Projekt do předmětu FAV – Nástroj CPAchecker

Vojtěch Havlena, xhavle03

26. ledna 2016

1 Charakteristika nástroje

CPAchecker je volně dostupný framework a konfigurovatelný nástroj pro softwarovou verifikaci programů napsaných v jazyce C. V tomto projektu jsem využil verzi 1.4 pro operační systém Linux. Samotný nástroj je napsán v jazyce Java[1] a je vyvýjen na univerzitě v německém Passau. Nástroj CPAchecker je založen na konceptu konfigurovatelné analýze programů (configurable program analysis – CPA). CPA je koncept, který umožňuje pomocí stejného formálního základu vyjádřit různé verifikační přístupy. Tedy pomocí jediného formalismu je možné vyjádřit různé přístupy založené na analýze programů a na model checkingu [4].

1.1 Základní vlastnosti

CPAchecker je nástroj zahrnující model checking, založený na predikátové abstrakci (implementováno jako CPA, viz. níže). V mnoha ohledech je CPAchecker podobný nástroji Blast – pro predikátovou analýzu se stejně jako v nástroji Blast využívá lazy abstrakce a interpolace [4]. Výhodou nástroje CPAchecker je jeho snadná konfigurovatelnost. Nástroj může například provádět predikátovou analýzu použitím single-block encoding (SBE), large-block encoding (LBE), popřípadě adjustable-block encoding (ABE). O přístupu single-block encoding hovoříme v případě, kdy hrana v abstract reachibility graph (ARG) reprezentuje jeden blok v programu. V případě, že hrany v ARG reprezentují větší část programu, hovoříme o LBE [2]. ABE potom umožňuje vyjádřit oba předchozí přístupy.

Další výhodou, především oproti nástrojům, které implementují smyčku abstract-check-refine odděleně je, že on-the-fly přístup lazy abstrakce umožňuje použití efektivnějších a flexibilnějších algoritmů. Do nástroje CPAchecker je také integrován bounded model checker pro kontrolu proveditelnosti chybových cest. Je rovněž využíván přístup CEGAR.

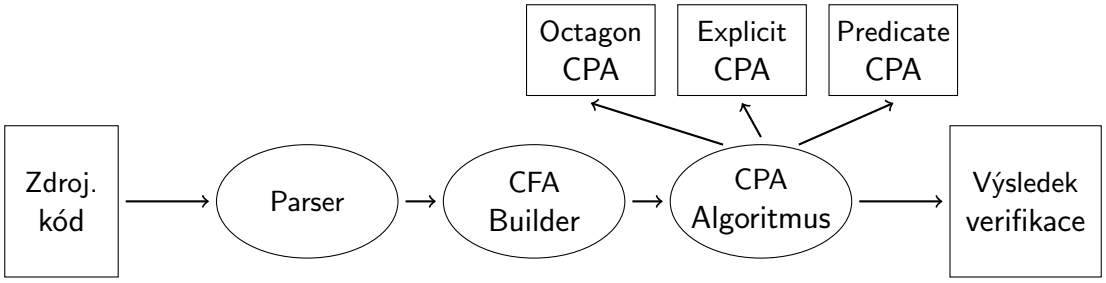
1.2 Architektura

Celý nástroj CPAchecker se skládá z několika částí. V první části se vstupní program ve zdrojovém kódu převede do syntaktického stromu. Na základě tohoto

[1] pro běh je nutný Java Runtime Environment verze 7+

stromu se potom vytvoří control-flow automaty, které reprezentují program [4]. Z těchto automatů se potom postupně buduje abstract reachibility graph (ARG).

Jádrem celého nástroje je CPA algoritmus [3]. CPA algoritmus provádí analýzu dosažitelnosti s tím, že pracuje nad objekty abstraktního datového typu CPA. CPA tedy tvoří rozhraní pro operace, které jsou využívány v CPA algoritmu, který nad těmito operacemi pracuje bez znalosti jejich konkrétní implementace. Konkrétní použitý CPA může být také kombinací více různých CPA (v tom případě se hovoří o složené CPA) [4]. CPAchecker obsahuje konkrétní implementaci CPA např. pro predikátovou abstrakci. Stručné schéma celého nástroje je na obr. 1.



Obrázek 1: Schéma nástroje CPAchecker. Převzato a upraveno z [4]

V případě rozšíření nástroje CPAchecker o další CPA pro novou abstraktní doménu, je nutné provést následující kroky. Nejprve je nutné pro nový CPA vytvořit záznam v globálním konfiguračním souboru. Dále je nutné, aby nový CPA implementoval požadované CPA rozhraní a všechny operace v něm obsažené. Například pokud chceme implementovat CPA pro shape analýzu, vytvoříme nový CPA obsahující konkrétní implementaci operací. Je možné také vytvořit několik různých implementací stejné operace pro daný CPA [4].

1.3 Použití

Nástroj CPAchecker provádí analýzu programů zapsaných v jazyce C. Podle vývojářů je nástroj schopný parsovat a analyzovat velkou podmnožinu jazyka (GNU) C. Před samotnou verifikací je nutné provést předzpracování vstupního zdrojového kódu preprocesorem jazyka C – zdrojový kód nesmí obsahovat direktivy `#define` a `#include` (případně lze využít přepínač `-preprocess` a CPAchecker sám toto předzpracování provede). CPAchecker také experimentálně poskytuje možnost spojit více C souborů a provést nad nimi verifikaci.

Specifikace pro verifikaci se zadává pomocí jednoduchého automatu (jazyk pro popis těchto automatů je podobný jazyku Blast Query Language [1]). S nástrojem jsou dodávány i již předdefinované specifikace. Specifikace je vyjádřena pomocí vlastností dosažitelnosti (ve výchozí specifikaci to je kontrola příkazu `assert`, dosažení labelu `ERROR` a volání funkcí `abort()` a `exit()`).

Pro běh nástroje CPAchecker se nejčastěji používá kombinace specifikace konfiguračního souboru pro verifikaci (přepínač `-config`, definuje parametry verifikace, ověřovanou specifikaci apod.) a samotného souboru se zdrojovým textem, který

se má verifikovat. Výběr vhodné konfigurace záleží na konkrétní podobě programu, který se má verifikovat. Mezi předdefinované přepínače zastupující konfigurační soubory mj. patří:

- `-predicateAnalysis` – predikátová analýza s přístupem CEGAR, doporučeno pro obecné použití
- `-valueAnalysis` – analýza hodnot (value analysis) celočíselných proměnných
- `-bmc-induction` – bounded model checking, použití indukce pro dokázání bezpečnosti. Tato konfigurace je zatím označena jako experimentální.
- A další (`-sv-comp15`, `-octagonAnalysis`, ...)

Nevýhodou nástroje CPAchecker je nepodporování verifikace programů, které obsahují rekurzi (existuje ale nástroj založený na CPAcheckeru, nazvaný CPArec, který umožňuje verifikaci rekurzivních C programů). Dále CPAchecker nepodporuje verifikaci programů obsahující souběžnost. Další nevýhodou je absence tutoriálu, popřípadě nějaké podrobnější uživatelské dokumentace k používání nástroje.

I přesto se tento nástroj v praxi používá (používá jej např. výzkumná skupina zabývající se verifikací Linuxových ovladačů). Pomocí tohoto nástroje bylo také odhaleno několik chyb v Linuxovém jádře. Navíc na soutěži Competition on Software verification se nástroj CPAchecker v určitých kategoriích pravidelně umísťuje na čelních pozicích[2].

2 Experimenty na stávajících příp. studiích

Spolu s nástrojem jsou dodávány různé zdrojové soubory v jazyce C, které byly použity např. pro srovnávací experimenty s jinými nástroji. Na těchto souborech je možné vyzkoušet běh nástroje CPAchecker. Po ukončení běhu je vygenerován soubor se shrnujícími informacemi a statistikami o proběhlé verifikaci. Všechny experimenty jsem prováděl na systému GNU/Linux (Debian 8) 64 bit, 4-jádrový procesor AMD, 6 GB RAM.

Asi nejjednodušším z těchto dodávaných souborů je `example.c`, který obsahuje funkci `main`, dvě celočíselné proměnné, cyklus while a několik podmínek. Chybový stav je označen návěštím `ERROR`. Zdrojový soubor neobsahuje direktivy preprocesoru, není tedy nutné používat parametr `-preprocess`. Vzhledem k použitému chybovému návěští, jsem zvolil výchozí specifikaci, která obsahuje dosažení labelu `ERROR`. Samotnou verifikaci jsem potom provedl s následujícími konfiguracemi:

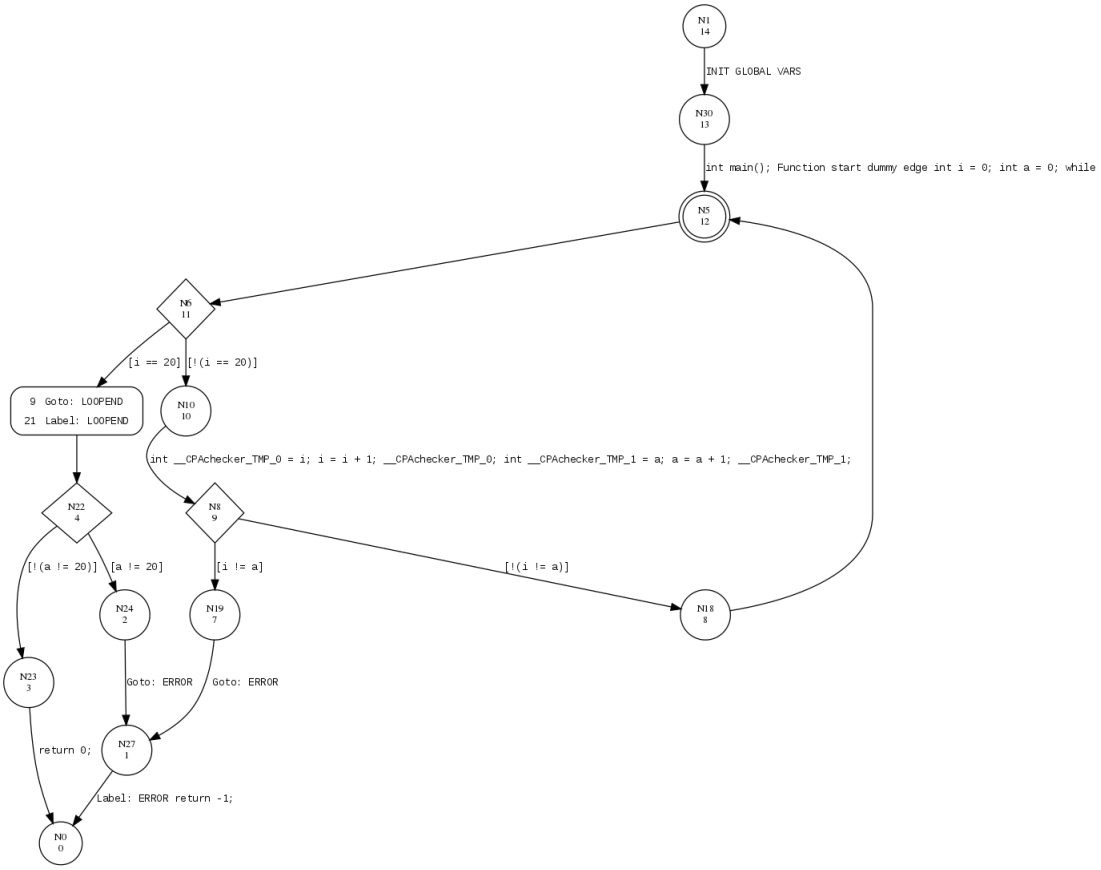
- `-predicateAnalysis`. Pro tuto konfiguraci verifikace skončila, vzhledem k zadané specifikaci, s výsledkem Safe. Běh trval 2.75 s. Množství využité paměti hromady je 47 MB. Soubor se statistikami obsahuje i další podrobnější informace o verifikaci a využitých zdrojích během verifikace – počet abstrakcí:

[2] viz. `http://sv-comp.sosy-lab.org/2015/results/`

5, zdroje využité pro BDD (počet uzlů BDD: 205, ...), statistiky algoritmu CEGAR (počet zpřesnění: 1, ...), apod.

- `-valueAnalysis`. Pro tuto konfiguraci opět verifikace skončila s výsledkem SAFE. Běh trval 2.46 s. Množství využité paměti hromady je 48 MB.
- `-bmc-induction`. Výsledek verifikace byl opět SAFE. Při této konfiguraci běh trval 3.46 s. Množství využité paměti hromady je 43 MB
- `-sv-comp15`. Pro tuto konfiguraci bylo nutné provést úpravu konfiguračního souboru tak, aby bylo povoleno vytváření statistiky. Výsledek verifikace byl opět SAFE. Běh trval 2.20 s a množství využité paměti hromady je 50 MB.

Pomocí dodávaného skriptu je také možné vygenerovat control-flow automaty a abstract-reachibility graf, které byly vytvořeny ze zdrojového textu během procesu verifikace. Příklad CFA pro zdrojový soubor `example.c` je na obr. 2. Skript také umožňuje celý výsledek verifikace přehledně zobrazit v HTML souboru. V případě, že vstupní program obsahuje chybu, je v CFA a ARG vyznačena cesta, která vedla k chybě.



Obrázek 2: CFA použitý při verifikaci souboru `example.c`.

Kromě verifikace tohoto jednoduchého programu jsem provedl verifikaci na dalších dodávaných programech. Experimentoval jsem na jedné z dodávaných sadách

benchmarků. Sada, kterou jsem použil, tvoří zjednodušené ovladače v systému Windows. Kromě této sady je k dispozici např. sada, obsahující zjednodušené verze stavových automatů, které zabezpečují komunikaci v rámci SSH. Pro všechny provedené experimenty byla použita výchozí specifikace. Pro verifikaci jsem nejprve zvolil konfigurace `-predicateAnalysis` (predikátová analýza) a `-valueAnalysis` (analýza hodnot). Dodávané zdrojové soubory již neobsahují direktivy preprocesoru, není je tedy nutné upravovat. Verifikované programy jsou již komplexnější programy, obsahují podstatně více celočíselných proměnných, cyklů, podmínek a funkcí než základní soubor `example.c`. Samotné výsledky těchto experimentů lze nalézt v tabulce 1. V případě, že byla nalezena chyba, ve vygenerovaném CFA byla vyznačena cesta k této chybě. Výsledné CFA s chybou zde ale kvůli rozměru neuvádím.

Program	**Oček.**	**-predicateAnalysis**			**-valueAnalysis**		
	výsledek	Výsl.	Čas [s]	Paměť	Výsl.	Čas [s]	Paměť
`cdaudio_simpl1`	SAFE	SAFE	19.46	117	SAFE	19.85	407
`cdaudio_simpl1_BUG`	BUG	BUG	16.01	141	BUG	10.19	199
`diskperf_simpl1`	SAFE	SAFE	12.78	112	SAFE	13.67	380
`floppy_simpl3`	SAFE	SAFE	10.55	110	SAFE	11.09	300
`floppy_simpl3_BUG`	BUG	BUG	12.74	110	BUG	12.24	327
`floppy_simpl4`	SAFE	SAFE	11.26	108	SAFE	12.69	374
`floppy_simpl4_BUG`	BUG	BUG	13.46	113	BUG	13.17	375
`kbfiltr_simpl1`	SAFE	SAFE	4.80	60	SAFE	6.62	103
`kbfiltr_simpl2`	SAFE	SAFE	5.95	59	SAFE	7.36	105
`kbfiltr_simpl2_BUG`	BUG	BUG	8.02	75	BUG	8.79	110

Tabulka 1: Výsledek verifikace dodávaných souborů (zjednodušené ovladače systému Windows) pro konfigurace `-predicateAnalysis` a `-valueAnalysis`. Pro každou konfiguraci a soubor je uveden výsledek verifikace, doba v sekundách a využitá paměť hromady v MB.

Kromě výše zmíněných konfigurací jsem provedl experimenty také s konfigurací `-bmc-inductive`. Výsledky experimentu jsou uvedeny v tabulce 2. V případě, že byla v programu chyba, CPACHECKER ji sice odhalil (vypsal, že v programu existuje chyba), ale již se nepodařilo vytvořit chybovou cestu. Tudíž verifikace skončila s výsledkem SAFE. Toto chování může být způsobeno tím, že konfigurace `-bmc-induc tive` je zatím označena jako experimentální a tudíž ne vše může být funkční.

Program	Oček. výsledek	-bmc-inductive		
		Výsl.	Čas [s]	Paměť
cdaudio_simpl1	SAFE	SAFE	10.41	113
cdaudio_simpl1_BUG	BUG	SAFE	8.89	113
diskperf_simpl1	SAFE	SAFE	7.04	109
floppy_simpl3	SAFE	SAFE	6.60	108
floppy_simpl3_BUG	BUG	SAFE	6.60	108
floppy_simpl4	SAFE	SAFE	7.07	106
floppy_simpl4_BUG	BUG	SAFE	7.44	109
kbfiltr_simpl1	SAFE	SAFE	4.66	61
kbfiltr_simpl2	SAFE	SAFE	5.51	58
kbfiltr_simpl2_BUG	BUG	SAFE	6.11	58

Tabulka 2: Verifikace dodávaných souborů s konfigurací `-bmc-inductive`.

3 Experimenty na vlastních příp. studiích

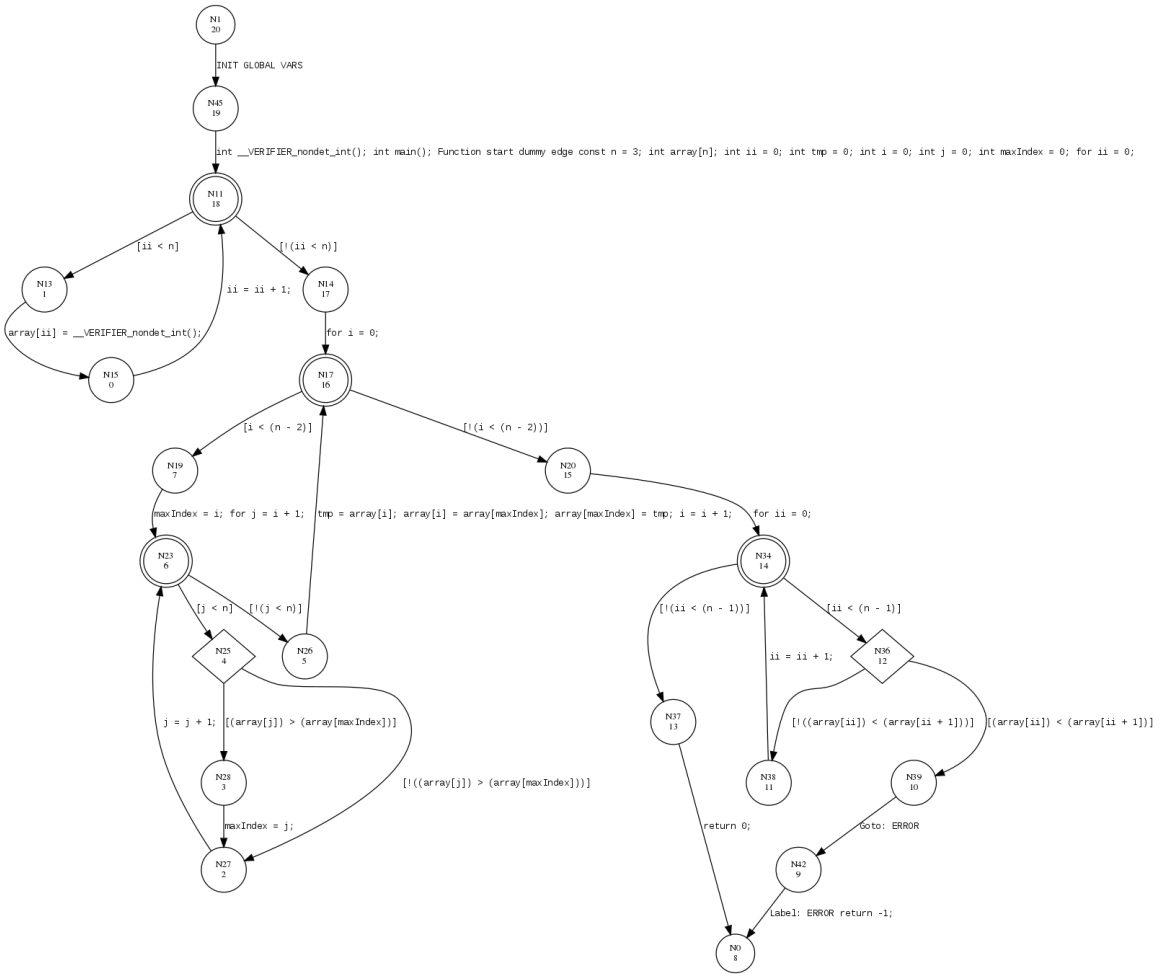
Vlastní experimenty jsem provedl na programu pracující s polem celočíselných hodnot. Program provádí řazení pole pomocí jednoduchého algoritmu selection sort. Zdrojový kód v jazyce C je uveden níže.

```
extern int __VERIFIER_nondet_int();
const n = 3;
int main()
{
   int array[n];
   int ii = 0;
   int tmp = 0, i = 0, j = 0, maxInd  = 0;

   for(ii = 0; ii < n; ii++)
      array[ii] = __VERIFIER_nondet_int();

   for (i = 0; i < n - 1; i++)
   {
      maxInd = i;
      for (j = i + 1; j < n; j++)
      {
          if (array[j] > array[maxInd])
                maxInd = j;
      }
      tmp = array[i];
      array[i] = array[maxInd];
      array[maxInd] = tmp;
   }
   for(ii = 0; ii < n - 1; ii++)
   {
      if(array[ii] < array[ii + 1])
         goto ERROR;
   }
   return 0;
   ERROR:
      return -1;
}
```

Nejprve se pole hodnot naplní celočíselnými hodnotami. Následně se pole seřadí pomocí algoritmu selection sort. V posledním cyklu `for` je kontrola, zda je pole opravdu seřazeno v sestupném pořadí hodnot. V rámci experimentu jsem postupně zvyšoval konstantu `n` a prováděl verifikaci. Vzhledem k použitému labelu `ERROR` jsem použil výchozí specifikaci. Co se týče konfigurací, nejprve jsem použil `-valueAnalysis` a `-predicateAnalysis-bitprecise`. Nicméně při použití této konfigurace byl výsledek verifikace UNKNOWN, proto jsem další experimenty prováděl s konfigurací `-predicateAnalysis`. CFA použitý při verifikaci je na obr. 3. Výsledky verifikace při této konfiguraci jsou potom uvedeny v tabulce 3.



Obrázek 3: CFA použitý při verifikaci řazení pole pomocí selection sortu.

Nyní se budu zabývat situací, kdy do programu zavedu chybu. Například, že 12. řádek původního programu nahradím za následující kód

```
for(i = 0; i < n-2; i++),
```

tedy poslední prvek pole není zařazen na správné místo a pole nemusí být seřazené. V tomto případě při verifikaci s konfigurací `-predicateAnalysis` je obdržen výsledek BUG v čase 9.71 s.

Konstanta `n`	**-predicateAnalysis**		
	Výsledek	Čas [s]	Paměť [MB]
`1`	SAFE	4.32	48
`2`	SAFE	6.94	59
`3`	SAFE	15.69	108
`4`	SAFE	47.08	205
`5`	UNKNOWN	TO	–

Tabulka 3: Výsledky verifikace pragramu provádějící řazení staticky alokovaného pole, pro různé hodnoty konstanty `n`. Při verifikaci s konstantou `n = 5` vypršel časový limit, který byl nastaven na 900s.

V rámci dalšího experimentu jsem provedl modifikaci původního programu tak, aby pracoval s dynamicky alokovaným polem místo se statickým polem. Tedy 5. řádek původního programu jsem zaměnil za následující kód

```
int *array = (int *)malloc(n * sizeof(int));
if(array == NULL)
   return 0;
```

Vzhledem k použití knihovní funkce `malloc` je nutné přidat i hlavičkový soubor `#include<stdlib.h>`. Při verifikaci je potom ale nutné použít přepínač `-preprocess`, protože v kódu je použita direktiva `#include`. Pro verifikaci jsem opět použil standardní specifikaci spolu s konfigurací `-predicateAnalysis` (stejně jako v předchozím experimentu). Na řádku zdrojového kódu programu, který provádí alokaci pole je použito explicitní přetypování `(int *)`. V případě, že bych toto přetypování nepoužil, CPACHECKER při verifikaci objeví chybu. Experiment opět spočíval ve verifikaci modifikovaného programu pro různé konstanty `n`. Výsledky experimentu jsou shrnuty v tabulce 4.

Konstanta `n`	**-predicateAnalysis**		
	Výsledek	Čas [s]	Paměť [MB]
`1`	SAFE	4,78	57
`2`	SAFE	7.62	71
`3`	SAFE	7.76	75
`4`	SAFE	10.52	111
`5`	SAFE	29.16	204
`6`	SAFE	45.27	381

Tabulka 4: Výsledky verifikace pragramu provádějící řazení dynamicky alokovaného pole, pro různé hodnoty konstanty `n`

Z tabulky plyne, že při použití dynamicky alokovaného pole v programu je doba běhu nástroje CPAchecker pro danou konstantu `n` nižší než při použití staticky alokovaného pole. Paměťové nároky jsou v obou případech srovnatelné.

4 Závěr

Tato esej shrnuje poznatky získané při používání a studiu nástroje CPAchecker. V první části práce je popsán princip, na kterém je nástroj postaven. Druhá část je věnována experimentům na existujících případových studiích. Na existujících programech jsem experimentoval s verifikací pro různé konfigurace. Poslední část je potom věnována vlastním případovým studiím, kde je věnována pozornost verifikaci programu provádějící řazení celých čísel.

Reference

[1] Beyer, D., Chlipala, A. J., Henzinger, T. A. et al. The Blast Query Language for Software Verification. *Static Analysis*. 2004. S. 2–18.

[2] Beyer, D., Cimatti, A., Griggio, A. et al. Software Model Checking via Large-Block Encoding. *CoRR*. 2009, abs/0904.4709.

[3] Beyer, D., Henzinger, T. a Théoduloz, G. *Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis*. Sv. 4590. [b.m.]: Springer Berlin Heidelberg, 2007. S. 504–518. ISBN 978-3-540-73367-6.

[4] Beyer, D. a Keremoglu, M. *CPAchecker: A Tool for Configurable Software Verification*. Sv. 6806. [b.m.]: Springer Berlin Heidelberg, 2011. S. 184–190. ISBN 978-3-642-22109-5.