

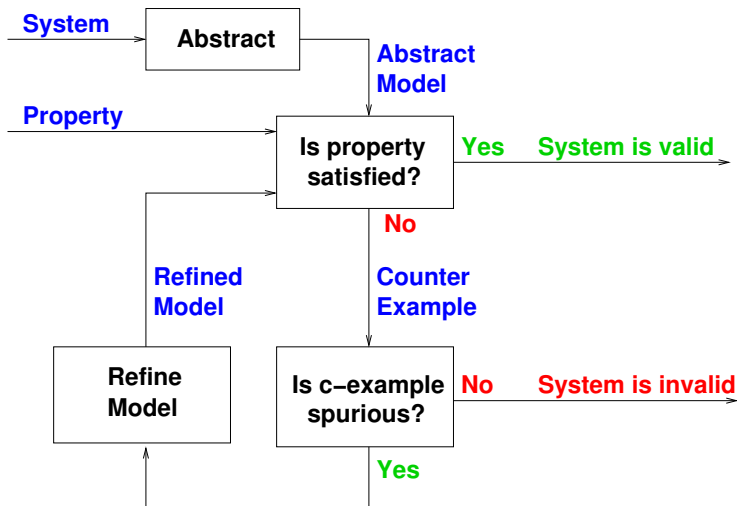
*IV101 – BLAST a verifikace metodou CEGAR*

Jiří Barnat

# *BLAST*

- model-checker pro programy v jazyce C
- CEGAR metoda
- nástroj vyvinut skupinou Thomase A. Henzingera
- aktuální verze 2.5 (11. 7. 2008)
- <http://mtc.epfl.ch/blast/>

CEGAR – Counter-Example Guided Abstraction Refinement

*CEGAR*

## Abstrakce

- metoda vyžaduje, aby abstrakce systému byla *overapproximace*
- prvotní abstrakce systému je dána pouze grafem toku řízení (control-flow graph), tj. v každém stavu abstraktního systému se dovoluje libovolná valuace proměnných
- pro některé běhy v abstraktním systému neexistují běhy původního (neabstrahovaného modelu)
- protipříklady, tvořeny nereálnými běhy jsou tzv. *spurious*
- analýzou neplatných protipříkladů získáváme omezení na valuace proměnných (predikáty)
- pro odvozování predikátů z protipříkladů se používá theorem prover *Simplify*

## *Odstranění neplatného protipříkladu predikátem*

- stupid.c

```
22 ...
23 if (!(x>y)) {
24     while (x>y)
25         x=x-y;
26 }
27 printf("%d\n",x);
28 ...
```

- je nevyhnutelně dosažitelný řádek 27 pokud uvážíme prázdnou množinou predikátů ?

## *Odstranění neplatného protipříkladu predikátem*

- stupid.c

```
22 ...
23 if (!(x>y)) {
24     while (x>y)
25         x=x-y;
26 }
27 printf("%d\n",x);
28 ...
```

- je nevyhnutelně dosažitelný řádek 27 pokud uvážíme prázdnou množinou predikátů ? NE

## *Odstranění neplatného protipříkladu predikátem*

- stupid.c

```
22 ...
23 if (!(x>y)) {
24     while (x>y)
25         x=x-y;
26 }
27 printf("%d\n",x);
28 ...
```

- je nevyhnutelně dosažitelný řádek 27 pokud uvážíme prázdnou množinou predikátů ? **NE**
- jak vypadá spurious counterexample?

## Odstranění neplatného protipříkladu predikátem

- stupid.c

```
22 ...
23 if (!(x>y)) {
24     while (x>y)
25         x=x-y;
26 }
27 printf("%d\n",x);
28 ...
```

- je nevyhnutelně dosažitelný řádek 27 pokud uvážíme prázdnou množinou predikátů ? **NE**
- jak vypadá spurious counterexample? **... 22.23.{24.25}<sup>ω</sup>**

## Odstranění neplatného protipříkladu predikátem

- stupid.c

```
22 ...
23 if (!(x>y)) {
24     while (x>y)
25         x=x-y;
26 }
27 printf("%d\n",x);
28 ...
```

- je nevyhnutelně dosažitelný řádek 27 pokud uvážíme prázdnou množinou predikátů ? **NE**
- jak vypadá spurious counterexample? ... 22.23.{24.25}<sup>ω</sup>
- je nevyhnutelně dosažitelný řádek 27 pokud uvážíme predikát (x>y)?

## Odstranění neplatného protipříkladu predikátem

- stupid.c

```
22 ...
23 if (!(x>y)) {
24     while (x>y)
25         x=x-y;
26 }
27 printf("%d\n",x);
28 ...
```

- je nevyhnutelně dosažitelný řádek 27 pokud uvážíme prázdnou množinou predikátů ? **NE**
- jak vypadá spurious counterexample? **... 22.23.{24.25}<sup>ω</sup>**
- je nevyhnutelně dosažitelný řádek 27 pokud uvážíme predikát  $(x>y)$ ? **ANO**

BLAST – Berkeley Lazy Abstraction Software Verification Tool

## *Poznámka o nerozhodnutelnosti*

- *safety* vlastnost = absence chybového stavu systému
- programovací jazyk C má Turingovskou vyjadřovací sílu
- problém dosažení dané konfigurace Turingova stroje je nerozhodnutelný, tj.
- **neexistuje algoritmus, pro daný problém verifikace**
- pro daný program a danou vlastnost nastává jedna z následujících možností
  - BLAST prokáže, že chybový stav je nedosažitelný
  - BLAST najde chybový stav a vrátí protipříklad
  - BLAST neskončí (počítá dokud nedojdou systémové zdroje)

## *Ověřování dosažitelnosti chybového stavu*

- chybový stav není rozpoznán nástrojem BLAST
- chybový stav je označen programátorem
- chybový stav je označen návěstím ve zdrojovém souboru  
    ERROR: goto ERROR
  
- lze použít pro detekci porušených *assertů*
- předefinuje funkci `assert()`, tak aby obsahovala chybový stav
  
- verifikace není simulace jednoho běhu!
- má šanci detekovat chyby, které neodhalí testování

## *Ověřování dosažitelnosti chybového stavu BLASTem*

- `-main XXX`
  - označuje hlavní funkci programu
- `-L label`
  - specifikuje návěští, které identifikuje neplatný stav
  - pokud neuvedeno hledá se návěští `ERROR`
- detekce chybových stavů
  - `pblast.opt -main foo -L ERROR muj.c`
- detekce neplatných assertů
  - předpokládá `assert.h`, která vhodně definuje `assert()`
  - `cpp -I. muj.c >muj.i`  
`pblast.opt -main foo -L ERROR muj.i`

## *Ověřování temporálních safety vlastností*

- temporální safety vlastnosti jsou vlastnosti systému, které lze posoudit pomocí externího monitoru systému
- temporální safety vlastnost je nesplněna, pokud monitor projde chybovým stavem
- příklad
  - uvažme systém, který zamyká a odemyká přístup k nějakému sdílenému prostředku
  - ptáme se, zda použití zámků je konzistentní, tj. na každém běhu striktně alternuje volání funkcí pro zamčení a odemčení zámku

## *Ověřování temporálních safety vlastností BLASTem*

### interní monitor

- monitor se implementuje přímo do zdrojového kódu
- nutno provádět pro každou instanci verifikace

### externí monitor

- monitor je popsán v externím souboru
- BLAST automaticky instrumentuje kód
  
- v obou případech se problém ověření temporálních safety vlastností redukuje na problém dosažitelnosti

## *Soubor specifikující externí monitor*

- definice globálních proměnných monitoru
  - klíčové slovo `global`
- abstraktní typy
  - klíčové slovo `shadow`
  - typicky pro použití s knihovnamy ke kterým nejsou dostupné zdrojové kódy
- události (events)
  - mění stav monitoru dle chování původního kódu
  - event {  
    pattern { FSMLock(); }  
    guard { lockStatus == 0 }  
    action { lockStatus = 1; }  
}
- nesplnění sekce `guard` vede k přechodu do chybového stavu

## *Soubor specifikující externí monitor*

- možnost přenosu výsledků volání funkcí v originálním souboru do proměnných monitoru pomocí \$? a \$1, \$2, ...
- nesmí volat `system()` dokud nezmění své EUID na nenula  
`global int _monvar_ = 0;`

```
event {  
  pattern{ $? = seteuid($1); }  
  action{ _monvar_ = $1; }  
}
```

```
event {  
  pattern{ $? = system($?); }  
  guard{ _monvar_ != 0; }  
}
```

## *Ověřování temporálních safety vlastností BLASTem*

- `spec.opt mojespecifikace.spc muj.c`
- vytvoří soubory
  - `instrumented.c`  
obsahuje kombinaci původního kódu a monitoru
  - `instrumented.pred`  
výchozí predikáty vygenerované při instrumentaci kódu
- `pblast.opt instrumented.c`

## *Přídavné predikáty*

- existují případy verifikace, v kterých BLAST neumí odvodit správně predikáty pro dokončení verifikace
- obecně je to nerozhodnutelný problém
- uživatel nástroje může díky svému chápání verifikovaného systému některé predikáty znát
  
- přídavné predikáty lze specifikovat v souboru \*.pred
- pomocí @ lze upřesnit rozsah platnosti predikátu
  - `x@foo >= 0;`

## *Verifikovatelný zdrojový kód*

- umí pointer aliasing
- nedeterminismus pomocí `__BLAST_NONDET`

```
if (__BLAST_NONDET) {  
    // then branch  
} else {  
    // else branch  
}
```

Ukázka práce s BLASTem

## *Další aspekty použití nástroje BLAST*

- ukládá předchozí nalezené predikáty pro další použití
  - několik voleb ovlivňující způsob nalezení predikátů
  - a mnoho dalších voleb viz manuál
- 
- nástroj je veřejně dostupný, ale působí spíš jako zveřejněná prototypová implementace
  - GUI existuje, ale není aktuální pro poslední verzi