BLAST

David Dueck

12.07.2007

Was ist BLAST?

- Berkeley Lazy Abstraction Software Verification Tool
- statische Prüfung von “Sicherheitseigenschaften”

Welches Problem löst BLAST?

Einfachste Form dieser Sicherheitseigenschaften Assertions:

- Laufzeitchecks

Welches Problem löst BLAST?

Einfachste Form dieser Sicherheitseigenschaften Assertions:

- Laufzeitchecks
- Programmierer fügt *assert(e)* in den Code ein

Welches Problem löst BLAST?

Einfachste Form dieser Sicherheitseigenschaften Assertions:

- Laufzeitchecks
- Programmierer fügt *assert(e)* in den Code ein
- Wenn e zur Laufzeit zu 0 auswertet $\Rightarrow$ Programmabbruch

Welches Problem löst BLAST?

Einfachste Form dieser Sicherheitseigenschaften Assertions:

- Laufzeitchecks
- Programmierer fügt *assert(e)* in den Code ein
- Wenn e zur Laufzeit zu 0 auswertet $\Rightarrow$ Programmabbruch
- Keine Fehlerabfrage bzw. Fehlerbehandlung. Spezifizieren Invarianten

Welches Problem löst BLAST?

Einfachste Form dieser Sicherheitseigenschaften Assertions:

- Laufzeitchecks
- Programmierer fügt *assert(e)* in den Code ein
- Wenn e zur Laufzeit zu 0 auswertet $\Rightarrow$ Programmabbruch
- Keine Fehlerabfrage bzw. Fehlerbehandlung. Spezifizieren Invarianten
- Dürfen keine Seiteneffekte enthalten

Welches Problem löst BLAST?

Einfachste Form dieser Sicherheitseigenschaften Assertions:

- Laufzeitchecks
- Programmierer fügt *assert(e)* in den Code ein
- Wenn e zur Laufzeit zu 0 auswertet $\Rightarrow$ Programmabbruch
- Keine Fehlerabfrage bzw. Fehlerbehandlung. Spezifizieren Invarianten
- Dürfen keine Seiteneffekte enthalten
- Meist benutzt während der Entwicklung, entfernt vor der Auslieferung

Welches Problem löst BLAST?

Einfachste Form dieser Sicherheitseigenschaften Assertions:

- Laufzeitchecks
- Programmierer fügt *assert(e)* in den Code ein
- Wenn e zur Laufzeit zu 0 auswertet $\Rightarrow$ Programmabbruch
- Keine Fehlerabfrage bzw. Fehlerbehandlung. Spezifizieren Invarianten
- Dürfen keine Seiteneffekte enthalten
- Meist benutzt während der Entwicklung, entfernt vor der Auslieferung
- (Im Grunde: Erreichbarkeit eines Labels)

Welches Problem löst BLAST?

Probleme mit Assertions:

- Bei Auslieferung mit Assertions: Laufzeiteinbußen

Welches Problem löst BLAST?

Probleme mit Assertions:

- Bei Auslieferung mit Assertions: Laufzeiteinbußen
- Bei Entfernung: keine Sicherheit, Fehlschlag bedeutet im schlimmsten Fall undefined behaviour

Welches Problem löst BLAST?

Probleme mit Assertions:

- Bei Auslieferung mit Assertions: Laufzeiteinbußen
- Bei Entfernung: keine Sicherheit, Fehlschlag bedeutet im schlimmsten Fall undefined behaviour
- In jedem Fall: späte Fehlermeldung

Welches Problem löst BLAST?

BLAST versucht **statisch** d.h. zur **compile time** zu beweisen, dass keine Assertion fehlschlägt.
3 Möglichkeiten: Entweder BLAST

- beweist Assertion schlägt nie fehl

Welches Problem löst BLAST?

BLAST versucht **statisch** d.h. zur **compile time** zu beweisen, dass keine Assertion fehlschlägt.
3 Möglichkeiten: Entweder BLAST

- beweist Assertion schlägt nie fehl
- gibt einen Pfad durch das Programm an, bei dem die Assertion fehlschlägt

Welches Problem löst BLAST?

BLAST versucht **statisch** d.h. zur **compile time** zu beweisen, dass keine Assertion fehlschlägt.
3 Möglichkeiten: Entweder BLAST

- beweist Assertion schlägt nie fehl
- gibt einen Pfad durch das Programm an, bei dem die Assertion fehlschlägt
- überschreitet die Zeitschranken

Welches Problem löst BLAST?

BLAST versucht **statisch** d.h. zur **compile time** zu beweisen, dass keine Assertion fehlschlägt.
3 Möglichkeiten: Entweder BLAST

- beweist Assertion schlägt nie fehl
- gibt einen Pfad durch das Programm an, bei dem die Assertion fehlschlägt
- überschreitet die Zeitschranken

Ziel: Sicherheit bzgl. der Assertions. Kann bedeuten dass Laufzeitchecks eingefügt werden müssen die nie fehlschlagen.

Welches Problem löst BLAST?

BLAST löst/verbessert alle genannten Probleme

- Minimierte Laufzeiteinbußen, da viele Assertions entfernt werden können

Welches Problem löst BLAST?

BLAST löst/verbessert alle genannten Probleme

- Minimierte Laufzeiteinbußen, da viele Assertions entfernt werden können
- Viele Bugs können zur compiletime gefunden werden

Welches Problem löst BLAST?

BLAST löst/verbessert alle genannten Probleme

- Minimierte Laufzeiteinbußen, da viele Assertions entfernt werden können
- Viele Bugs können zur compiletime gefunden werden
- Im Timeoutfall kann der Programmierer einzelne Programmstellen untersuchen

Welches Problem löst BLAST?

BLAST löst/verbessert alle genannten Probleme

- Minimierte Laufzeiteinbußen, da viele Assertions entfernt werden können
- Viele Bugs können zur compiletime gefunden werden
- Im Timeoutfall kann der Programmierer einzelne Programmstellen untersuchen
- Rest der Assertions muss bleiben

Weitere Möglichkeiten “Sicherheitseigenschaften” zu spezifizieren und zu beweisen

Man möchte “Sicherheitseigenschaften” auf einer höheren Abstraktions Ebene spezifizieren, ohne den Code umfangreich zu ändern bzw. überall Assertions einzufügen

- BLAST Specification Language
- Externe Programme, beispielsweise CCured

BLAST Specification Language

Angenommen wir haben ein Programm, das ein Lock benutzt. Wir wollen sicherstellen, dass Aufrufe zu Lock und Unlock sich abwechseln. Funktionen:

- FSMInit()
- FSMLock()
- FSMUnLock()

BLAST Specification Language

Beispiel Spezifikation:

```
global int lockStatus = 0;

event {
  pattern { FSMInit(); }
  action { lockStatus = 0; }
}

event {
  pattern { FSMLock(); }
  guard { lockStatus == 0 }
  action { lockStatus = 1; }
}
```

BLAST Specification Language

```
event {
   pattern { FSMUnLock(); }
   guard { lockStatus == 1 }
   action { lockStatus = 0; }
}
```

BLAST Specification Language

lockStatus: observer variable

- Bei erreichen des pattern wird das event ausgelöst
- Die guard Bedingung muss erfüllt sein
- action wird beim Einhalten der guard Bedingung ausgeführt

BLAST Specification Language

- Spezifikation kann auf den Assertion Fall zurückgeführt werden
- $\Rightarrow$ BLAST kann Einhaltung der Spezifikation prüfen

CCured

Legt für Zeiger eine Ebene über das C Typsystem

- Teilt Zeiger in verschiedene Klassen ein
- Je nach Zeigerklasse fügt es möglichst wenig Assertions in das Programm ein
- BLAST kann viele der Checks entfernen

Funktionsweise

Überblick:

- statische Erreichbarkeitsanalyse

Funktionsweise

Überblick:

- statische Erreichbarkeitsanalyse
- beobachtet relevante Fakten (Prädikate) über einzelne Variablen statt des gesamten Programmzustandes (Abstraction)

Funktionsweise

Überblick:

- statische Erreichbarkeitsanalyse
- beobachtet relevante Fakten (Prädikate) über einzelne Variablen statt des gesamten Programmzustandes (Abstraction)
- Unterscheidung zwischen *spurious* und *genuine Counterexample*

Funktionsweise

Überblick:

- statische Erreichbarkeitsanalyse
- beobachtet relevante Fakten (Prädikate) über einzelne Variablen statt des gesamten Programmzustandes (Abstraction)
- Unterscheidung zwischen *spurious* und *genuine Counterexample*
- Verfeinerung solange bis ein Bug gefunden wird oder bewiesen werden kann, dass das Error Label nie erreicht wird

Funktionsweise

Überblick:

- statische Erreichbarkeitsanalyse
- beobachtet relevante Fakten (Prädikate) über einzelne Variablen statt des gesamten Programmzustandes (Abstraction)
- Unterscheidung zwischen *spurious* und *genuine Counterexample*
- Verfeinerung solange bis ein Bug gefunden wird oder bewiesen werden kann, dass das Error Label nie erreicht wird
- Verfeinerung geschieht durch Interpolantenbildung (Craig'sche Interpolanten bekannt aus Formale Systeme)

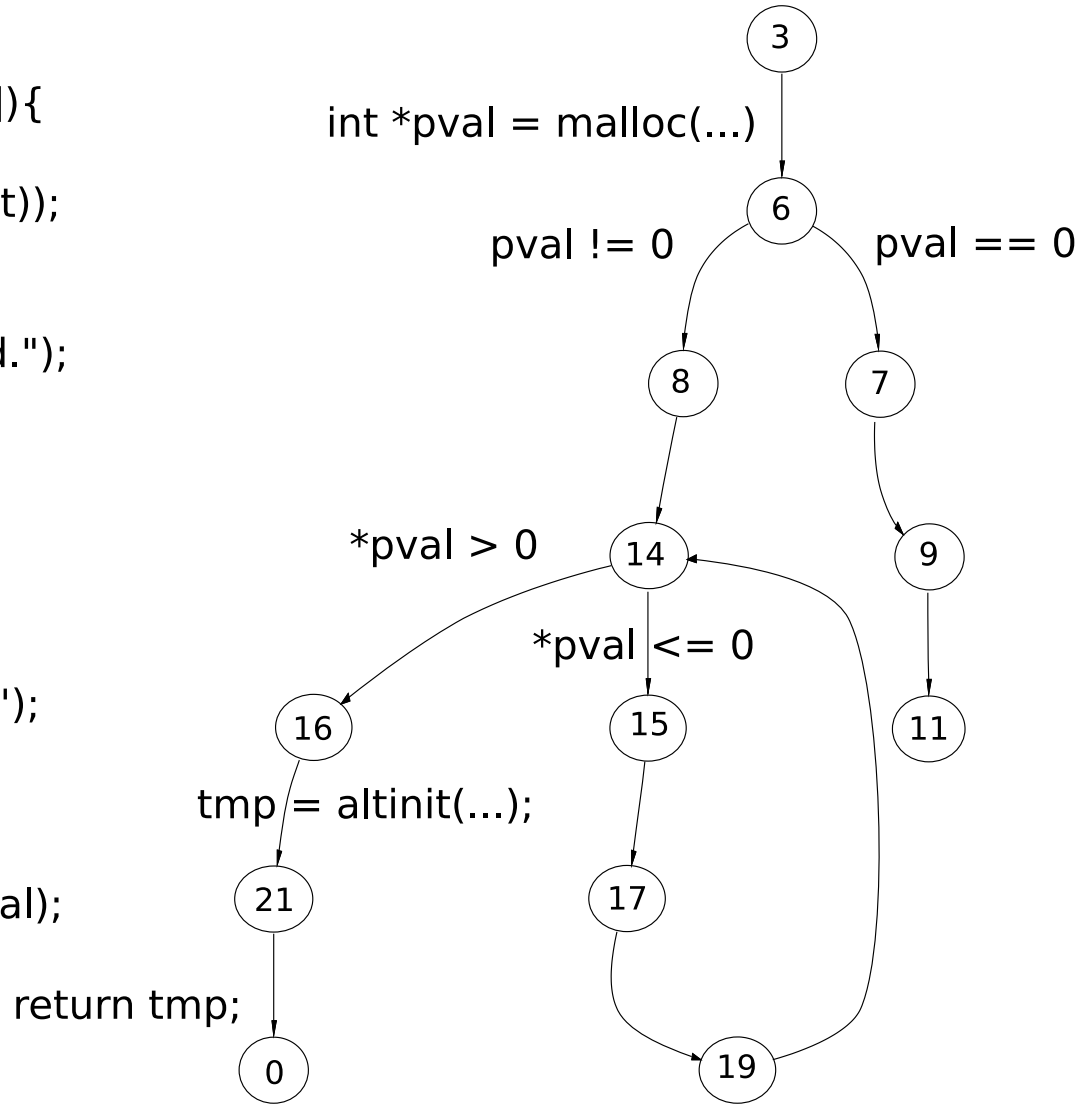
Interne Programmrepräsentation

- Für jede Funktion des Programms wird ein Kontrollfluss Automat erzeugt
- gerichteter Graph
- Kanten repräsentieren den Kontrollfluss
- Knoten sind die Kontrollpunkte im Programm (“Programm Counter Werte”)

Interne Programmrepräsentation

- Kanten sind mit der zugehörigen “Instruktion” markiert
- Instruktionen sind Grundblöcke, Funktionsaufrufe, “Annahmen” oder ein Return
- Funktionen hier Call-by-Value

```
int main(int argc, char *argv []){

 1: int *pval = malloc(sizeof(int));

 2: if (pval == 0) {
 3:   printf("Memory exhausted.");
 4:   exit(1);
}

5: *pval = 0;

6: while(*pval <= 0) {
7:  printf("Give a number > 0:");
8:  myscanf("%d", pval);
}

9: return altInit(*pval, pval, pval);
}
```



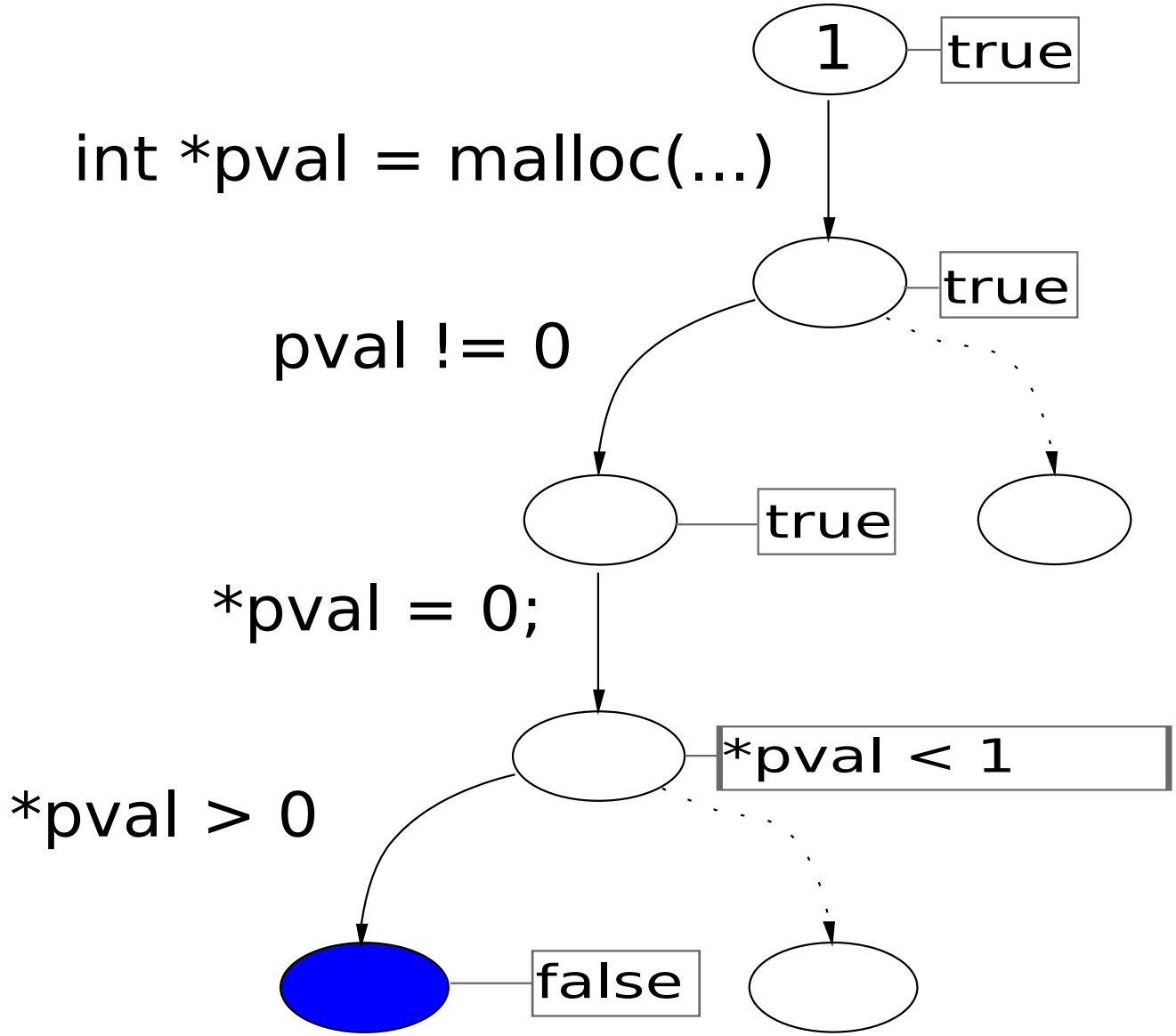
Abstract Reachability Trees

- Um Unereichbarkeit von Labels zu beweisen wird ein ART konstruiert
- Ein Pfad im ART repräsentiert eine Menge von Programmausführungen
- Knoten gehören zu einem Knoten in einem CFA
- Kanten sind markiert wie in den CFAs, können aber nun auch von einem CFA in einen anderen führen

Abstract Reachability Trees

Jeder Knoten n ist markiert mit (q, s, φ):

- einem Zugehörigen Knoten eines CFA (q)
- dem aktuellen Call Stack (s)
- der *reachable region*, die Menge der möglichen Datenzustände in diesem Kontrollschritt, spezifiziert durch eine prädikatenlogische Formel (φ) ohne Quantoren


1
true
int *pval = malloc(...)
true
pval != 0
true
*pval = 0;
*pval < 1
*pval > 0
false

Abstract Reachability Trees

- Gegeben: eine Menge von Programmzuständen φ, eine Operation op vom Typ Basisblock oder "Annahme":
  Sei $post(\varphi, \text{op})$ die Menge der erreichbaren Datenzustände nach Ausführung von op

Abstract Reachability Trees

- Gegeben: eine Menge von Programmzuständen φ, eine Operation op vom Typ Basisblock oder "Annahme":
  Sei *post*(φ, op) die Menge der erreichbaren Datenzustände nach Ausführung von op
- Für Operationen op vom Typ Funktionsaufruf:
  Sei *post*(φ, op) die Menge der erreichbaren Datenzustände nach Zuweisung des Arguments an den Parameter

Abstract Reachability Trees

- Gegeben: eine Menge von Programmzuständen φ, eine Operation op vom Typ Basisblock oder "Annahme":
  Sei *post*(φ, op) die Menge der erreichbaren Datenzustände nach Ausführung von op
- Für Operationen op vom Typ Funktionsaufruf:
  Sei *post*(φ, op) die Menge der erreichbaren Datenzustände nach Zuweisung des Arguments an den Parameter
- Für return Operationen op und eine Variable x:
  Sei *post*(φ, op, x) die Menge der erreichbaren Datenzustände nach Zuweisung des return Wertes an x

Abstract Reachability Trees

- Für zwei *reachable regions* φ, φ' gilt $\varphi \subseteq \varphi$' genau dann, wenn für jeden Datenzustand x für den φ(x) gilt auch φ'(x) gilt. Sprich jeder Datenzustand der von φ erreichbar ist, ist auch von φ' erreichbar.

Abstract Reachability Trees

- Für zwei *reachable regions* φ, φ' gilt $\varphi \subseteq \varphi$' genau dann, wenn für jeden Datenzustand x für den φ(x) gilt auch φ'(x) gilt. Sprich jeder Datenzustand der von φ erreichbar ist, ist auch von φ' erreichbar.
- Ein Knoten n:(q, s, φ) heisst covered durch einen innernen Knoten n':(q, s, φ') wenn $\varphi \subseteq \varphi$'.

Abstract Reachability Trees

Ein ART ist abgeschlossen unter Nachbedingungen wenn für jeden inneren Knoten n:(q, s, φ) und jede Vorwärts-Kante q$\overset{op}{\longmapsto}$q' im CFA von q gilt:

Abstract Reachability Trees

Ein ART ist abgeschlossen unter Nachbedingungen wenn für jeden inneren Knoten n:(q, s, φ) und jede Vorwärts-Kante q$\overset{op}{\longmapsto}$q' im CFA von q gilt:

1. Fall: op ist ein Basisblock oder eine "Annahme"
   - Es muss gelten: n hat einen Nachfolger Knoten n':(q', s, φ'), sodass n$\overset{op}{\longmapsto}$n' und post($\varphi$, op) $\subseteq \varphi$'

Abstract Reachability Trees

Ein ART ist abgeschlossen unter Nachbedingungen wenn für jeden inneren Knoten n:(q, s, φ) und jede Vorwärts-Kante q$\overset{op}{\longmapsto}$q' im CFA von q gilt:

1. Fall: op ist ein Basisblock oder eine "Annahme"
   - Es muss gelten: n hat einen Nachfolger Knoten n':(q', s, φ'), sodass n$\overset{op}{\longmapsto}$n' und post($\varphi$, op) $\subseteq$ φ'
2. Fall: op ist ein Funktionsaufruf der Form x = f(argumte)
   - Es muss gelten: n hat einen Nachfolger Knoten n':(q", s', φ'), sodass n $\overset{op}{\longmapsto}$ n', q" der Startknoten des CFA von f, s' ist push(s, (return adresse, x)) und post(φ, op) $\subseteq$ φ'

Abstract Reachability Trees

Ein ART ist abgeschlossen unter Nachbedingungen wenn für jeden inneren Knoten n:(q, s, φ) und jede Vorwärts-Kante q$\overset{op}{\longmapsto}$q' im CFA von q gilt:

3. Fall: op ist eine return Instruktion

- Es muss gelten: n hat einen Nachfolger Knoten n':(q'', s', φ'), sodass n$\overset{op}{\longmapsto}$n', (q'', x) ist das erste Element auf s, s' = pop(s) und post(φ, op, x) $\subseteq$ φ'

Abstract Reachability Trees

Ein ART ist vollständig wenn:

- 1. Die Wurzel ist mit dem Anfangszustand des Programmes markiert
- 2. Der Baum ist abgeschlossen unter Nachbedingungen (post)
- 3. Für jedes Blatt n:(q, s, φ) gilt: Entweder q hat keine ausgehende Kante, $\varphi = \emptyset$ oder es existiert ein n', sodass gilt: n ist covered durch n'

Abstract Reachability Trees

- Ein vollständiger ART approximiert die Menge der Erreichbaren Datenzustände.
- Ein vollständiger ART ist sicher bzgl. eines CFA Knotens q genau dann wenn für jeden Knoten n:(q, - , φ) des ART $\varphi = \emptyset$ gilt.
- Ein bzgl. q sicherer ART ist ein Zertifikat/Beweis, dass q auf keinem Ausführungspfad erreicht wird.

ART Konstruktion

- Ausrollen der CFAs, anfangen bei *main*

ART Konstruktion

- Ausrollen der CFAs, anfangen bei *main*
- Zuerst alle *reachable regions* zu $\varphi(x) = \text{true}$ gesetzt

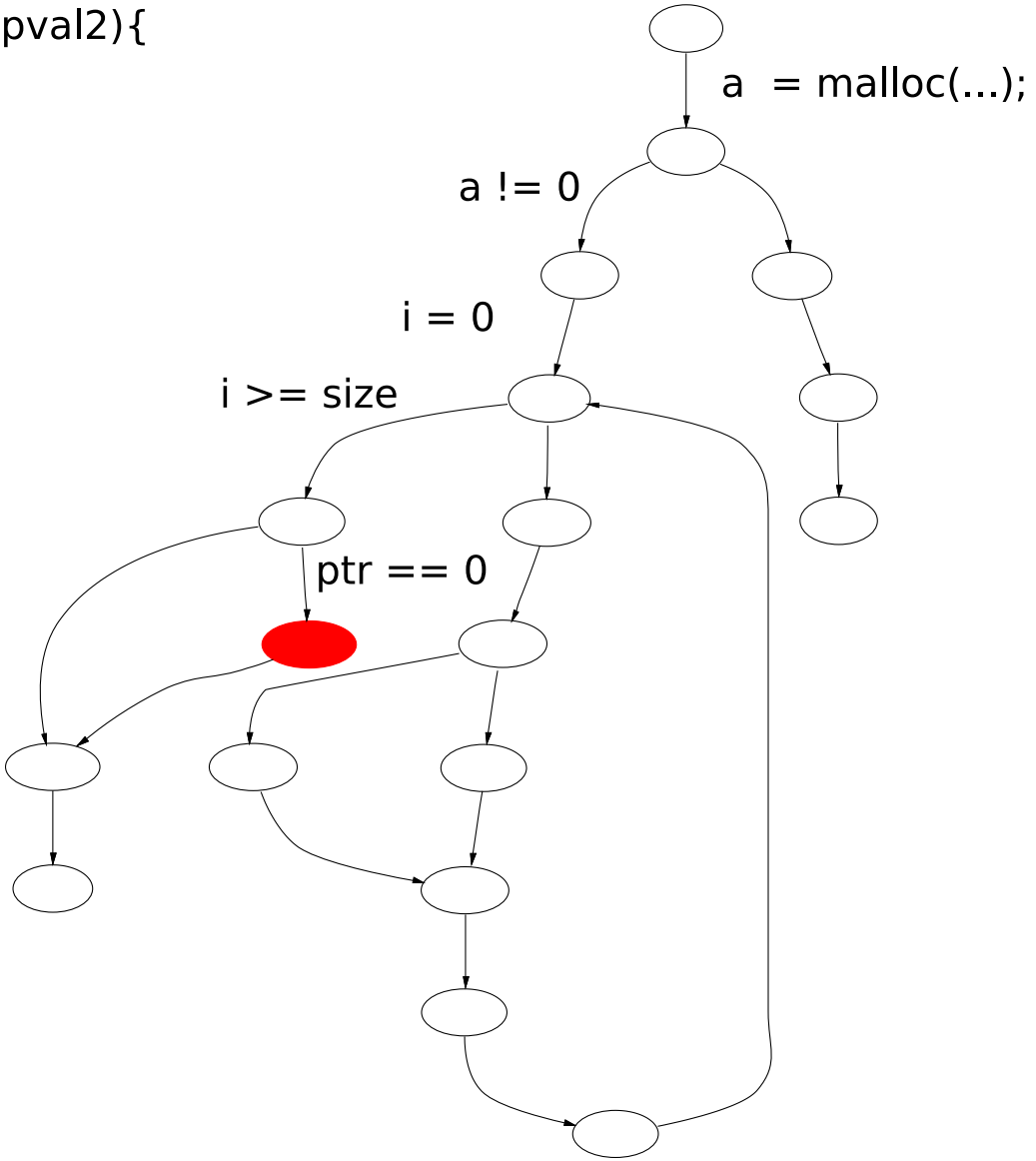
ART Konstruktion

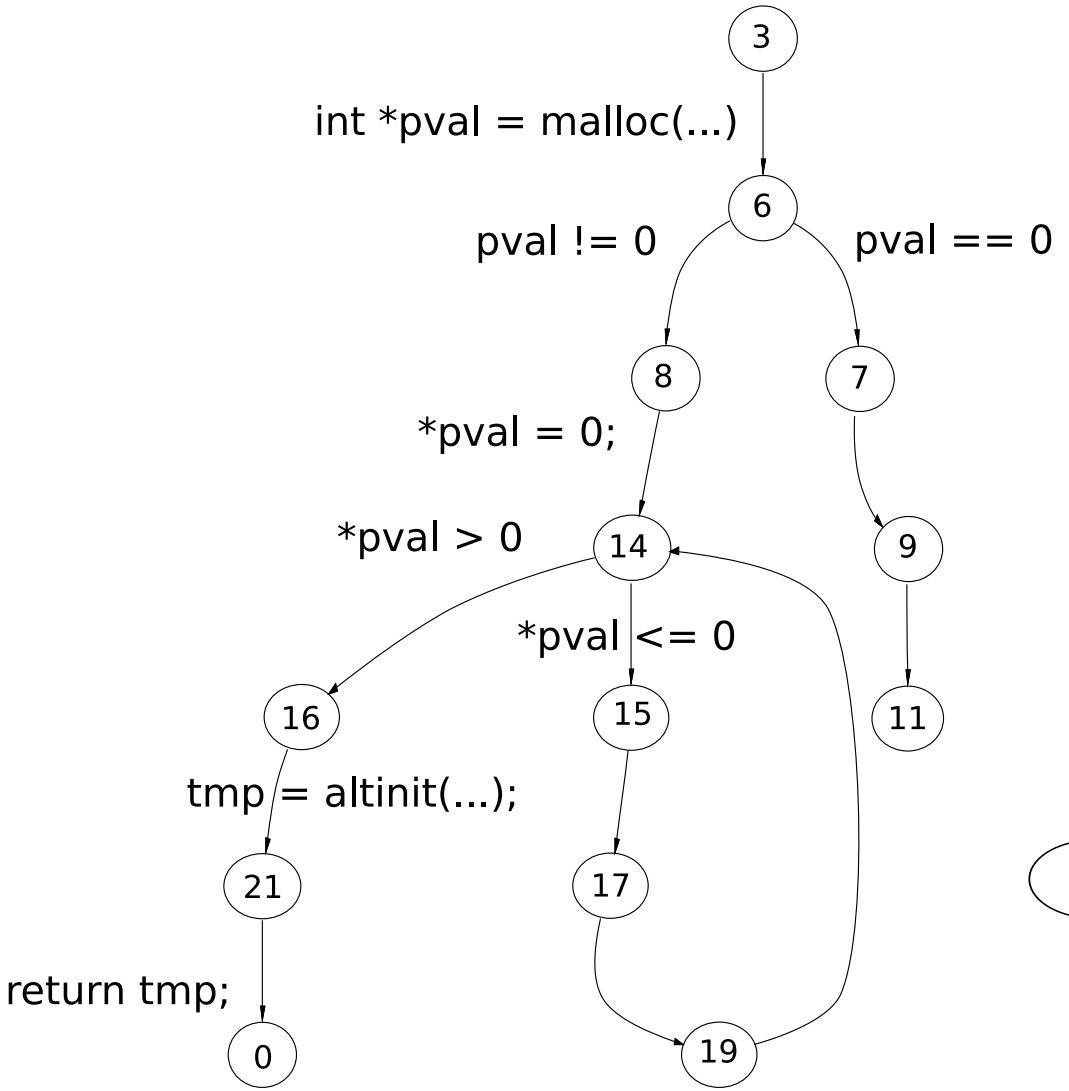
- Ausrollen der CFAs, anfangen bei *main*
- Zuerst alle *reachable regions* zu $\varphi(\mathrm{x}) = \mathrm{true}$ gesetzt
- Solange ausrollen, bis ein Fehlerlabel erreicht wurde oder der ART vollständig ist

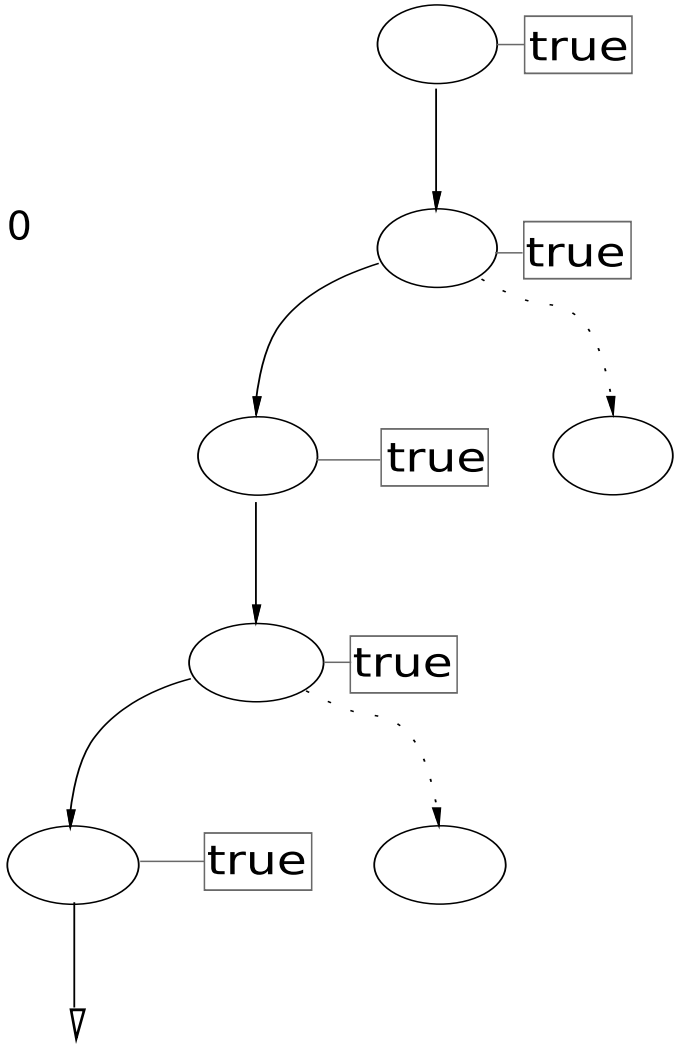
ART Konstruktion

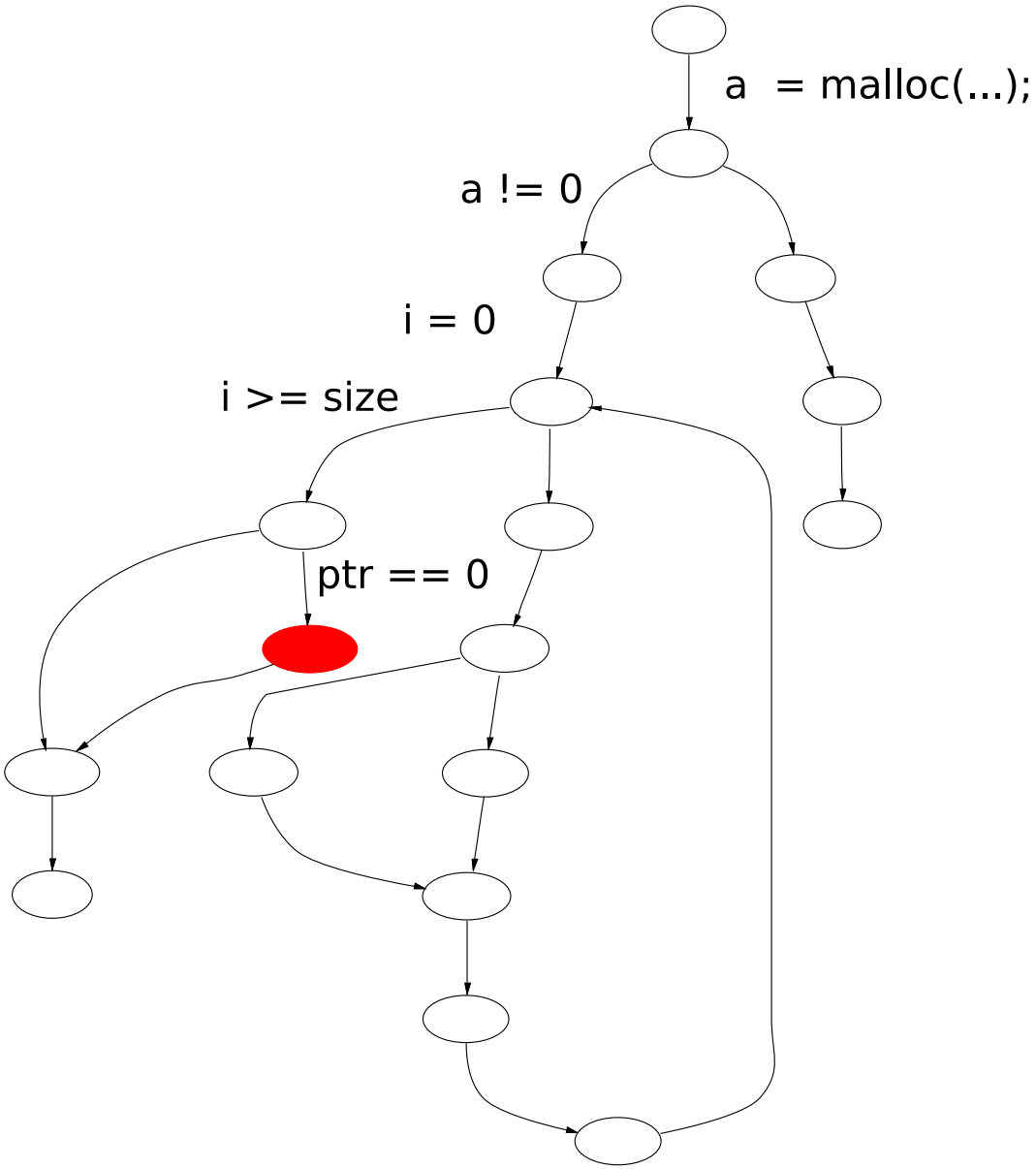
- Ausrollen der CFAs, anfangen bei *main*
- Zuerst alle *reachable regions* zu $\varphi(x) = \text{true}$ gesetzt
- Solange ausrollen, bis ein Fehlerlabel erreicht wurde oder der ART vollständig ist
- Wenn ein Fehlerlabel erreicht wird: *Counterexample Analysis*

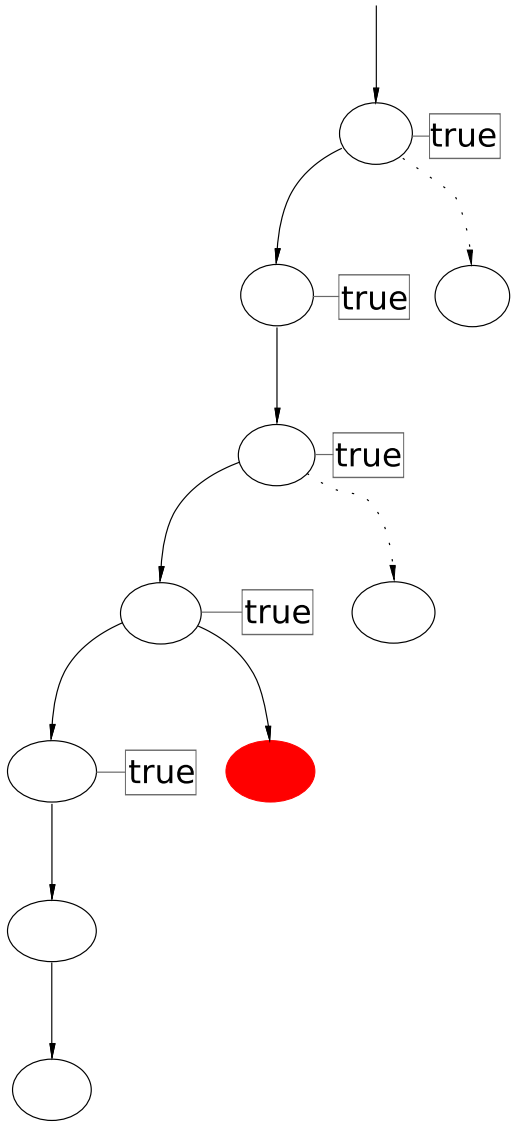
```
int altInit(int size, int *pval1, int *pval2){
1: int i, *ptr;
2: a = malloc(sizeof(int) * size);
3: if (a == 0) {
4:   printf("Memory exhausted.");
5:   exit(1);
6: }
7: i = 0;
8: while(i < size) {
9:    i = i + 1;

10:  if (i % 2 == 0) {
11:  ptr = pval1;
12: } else {
13:  ptr = pval2;
14: }

15: a[i] = *ptr;
16: printf("%d. iteration", i);
17: }  // end while

18: if (ptr == 0) ERR: ;
19: return *ptr;
}
```




3
int *pval = malloc(...)
6
pval != 0
pval == 0
8
7
*pval = 0;
*pval > 0
14
9
*pval <= 0
16
15
11
tmp = altinit(...);
21
17
return tmp;
0
19


true
true
true
true
true


a = malloc(...);
a != 0
i = 0
i >= size
ptr == 0


true
true
true
true
true

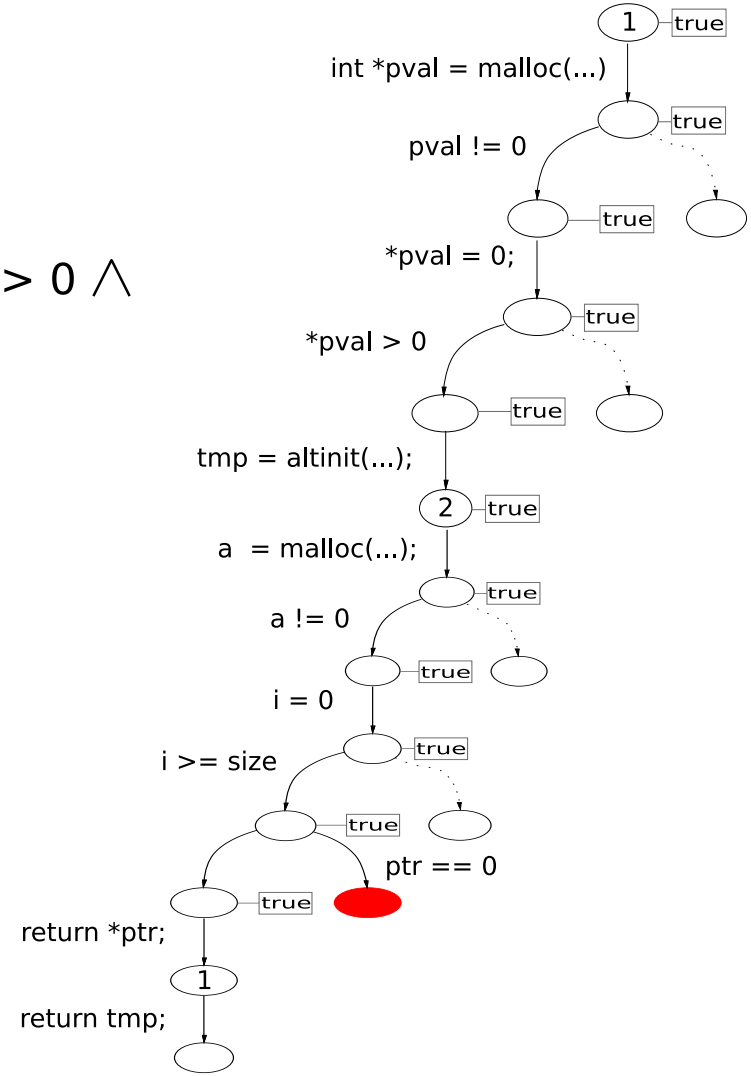
Counterexample Analysis

- Es soll zwischen *spurious* und *genuine counterexamples* unterschieden werden
- Aufstellen der *trace forumla*: Formel die genau dann erfüllbar ist, wenn der Pfad genommen werden kann
- Fehlerpfad wird in SSA Form transformiert und die *trace formula* wird erzeugt

$\langle pval, 1\rangle = malloc\_0 \wedge \langle pval, 1\rangle = 0 \wedge$
$\langle *(\langle pval, 1\rangle), 1\rangle = 0 \wedge \langle *(\langle pval, 1\rangle), 1\rangle\ 0 \wedge$

$\langle size, 1\rangle = \langle *(\langle pval, 1\rangle), 1\rangle \wedge$
$\langle pval1, 1\rangle = \langle pval, 1\rangle \wedge$
$\langle pval2, 1\rangle = \langle pval, 1\rangle \wedge$

$\langle a, 1\rangle = malloc\_1 \wedge \langle a, 1\rangle = 0 \wedge$
$\langle i, 1\rangle = 0 \wedge \langle i, 1\rangle \geq \langle size, 1\rangle \wedge$
$\langle ptr, 1\rangle = 0$



Predicate Discovery

- Jetzt sollen neue Prädikate eingefügt werden, sodass ein vollständiger bzgl. der Fehlerstelle sicherer ART entsteht
- Dazu werden alle *cuts* des Fehlerpfades betrachtet
- *Cuts* sind Knoten n im ART, die *trace formula* wird in 2. Teile aufgeteilt, ein Teil bis zu n, ein Teil nach n (φ^- und φ^+)
- Jetzt werden Interpolanten gebildet

Craigsches Interpolationslemma

Seien φ^- und φ^+ Formeln der Prädikatenlogik und es gelte:

- $\models \neg(\varphi^- \wedge \varphi^+)$

Dann gibt es eine Formel ψ sodass:

- $\models \varphi^- \longrightarrow \psi$

Craigsches Interpolationslemma

Seien φ^- und φ^+ Formeln der Prädikatenlogik und es gelte:

- $\models \neg(\varphi^- \wedge \varphi^+)$

Dann gibt es eine Formel ψ sodass:

- $\models \varphi^- \longrightarrow \psi$
- $\models \neg(\psi \wedge \varphi^+)$

Craigsches Interpolationslemma

Seien φ^- und φ^+ Formeln der Prädikatenlogik und es gelte:

- $\models \neg(\varphi^- \wedge \varphi^+)$

Dann gibt es eine Formel ψ sodass:

- $\models \varphi^- \longrightarrow \psi$
- $\models \neg(\psi \wedge \varphi^+)$
- ψ enthält nur Variablen die in beiden Formeln vorkommen

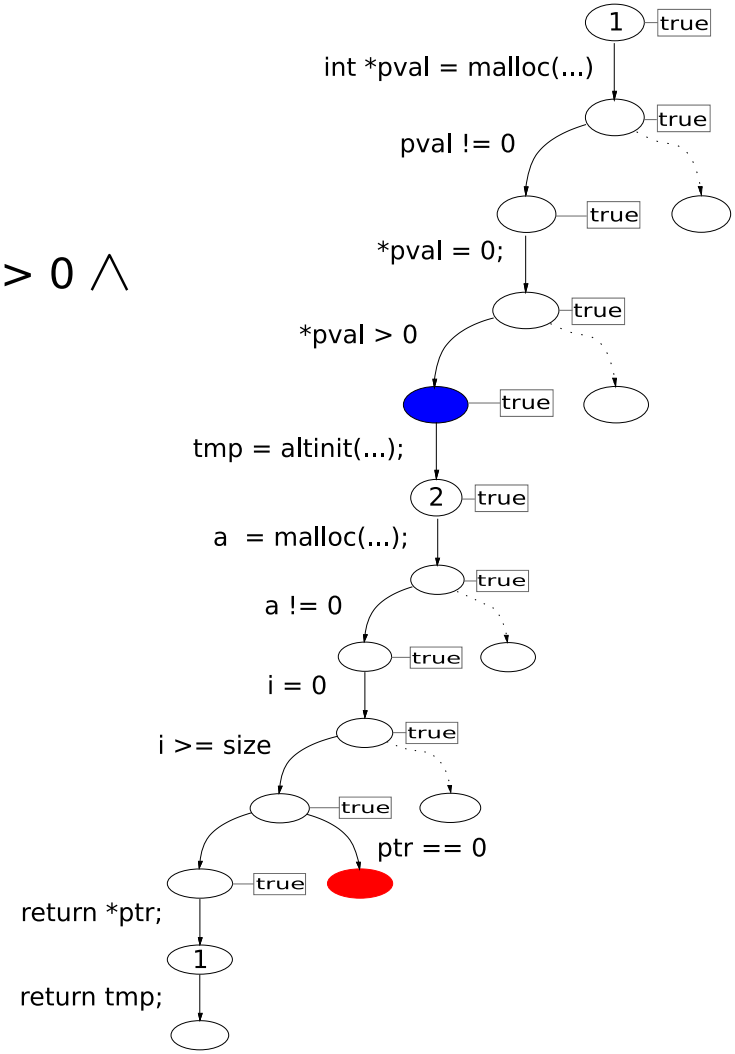
$\varphi^- =$

<pval, 1> = malloc_0 ∧ <pval, 1> = 0 ∧
<∗(<pval, 1>), 1> = 0 ∧ <∗(<pval, 1>), 1> 0 ∧

$\varphi^+ =$

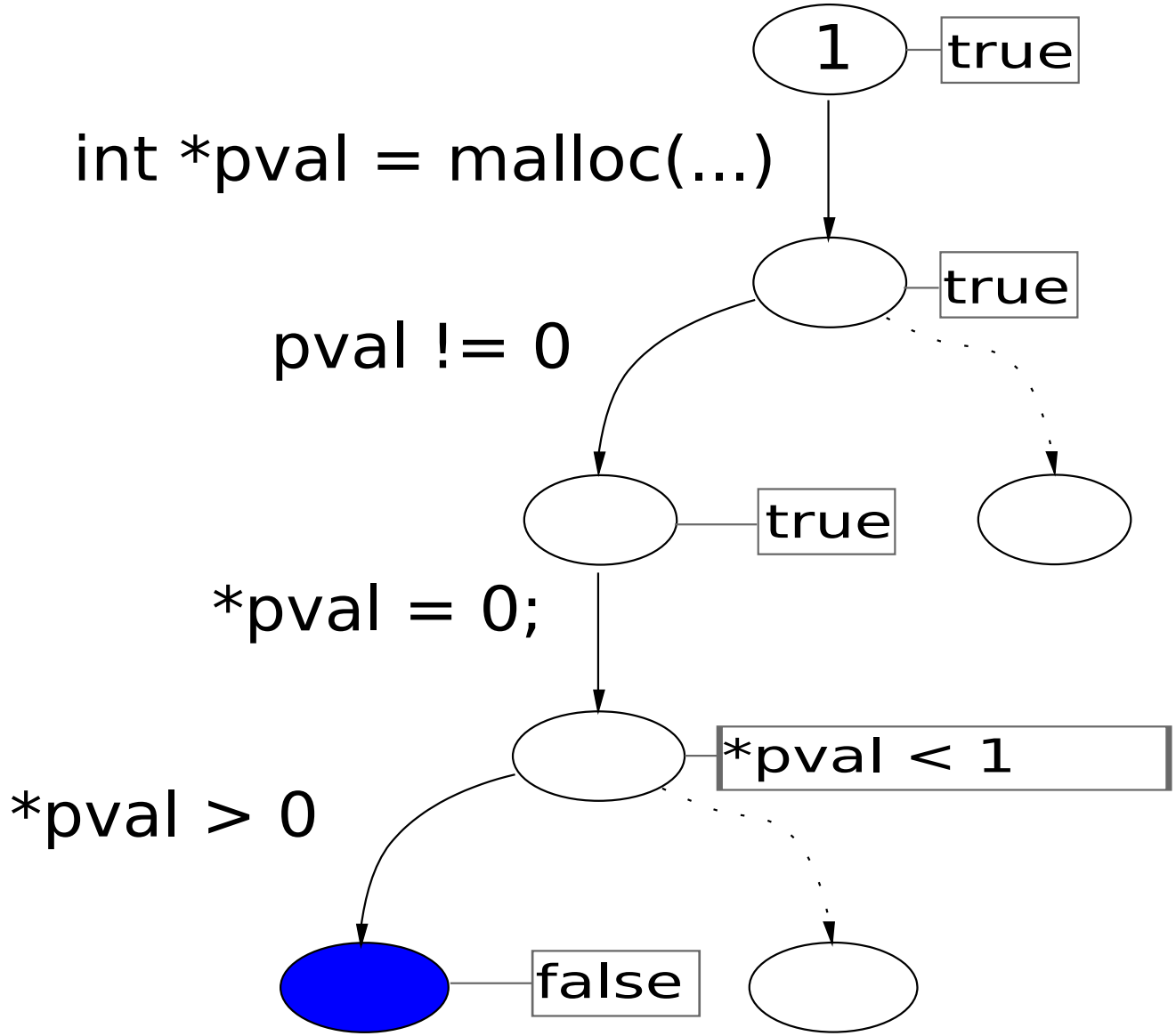
<size, 1> = <∗(<pval, 1>), 1> ∧
<pval1, 1> = <pval, 1> ∧
<pval2, 1> = <pval, 1> ∧
<a, 1> = malloc_1 ∧ <a, 1> = 0 ∧
<i, 1> = 0 ∧ <i, 1> ≥ <size, 1> ∧
<ptr, 1> = 0

ψ = <*(<pval, 1>), 1> >= 0



Verfeinerung des ART

- ψ leistet das gewünschte
- Jetzt wird der Fehlerpfad erneut abgelaufen und ψ beobachtet
- Nur Unterbäume von Knoten zu denen Prädikate hinzugefügt wurden werden aktualisiert
- Verfeinerung bis der ART sicher ist oder ein *genuine Counterexample* gefunden wird


1
true
int *pval = malloc(...)
true
pval != 0
true
*pval = 0;
*pval < 1
*pval > 0
false

Nach weiteren Iterationen ist der ART vollständig und sicher bzgl. des Errorlabels.

Unsicheres Testprogramm

```
void funcfoo(int x, int y){
  if(x > y){
    x = x - y;

  if(x > 0)
    ERROR: x++;

  }
}
```

BLAST Aufruf und Output

- gcc -E funcfoo.c -o funcfoo.i

BLAST Aufruf und Output

- gcc -E funcfoo.c -o funcfoo.i
- pblast.opt funcfoo.i -main funcfoo

BLAST Aufruf und Output

- gcc -E funcfoo.c -o funcfoo.i
- pblast.opt funcfoo.i -main funcfoo

Output: [...] Error found! The system is unsafe :-(
Error trace:

Sicheres Testprogramm

```
void funcfoo(int x, int y){
  if(x > y){
    x = x - y;

  if(x <= 0)
    ERROR: x++;

  }
}
```

BLAST Output

Output: [...]
Exception raised :(Failure("No new preds found !– and not running allPreds ...") Ack! The gremlins again!: Failure("No new preds found !– and not running allPreds ...") Ack! The gremlins again!: Failure("No new preds found !– and not running allPreds ...")
Fatal error: exception Failure("No new preds found !– and not running allPreds ...")