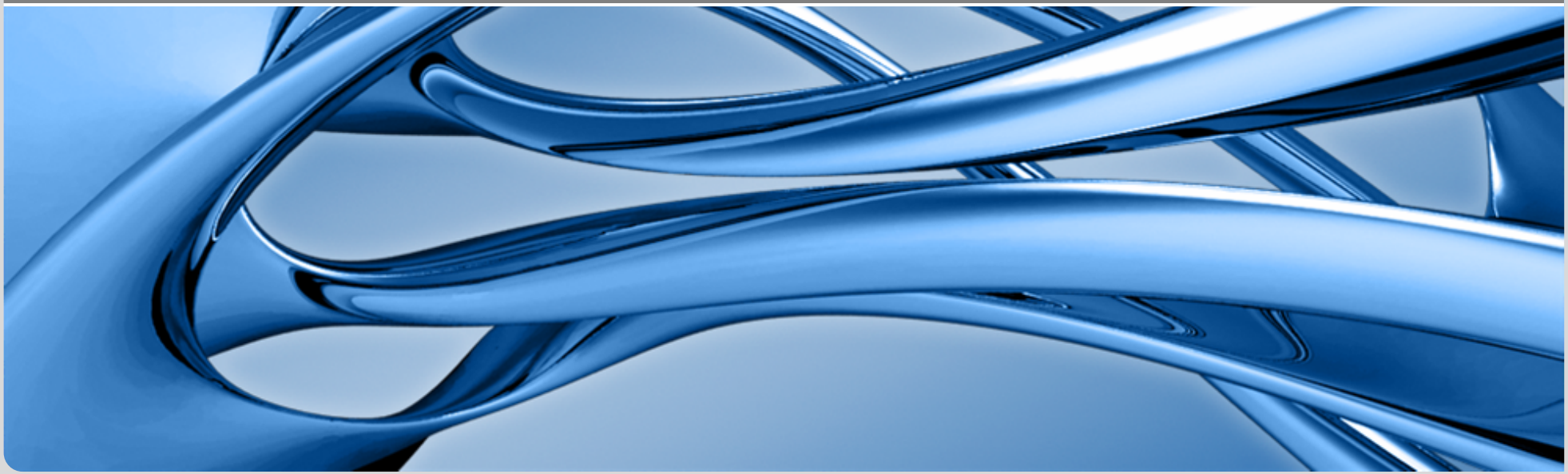
Karlsruher Institut für Technologie

LLBMC: Bounded Model Checking
KLEE: Symbolic Execution

Entscheidungsverfahren mit Anwendungen in der Softwareverifikation

STEPHAN FALKE — INSTITUT FÜR THEORETISCHE INFORMATIK (ITI)



Inhalt

Inhalt

Inhalt

Karlsruher Institut für Technologie

Teil I

Einführung

Verifikation auf der Ebene der Zwischensprache

- Ziel: Verifikationssystem das alle Features einer komplexen Programmiersprache (C, C++, Java, . . .) unterstützt

Verifikation auf der Ebene der Zwischensprache

- Ziel: Verifikationssystem das alle Features einer komplexen Programmiersprache (C, C++, Java, . . .) unterstützt
- Probleme:
  - Komplexer Syntax
  - Komplexe Semantik
  - Viele Grenzfälle

Verifikation auf der Ebene der Zwischensprache

- Ziel: Verifikationssystem das alle Features einer komplexen Programmiersprache (C, C++, Java, . . .) unterstützt
- Probleme:
  - Komplexer Syntax
  - Komplexe Semantik
  - Viele Grenzfälle
- Hohe Anfangshürde, insbesondere für Forschungsprojekte

Verifikation auf der Ebene der Zwischensprache

- Ziel: Verifikationssystem das alle Features einer komplexen Programmiersprache (C, C++, Java, . . .) unterstützt
- Probleme:
  - Komplexer Syntax
  - Komplexe Semantik
  - Viele Grenzfälle
- Hohe Anfangshürde, insbesondere für Forschungsprojekte
- Lösung: Benutze die Zwischensprache eines Compilers
  - Die Zwischensprache hat in der Regel einen einfachen Syntax und eine weniger komplexe Semantik
  - Das Verifikationssystem kann für alle Programmiersprachen benutzt werden die in die Zwischensprache übersetzt werden können
  - Das untersuchte Programm ist “näher” am ausgeführten Programm

LLVM

- LLVM ist ein beliebtes Compiler Framework

LLVM

- LLVM ist ein beliebtes Compiler Framework
- Front-Ends für
  - C, C++, Objective-C, Objective-C++
  - Fortran
  - Ada
  - D, Lua, Hydra, ...

LLVM

- LLVM ist ein beliebtes Compiler Framework
- Front-Ends für
  - C, C++, Objective-C, Objective-C++
  - Fortran
  - Ada
  - D, Lua, Hydra, ...
- Zwischensprache:
  - RISC-artige Assemblersprache für eine Registermachine mit einer unbeschränkten Anzahl von Registern

LLVM

- LLVM ist ein beliebtes Compiler Framework
- Front-Ends für
  - C, C++, Objective-C, Objective-C++
  - Fortran
  - Ada
  - D, Lua, Hydra, ...
- Zwischensprache:
  - RISC-artige Assemblersprache für eine Registermachine mit einer unbeschränkten Anzahl von Registern
  - Programme sind in SSA-Form (static single assignment), d.h., jedes Register wird im (statischen) Programmcode nur einmal geschrieben

LLVM

- LLVM ist ein beliebtes Compiler Framework
- Front-Ends für
  - C, C++, Objective-C, Objective-C++
  - Fortran
  - Ada
  - D, Lua, Hydra, ...
- Zwischensprache:
  - RISC-artige Assemblersprache für eine Registermachine mit einer unbeschränkten Anzahl von Registern
  - Programme sind in SSA-Form (static single assignment), d.h., jedes Register wird im (statischen) Programmcode nur einmal geschrieben
- Codegeneratoren für
  - X86
  - PowerPC
  - ARM
  - Mips
  - Sparc
  - AArch64, Hexagon, MBlaze, MSP430, NVPTX, R600, SystemZ, XCore, ...

Beispiel

```
int power(int x, int y)
{
    int r = 1;
    while (y > 0) {
        r = r*x;
        y = y - 1;
    }
    return r;
}
```

```
define i32 @power(i32 %x, i32 %y) {
entry:
  br label %bb1

bb1:
  %y.0 = phi i32 [ %y, %entry ], [ %2, %bb ]
  %r.0 = phi i32 [ 1, %entry ], [ %1, %bb ]
  %0 = icmp sgt i32 %y.0, 0
  br i1 %0, label %bb, label %return

bb:
  %1 = mul i32 %r.0, %x
  %2 = sub i32 %y.0, 1
  br label %bb1

return:
  ret i32 %r.0
}
```

Verifikation mit LLVM

In den letzten Jahren wurde viele Verifikationssysteme entwickelt die auf LLVMs Zwischensprache aufbauen

- Calysto: Prüfung von Speicherkorrektheit und Assertions (“proof localization”)

Verifikation mit LLVM

In den letzten Jahren wurde viele Verifikationssysteme entwickelt die auf LLVMs Zwischensprache aufbauen

- Calysto: Prüfung von Speicherkorrektheit und Assertions (“proof localization”)
- KLEE: Prüfung diverser Eigenschaften (“symbolic execution”)

Verifikation mit LLVM

In den letzten Jahren wurde viele Verifikationssysteme entwickelt die auf LLVMs Zwischensprache aufbauen

- Calysto: Prüfung von Speicherkorrektheit und Assertions (“proof localization”)
- KLEE: Prüfung diverser Eigenschaften (“symbolic execution”)
- LLBMC: Prüfung diverser Eigenschaften (“bounded model checking”)

Verifikation mit LLVM

In den letzten Jahren wurde viele Verifikationssysteme entwickelt die auf LLVMs Zwischensprache aufbauen

- Calysto: Prüfung von Speicherkorrektheit und Assertions (“proof localization”)
- KLEE: Prüfung diverser Eigenschaften (“symbolic execution”)
- LLBMC: Prüfung diverser Eigenschaften (“bounded model checking”)
- KITTeL: Terminierungsbeweiser

Verifikation mit LLVM

In den letzten Jahren wurde viele Verifikationssysteme entwickelt die auf LLVMs Zwischensprache aufbauen

- Calysto: Prüfung von Speicherkorrektheit und Assertions (“proof localization”)
- KLEE: Prüfung diverser Eigenschaften (“symbolic execution”)
- LLBMC: Prüfung diverser Eigenschaften (“bounded model checking”)
- KITTeL: Terminierungsbeweiser
- LAV: Prüfung diverser Eigenschaften (“symbolic execution”/”bounded model checking”)

Verifikation mit LLVM

In den letzten Jahren wurde viele Verifikationssysteme entwickelt die auf LLVMs Zwischensprache aufbauen

- Calysto: Prüfung von Speicherkorrektheit und Assertions (“proof localization”)
- KLEE: Prüfung diverser Eigenschaften (“symbolic execution”)
- LLBMC: Prüfung diverser Eigenschaften (“bounded model checking”)
- KITTeL: Terminierungsbeweiser
- LAV: Prüfung diverser Eigenschaften (“symbolic execution”/”bounded model checking”)
- Ufo: Erreichbarkeitsprüfung (“interpolation”/”abstract interpretation”)

LLBMC und KLEE

- LLBMC und KLEE sind hauptsächlich zur Prüfung von C Programmen entwickelt worden

LLBMC und KLEE

- LLBMC und KLEE sind hauptsächlich zur Prüfung von C Programmen entwickelt worden
- Von beiden Systemen überprüfte Eigenschaften:
  - Division durch Null

LLBMC und KLEE

- LLBMC und KLEE sind hauptsächlich zur Prüfung von C Programmen entwickelt worden
- Von beiden Systemen überprüfte Eigenschaften:
  - Division durch Null
  - Korrektheit von Speicherzugriffen

LLBMC und KLEE

- LLBMC und KLEE sind hauptsächlich zur Prüfung von C Programmen entwickelt worden
- Von beiden Systemen überprüfte Eigenschaften:
  - Division durch Null
  - Korrektheit von Speicherzugriffen
  - Vom Programmierer gegebene Assertions

LLBMC und KLEE

- LLBMC und KLEE sind hauptsächlich zur Prüfung von C Programmen entwickelt worden
- Von beiden Systemen überprüfte Eigenschaften:
  - Division durch Null
  - Korrektheit von Speicherzugriffen
  - Vom Programmierer gegebene Assertions
- Nur von LLBMC überprüfte Eigenschaften:
  - Arithmetische Unter- und Überläufe

LLBMC und KLEE

- LLBMC und KLEE sind hauptsächlich zur Prüfung von C Programmen entwickelt worden
- Von beiden Systemen überprüfte Eigenschaften:
  - Division durch Null
  - Korrektheit von Speicherzugriffen
  - Vom Programmierer gegebene Assertions
- Nur von LLBMC überprüfte Eigenschaften:
  - Arithmetische Unter- und Überläufe
  - Undefinierte Shift Operationen

LLBMC und KLEE

- LLBMC und KLEE sind hauptsächlich zur Prüfung von C Programmen entwickelt worden
- Von beiden Systemen überprüfte Eigenschaften:
  - Division durch Null
  - Korrektheit von Speicherzugriffen
  - Vom Programmierer gegebene Assertions
- Nur von LLBMC überprüfte Eigenschaften:
  - Arithmetische Unter- und Überläufe
  - Undefinierte Shift Operationen
  - Korrekte Benutzung von Speicherallokationen (malloc/free)

LLBMC und KLEE

- LLBMC und KLEE sind hauptsächlich zur Prüfung von C Programmen entwickelt worden
- Von beiden Systemen überprüfte Eigenschaften:
  - Division durch Null
  - Korrektheit von Speicherzugriffen
  - Vom Programmierer gegebene Assertions
- Nur von LLBMC überprüfte Eigenschaften:
  - Arithmetische Unter- und Überläufe
  - Undefinierte Shift Operationen
  - Korrekte Benutzung von Speicherallokationen (malloc/free)
  - ...

LLBMC und KLEE

- LLBMC und KLEE sind hauptsächlich zur Prüfung von C Programmen entwickelt worden
- Von beiden Systemen überprüfte Eigenschaften:
  - Division durch Null
  - Korrektheit von Speicherzugriffen
  - Vom Programmierer gegebene Assertions
- Nur von LLBMC überprüfte Eigenschaften:
  - Arithmetische Unter- und Überläufe
  - Undefinierte Shift Operationen
  - Korrekte Benutzung von Speicherallokationen (malloc/free)
  - ...
- Beide Systeme spüren Fehler auf und können i.A. die Abwesenheit von Fehlern nicht beweisen

LLBMC und KLEE

- LLBMC und KLEE sind hauptsächlich zur Prüfung von C Programmen entwickelt worden
- Von beiden Systemen überprüfte Eigenschaften:
  - Division durch Null
  - Korrektheit von Speicherzugriffen
  - Vom Programmierer gegebene Assertions
- Nur von LLBMC überprüfte Eigenschaften:
  - Arithmetische Unter- und Überläufe
  - Undefinierte Shift Operationen
  - Korrekte Benutzung von Speicherallokationen (malloc/free)
  - ...
- Beide Systeme spüren Fehler auf und können i.A. die Abwesenheit von Fehlern nicht beweisen
- Trotz dieser Gemeinsamkeiten benutzen LLBMC und KLEE unterschiedliche Techniken

Karlsruher Institut für Technologie

Teil II

Bounded Model Checking mit LLBMC

Bounded Model Checking – Idee

- Es ist i.A. unentscheidbar ob ein Programm eine gegebene Eigenschaft erfüllt

Bounded Model Checking – Idee

- Es ist i.A. unentscheidbar ob ein Programm eine gegebene Eigenschaft erfüllt
- Aber: Verletzungen finden immer nach endlichen Läufen des Programms statt

Bounded Model Checking – Idee

- Es ist i.A. unentscheidbar ob ein Programm eine gegebene Eigenschaft erfüllt
- Aber: Verletzungen finden immer nach endlichen Läufen des Programms statt
- Idee des Bounded Model Checking:
  - Analysiere nur beschränkte Läufe des Programms

Bounded Model Checking – Idee

- Es ist i.A. unentscheidbar ob ein Programm eine gegebene Eigenschaft erfüllt
- Aber: Verletzungen finden immer nach endlichen Läufen des Programms statt
- Idee des Bounded Model Checking:
  - Analysiere nur beschränkte Läufe des Programms
    - Schränke die Tiefe von betrachteten Funktionsaufrufen ein und inline Funktionen an ihre Aufrufstellen

Bounded Model Checking – Idee

- Es ist i.A. unentscheidbar ob ein Programm eine gegebene Eigenschaft erfüllt
- Aber: Verletzungen finden immer nach endlichen Läufen des Programms statt
- Idee des Bounded Model Checking:
  - Analysiere nur beschränkte Läufe des Programms
    - Schränke die Tiefe von betrachteten Funktionsaufrufen ein und inline Funktionen an ihre Aufrufstellen
    - Schränke die Anzahl der Schleifeniterationen ein und rolle die Schleifen entsprechend ab

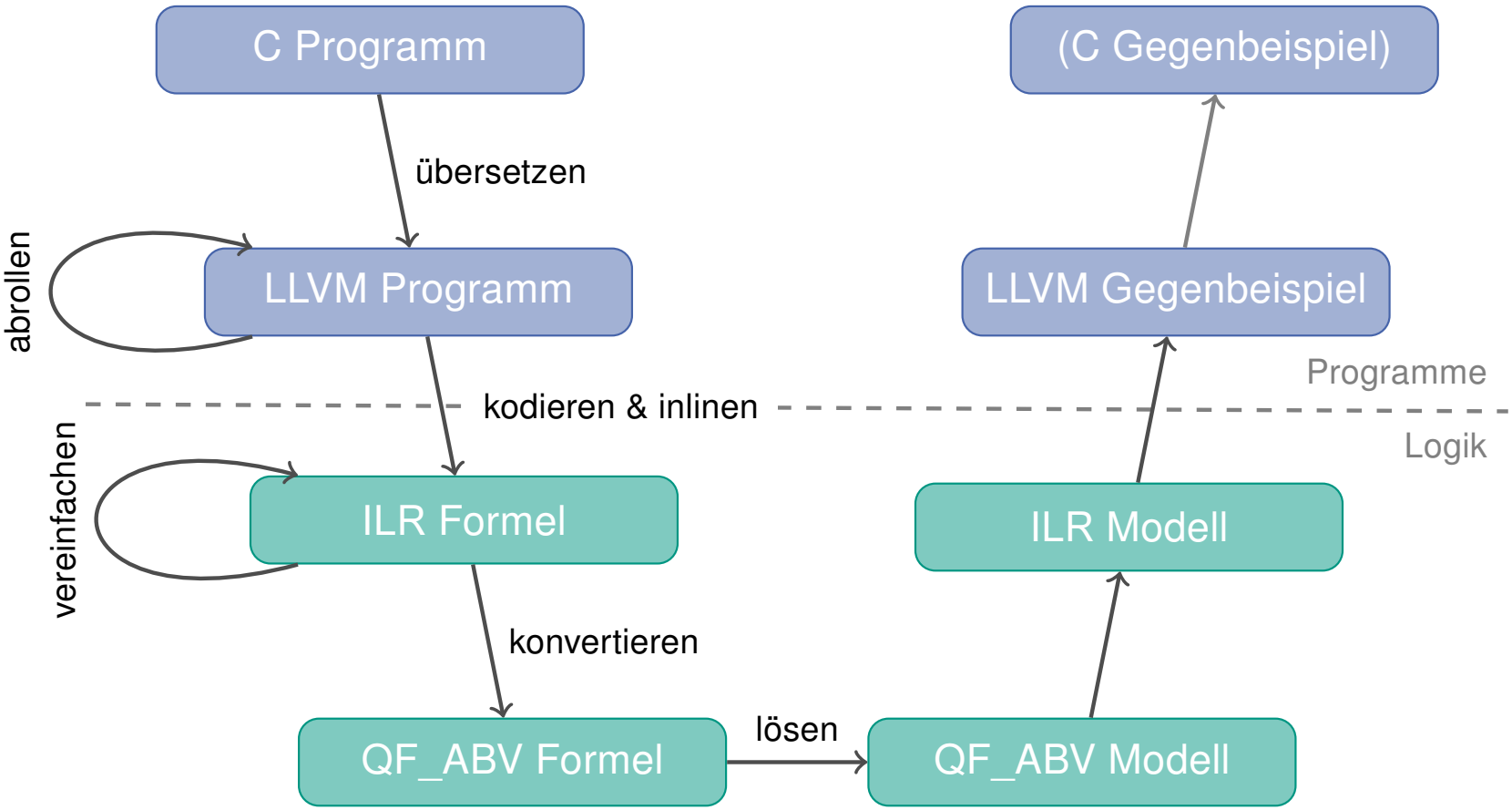
Bounded Model Checking – Idee

- Es ist i.A. unentscheidbar ob ein Programm eine gegebene Eigenschaft erfüllt
- Aber: Verletzungen finden immer nach endlichen Läufen des Programms statt
- Idee des Bounded Model Checking:
  - Analysiere nur beschränkte Läufe des Programms
    - Schränke die Tiefe von betrachteten Funktionsaufrufen ein und inline Funktionen an ihre Aufrufstellen
    - Schränke die Anzahl der Schleifeniterationen ein und rolle die Schleifen entsprechend ab
  - Es ist dann entscheidbar ob eine Eigenschaft innerhalb der Schranken verletzt wird

Bounded Model Checking – Idee

- Es ist i.A. unentscheidbar ob ein Programm eine gegebene Eigenschaft erfüllt
- Aber: Verletzungen finden immer nach endlichen Läufen des Programms statt
- Idee des Bounded Model Checking:
  - Analysiere nur beschränkte Läufe des Programms
    - Schränke die Tiefe von betrachteten Funktionsaufrufen ein und inline Funktionen an ihre Aufrufstellen
    - Schränke die Anzahl der Schleifeniterationen ein und rolle die Schleifen entsprechend ab
  - Es ist dann entscheidbar ob eine Eigenschaft innerhalb der Schranken verletzt wird
  - Aber: Abwesenheit von Fehlern kann i.A. nicht bewiesen werden

LLBMC – Überblick



LLBMC – Details

- ILR erweitert QF_ABV um spezielle Intrinsics zum Prüfen der eingebauten Eigenschaften

LLBMC – Details

- ILR erweitert QF_ABV um spezielle Intrinsics zum Prüfen der eingebauten Eigenschaften
  - Diese müssen nach QF_ABV umgesetzt werden bevor ein SMT-Solver aufgerufen wird

LLBMC – Details

- ILR erweitert QF_ABV um spezielle Intrinsics zum Prüfen der eingebauten Eigenschaften
  - Diese müssen nach QF_ABV umgesetzt werden bevor ein SMT-Solver aufgerufen wird
- LLBMC kann die folgenden SMT-Solver ansteuern:
  - STP
  - Boolector
  - SONOLAR
  - Yices2
  - CVC4
  - Z3

LLBMC – Details

- ILR erweitert QF_ABV um spezielle Intrinsics zum Prüfen der eingebauten Eigenschaften
  - Diese müssen nach QF_ABV umgesetzt werden bevor ein SMT-Solver aufgerufen wird
- LLBMC kann die folgenden SMT-Solver ansteuern:
  - STP
  - Boolector
  - SONOLAR
  - Yices2
  - CVC4
  - Z3
- Erzeugte QF_ABV Formeln haben komplexe Boolesche Struktur

Formelerzeugung in LLBMC

Ziel: Zeige dass

$$C \rightarrow P$$

gültig ist

Hierbei kodiert C die beschränkten Läufe des Programms und P die Eigenschaften die erfüllt seien sollen

Formelerzeugung in LLBMC

Ziel: Zeige dass

$$C \rightarrow P$$

gültig ist

Hierbei kodiert C die beschränkten Läufe des Programms und P die Eigenschaften die erfüllt seien sollen

Noch zu tun:

Kodiere die Eigenschaften P ☐

Kodiere die beschränkten Programmläufe C ☐

Kodierung der Eigenschaften

Für eine Instruktion

l: **int** z = x / y;

soll geprüft werden dass

$$y \neq 0$$

gilt wann immer die Instruktion ausgeführt wird

Kodierung der Eigenschaften

Für eine Instruktion

```
l: int z = x / y;
```

soll geprüft werden dass

$$y \neq 0$$

gilt wann immer die Instruktion ausgeführt wird

Es muss also gezeigt werden dass

$$c_{exec}(l) \rightarrow y \neq 0$$

gilt wobei

$$c_{exec}(l) = \begin{cases} \top & \text{Instruktion } l \text{ wird ausgeführt} \\ \bot & \text{Instruktion } l \text{ wird nicht ausgeführt} \end{cases}$$

die Ausführungsbedingung von l ist

Formelerzeugung in LLBMC

Ziel: Zeige dass

$$C \rightarrow P$$

gültig ist

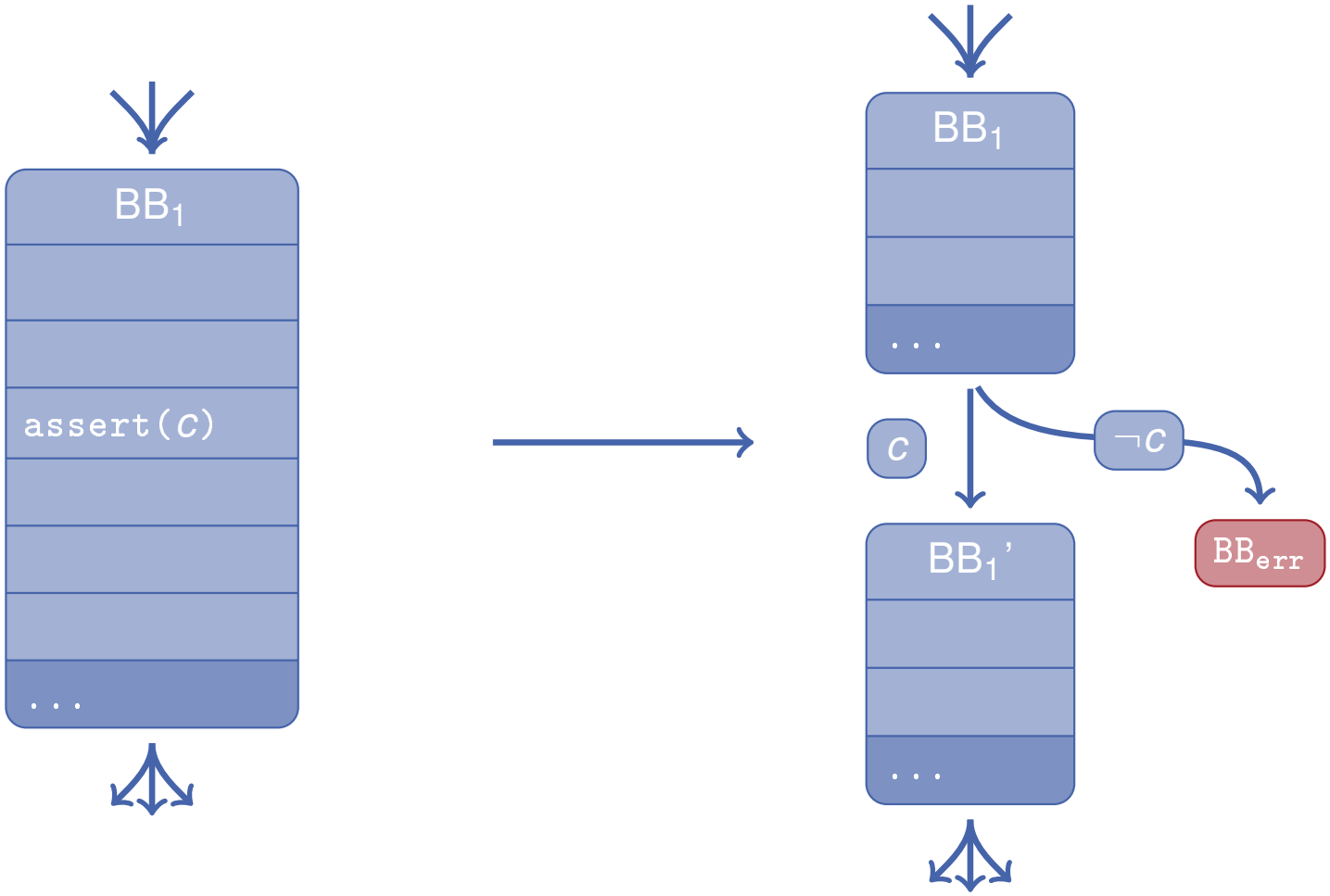
Noch zu tun:

Kodiere die Eigenschaften P ☐

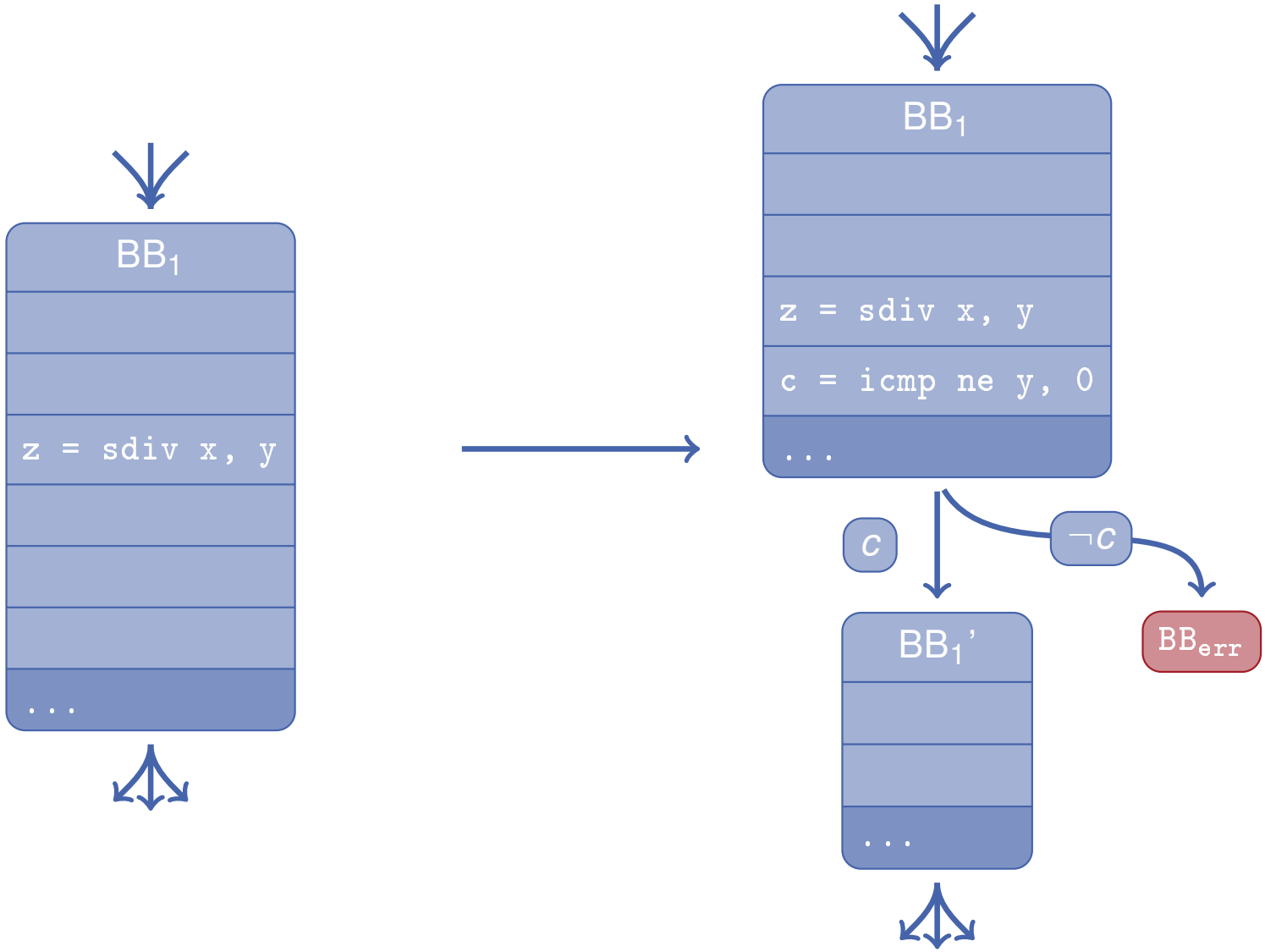
- Kodiere die Tests ☐
- Kodiere die Ausführunsbedingungen ☐

Kodiere die beschränkten Programmläufe C ☐

Kodierung von Assertions



Kodierung der Tests



Formelerzeugung in LLBMC

Ziel: Zeige dass

$$C \rightarrow P$$

gültig ist

Noch zu tun:

Kodiere die Eigenschaften P ☐

- Kodiere die Tests ☑
- Kodiere die Ausführunsbedingungen ☐

Kodiere die beschränkten Programmläufe C ☐

Formelerzeugung in LLBMC

Ziel: Zeige dass

$$c_{exec}(BB_{\text{err}})$$

unerfüllbar ist

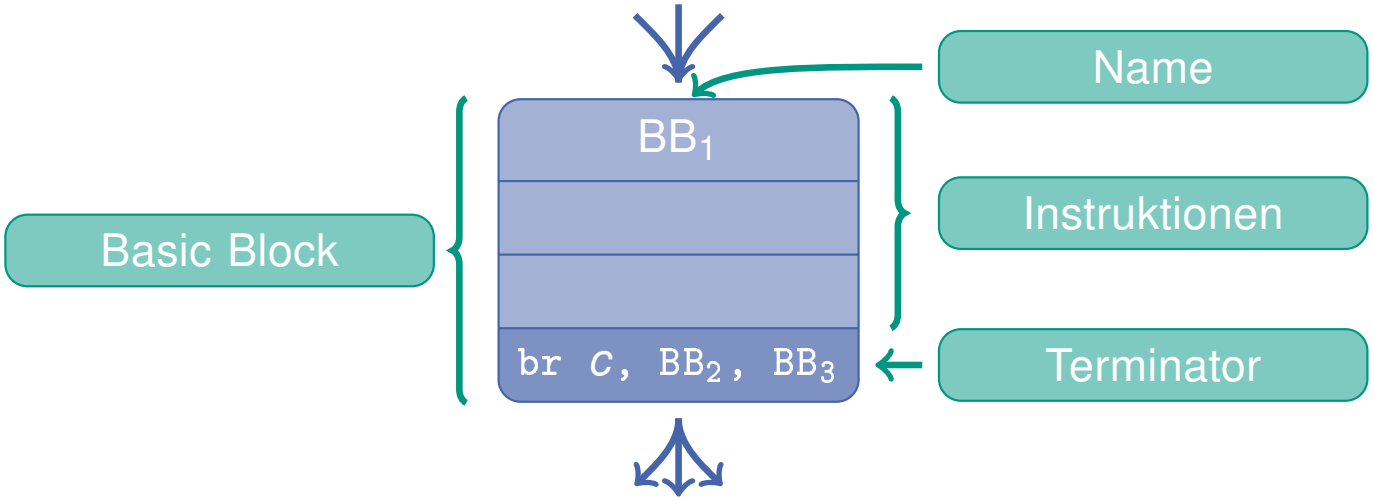
Noch zu tun:

Kodiere die Eigenschaften P ☐

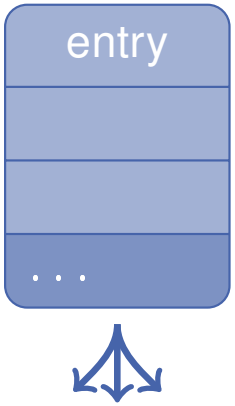
- Kodiere die Tests ☑
- Kodiere die Ausführunsbedingungen ☐

Kodiere die beschränkten Programmläufe C ☐

Basic Blocks

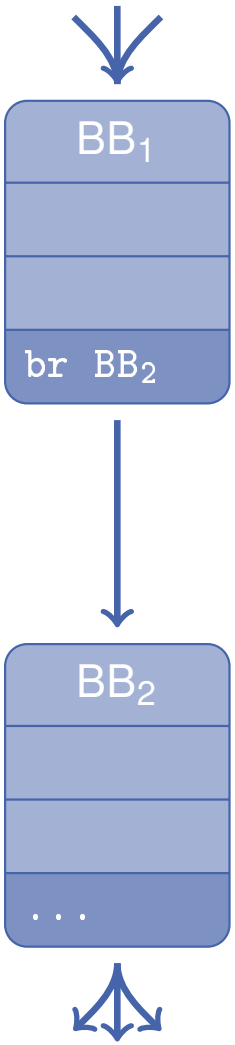


Ausführungsbedingungen – `entry` Block



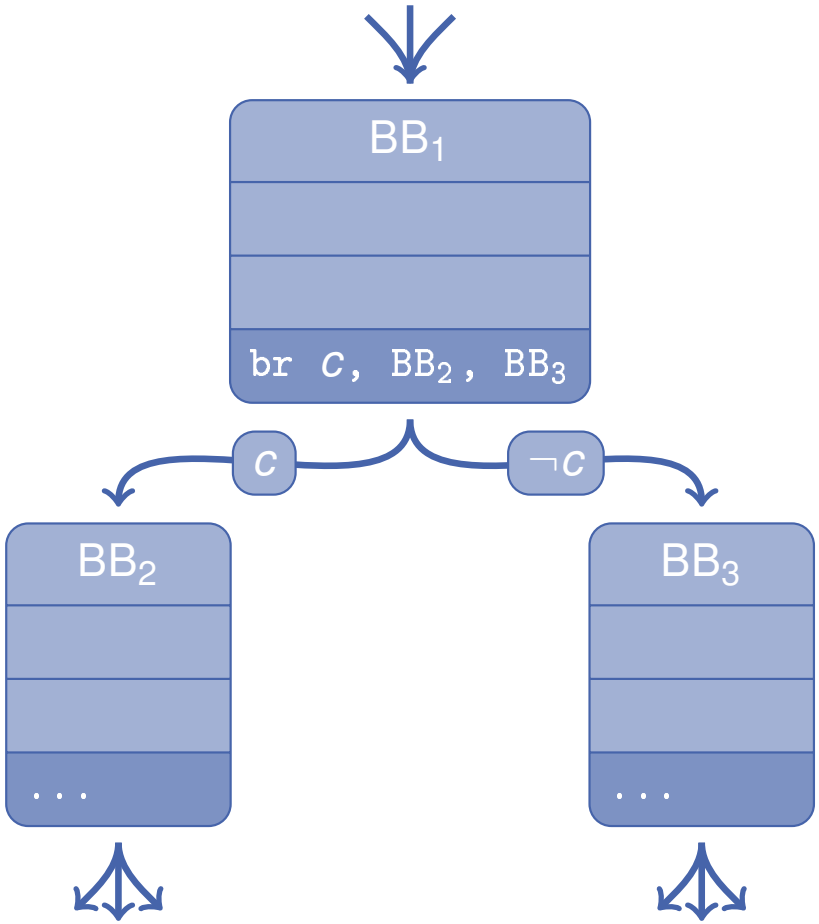
$$c_{exec}(entry) = true$$

Ausführungsbedingungen – Unbedingte Sprünge



$$c_{exec}(BB_2) = c_{exec}(BB_1)$$

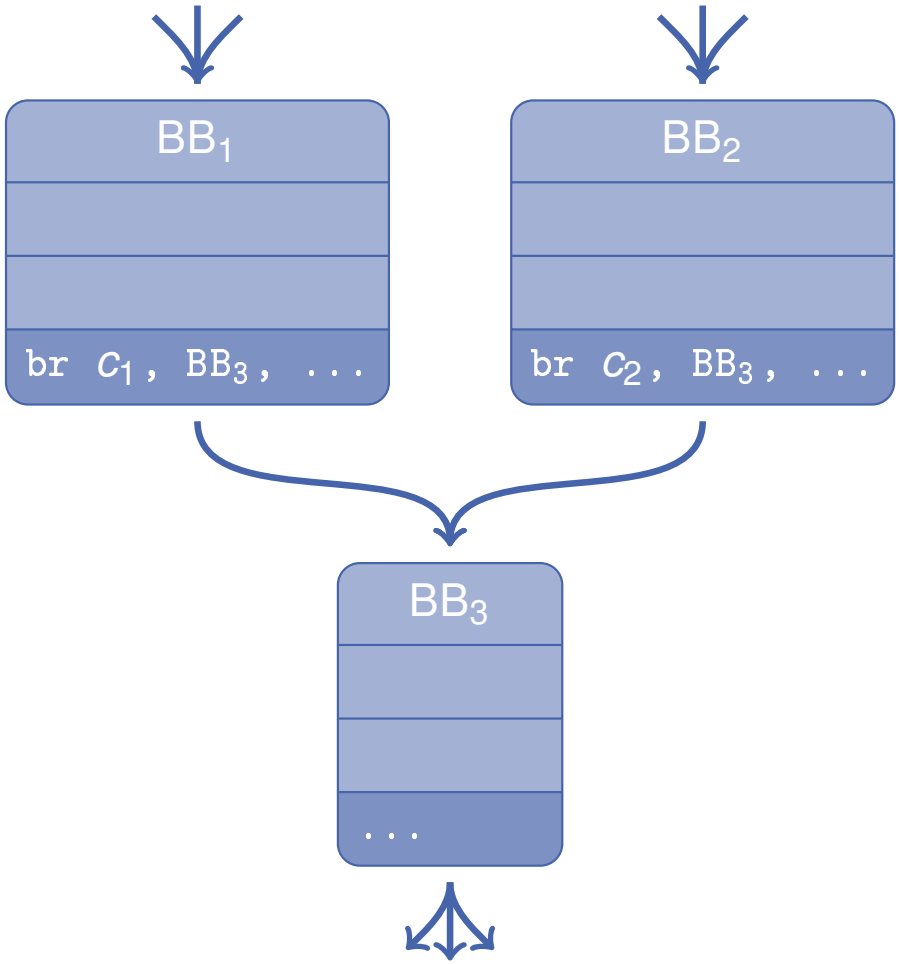
Ausführungsbedingungen – Bedingte Sprünge



$$c_{exec}(BB_2) = c_{exec}(BB_1) \wedge c$$

$$c_{exec}(BB_3) = c_{exec}(BB_1) \wedge \neg c$$

Ausführungsbedingungen – Mehrere Vorgänger



$$c_{exec}(BB_3) = c_{exec}(BB_1) \wedge c_1 \vee c_{exec}(BB_2) \wedge c_2$$

Formelerzeugung in LLBMC

Ziel: Zeige dass

$$c_{exec}(BB_{\text{err}})$$

unerfüllbar ist

Noch zu tun:

Kodiere die Eigenschaften P ☑

- Kodiere die Tests ☑
- Kodiere die Ausführunsbedingungen ☑

Kodiere die beschränkten Programmläufe C ☐

- innerhalb eines Basic Blocks ☐
- über Basic Blöcke hinweg ☐
- Speicherzugriffe ☐
- Speicherallokationen ☐

Kodierung eines Basic Blocks

Beispiel

```
int f(int x, int y) {
    return ((x − y > 0) == (x > y));
}
```

```
define i32 @f(i32 %x, i32 %y) {
entry:
  %sub = sub nsw i32 %x, %y
  %cmp = icmp sgt i32 %sub, 0
  %conv = zext i1 %cmp to i32
  %cmp1 = icmp sgt i32 %x, %y
  %conv2 = zext i1 %cmp1 to i32
  %cmp3 = icmp eq i32 %conv, %conv2
  %conv4 = zext i1 %cmp3 to i32
  ret %conv4
}
```

$$
\begin{aligned}
& sub = \texttt{bvsub}\ x\ y \\
\wedge\ & cmp = (\texttt{bvsgt}\ sub\ \texttt{bv}_{32,0})\ ?\ \texttt{bv}_{1,1}\ :\ \texttt{bv}_{1,0} \\
\wedge\ & conv = \texttt{zero\_extend}_{31}\ cmp \\
\wedge\ & cmp1 = (\texttt{bvsgt}\ x\ y)\ ?\ \texttt{bv}_{1,1}\ :\ \texttt{bv}_{1,0} \\
\wedge\ & conv2 = \texttt{zero\_extend}_{31}\ cmp1 \\
\wedge\ & cmp3 = (conv = conv2)\ ?\ \texttt{bv}_{1,1}\ :\ \texttt{bv}_{1,0} \\
\wedge\ & conv4 = \texttt{zero\_extend}_{31}\ cmp3
\end{aligned}
$$

Formelerzeugung in LLBMC

Ziel: Zeige dass

$$c_{exec}(BB_{\text{err}})$$

unerfüllbar ist

Noch zu tun:

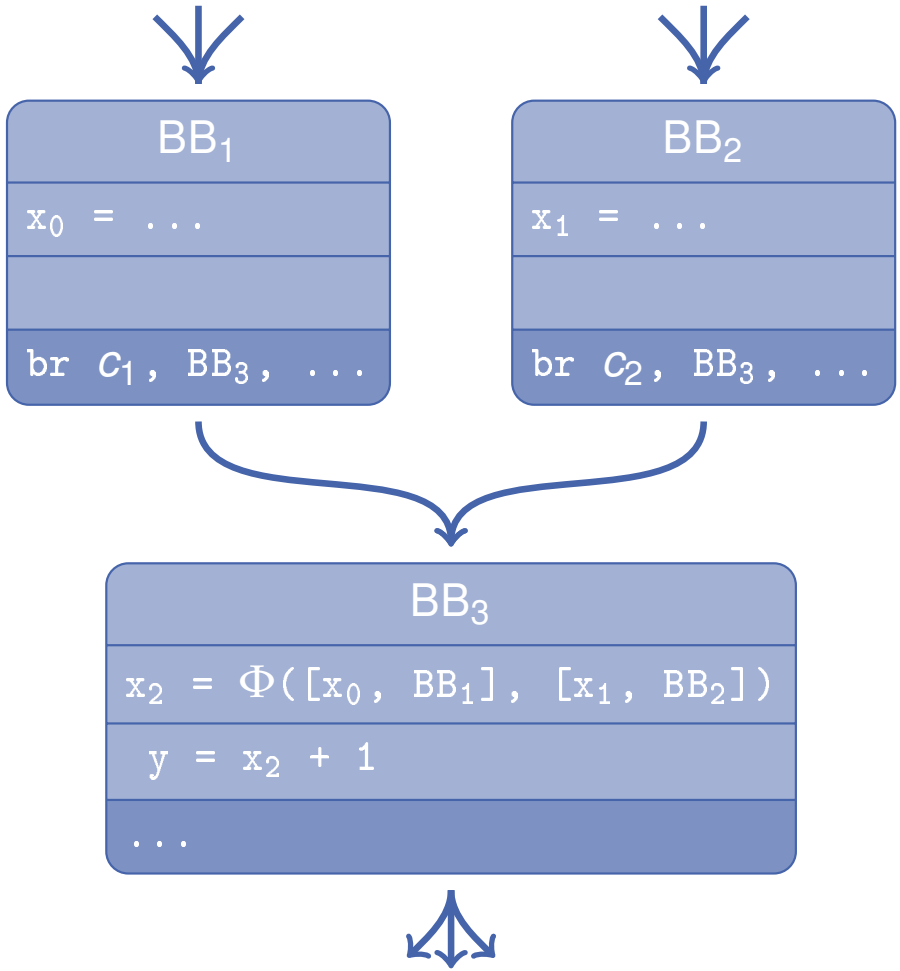
Kodiere die Eigenschaften P ☑

- Kodiere die Tests ☑
- Kodiere die Ausführunsbedingungen ☑

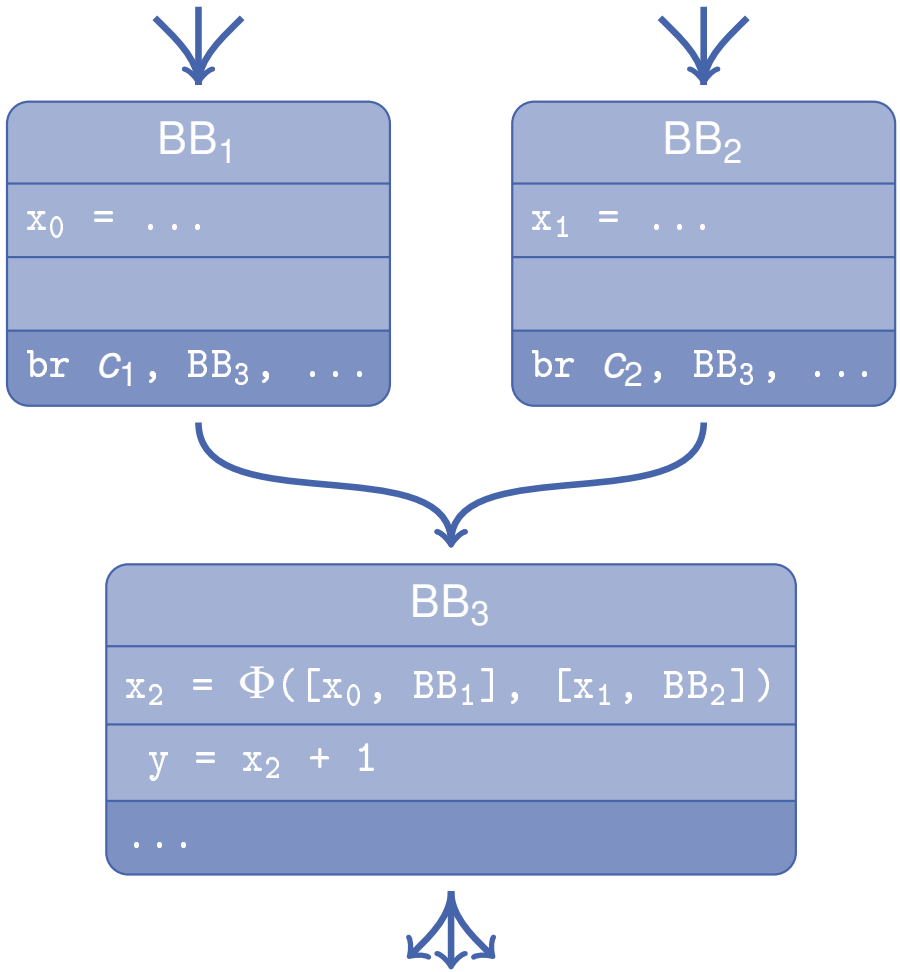
Kodiere die beschränkten Programmläufe C ☐

- innerhalb eines Basic Blocks ☑
- über Basic Blöcke hinweg ☐
- Speicherzugriffe ☐
- Speicherallokationen ☐

Sprünge

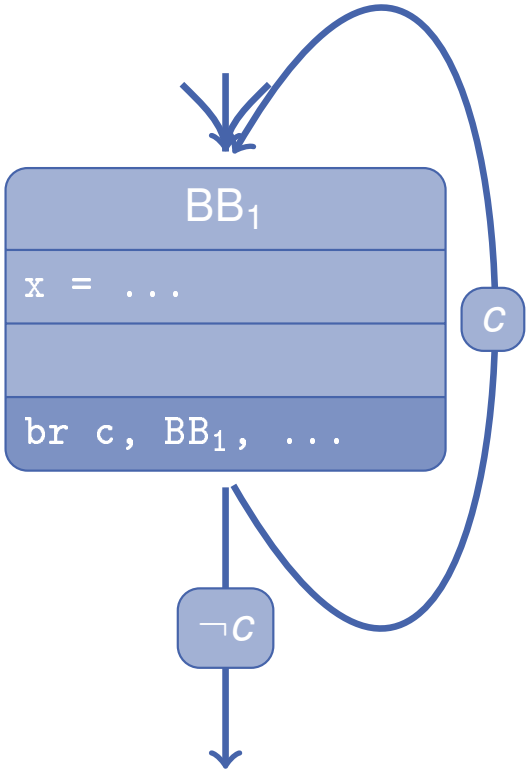


Sprünge

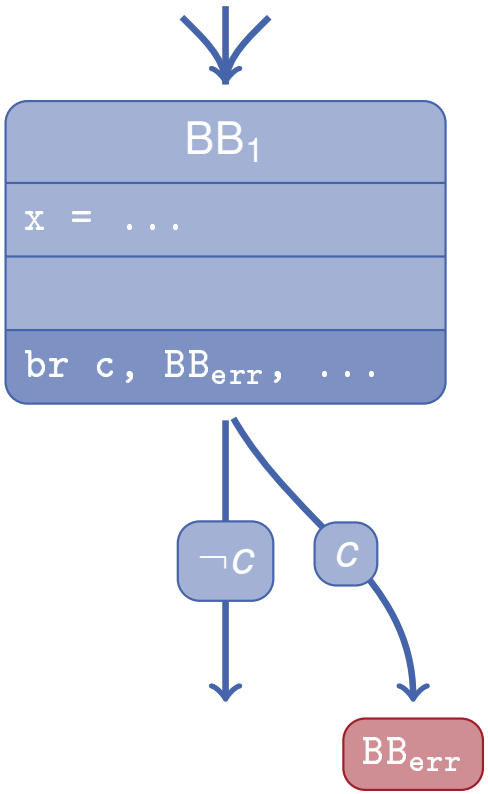


$$x_2 = c_{exec}(BB_1) \; ? \; x_0 \; : \; x_1$$

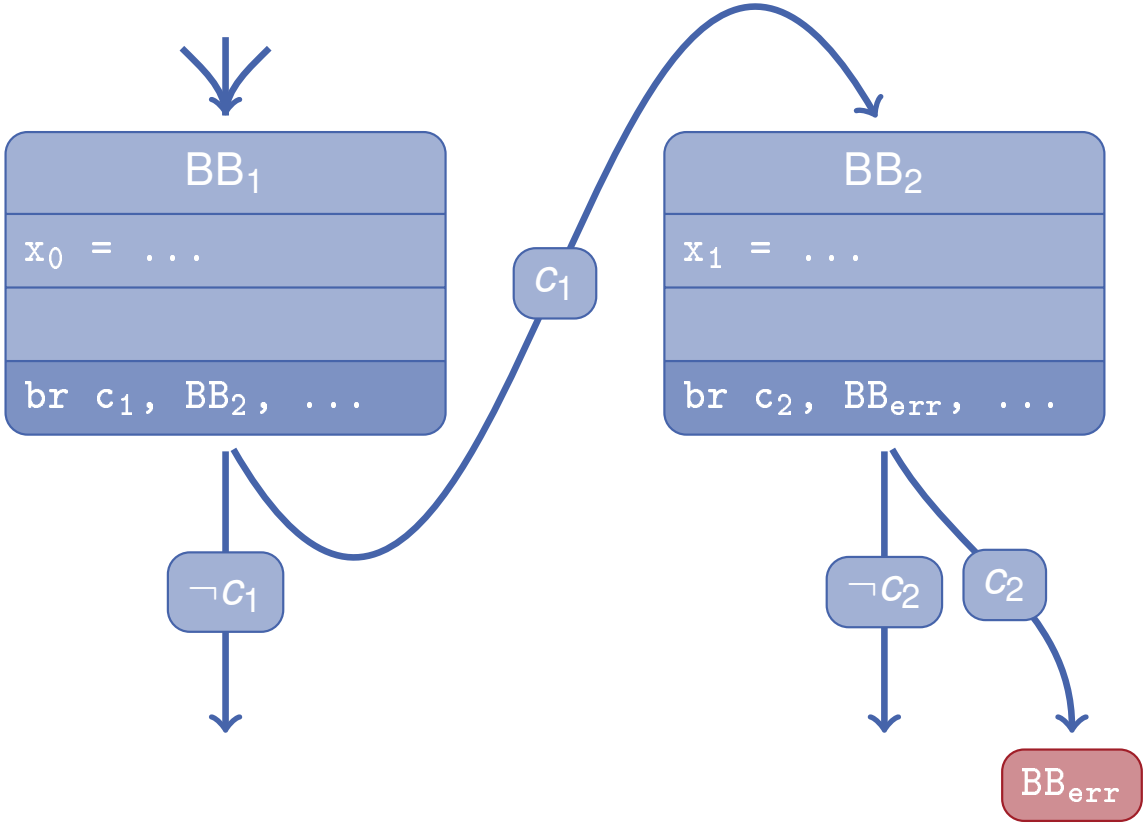
Schleifen



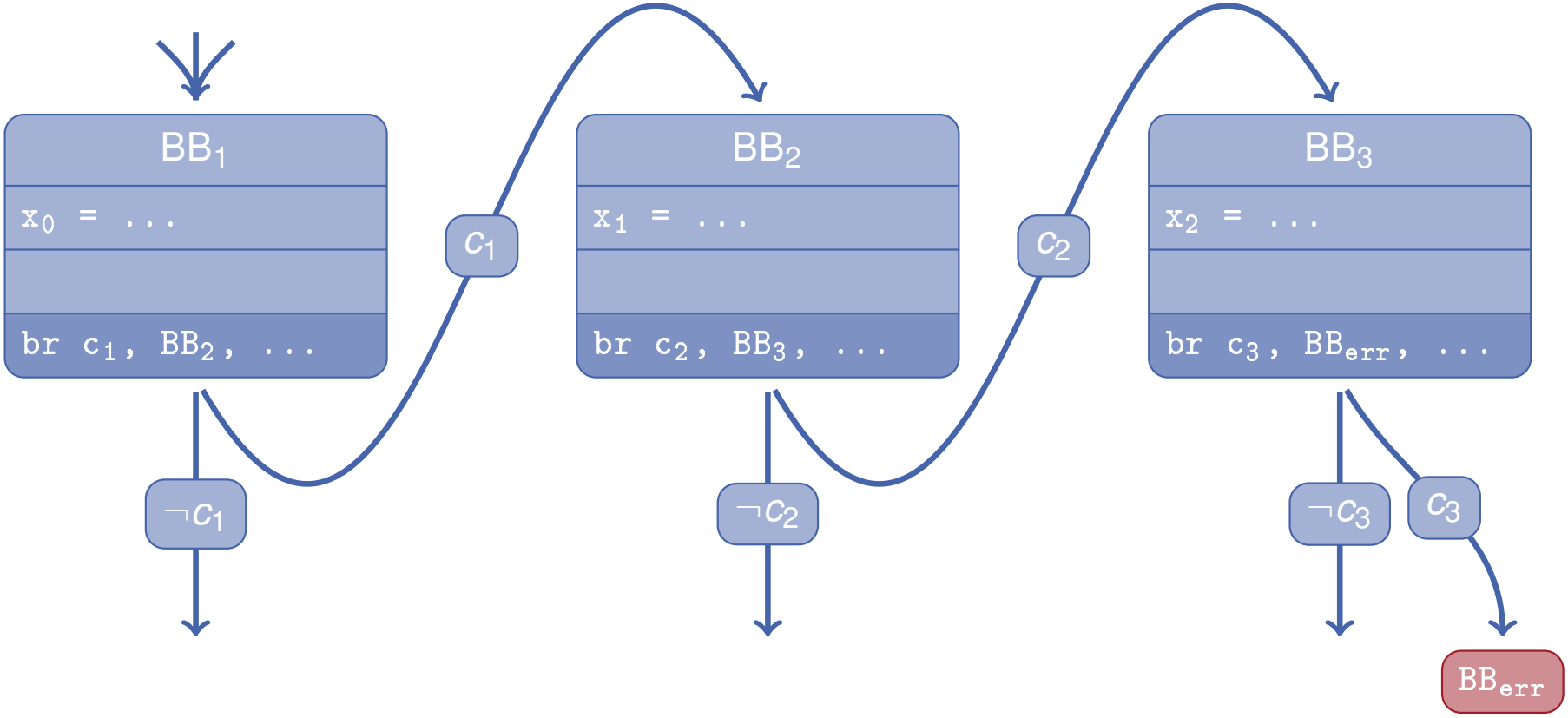
Schleifen



Schleifen



Schleifen


BB1
x0 = ...
br c1, BB2, ...
BB2
x1 = ...
br c2, BB3, ...
BB3
x2 = ...
br c3, BBerr, ...
c1
¬c1
c2
¬c2
¬c3
c3
BBerr

Formelerzeugung in LLBMC

Ziel: Zeige dass

$$c_{exec}(BB_{\text{err}})$$

unerfüllbar ist

Noch zu tun:

Kodiere die Eigenschaften P ☑

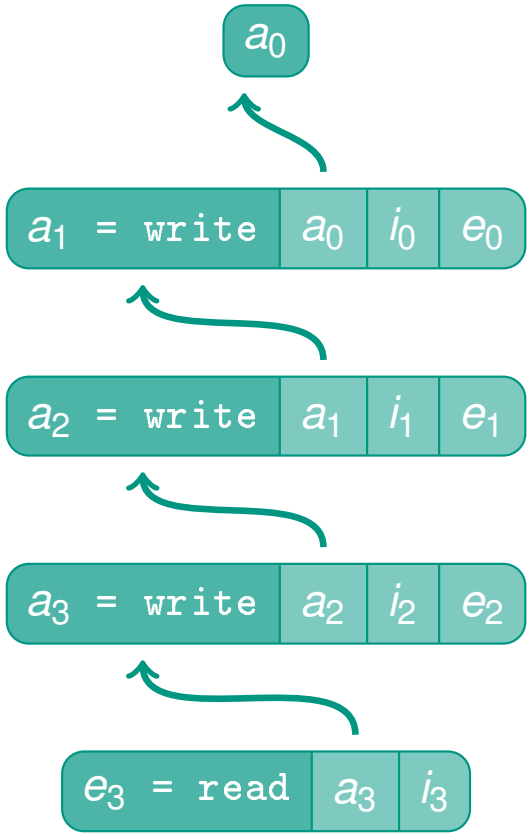
- Kodiere die Tests ☑
- Kodiere die Ausführunsbedingungen ☑

Kodiere die beschränkten Programmläufe C ☐

- innerhalb eines Basic Blocks ☑
- über Basic Blöcke hinweg ☑
- Speicherzugriffe ☐
- Speicherallokationen ☐

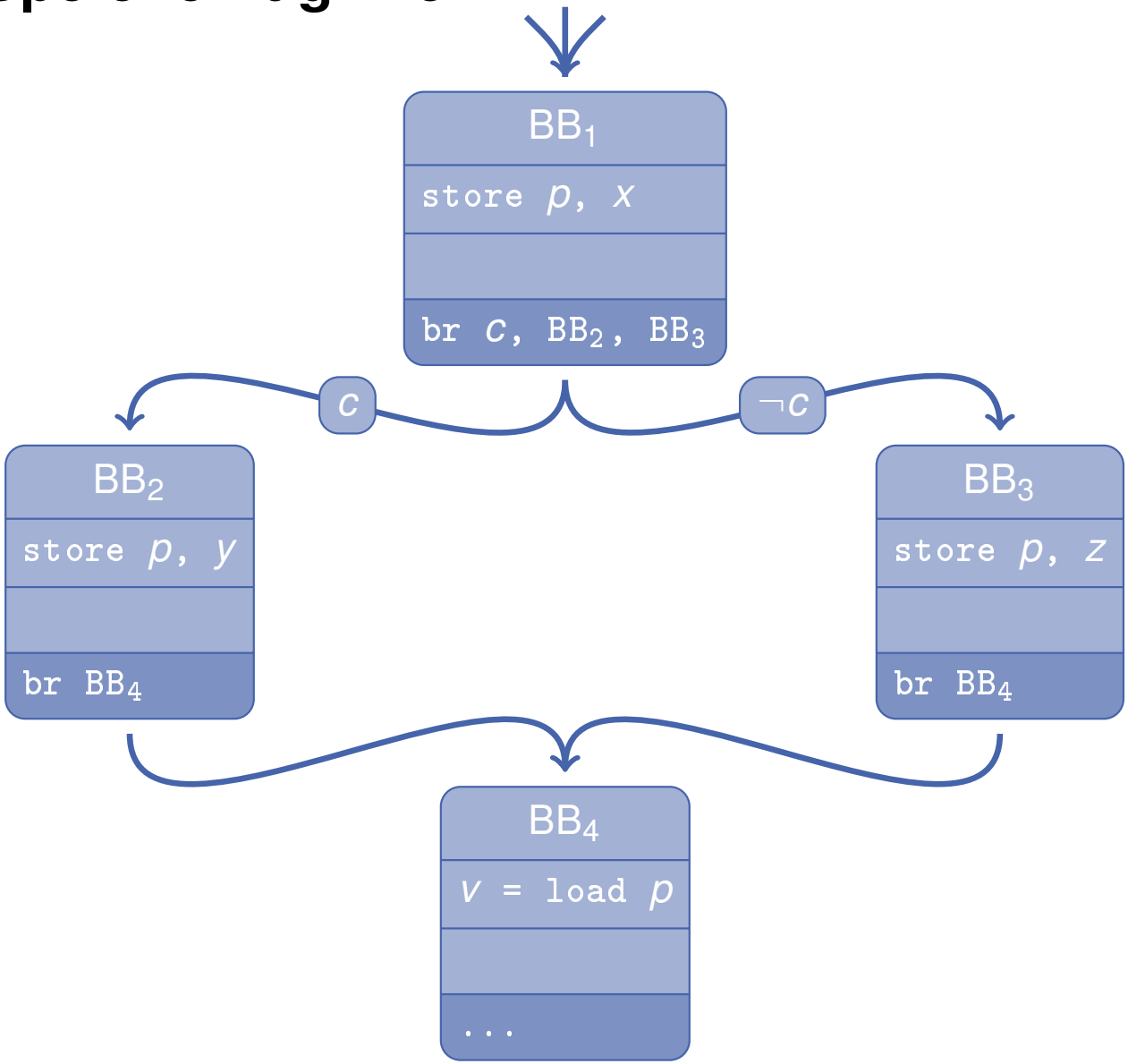
Theorie der Arrays

$$write : A \times I \times E \rightarrow A$$
$$read : A \times I \rightarrow E$$

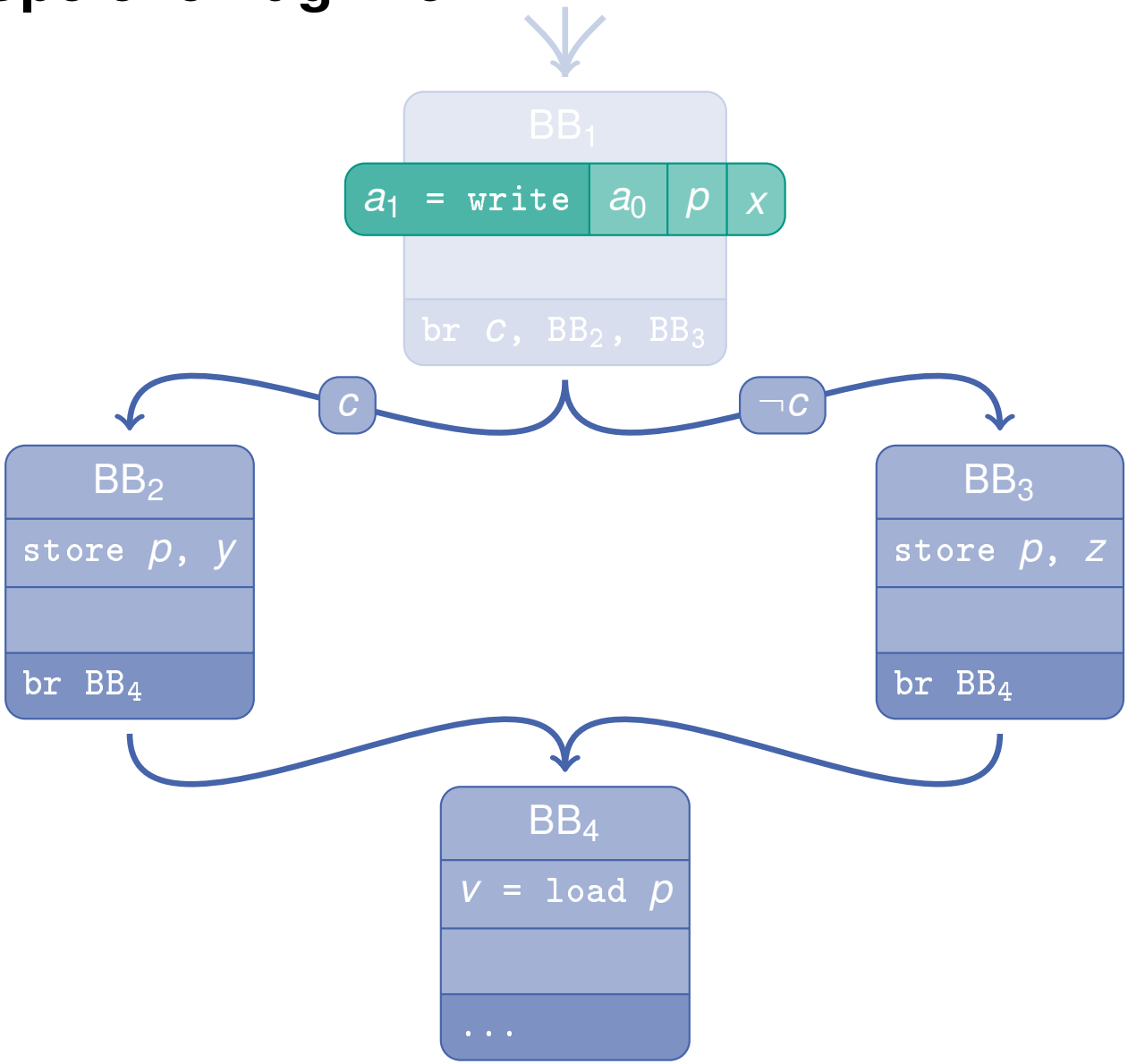


$$read(write(write(write(a_0, i_0, e_0), i_1, e_1), i_2, e_2), i_3)$$

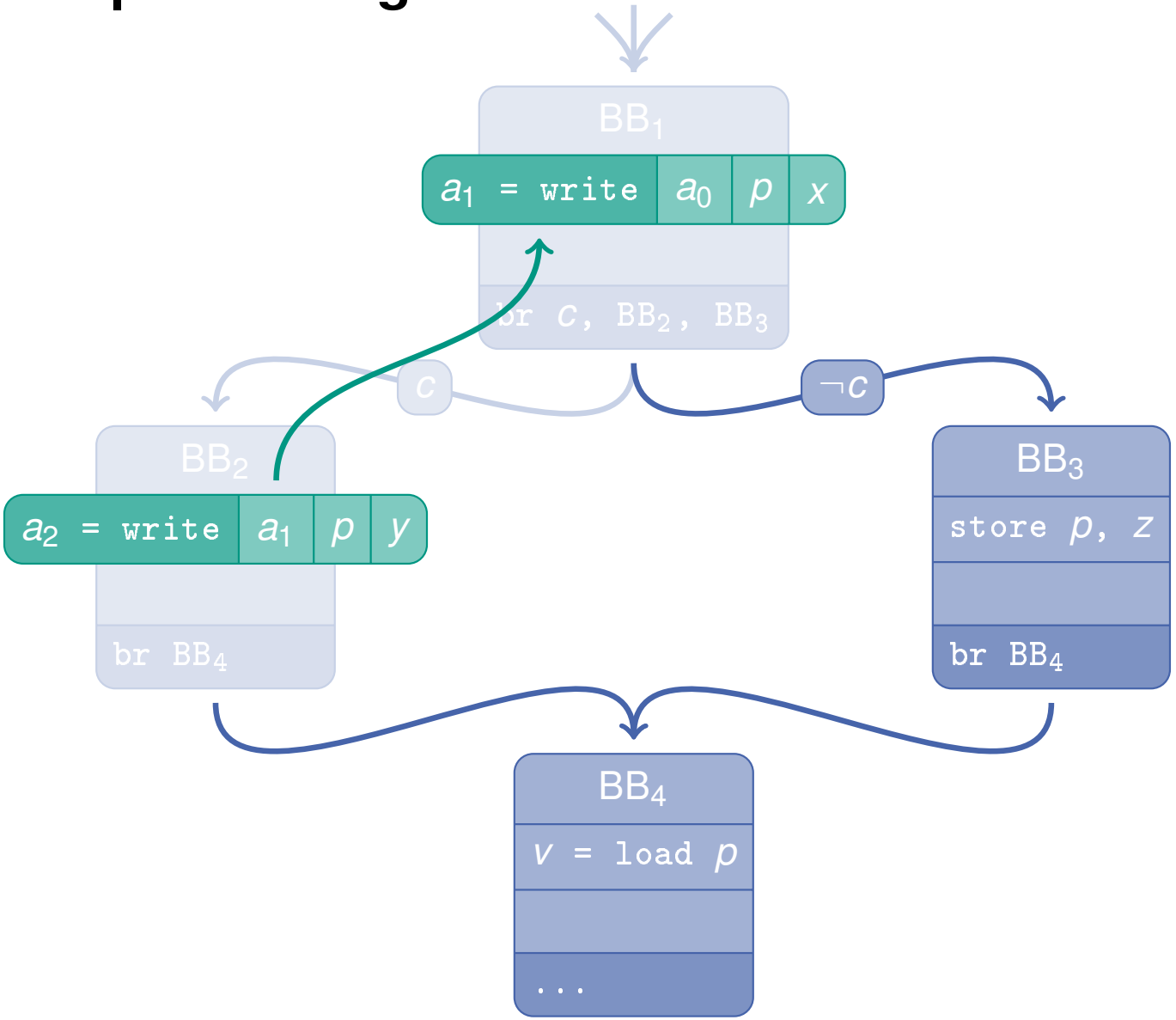
SSA für Speicherzugriffe



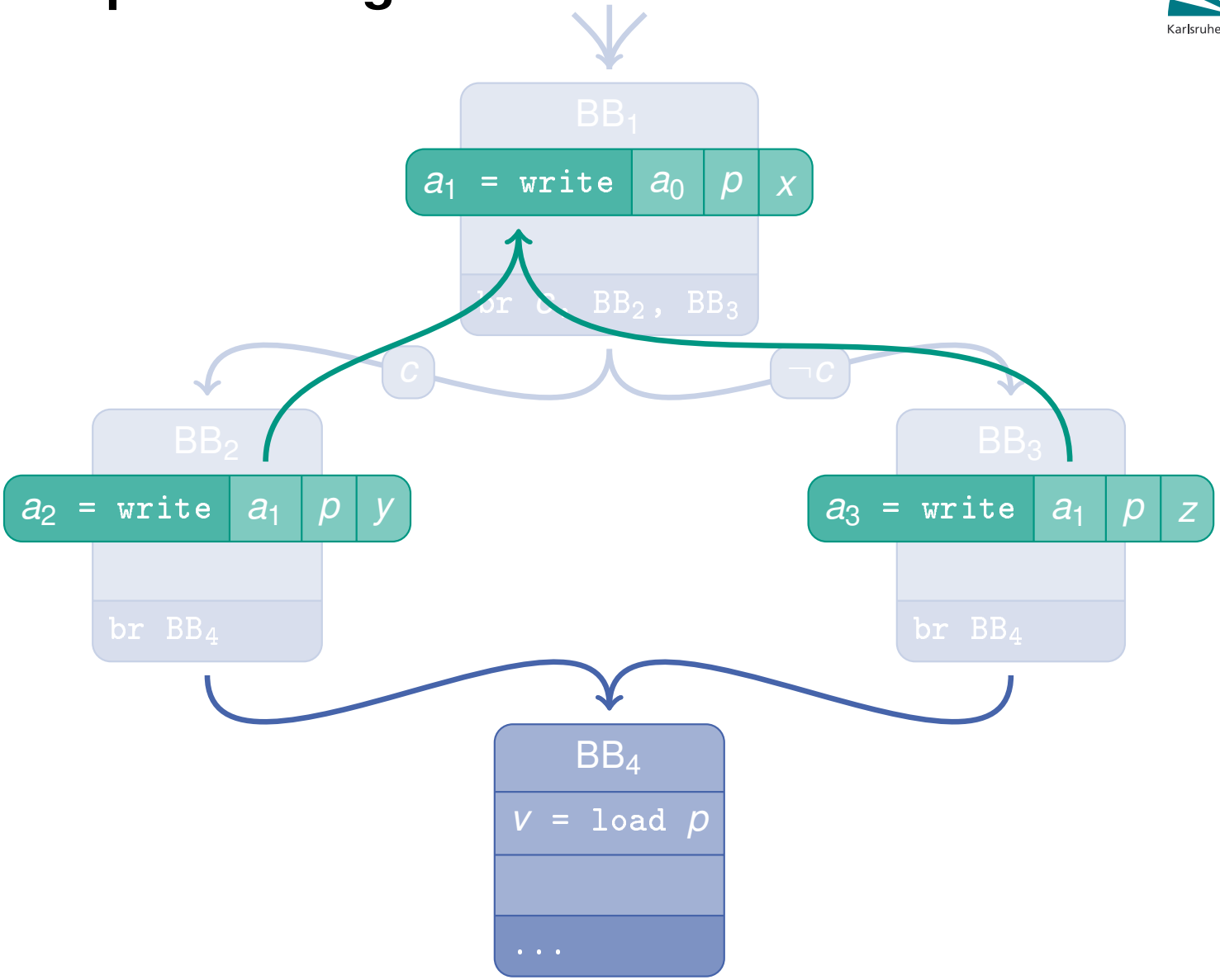
SSA für Speicherzugriffe



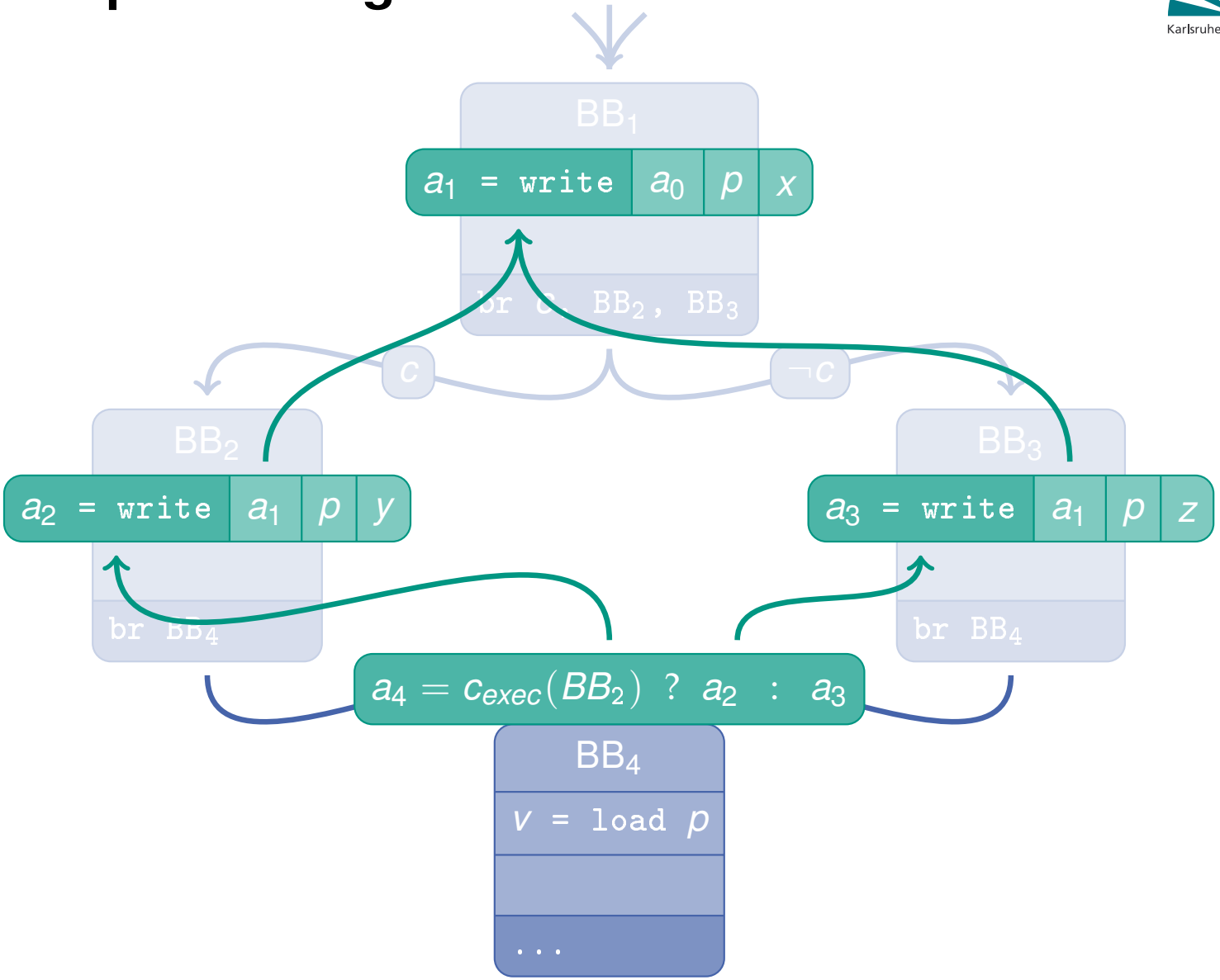
SSA für Speicherzugriffe



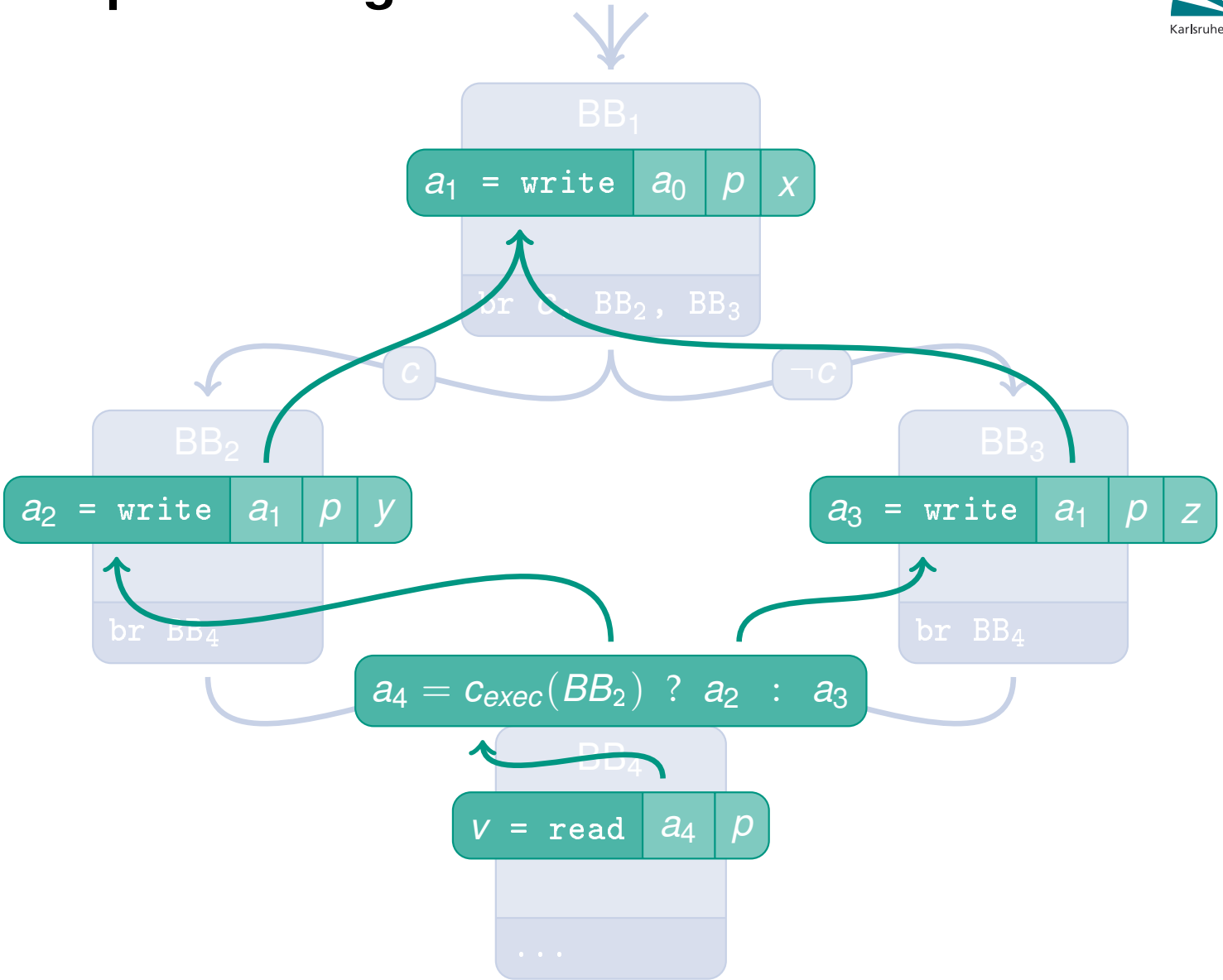
SSA für Speicherzugriffe



SSA für Speicherzugriffe



SSA für Speicherzugriffe



Formelerzeugung in LLBMC

Ziel: Zeige dass

$$c_{exec}(BB_{\text{err}})$$

unerfüllbar ist

Noch zu tun:

Kodiere die Eigenschaften P ☑

- Kodiere die Tests ☑
- Kodiere die Ausführunsbedingungen ☑

Kodiere die beschränkten Programmläufe C ☐

- innerhalb eines Basic Blocks ☑
- über Basic Blöcke hinweg ☑
- Speicherzugriffe ☑
- Speicherallokationen ☐

Theorie der Speicherallokationen 1

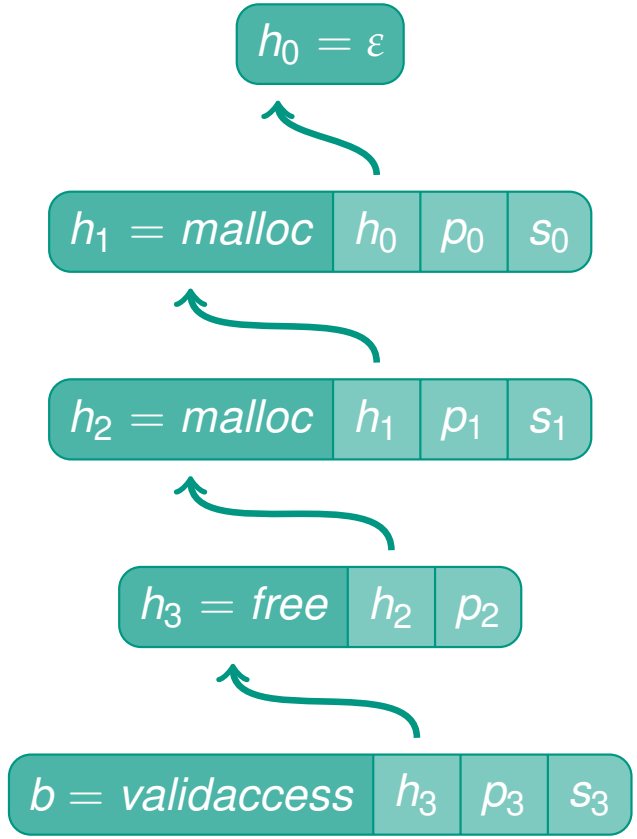
- Idee: Trenne Speicherinhalt von Speicherallokationen
- $\Longrightarrow$ Theorie der Speicherallokationen

$$\varepsilon : \rightarrow H$$
$$\mathit{malloc} : H \times I \times S \rightarrow H$$
$$\mathit{free} : H \times I \rightarrow H$$
$$\mathit{validaccess} : H \times I \times S \rightarrow \mathbb{B}$$
$$\mathit{validmalloc} : H \times I \times S \rightarrow \mathbb{B}$$
$$\mathit{validfree} : H \times I \rightarrow \mathbb{B}$$

H : Allokationszustand

I : Indexsorte

S : Größensorte



Theorie der Speicherallokationen 2

$validfree(h, q)$	Zeigt q auf den Anfang eines allozierten Speicherbereichs?

Theorie der Speicherallokationen 2

$validfree(h, q)$ Zeigt q auf den Anfang eines allozierten Speicherbereichs?

$$validfree(\varepsilon, q) \Leftrightarrow \bot$$

Theorie der Speicherallokationen 2

validfree(h, q) Zeigt q auf den Anfang eines allozierten Speicherbereichs?

$$\mathit{validfree}(\varepsilon, q) \Leftrightarrow \bot$$

$$\begin{aligned}
\mathit{validmalloc}(h, p, s) \ \wedge\ p = q \quad &\Longrightarrow \quad \mathit{validfree}(\mathit{malloc}(h, p, s), q) \Leftrightarrow \top \\
\neg\mathit{validmalloc}(h, p, s) \ \vee\ p \neq q \quad &\Longrightarrow \quad \mathit{validfree}(\mathit{malloc}(h, p, s), q) \Leftrightarrow \mathit{validfree}(h, q)
\end{aligned}$$

Theorie der Speicherallokationen 2

validfree(h, q) Zeigt q auf den Anfang eines allozierten Speicherbereichs?

$$\mathit{validfree}(\varepsilon, q) \Leftrightarrow \bot$$

$$\begin{aligned}
\mathit{validmalloc}(h, p, s) \ \wedge\ p = q \quad &\Longrightarrow \quad \mathit{validfree}(\mathit{malloc}(h, p, s), q) \Leftrightarrow \top \\
\neg\mathit{validmalloc}(h, p, s) \ \vee\ p \neq q \quad &\Longrightarrow \quad \mathit{validfree}(\mathit{malloc}(h, p, s), q) \Leftrightarrow \mathit{validfree}(h, q)
\end{aligned}$$

$$\begin{aligned}
\mathit{validfree}(h, p) \ \wedge\ p = q \quad &\Longrightarrow \quad \mathit{validfree}(\mathit{free}(h, p), q) \Leftrightarrow \bot \\
\neg\mathit{validfree}(h, p) \ \vee\ p \neq q \quad &\Longrightarrow \quad \mathit{validfree}(\mathit{free}(h, p), q) \Leftrightarrow \mathit{validfree}(h, q)
\end{aligned}$$

Theorie der Speicherallokationen 2

validfree(h, q) Zeigt q auf den Anfang eines allozierten Speicherbereichs?

$$\mathit{validfree}(\varepsilon, q) \Leftrightarrow \bot$$

$$\begin{aligned}
\mathit{validmalloc}(h, p, s) \ \wedge\ p = q &\implies \mathit{validfree}(\mathit{malloc}(h, p, s), q) \Leftrightarrow \top \\
\neg\mathit{validmalloc}(h, p, s) \ \vee\ p \neq q &\implies \mathit{validfree}(\mathit{malloc}(h, p, s), q) \Leftrightarrow \mathit{validfree}(h, q)
\end{aligned}$$

$$\begin{aligned}
\mathit{validfree}(h, p) \ \wedge\ p = q &\implies \mathit{validfree}(\mathit{free}(h, p), q) \Leftrightarrow \bot \\
\neg\mathit{validfree}(h, p) \ \vee\ p \neq q &\implies \mathit{validfree}(\mathit{free}(h, p), q) \Leftrightarrow \mathit{validfree}(h, q)
\end{aligned}$$

Axiome für *validaccess* und *validmalloc* haben ähnliche Struktur

Theorie der Speicherallokationen 3

- Die Theorie der Speicherallokationen ist entscheidbar (mit Bitvektoren oder ganzen Zahlen als Indexsorte)

Theorie der Speicherallokationen 3

- Die Theorie der Speicherallokationen ist entscheidbar (mit Bitvektoren oder ganzen Zahlen als Indexsorte)
- Einfaches Entscheidungsverfahren:

Theorie der Speicherallokationen 3

- Die Theorie der Speicherallokationen ist entscheidbar (mit Bitvektoren oder ganzen Zahlen als Indexsorte)
- Einfaches Entscheidungsverfahren:
  - Ähnlich zum quantorenfreien Fragment der Arraytheorie
  - Benutze die Axiome und führe eine Fallunterscheidung durch
  - Reduziert die Theorie der Speicherallokationen auf die Theorie der Indexsorte

Theorie der Speicherallokationen 3

- Die Theorie der Speicherallokationen ist entscheidbar (mit Bitvektoren oder ganzen Zahlen als Indexsorte)
- Einfaches Entscheidungsverfahren:
  - Ähnlich zum quantorenfreien Fragment der Arraytheorie
  - Benutze die Axiome und führe eine Fallunterscheidung durch
  - Reduziert die Theorie der Speicherallokationen auf die Theorie der Indexsorte
- “Besseres” Entscheidungsverfahren:

Theorie der Speicherallokationen 3

- Die Theorie der Speicherallokationen ist entscheidbar (mit Bitvektoren oder ganzen Zahlen als Indexsorte)
- Einfaches Entscheidungsverfahren:
  - Ähnlich zum quantorenfreien Fragment der Arraytheorie
  - Benutze die Axiome und führe eine Fallunterscheidung durch
  - Reduziert die Theorie der Speicherallokationen auf die Theorie der Indexsorte
- “Besseres” Entscheidungsverfahren:

?

Formelerzeugung in LLBMC

Ziel: Zeige dass

$$c_{exec}(BB_{\text{err}})$$

unerfüllbar ist

Noch zu tun:

Kodiere die Eigenschaften P ..... ☑

- Kodiere die Tests ..... ☑
- Kodiere die Ausführunsbedingungen ..... ☑

Kodiere die beschränkten Programmläufe C ..... ☑

- innerhalb eines Basic Blocks ..... ☑
- über Basic Blöcke hinweg ..... ☑
- Speicherzugriffe ..... ☑
- Speicherallokationen ..... ☑

Teil III

Symbolic Execution mit KLEE

Symbolic Execution – Idee 1

- Idee der Symbolic Execution:
  - Führe eine (symbolische) Pfaderkundung durch

Symbolic Execution – Idee 1

- Idee der Symbolic Execution:
  - Führe eine (symbolische) Pfaderkundung durch
    - Baue mehrere Formeln auf, jede Formel beschreibt genau einen Pfad

Symbolic Execution – Idee 1

- Idee der Symbolic Execution:
  - Führe eine (symbolische) Pfaderkundung durch
    - Baue mehrere Formeln auf, jede Formel beschreibt genau einen Pfad
    - Eine Pfadformel φ wird gemäß des Programs erweitert bis ein bedingter Sprung

      $$br\ c, BB_1, BB_2$$

      erreicht wird

Symbolic Execution – Idee 1

- Idee der Symbolic Execution:
  - Führe eine (symbolische) Pfaderkundung durch
    - Baue mehrere Formeln auf, jede Formel beschreibt genau einen Pfad
    - Eine Pfadformel φ wird gemäß des Programs erweitert bis ein bedingter Sprung

      $$br\ c, BB_1, BB_2$$

      erreicht wird
    - Falls $\varphi \wedge c$ erfülbar ist, so setze die Pfaderkundung mit $\varphi \wedge c$ in BB_1 fort

Symbolic Execution – Idee 1

- Idee der Symbolic Execution:
  - Führe eine (symbolische) Pfaderkundung durch
    - Baue mehrere Formeln auf, jede Formel beschreibt genau einen Pfad
    - Eine Pfadformel φ wird gemäß des Programs erweitert bis ein bedingter Sprung

      $$br\ c, BB_1, BB_2$$

      erreicht wird
    - Falls $\varphi \wedge c$ erfülbar ist, so setze die Pfaderkundung mit $\varphi \wedge c$ in BB_1 fort
    - Falls $\varphi \wedge \neg c$ erfülbar ist, so setze die Pfaderkundung mit $\varphi \wedge \neg c$ in BB_2 fort

Symbolic Execution – Idee 2

- Idee der Symbolic Execution (Fortsetzung):
  - Sobald eine Stelle erreicht wird an der Eigenschaft *P* geprüft werden soll

Symbolic Execution – Idee 2

- Idee der Symbolic Execution (Fortsetzung):
  - Sobald eine Stelle erreicht wird an der Eigenschaft P geprüft werden soll
    - Falls $\varphi \wedge \neg P$ erfüllbar ist, so führt eine Konkretisierung dieses Pfads zu einer Verletzung der Eigenschaft

Symbolic Execution – Idee 2

- Idee der Symbolic Execution (Fortsetzung):
  - Sobald eine Stelle erreicht wird an der Eigenschaft P geprüft werden soll
    - Falls $\varphi \wedge \neg P$ erfüllbar ist, so führt eine Konkretisierung dieses Pfads zu einer Verletzung der Eigenschaft
    - Setze die Pfaderkundung mit $\varphi \wedge P$ fort

Symbolic Execution – Idee 2

- Idee der Symbolic Execution (Fortsetzung):
  - Sobald eine Stelle erreicht wird an der Eigenschaft P geprüft werden soll
    - Falls $\varphi \wedge \neg P$ erfüllbar ist, so führt eine Konkretisierung dieses Pfads zu einer Verletzung der Eigenschaft
    - Setze die Pfaderkundung mit $\varphi \wedge P$ fort
- Pfadformeln können zur automatischen Erzeugung von Test Cases verwendet werden
  - Hohe Coverage

Symbolic Execution – Idee 2

- Idee der Symbolic Execution (Fortsetzung):
  - Sobald eine Stelle erreicht wird an der Eigenschaft P geprüft werden soll
    - Falls $\varphi \wedge \neg P$ erfüllbar ist, so führt eine Konkretisierung dieses Pfads zu einer Verletzung der Eigenschaft
    - Setze die Pfaderkundung mit $\varphi \wedge P$ fort
- Pfadformeln können zur automatischen Erzeugung von Test Cases verwendet werden
  - Hohe Coverage
- Abwesenheit von Fehlern kann i.A. nicht bewiesen werden

KLEE – Details

- Pfadformeln sind QF_ABV Formeln mit einfacher Boolescher Struktur (Konjunktion)

KLEE – Details

- Pfadformeln sind QF_ABV Formeln mit einfacher Boolescher Struktur (Konjunktion)
- KLEE kann nur STP als SMT-Solver ansteuern

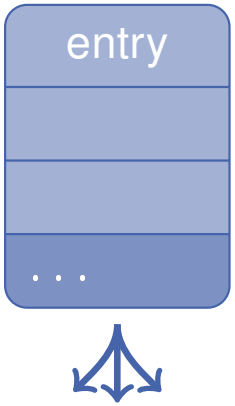
KLEE – Details

- Pfadformeln sind QF_ABV Formeln mit einfacher Boolescher Struktur (Konjunktion)
- KLEE kann nur STP als SMT-Solver ansteuern
- Verschiedene Heuristiken zur Auswahl einer Pfadformel von der aus die Pfaderkundung weiter geführt wird

KLEE – Details

- Pfadformeln sind QF_ABV Formeln mit einfacher Boolescher Struktur (Konjunktion)
- KLEE kann nur STP als SMT-Solver ansteuern
- Verschiedene Heuristiken zur Auswahl einer Pfadformel von der aus die Pfaderkundung weiter geführt wird
- KLEE kann Dateien modellieren
  - `open`
  - `read`
  - `write`
  - `stat`
  - `lseek`
  - `ftruncate`
  - `ioctl`
  - `...`

Pfadformel – `entry` Block

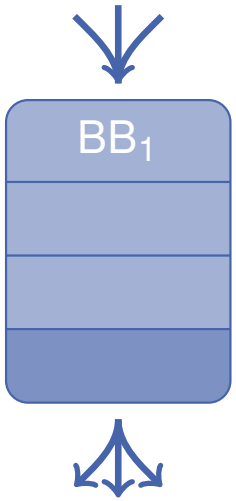


Erzeuge

$$\langle \mathit{true}, \mathit{entry}, \uparrow \rangle$$

Pfadformel: *true*
Basic Block: *entry*
Position im Basic Block: Anfang

Pfadformel – Basic Blöcke



Ersetze

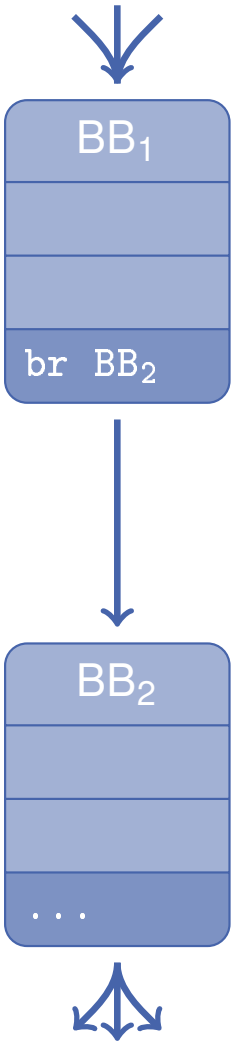
$$\langle \varphi, BB_1, \uparrow \rangle$$

durch

$$\langle \varphi \wedge \psi, BB_1, \downarrow \rangle$$

ψ kodiert die Instruktionen in BB_1
(außer Φ-Instruktionen und Terminatoren)

Pfadformel – Unbedingte Sprünge



Ersetze

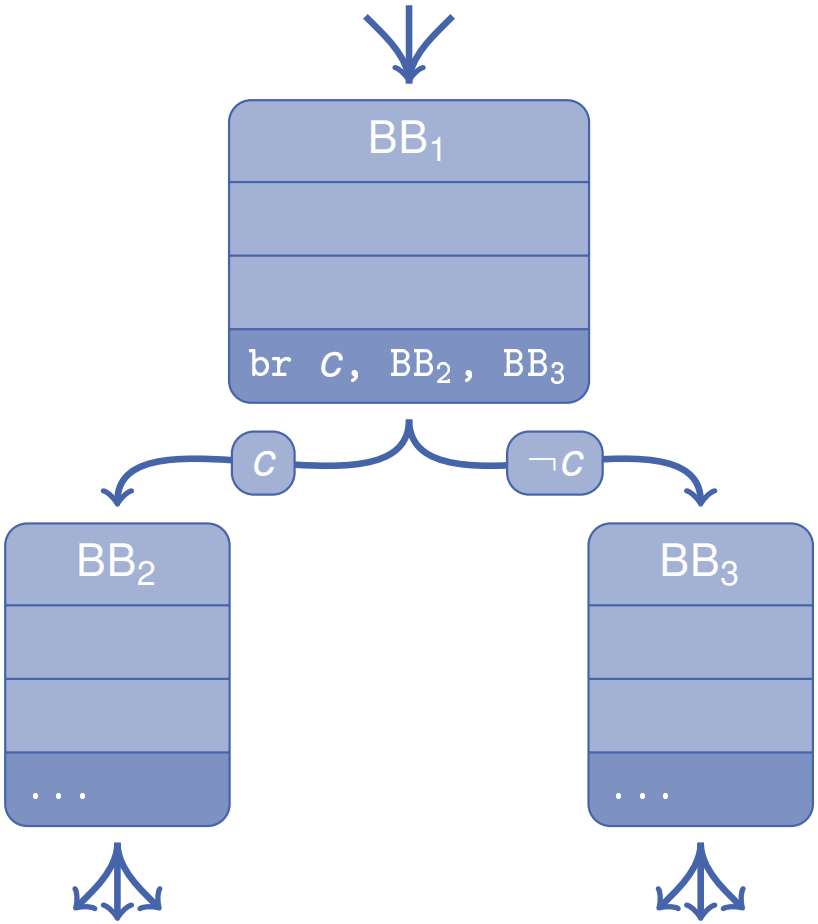
$$\langle \varphi, BB_1, \downarrow \rangle$$

durch

$$\langle \varphi \wedge \psi, BB_2, \uparrow \rangle$$

ψ kodiert Φ-Instruktionen in BB_2 indem es die passenden Werte auswählt

Pfadformel – Bedingte Sprünge



Ersetze

$$\langle \varphi, BB_1, \downarrow \rangle$$

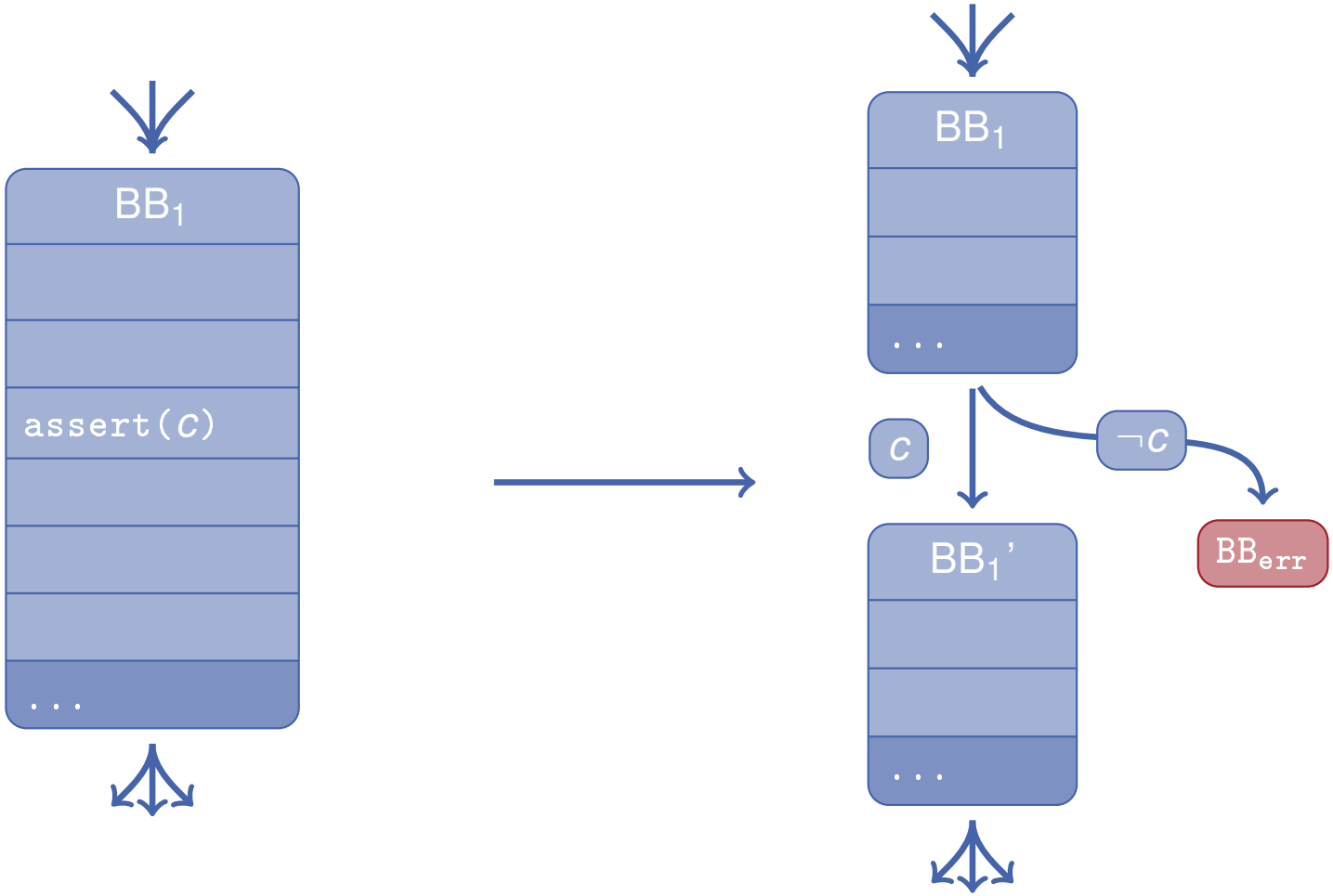
durch die Tupel aus

$$\langle \varphi \wedge c \wedge \psi_1, BB_2, \uparrow \rangle$$
$$\langle \varphi \wedge \neg c \wedge \psi_2, BB_3, \uparrow \rangle$$

für die die Pfadformel erfüllbar ist

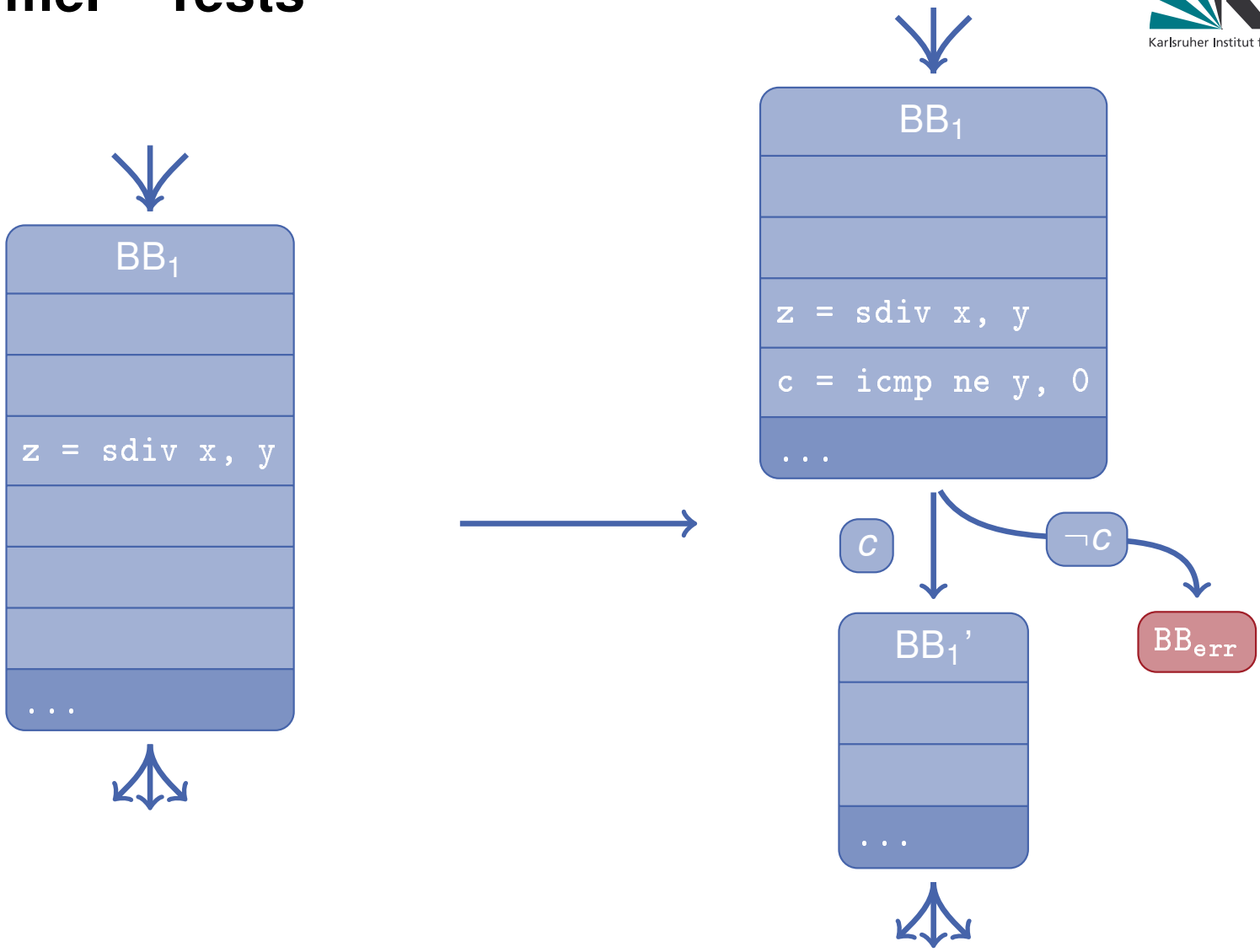
ψ_1 und ψ_2 kodieren Φ-Instruktionen in BB_2 bzw. BB_3 indem es die passenden Werte auswählt

Pfadformel – Assertions



Falls Pfadformel für Sprung nach BB_{err} erfüllbar ist melde Fehler

Pfadformel – Tests

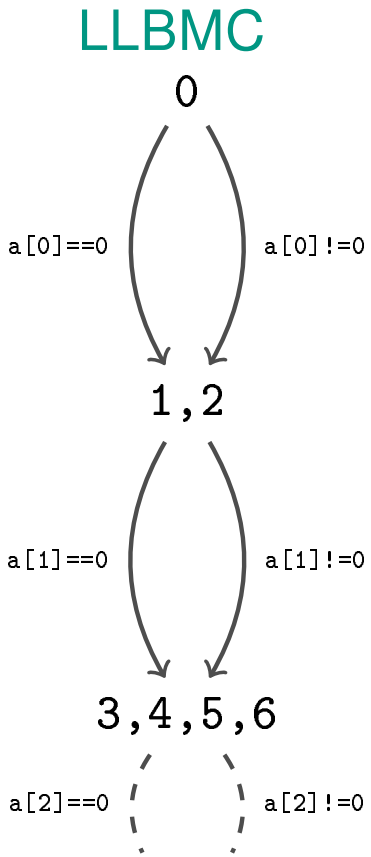


Falls Pfadformel für Sprung nach BB_{err} erfüllbar ist melde Fehler

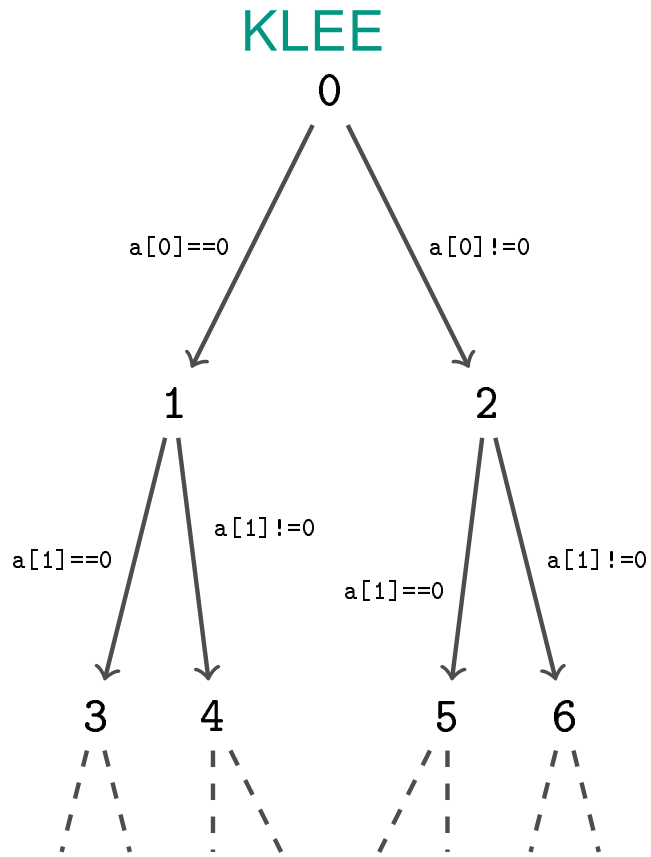
Gegenüberstellung LLBMC und KLEE

Programm

```
int a[32];
for (int i=0; i!=32; ++i)
{
   if (a[i]==0) {
      // then
   } else {
      // else
   }
}
```



Eine Formel



2^{32} Formeln