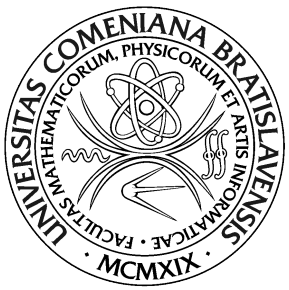
UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

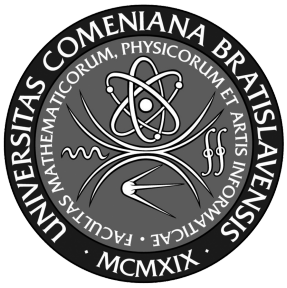


VERIFIKAČNÝ NÁSTROJ PRE UNITY

Diplomová práca

2017 Bc. Filip Špaldoň

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY



VERIFIKAČNÝ NÁSTROJ PRE UNITY

Diplomová práca

Študijný program:	Aplikovaná informatika
Študijný odbor:	2511 Aplikovaná informatika
Školiace pracovisko:	Katedra aplikovanej informatiky
Školiteľ:	doc. RNDr. Damas Gruska, PhD.

Bratislava, 2017 Bc. Filip Špaldoň

Čestne prehlasujem, že túto diplomovú prácu som vypracoval samostatne len s použitím uvedenej literatúry a za pomoci konzultácií u môjho školiteľa.

.....

Bratislava, 2017 Bc. Filip Špaldoň

Poďakovanie

Abstrakt

Abstrakt EN

Obsah

Kapitola 1

Úvod

Kapitola 2

Motivácia

Kapitola 3

Východiská

V tejto kapitole je cieľom poskytnúť prehľad základným pojmom a postupov pri tvorbe verifikačného nástroja pre programovaní jazyk UNITY.

3.1 UNITY

UNITY vychádza z knihy Parallel Program Design - A Foundation, v ktorej bol UNITY popísaný a navrhnutý autormi K. Mali Chandy a Jayadev Misra z Univerzity of Texax. Je to teoretický jazyk, ktorý sa zameriava na to **čo**, namiesto toho **kde**, **kedy** alebo **ako**. Jazyk neobsahuje žiadnu metódu **riadenia toku** a príkazy programu prebiehajú **nedeterministickým spôsobom synchrónne a asynchrónne**, kým sa **priradenia** nedostanú do konečného **stavu**. To umožňuje, aby programy bežali na neurčito, ako napríklad autopilot alebo elektráreň, ktoré by normálne skončili.

3.2 Vlastnosti UNITY

- Nedeterminizmus

- Absencia toku riadenia (control-flow)
- Synchrónnosť a asynchrónnosť
- Stavy a priradenia

3.2.1 Nedeterminizmus

Je algoritmus, ktorý si môže vybrať z viacerých možností, ktoré má k dispozicií. Nedeterministický algoritmus môže pri rovnakých vstupoch dávať rozdielné výsledky. Konkrétne v UNITY to znamená, že jednotilvé príkazy a priradenia sa budú vykonávať v rozdielnom poradí, čo môže mať za dôsledok rozdielné výsledky programu.

3.2.2 Absencia riadenia toku (control-flow)

Takýto tok nám zabezpečí paralelné vykonávanie programu. V prvotných programovacích jazykoch sa control-flow používal na postupné riadenie procesov. V prípade UNITY je takéto riadenie procesov zabezpečené paralelizmom.

3.2.3 Synchrónnosť a asynchrónnosť

Ako všetky programovacie jazyky, ktoré sú založené na paralelizme aj UNITY využíva synchronné a asynchrónne operácie.

3.2.4 Stavy a priradenia

Stavy a priradia sú základom UNITY programu. Konkrétne tento prechodový systém pozostáva z počiatočného stavu a transformácií, ktoré sú reprezentované premennými a priradeniami. Do výsledného stavu sa program dostane

pomocou niekoľkých priradení, pri ktorých premenné nadobúdajú výsledné hodnoty.

3.3 Telo programu

UNITY obsahuje štyri základné sekcie: decleráciu premmenných, množinu skratiek, počiatočné hodnoty premenných a množinu priraďovacích príkazov. V tele programu sa tieto sekcie vyskytujú pod názvamy declare, always, initially, assign. Telo programu obsahuje aj program-name, názov programu, ktorý môžeme vynechať, v tom prípade z tela programu vynechávame aj sekciu program-name. UNITY program má nasledujúcu formu:

```
Program      program-name
    declare  declare-section
    always   always-section
    initially initially-section
    assign   assign-section
end
```

Obr. 3.1: Ukážka tela programu

3.3.1 Declare-section

Táto sekcia obsahuje dekleráciu premenných použité v programe a ich súvisiace typy. V nasledujúcej ukážke môžete vidieť dekleráciu premenných x a y typu integer. Syntax je podobná ako v programovacom jazyku PASCAL. Medzi základné typy patria:

- Integer
- Boolean

Príklad deklarácie:

```
declare
   x, y : integer
   b : boolean
```

Taktiež sa využívajú n-rozmerné polia v nasledujúcom tvare:

```
declare
   p: Array[a1, a2, ..., an] of integer
```

3.3.2 Always-section

Sekcia always definuje skratky, ktoré slúžia na stručné spísanie programu. Konkrétnejšie to sú premenné, ktoré definujú funkcie alebo podmienky. Takéto premenné sú známe ako transparentné premenné. Transparentné premenné poskytujú vhodný spôsob skrátenia výrazov, ktoré sa často vyskytujú v programe. Táto sekcia nie je nevyhnutná v tele programu UNITY. Transparentné premenné môžeme definovať následovne pomocou ||:

```
always
      decx = x > y
   ||
      decy = y > z
```

Tieto premenné je možné zapísať aj jednoriadkovo bez použitia spojovníka:

```
always
```

```
  decx, decy = z > y, y > z
```

3.3.3 Initially-section

Initially sekcia je súbor rovníc, ktoré definujú počiatočné hodnoty pre niektoré programové premenné. Premenné, ktoré nie sú inicializované majú ľubovoľné počiatočné hodnoty. Premenné x a y môže byť definované:

```
initially
     x = X
  ||
     y = Y
```

alebo takto:

```
initially
  x, y = X, Y
```

3.3.4 Assign-section

Táto sekcia je konečná a neprázdna množina, ktorá sa skladá z konečných príkazov, tie sú oddelené znakom []. Znak [] predstavuje nedeterminizmus. Príkazy sa skladájú z konečných priradení, tie sú oddelené znakom ||. Tento znak predstavuje paralelizmus. Príklady:

```
  assign
     x, y := 1, 2
  [] x, y := y, x if x > y
```

V týchto príkladoch môžeme vidieť dve rôzne priradenia, podmienené a nepodmienené. Podmienené priradenie obsahuje podmienku *if* $x > y$, ktorá musí byť splnená ak dané priradenie má byť vykonané. V tomto konkrétnom príklade ak je x väčšie ako y tak x sa rovná y a y sa rovná x. Nepodmienené priradenie je teda jednoduché priradenie, v ktorom nie je žiadna podmienka a priradenie sa vykoná. Ďaľšie priradenia, ktoré sa môžu v assign sekcii vyskytnúť sú **kvantifikované priradenia** a **kvantifikované výrazy**.

Kvantifikované priradenia

Kvantifikované priradenia slúžia na zapísannie konečnej množiny priradení. Príklad:

```
assign
   <|| i, j : 1 =< i, j =< 10 :: A[i, j] := 0 >
resp.
   <[] i, j : 1 =< i, j =< 10 :: A[i, j] := 0 >
```

Z tohto príkladu môžeme vidieť, že sa jedná o dvojtý for cyklus, ktorý prechádza dvajrozmerné pole A. Časť *i, j* nám hovorí o vytvorení lokálnych premenných, ktoré sa kontrolujú booleovskou podmienkou *1 =< i, j =< 10*. Ak je podmienka splnená vykoná sa daný príkaz *A[i, j] := 0*. Znak || nám hovorí, že sa ma príkaz bude vykonávať paralelne a znak [], že sa bude vykonávať nedeterministicky. Od takéhoto priradenia požadujeme aby sa nestal zacykleným resp. bol konečným a v prípade, že bude obsahovať vnorené kvantifikované priradenie musí byť konečné.

Kvantifikované výrazy

Kvantifikované výrazy obsahujú binárnu operáciu. Napríklad tento výraz

```
assign
   <max i, j, k : 1 =< i, j, k =< N :: A[i, j, k] >
```

nám vráti maximálny prvok z trojrozmenrného poľa A. Ak nám daný výraz vráti prázdnu množinu tak nadobúda hodnotu, ktorá patrí neutrálnemu prvku operácie. Binárne operácie a jej neutrálny prvok sú:

Binárna operácia	Neutrálny prvok
min	∞
max	-∞
$+$	0
*	1
$\wedge$	true
$\vee$	false
$\equiv$	true

3.3.5 Vykonanie programu UNITY

Na začiatku programu je najdôležitejšia declare sekcia, kde sa deklerujú všetky premenné. Následne na to program postupuje do initial sekcie, tú sa deklerované premenné inicializujú, tie ktoré inicializované nie sú nadobúdajú náhodnú hodnotu. Ak je zadefinovaná always sekcia vykoná sa aj tá, priradia sa všetky transparentné premenné. V poslednom kroku program pokračuje do assing sekcie. Spustí sa nekonečný cyklus nedeterministického vyberania príkazov, ktoré obsahujú priradenia vykonávajúce sa paralelne. Každý z týchto krokov v assing sekcii sa vykonáva nekonečne veľa krát.

Pevný bod

Počas vykonávania programu môže nastať situácia vytvorenia tzv. **pevného bodu**, alebo **Fixed Point**. Tento bod predstavuje stav programu, kedy sa aktuálny stav programu už nikdy nezmení. V našom prípade to bude znamenať ukončenie programu.

3.3.6 Ukážka programu

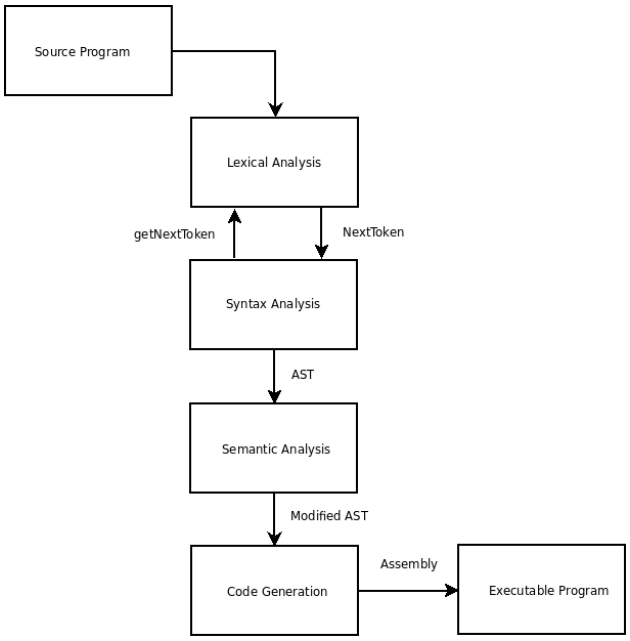
Nasledujúci UNITY program predsavuje Euclidovský algoritmus pre nájdenie najväčšieho spoločného deliteľa čísel X, Y:

Program *GCD*
declare x,y : *natural*
initially $x, y = X, Y$
assign $x := x - y$ **if** $x > y$
[] $y := y - x$ **if** $y > x$
end

Obr. 3.2: GCD algoritmus

3.4 Kompilátory

Softvér, ktorý transformuje zdrojový kód v jednom programovacom jazyku do ekvivalentného zápisu v inom programovacom jazyku. Pojem kompilátor sa najčastejšie používa pre prekladače, ktorých zdrojovým jazykom je vysokoúrovňový jazyk a cieľovým jazykom je nízkoúrovňový jazyk (napr. jazyk symbolických inštrukcií, objektový kód alebo strojový kód) za účelom vytvorenia spustiteľného programu [3]. Výhodov skompilovanej aplkácie je podpora na rôznych zariadeniach. Kompilované programy sú rádovo 10 - 100x rýchlejšie ako interpretované programy.



Obr. 3.3: Kopmilátor

3.4.1 Štruktúra kompilátoru

Kompilátor sa skladá z troch častí: front-end, middle-end a back-end.

1. Front-end (Predná časť) - overuje syntax a sémantiku špecifickú pre daný zdrojový kód. Pokiaľ je program na vstupe syntakticky nesprávny, obsahuje syntaktickú chybu, kompilátor by mal vhodným spôsobom na to reagovať. Predná časť spravidla zahŕňa lexikálnu analýzu, syntaktickú analýzu a sémantickú analýzu. Výstupom prác prednej časti kompilátoru býva program v intermediárnom kóde, ktorý je poskytovaný na spracovanie nasledujúcim častiam kompilátora.

2. Middle-end (Stredná časť) - vykonáva optimalizácie nad intermediárnym kódom. Tieto optimalizácie sú nezávislé na architektúre cieľového počítača. Príkladom optimalizácií v strednej časti prekladu je odstraňovanie zbytočných alebo nedosiahnuteľných častí kódu, či optimalizácia

cyklov. Výstupom tejto časti kompilátora je öptimalizovanýïntermediárny kód, ktorý je následne používaný zadnou časťou kompilátora.

3. Back-end (Zadná časť) - môže vykonávať dodatočnú analýzu a optimalizácie, ktoré sú špecifické pre cieľový počítač. V každom prípade je však jej hlavnou úlohou generovanie cieľového kódu. Typicky je jej výstupom strojový kód pre konkrétny procesor.

3.4.2 Lexikálna analýza, syntaktická analýza a sémantická analýza

Lexikálna analýza

Lexikálna analýza je činnosť, ktorú má na starosť tzv. lexikálny analyzátor - je súčasťou prekladača (komplilátora). Lexikálny analyzátor rozdelí vstupnú postupnosť znakov na **lexémy** - lexikálne jednotky (napr. identifikátory, čísla, kľúčové slová, operátory). Tieto lexémy sú reprezentované ve forme tokenov (symbolov), tie sú poskytnuté ku spracovaniu syntaktickému analyzátoru.

Lexémy sú základné symboly programovacieho jazyka, patria sem identifikátory, kľúčové slová, konštanty rôznych typov, operátory.

Syntaktická analýza

Syntaktická analýza sa v informatike nazývá proces analýzy postupnosti formálnych prvkov s cieľom určiť ich gramatickú štruktúru voči predom danej formálnej gramatike. Program, ktorý vykonává tuto úlohu, se nazývá syntaktický analyzátor (parser) - vstupný text transformuje na určité datové struktury, syntaktický strom, ktorý zachováva hierarchické usporiadanie vstupných symbolov, ktoré sú vhodné pre daľšie spracovanie.

Sémantická analýza

Sémantická analýza postupne prechádza symboly či skupiny symbolov získané zo syntaktickej analýzy a priraďuje sa im význam. Pokiaľ napríklad skupina symbolov predstavuje použitie konkrétnej premennej, tak analyzátor zisťuje či je premenná už deklarována a či je správne použita vzhľadom k jej datovému typu.

3.4.3 Syntaktický strom

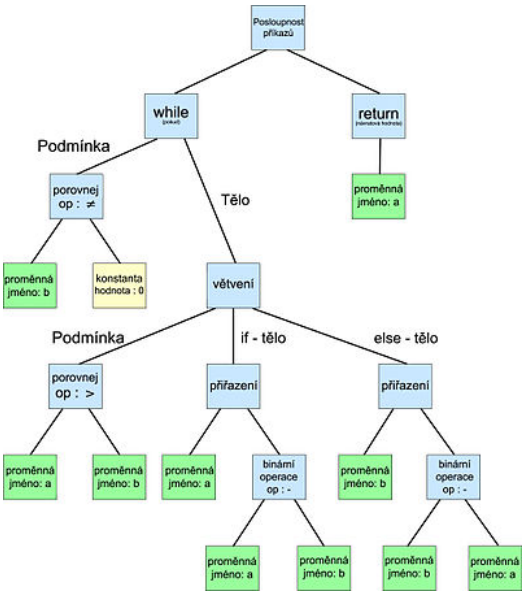
Abstraktný syntaktický strom je v informatike stromovou reprezentaciou abstraktnej syntaktickej štruktúry zdrojového kódu napísaného v programovacom jazyku. Abstraktný syntaktický strom se využívá primárne na preklad a optimalizaciu kódu.

Štruktúra syntaktického stromu

- vnútorné uzly stormu sú operátory
- listy stromu sú jeho operandy
- každá časť podstromu je samostatnou logickou jednotkou

Nasledujúci obrázok vychýdza z daného kódu

```
while b != 0:
   if a > b:
      a = a - b
   else:
      b = b - a
return a
```



Obr. 3.4: Syntaktický strom

3.5 Interprétre

Interpreter je počítačový program, ktorý interpretuje iný program napísaný v nejakom programovacom jazyku. Interpretre môžu program interpretovať riadok po riadku alebo preložia program do nejakého medzikódu a tento medzikód potom vykonávajú. Ak je v programe syntaktická chyba, prvý druh interpretra vykoná program po túto syntaktickú chybu a zahlási chybu na príslušnom riadku, druhý druh interpretra program nezačne vykonávať a počas prekladu do medzikódu zahlási chybu.

Programy vykonávané interpretrom bežia väčšinou pomalšie, ako kompilované programy.

Interpreter môže byť vo všeobecnosti interpretátorom ľubovoľného formalizovaného jazyka, napríklad interpreter matematických výrazov. Známe interpretre: PHP, Python MATLAB, Perl...

3.5.1 Model checking

Overovanie modelov alebo model checking je automatizovaná metóda formálnej verifikácie paralelného systému s konečným počtom stavov. Kontroluje sa, či zadaný model vyhovuje špecifikácii. Model sa zadáva ako systém prechodov stavov, kde vrcholy sú stavy, a postupnosť prechodov predstavuje vykonávanie správania sa modelu. Špecifikácia systému sa zadáva formulami temporálnej logiky. Výsledkom verifikácie je odpoveď na otázku, či model spĺňa špecifikáciu cite [4].

Temporálna logika je odvetvie logiky, ktoré skúma logickú štruktúru výrokov o čase s ktorými klasická výroková alebo predikátová logika nedokážu plnohodnotne narábať.

Výhody

- Žiadne dôkazy
- Rýchlosť
- Kontra príklady
- Žiadne problémy s čiastočnými špecifikáciami
- Logika môže vyjadriť veľa súbežných vlastností

Nevýhody

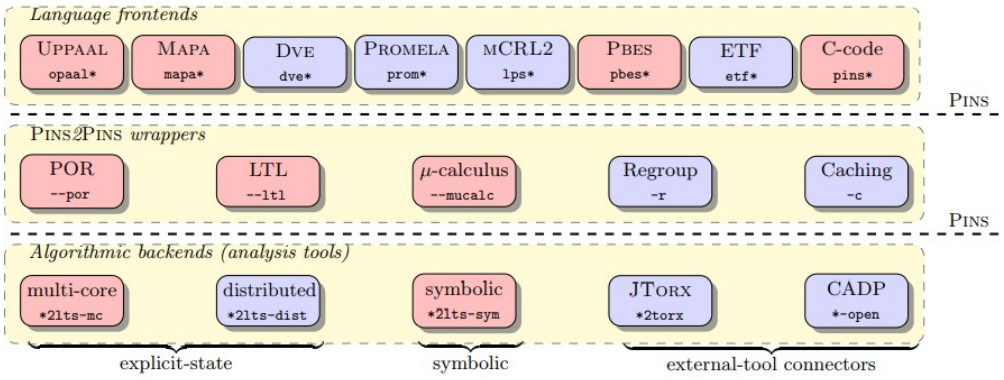
- Príliš veľa procesov
- Dátové cesty

3.5.2 LTSmin

LTSmin (model checker) začal ako všeobecná sada nástrojov na manipuláciu s označenými prechodovými systémami. Medzitým bola sada nástrojov rozšírená na plný overovací model, pri zachovaní jeho jazykovo nezávislých charakteristík.

Na získanie svojho vstupu LTSmin spája značný počet existujúcich overovacích nástrojov: muCRL , mCRL2 , DiVinE , SPIN (SpinS), UPPAAL (opaal), SCOOP , PNML , ProB a CADP.

LTSmin má modulárnu architektúru, ktorá umožňuje prepojenie viacerých front-end modelovacích jazykov s rôznymi analytickými algoritmami prostredníctvom spoločného rozhrania. Poskytuje symbolické aj explicitné algoritmy analýzy viacerých jazykov, ktoré umožňujú viaceré spôsoby riešenia problémov pri konflikte. Toto prepojovacie rozhranie sa nazýva Partitioned Next-State Interface (PINS), ktorého základom je definícia stavového vektora, počiatočný stav, funkcia NextState a funkcie označovania.



Obr. 3.5: PINS architektúra

Back-ends

LTSmin ponúka rôzne analytické algoritmy zahŕňajúce tri algoritmické backendy:

- Distribuované inštancie
- Viacjadrový model
- Symbolická kontrola modelu

Front-ends

LTSmin už spája značný počet existujúcich overovacích nástrojov ako jazykových modulov, čo umožňuje používať ich modelové formalizmy:

- muCRL
- mCRL2
- DiVinE
- SpinS
- UPPAAL
- SCOOP
- PNML
- ProB
- CADP

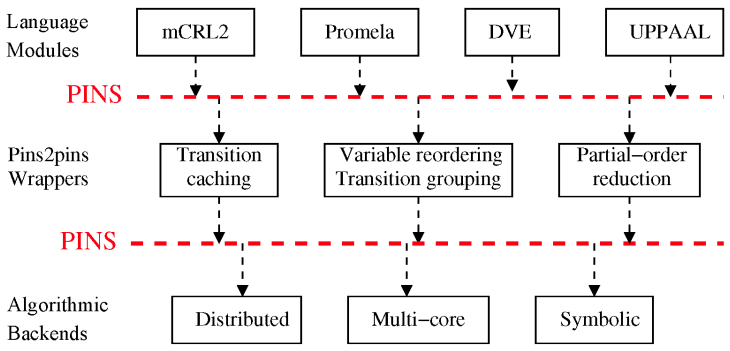
PINS2PINS

Rozhranie PINS čisto rozdeľuje naše kontrolné nástroje do dvoch nezávislých častí:

- jazykové moduly
- algoritmy kontroly modelov

Umožňuje však aj vytvorenie modulov PINS2PINS, ktoré sa nachádzajú medzi jazykovým modulom a algoritmom. Tieto moduly PINS2PINS môžu využívať všetky algoritmické backendy a môžu byť zapnuté a vypnuté na požiadanie:

- Transition caching do vyrovnávacej pamäte zvyšuje pomalé jazykové moduly
- Regrouping urýchľuje symbolické algoritmy pomocou optimalizácie závislostí
- Partial-order znižuje stavový priestor tým, že klesne irelevantné prechody



Obr. 3.6: PINS2PINS

Kapitola 4

Implementácia

Kapitola 5

Interpreter UNITY

Kapitola 6

Záver

Literatúra

[1] A UNITY - Style Programming Logic for a Shared Dataspace Language - H. Conrad Cunningham and Gruia-Catalin Roman - 1989

[2] Mechanizing UNITY in Isabelle - Lawrence C. Paulson - 2000

[3] PC Mag Staff. Encyclopedia: Definition of Compiler. 28 February 2017

[4] Baier, C., Katoen, J.: Principles of Model Checking. 2008.

[5] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom1 and Tom van Dijk: LTSmin: High-Performance Language-Independent Model Checking

Zoznam obrázkov