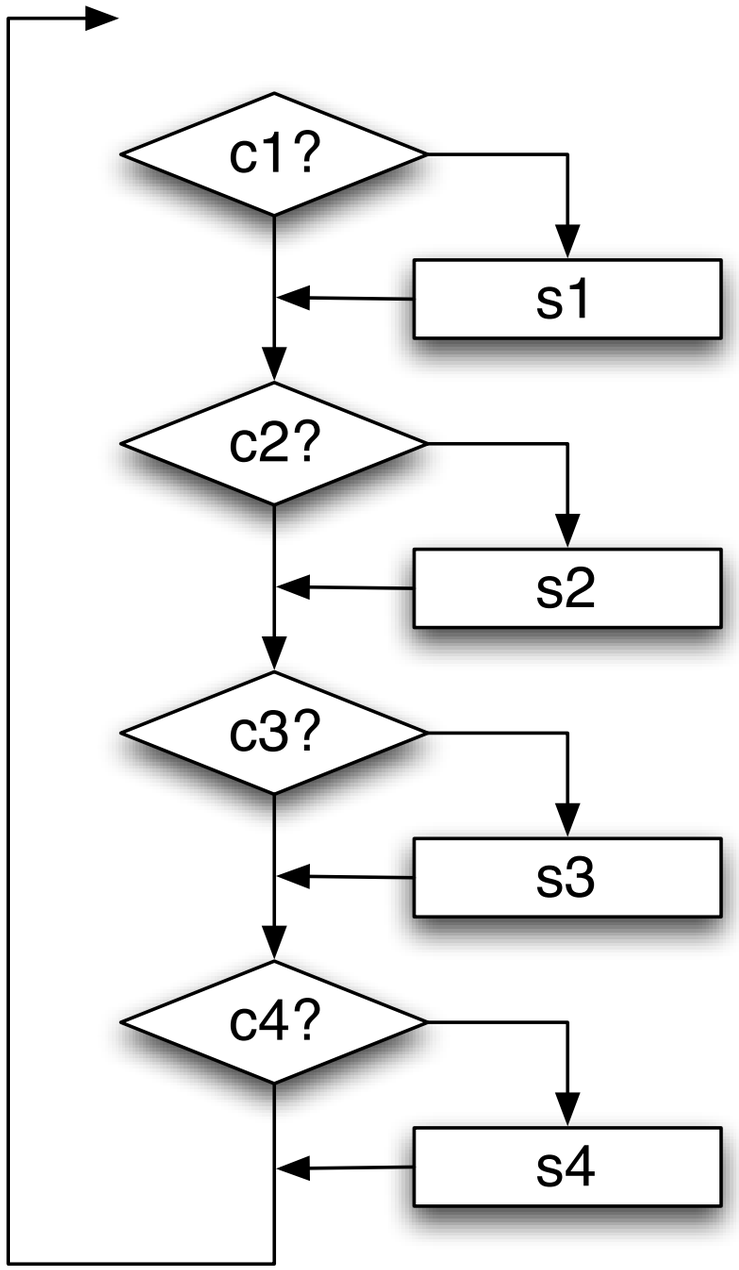


Original-CFG



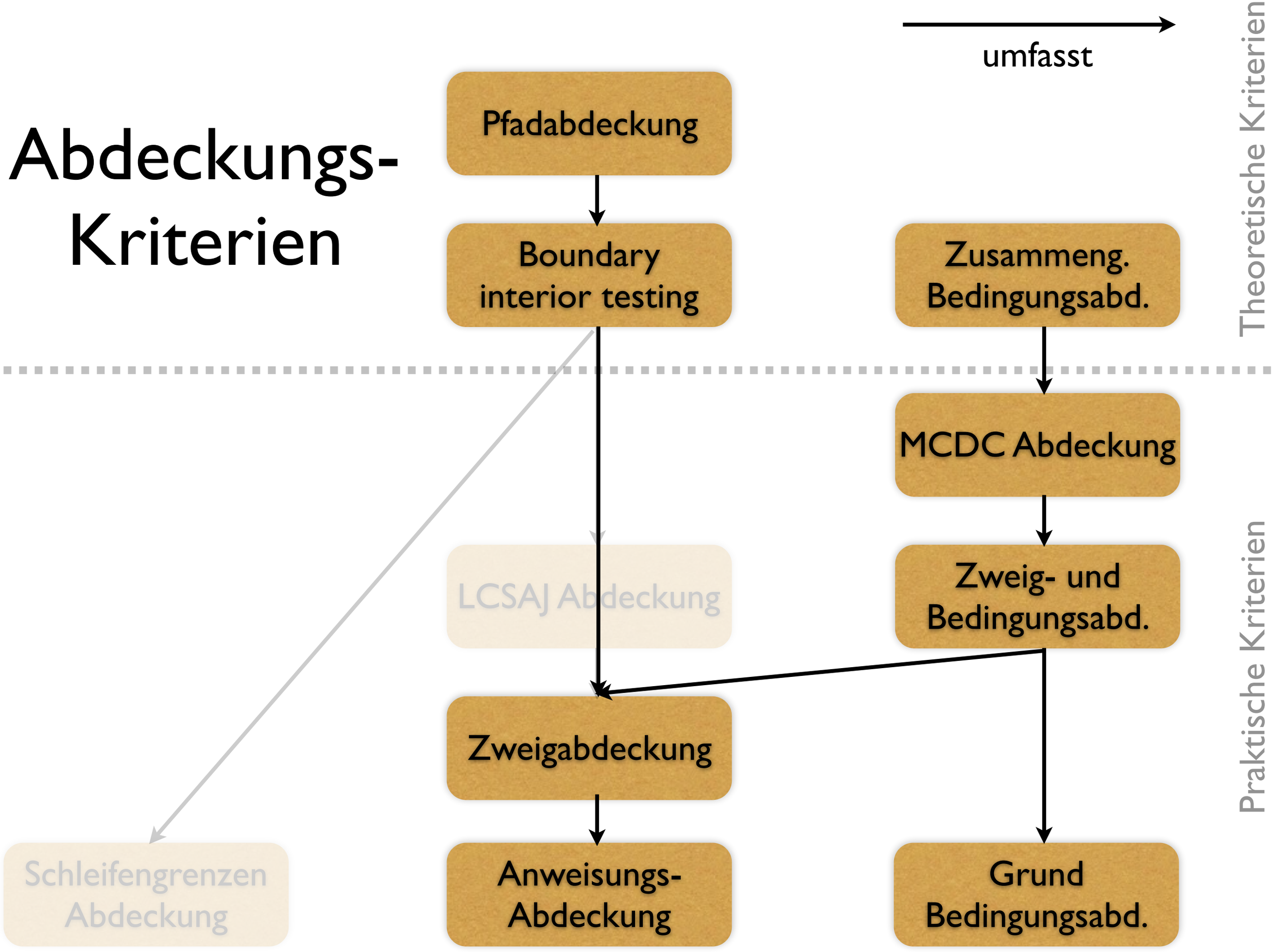
Abzudeckende Pfade

Probleme



- Die Zahl der Pfade kann immer noch exponentiell wachsen
  Im Beispiel links etwa $2^4 = 16$ Pfade
- Pfade können *unerreichbar* sein
  oder sogar *unmöglich*, wenn Bedingungen nicht unabhängig sind

Abdeckungs-Kriterien

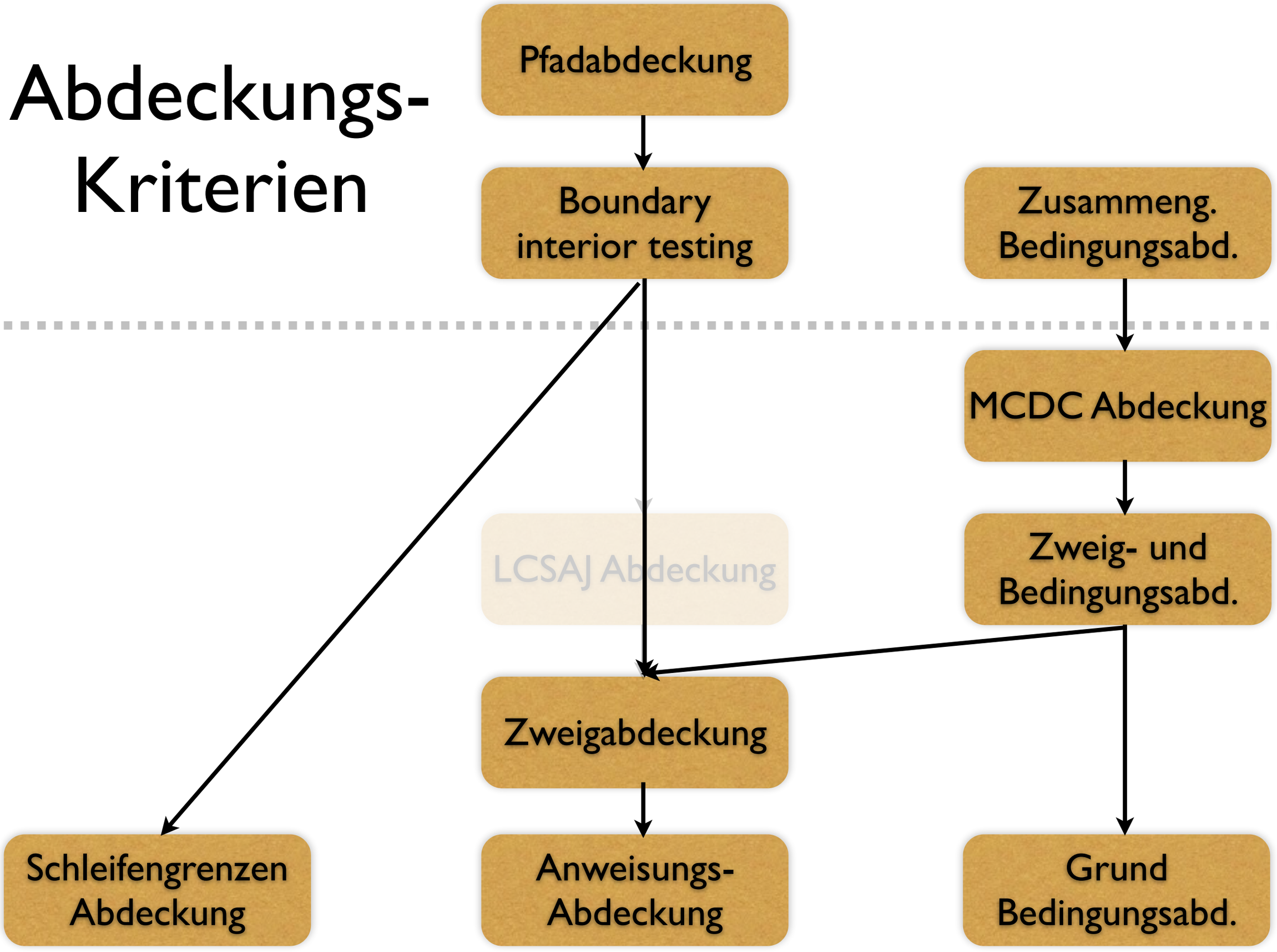


Abdeckungs-Kriterien

umfasst

Theoretische Kriterien

Praktische Kriterien



Schleifengrenzen-Abdeckung

Eine Testsuite erfüllt die Schleifengrenzen-Abdeckung, wenn für jede Schleife *S* gilt

- *Es gibt einen Testfall, der S 0x ausführt*
- *Es gibt einen Testfall, der S 1x ausführt*
- *Es gibt einen Testfall, der S* $\geq$ *2x ausführt*

Wird gewöhnlich mit anderen Kriterien wie MCDC kombiniert

int cgi_decode(char *encoded, char *decoded)

A
```
{ char *eptr = encoded;
  char *dptr = decoded;
  int ok = 0;
```

"" "a" "abc"

B
```
while (*eptr) {
```

0 1 ≥ 2

False

True

C
```
char c;
c = *eptr;
if (c == '+') {
```

False

True

D
```
elseif (c == '%') {
```

E
```
  *dptr = ' ';
}
```

False

True

F
```
else
*dptr = *eptr;
}
```

G
```
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
```

False

True

H
```
else {
*dptr = 16 * digit_high + digit_low;
}
```

I
```
ok = 1;
}
```

L
```
++dptr;
++eptr;
}
```

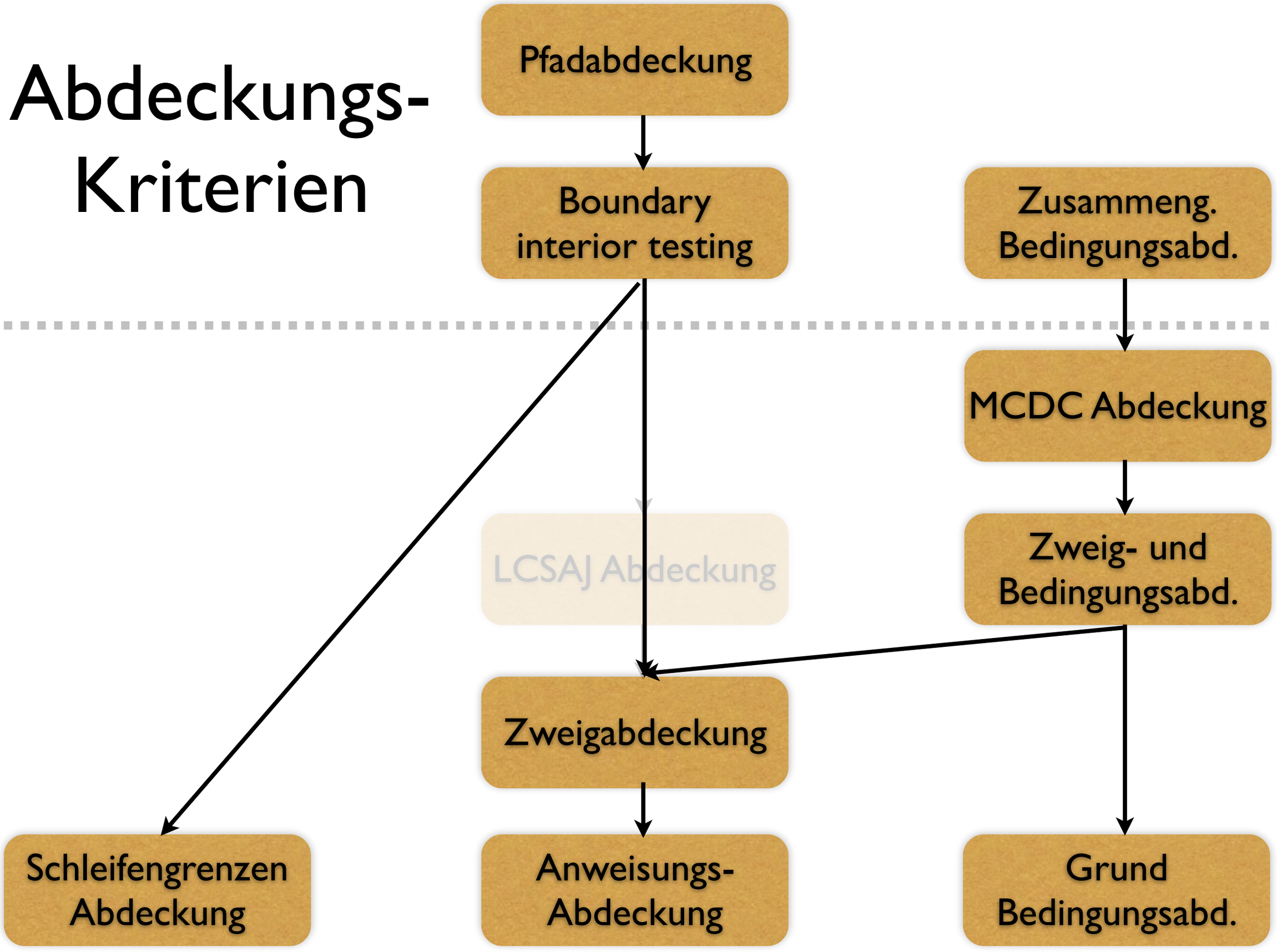
M
```
*dptr = '\0';
return ok;
}
```

Abdeckungs-Kriterien

umfasst

Theoretische Kriterien

Praktische Kriterien

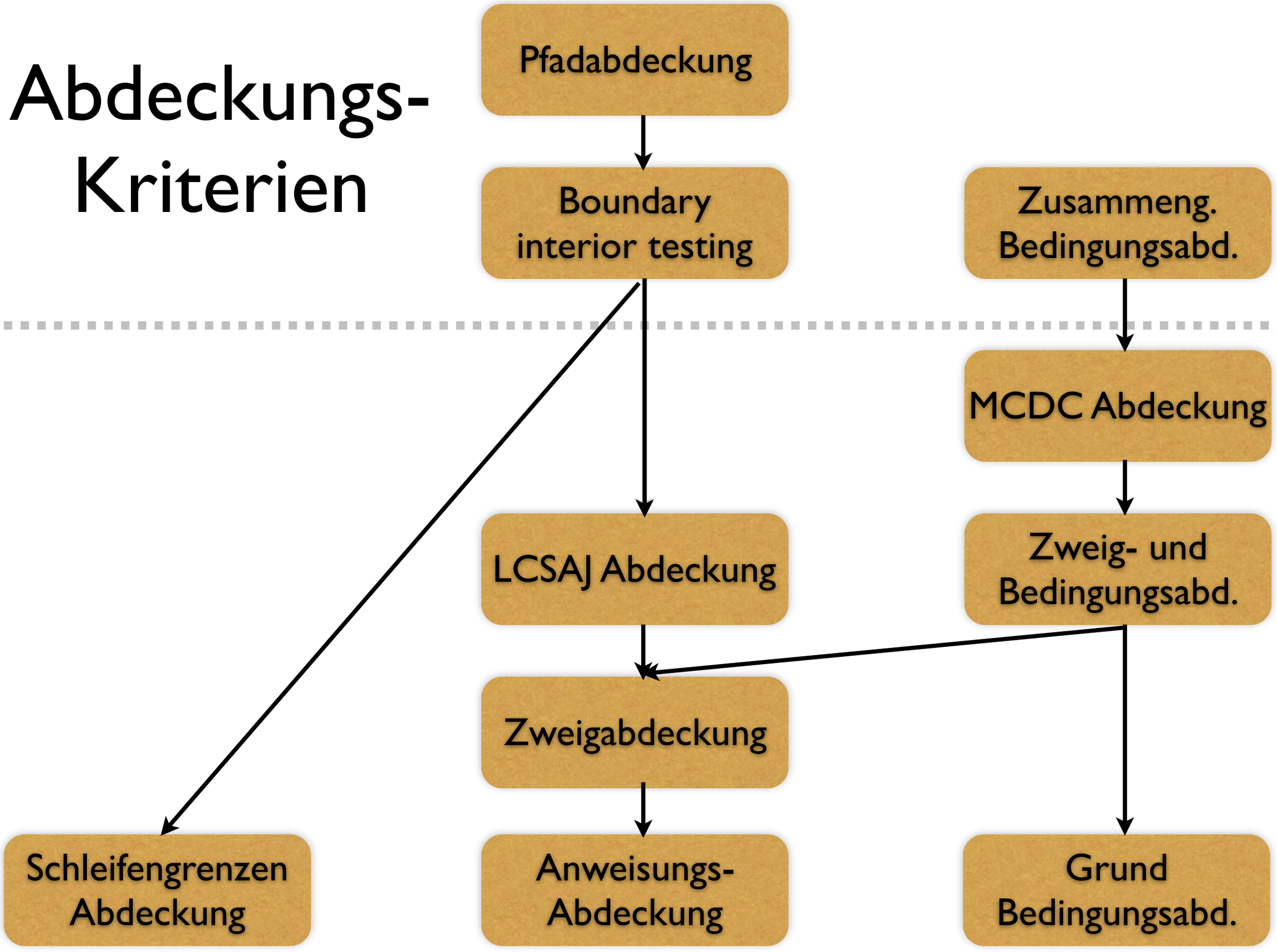


Abdeckungs-Kriterien

umfasst

Theoretische Kriterien

Praktische Kriterien



LCSAJ-Abdeckung

Alle Pfade bis zu bestimmter Länge testen

- LCSAJ = Linear Code Sequence And Jump
- Ein LCSAJ ist ein Unterpfad im CFG, der in einem Zweig beginnt und endet

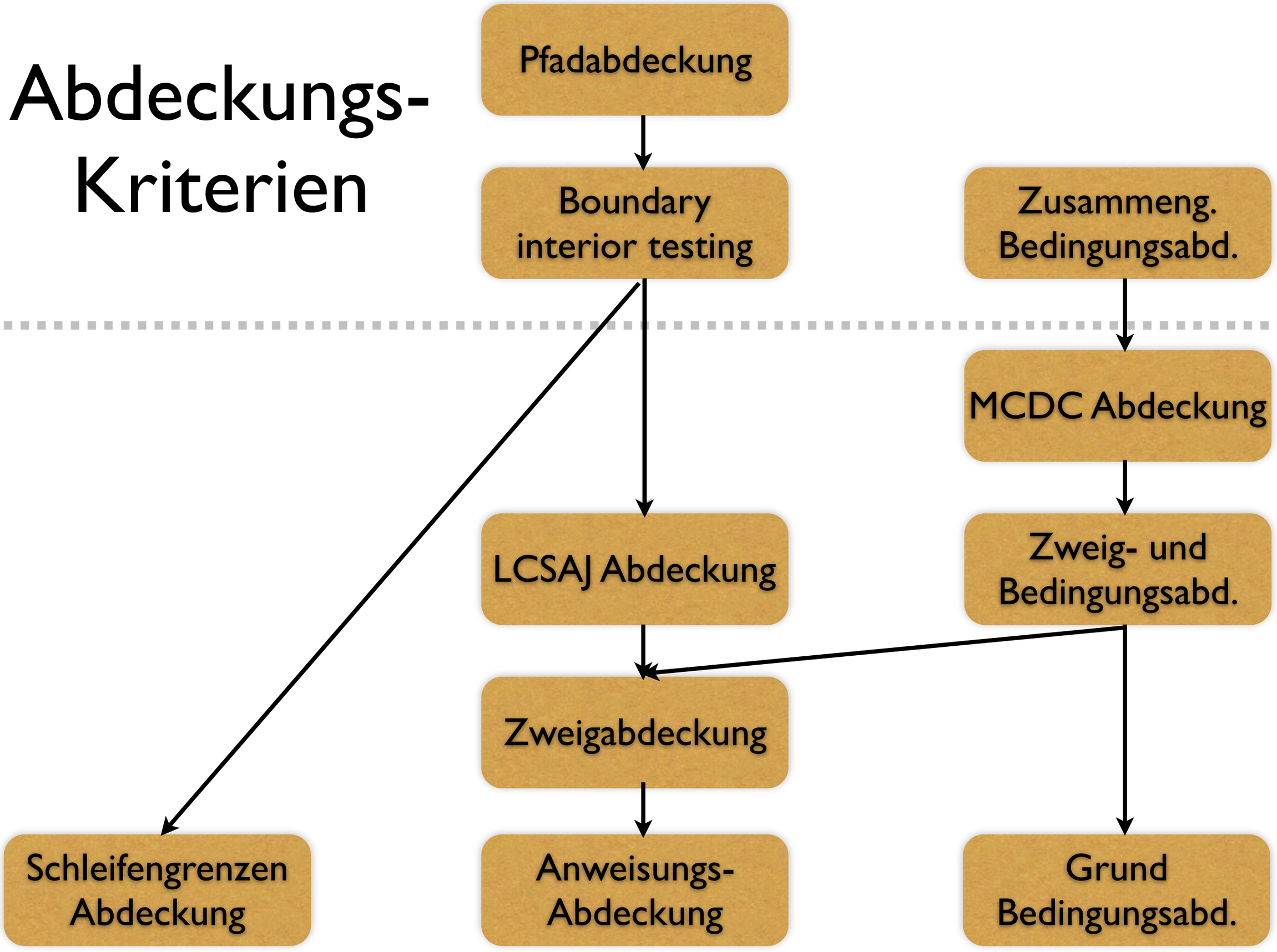
LCSAJ Länge	entspricht
1	Anweisungs-Abdeckung
2	Zweig-Abdeckung
n	Abdeckung von *n* aufeinanderfolgenden LCSAJs
∞	Pfad-Abdeckung

Abdeckungs-Kriterien

umfasst

Theoretische Kriterien

Praktische Kriterien



Weyukers Hypothese

Die Angemessenheit eines
Abdeckungs-Kriteriums
kann nur intuitiv definiert werden.

Kriterien erfüllen

Manchmal sind Kriterien nicht erfüllbar:

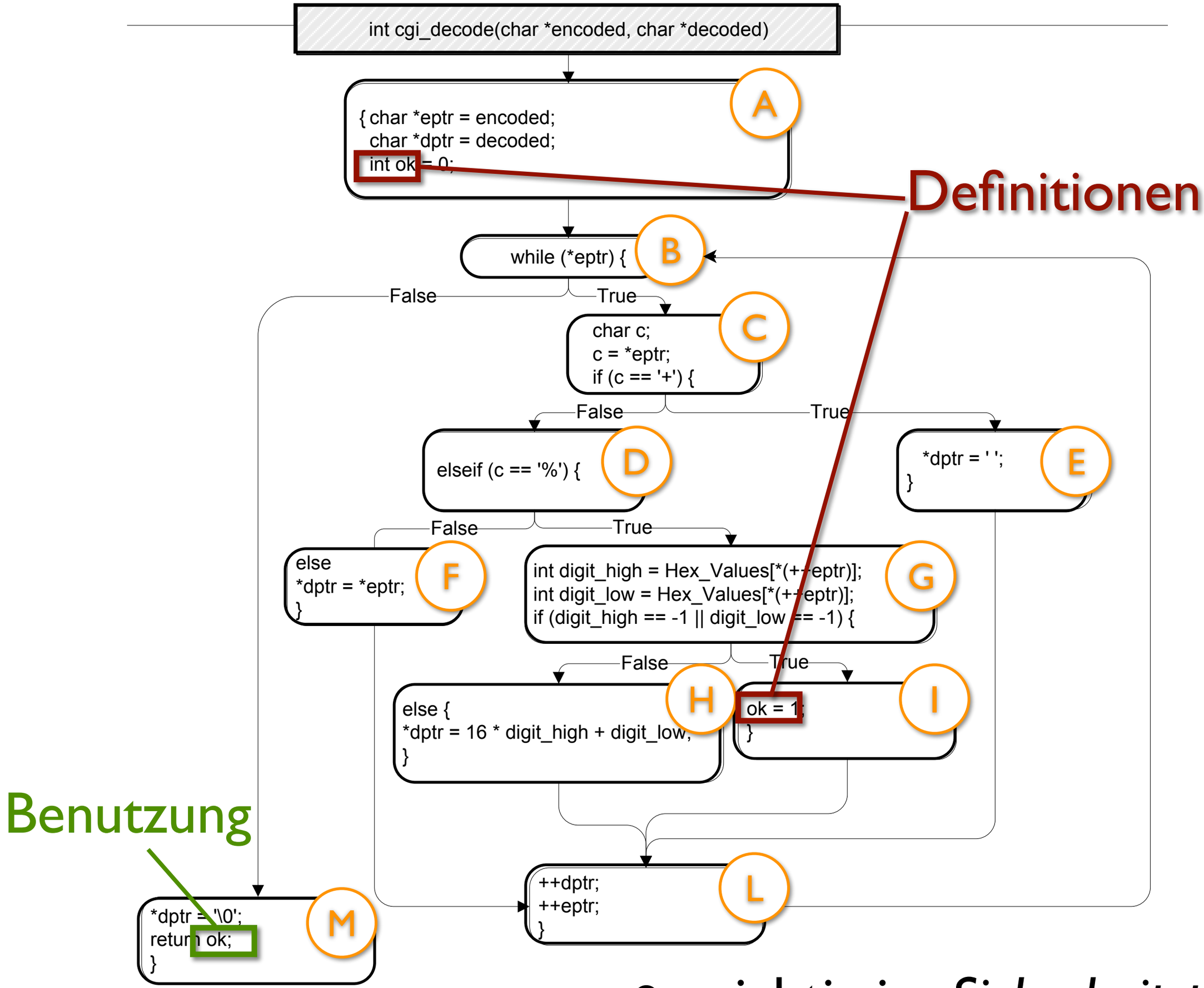
- *Anweisungen* werden u.U. nicht ausgeführt wegen *defensiver Programmierung* oder *Code-Wiederverwendung*
- *Bedingungen* sind u.U. nicht erfüllbar wegen *voneinander abhängigen Bedingungen*
- *Pfade* sind u.U. nicht erfüllbar wegen *voneinander abhängiger Zweige*

Kriterien erfüllen

- Eine bestimmte Stelle im Code zu erreichen kann sehr schwer sein!
- Selbst die besten Programme enthalten unerreichbaren Code
- Große Mengen an unerreichbarem Code sind ein schweres *Wartungsproblem*
- In der Praxis: Erlaube Abdeckung < 100%

Mehr Kriterien

- Objektorientierte Abdeckung
  z.B.“Jeder Übergang im endlichen Automaten des Objekts muss abgedeckt sein” oder “Jedes Methodenpaar in der Folge von Aufrufen muss abgedeckt sein”
- Interclass-Abdeckung
  z.B.“Jede Interaktion zwischen zwei Objekten muss abgedeckt sein”
- Datenfluss-Abdeckung
  z.B.“Jedes Paar aus Definition und Benutzung einer Variablen muss abgedeckt sein”



- wichtig im *Sicherheitstesten*

WHO WATCHES THE WATCHMEN?

Ein schlechter Test

```
void test_haha() {
  try {
    teste_alles();
    decke_alles_ab();
    wirklich_ALLES();
  } catch (Throwable th) {
  }
    assert(true);
}
```

Mutationstesten

DeMillo, Lipton, Sayward 1978

Programm

Eine Mutation

```
class Checker {
    public int compareTo(Object other)
    {
        return 1;
    }
}
```

Wird die Testsuite diesen Fehler finden?

Wenn nicht: *Welche Fehler wird sie noch übersehen?*

Mutationsoperatoren

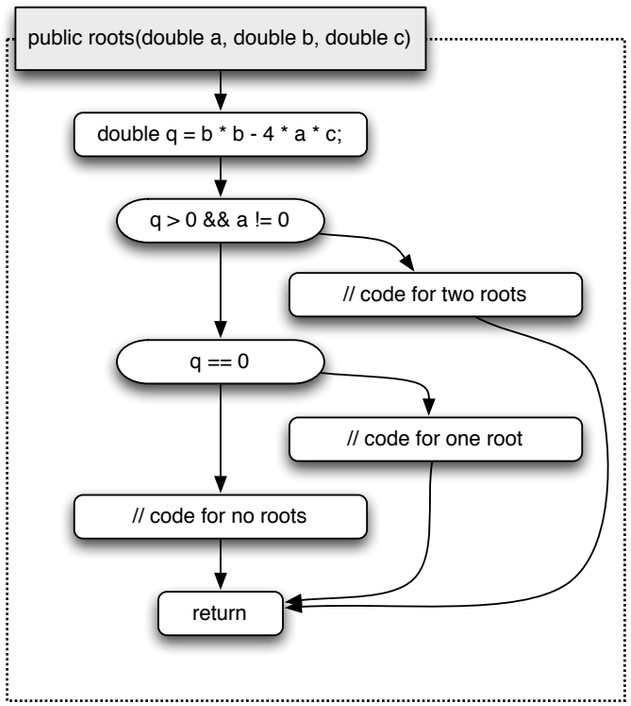
id	operator	description	constraint
Operand Modifications			
crp	constant for constant replacement	replace constant $C1$ with constant $C2$	$C1 \neq C2$
scr	scalar for constant replacement	replace constant C with scalar variable X	$C \neq X$
acr	array for constant replacement	replace constant C with array reference $A[I]$	$C \neq A[I]$
scr	struct for constant replacement	replace constant C with struct field S	$C \neq S$
svr	scalar variable replacement	replace scalar variable X with a scalar variable Y	$X \neq Y$
csr	constant for scalar variable replacement	replace scalar variable X with a constant C	$X \neq C$
asr	array for scalar variable replacement	replace scalar variable X with an array reference $A[I]$	$X \neq A[I]$
ssr	struct for scalar replacement	replace scalar variable X with struct field S	$X \neq S$
vie	scalar variable initialization elimination	remove initialization of a scalar variable	
car	constant for array replacement	replace array reference $A[I]$ with constant C	$A[I] \neq C$
sar	scalar for array replacement	replace array reference $A[I]$ with scalar variable X	$A[I] \neq X$
cnr	comparable array replacement	replace array reference with a comparable array reference	
sar	struct for array reference replacement	replace array reference $A[I]$ with a struct field S	$A[I] \neq S$
Expression Modifications			
abs	absolute value insertion	replace e by `abs(`e`)`	$e < 0$
aor	arithmetic operator replacement	replace arithmetic operator ψ with arithmetic operator ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
lcr	logical connector replacement	replace logical connector ψ with logical connector ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
ror	relational operator replacement	replace relational operator ψ with relational operator ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
uoi	unary operator insertion	insert unary operator	
cpr	constant for predicate replacement	replace predicate with a constant value	
Statement Modifications			
sdl	statement deletion	delete a statement	
sca	switch case replacement	replace the label of one case with another	
ses	end block shift	move `}` one statement earlier and later	

description	constraint
replace constant $C1$ with constant $C2$	$C1 \neq C2$
replace constant C with scalar variable X	$C \neq X$
replace constant C with array reference $A[I]$	$C \neq A[I]$
replace constant C with struct field S	$C \neq S$
replace scalar variable X with a scalar variable Y	$X \neq Y$
replace scalar variable X with a constant C	$X \neq C$
replace scalar variable X with an array reference $A[I]$	$X \neq A[I]$
replace scalar variable X with struct field S	$X \neq S$
remove initialization of a scalar variable	
replace array reference $A[I]$ with constant C	$A[I] \neq C$
replace array reference $A[I]$ with scalar variable X	$A[I] \neq X$
replace array reference with a comparable array reference	
replace array reference $A[I]$ with a struct field S	$A[I] \neq S$

Ihre Tests

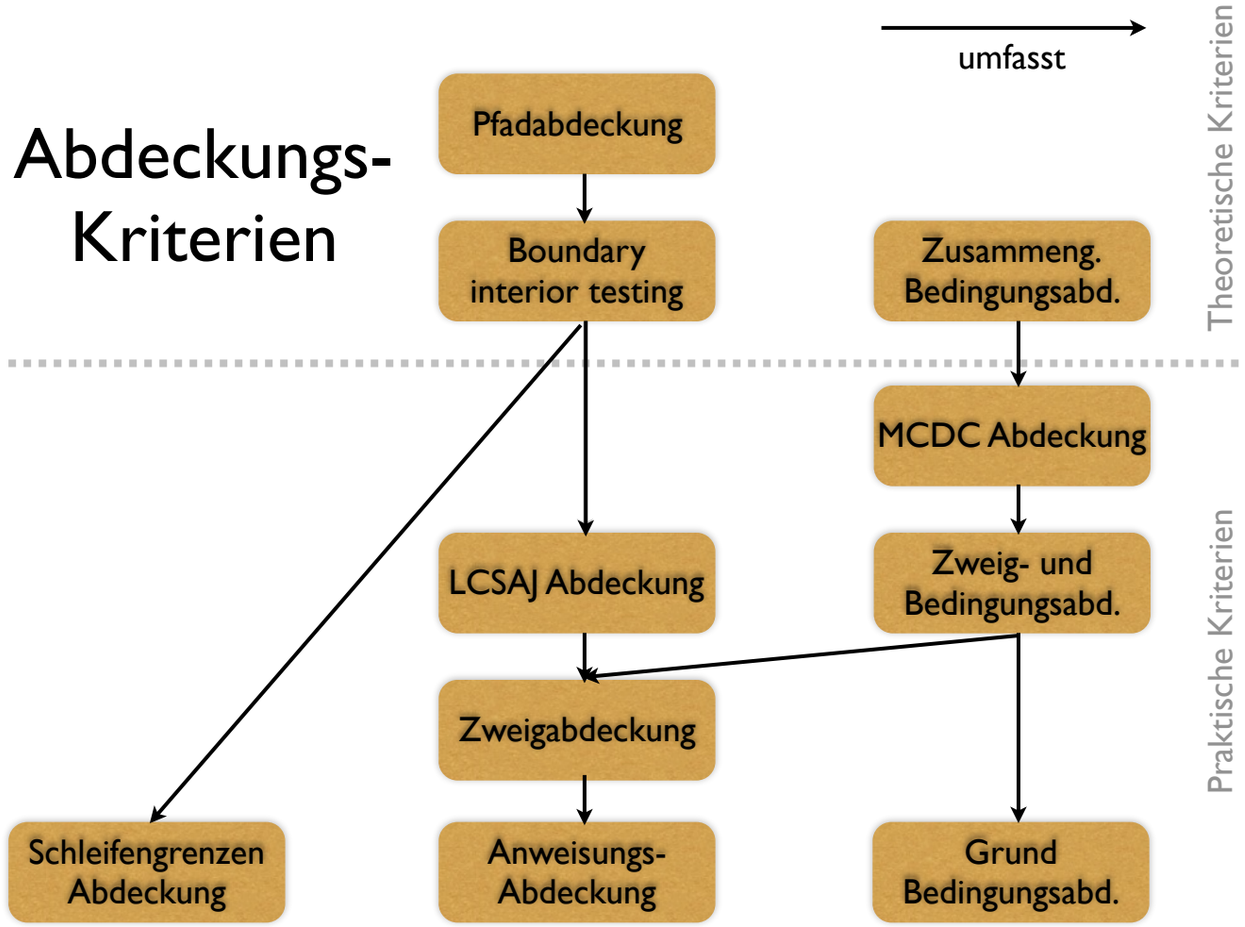
- Wir haben 20 Mutanten in unsere Referenzimplementierung eingeschleust
- Zwei Mal am Tag testen wir Ihre Testsuite gegen vier neue Mutanten
- Ihre Systemtests müssen bis zur Code-Abnahme *15 Mutanten finden*
- Die unveränderte Referenzimplementierung muss aus Sicht Ihrer Systemtests *fehlerfrei* sein

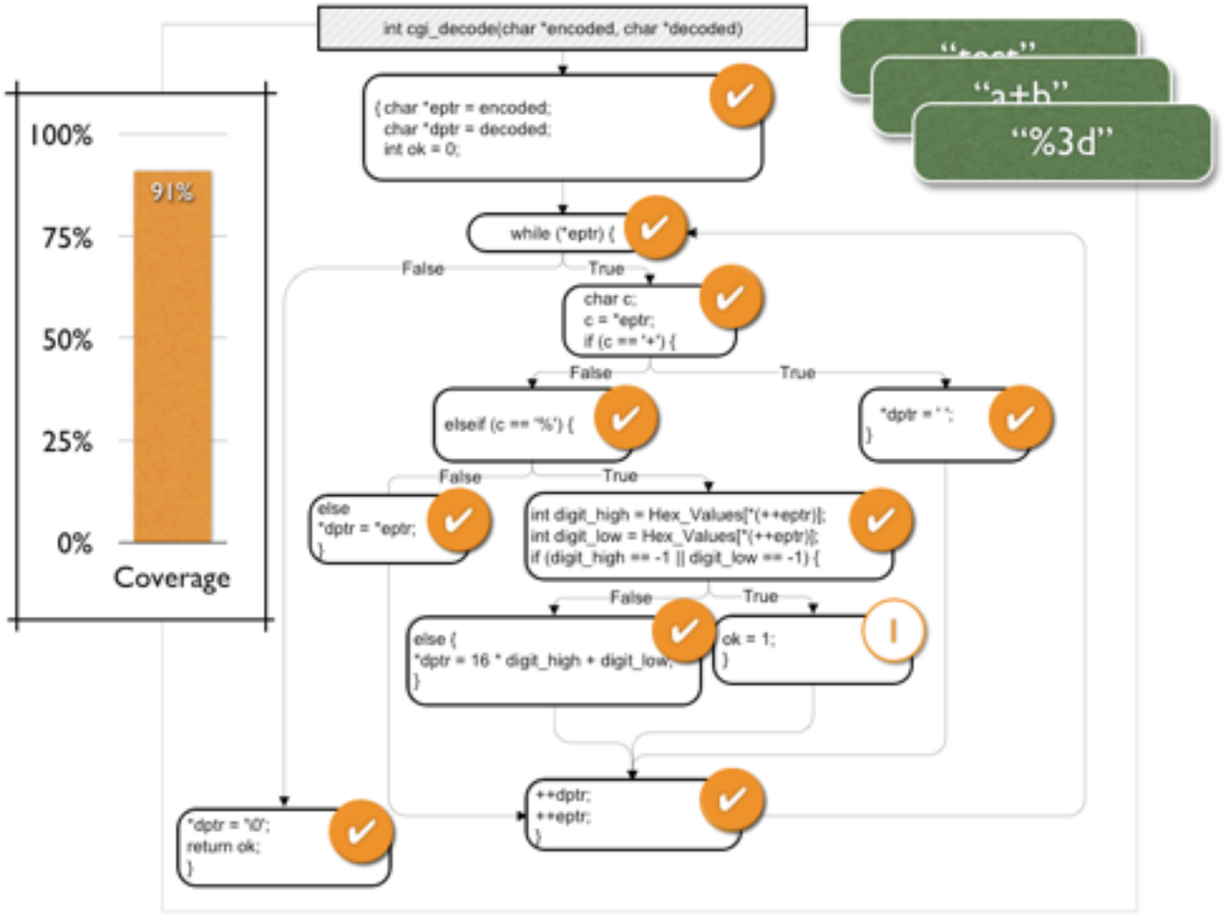
Kontrollflussgraph



- Ein *Kontrollflussgraph* (CFG) drückt mögliche Ausführungspfade eines Programms aus
- *Knoten* sind *Basic Blocks* – Anweisungsfolgen mit 1 Eingang und 1 Ausgang
- *Kanten* stellen *Kontrollfluss* dar – die Möglichkeit, dass ein Basic Block nach einem anderen ausgeführt wird

Abdeckungs-Kriterien





Eine Mutation

```
class Checker {
    public int compareTo(Object other)
    {
        return 1;
    }
}
```

Wird die Testsuite diesen Fehler finden?

Wenn nicht: *Welche Fehler wird sie noch übersehen?*