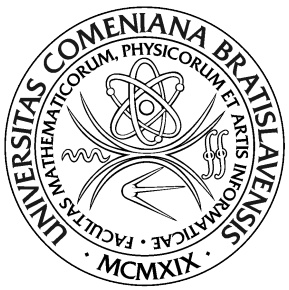
UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

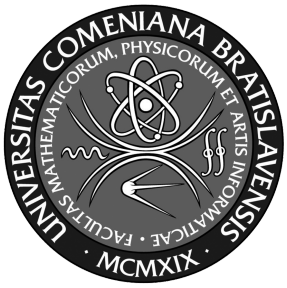


VERIFIKAČNÝ NÁSTROJ PRE FORMALIZMUS UNITY

Diplomová práca

2019 Bc. Filip Špaldoň

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY



VERIFIKAČNÝ NÁSTROJ PRE FORMALIZMUS UNITY

Diplomová práca

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

Bratislava, 2019 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, 2019 Bc. Filip Špaldoň

Poďakovanie

Ďakujem školiteľovi doc. RNDr. Damasovi Gruskovi, PhD. za rady a pomoc počas riešenia mojej diplomovej práce. Ďakujem aj svojím blízkym za morálnu podporu.

Abstrakt

Unbounded **N**ondeterministic **I**terative **T**ransformations of the program state (UNITY) je programovací jazyk popísaný v knihe Parallel Program Design - A Foundation a jeho autormi sú K. Mali Chandy a Jayadev Misra z University of Texas. Jeho hlavnými znakmi sú nedeterminizmus, paralelné operácie, absencia control-flow, synchrónnosť a asynchrónnosť, stavy a priradenia.

Všetky tieto znaky bolo potrebné aplikovať do vytvoreného interpretera, ktorý sa staral o činnosť verifikátoru. Práca popisuje podrobnú implementáciu lexikálneho a syntaktického analyzátoru interpretera. Tiež obsahuje popis skúmaných knižníc a nástrojov LTSmin, výsledné použité knižnice S2N a NuSMV. Výsledný verifikátor dokáže pracovať s logikou výpočtového stromu (CTL). Všetky tieto ciele sme mohli dosiahnuť vďaka webovej aplikácií postavenej na platformách GO, React, ktoré boli použité na implementáciu verifikátora.

Kľúčové slová : UNITY, CTL, LTSmin, S2N, NuSMV, intepreter, verifikátor

Abstrakt EN

Unbounded **N**ondeterministic **I**terative **T**ransformations of the program state (UNITY) is a programming language described in the book Parallel Program Design - A Foundation and the authors are K. Mali Chandy and Jayadev Misra from the University of Texas. Its main features are nondeterminism, paralell operations, absence of control flow, synchronity and asynchronity, states and assignments.

All these features were needed to be applied to the created interpreter that was managing the action of the verificator. The thesis describes a detailed implementation of the lexical and syntactic analyser of the interpreter. It also contains a description of the studied libraries and LTSmin tools, the resultant used libraries S2N and NuSMV. The resultant verificator is able to work with the logics of computational tree (CTL). All these aims were reached thanks to a web application based on the platform GO, React, which were used for the implementation of the verificator.

Keywords : UNITY, CTL, LTSmin, S2N, NuSMV, intepreter, verificator

Obsah

Zoznam obrázkov

Zoznam tabuliek

Kapitola 1

Úvod

UNITY je teoretický jazyk vytvorený za účelom riešenia paralelných problémov. V dnešnej dobe existuje iba málo takých paralelných programovacích jazykov, ktoré sa snažia dané problémy vyriešiť. Avšak neobsahujú všetky potrebné prvky na ich úplne vyriešenie. Preto je tu návrh za vytvorenie verifikačného nástroju podporujúci zápis a verifikáciu programov zapísaných v jazyku UNITY.

V nasledujúcich kapitolách tejto diplomovej práce sa dozviete všetky potrebné informácie o samotnom programe UNITY, návrhu a implementácie interpretera, ktorý je jadrom verifikačného nástroju, skúmanie použitých nástrojov a knižníc na dosiahnutie požadovaných výsledkov.

Kapitola 2

Východiská

V tejto kapitole je cieľom poskytnúť prehľad základným pojmom a postupov pri tvorbe verifikačného nástroja pre programovaní jazyk UNITY. Predpokladá sa znalosť Kripkeho štruktúry a sémantiky [11].

2.1 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 University of Texas. 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. [1]

2.2 Charakteristické vlastnosti UNITY

- Nedeterminizmus

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

2.2.1 Nedeterminizmus

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

2.2.2 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.

2.2.3 Synchrónnosť a asynchrónnosť

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

2.2.4 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.

2.3 Telo programu

UNITY obsahuje štyri základné sekcie: deklaráciu premenných, množinu skratiek, počiatočné hodnoty premenných a množinu priraďovacích príkazov. V tele programu sa tieto sekcie vyskytujú pod názvami **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. UNITY program má nasledujúcu formu: [6]

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

2.3.1 Declare-section

Táto sekcia obsahuje deklaráciu premenných použité v programe a ich súvisiace typy. V nasledujúcej ukážke môžete vidieť deklaráciu premenných `x`, `y` typu `integer` a `b` typu `boolean`. Syntax je podobná ako v programovacom jazyku PASCAL.

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

Medzi základné typy patria:

- Integer
- Boolean

Taktiež sa využívajú n-rozmerné polia v nasledujúcom tvare:

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

2.3.2 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 nasledovne pomocou `||`:

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

Tieto premenné je možné zapísať aj jednoriadkovo bez použitia spojovníka:

```
always
   decx, decy = x > y, y > x
```

2.3.3 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 = 10
  ||
     y = 5
```

alebo takto:

```
initially
  x, y = 10, 5
```

2.3.4 Assign-section

Táto sekcia je konečná a neprázdna množina, ktorá sa skladá z konečných príkazov, tie sú oddelené znakom `[]`. Znak `[]` predstavuje nedeterminizmus. Príkazy sa skladajú 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. Ďalšie priradenia, ktoré sa môžu v assign sekcii vyskytnúť sú **kvantifikované priradenia** a **kvantifikované výrazy**.

Kvantifikované priradenia

Kvantifikované priradenia slúžia na zapísanie 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 dvojitý for cyklus, ktorý prechádza dvojrozmerné 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 trojrozmerného poľa `A`. Ak nám daný výraz vráti prázdnu množinu tak nadobúda hodnotu, ktorá patrí neutrálnemu prvku operácie. Binárne operácie a jej neutrálny prvok môžete vidieť v tabuľke 2.1.

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

Tabuľka 2.1: Binárne operácie

2.3.5 Vykonanie programu UNITY

Na začiatku programu je najdôležitejšia declare sekcia, kde sa deklarujú všetky premenné. Následne na to program postupuje do initial sekcie, tú sa deklarované 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 [2].

2.3.6 Vlastnosti programu

Pevný bod

Počas vykonávania programu môže nastať situácia vytvorenia tzv. **pevného bodu**, alebo **Fixed Point** (FP). Tento bod predstavuje stav programu, kedy sa aktuálny stav programu už nikdy nezmení.

Leads-to

Lead-to označujeme znakom implikácie $\rightarrow$ a vzťahuje sa na viac krokov programu. Uvažujme, že máme program F a výraz $p \rightarrow q$, kde p a q sú výroky. Ak teda pre p nastane, že je $true$ tak q je alebo bude $true$. Môže ale nastať situácia, že p bude $false$ práve vtedy, keď q bude $true$.

2.3.7 Ukážka programu UNITY

```
Program         GCD
   declare      x, y : integer
   initially    x, y = X, Y
   assign       x := x - y if x > y
            ||  y := y - z if y > x
end
```

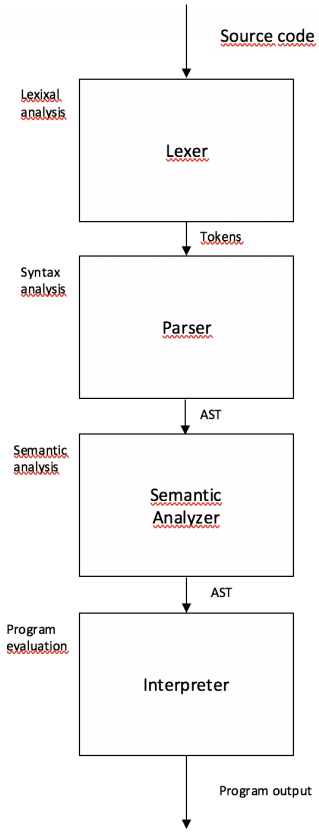
2.4 Interpreter

Interpreter (obr. 2.1) 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 interpretera vykoná program po túto syntaktickú chybu a zahlási chybu na príslušnom riadku, druhý druh interpretera program nezačne vykonávať a počas prekladu do medzikódu zahlási chybu.

Programy vykonávané interpreterom 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.



Obr. 2.1: Štruktúra interpretera

2.4.1 Štruktúra interpretera

Interpreter sa skladá z troch častí: front-end, middle-end a back-end [3].

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

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 interpretera je optimalizovaný intermediárny kód, ktorý je následne používaný zadnou časťou interpretera.

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.

2.4.2 Lexikálna analýza, syntaktická analýza a sémantická analýza

Lexikálna analýza

Lexikálna analýza je činnosť, ktorú má na starosť tzv. lexikálny analyzátor - je súčasťou prekladača. Lexikálny analyzátor rozdelí vstupnú postupnosť

znakov na **lexémy**. Tieto lexémy sú reprezentované vo 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 nazýva proces analýzy postupnosti formálnych prvkov s cieľom určiť ich gramatickú štruktúru voči predom danej formálnej gramatike. Program, ktorý vykonáva tuto úlohu, sa nazýva syntaktický analyzátor (parser) - vstupný text transformuje na určité dátové štruktury, syntaktický strom, ktorý zachováva hierarchické usporiadanie vstupných symbolov, ktoré sú vhodné pre ďalšie 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 použitá vzhľadom k jej dátovému typu.

2.4.3 Syntaktický strom

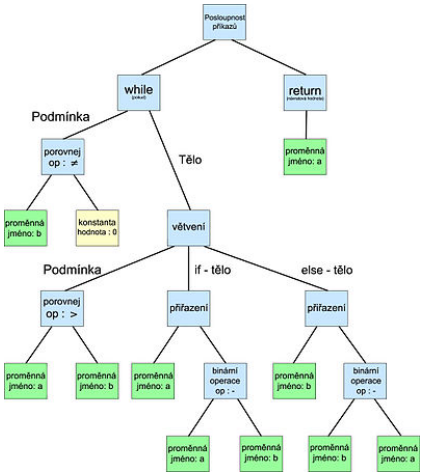
Abstraktný syntaktický strom je v informatike stromovou reprezentáciou abstraktnej syntaktickej štruktúry zdrojového kódu napísaného v programovacom jazyku. Abstraktný syntaktický strom sa využíva primárne na preklad a optimalizáciu kódu.

Štruktúra syntaktického stromu

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

Nasledujúci obrázok 2.2 vychádza z daného kódu

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



Obr. 2.2: Syntaktický strom

2.4.4 Model checking

Overovanie modelov alebo model checking je automatizovaná metóda formálnej verifikácie paralelného systému s konečným počtom stavov. Kontro-

luje 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 [4].

2.4.5 Temporálna logika

Tento špeciálny typ logiky je efektívnym nástrojom skúmania procesov v informatike, kde jednotlivé stavy systému (počítača, programu, atď.) majú časovú následnosť a podmienenosť. Kripkeho sémantika pre temporálnu logiku poskytuje transparentnú metódu pre pravdivostné ohodnotenie formúl, ktoré špecifikujú stavy informatického systému. Kripkeho model $M = (W, R, v)$ je teraz zjednodušene špecifikovaný tak, že množina W je nahradená lineárne usporiadanou množinou časových okamžikov (bodov) $T = \{t_1, t_2, t_3, ...\}$, pričom $0 \leq t_1 < t_2 < t_3 < ...$ [10].

2.4.6 CTL*

CTL* je temporálna logika, ktorá v sebe spája lineárnu časovú logiku LTL a logiku výpočtového stromu CTL. Vieme ňou vyjadriť všetko čo dokáže vyjadriť LTL a CTL. Spolu s tým dokážeme vytvárať formule, ktoré LTL a CTL nedokážu. Avšak zaobchádzať s touto logikou je dosť zložité. Preto je vhodnejšie pracovať jednotlivo buď s LTL alebo CTL logikou.

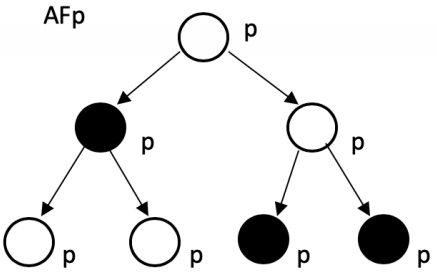
LTL - Linear Time Logic

Lineárna časová logika LTL rozširuje výrokovú logiku a je reprezentovaná stromom, ktorý má jedinú vetvu. Táto vetva obsahuje stavy programu, ktoré

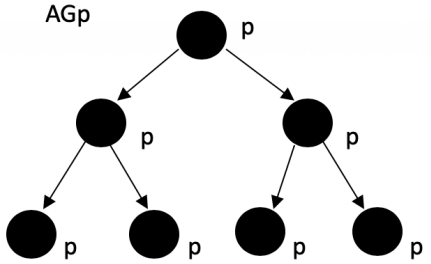
sú závislé na čase. LTL navyše od výrokovej logiky obsahuje aj temporálne operátory ako je napríklad **X** (ne**X**t), ktorým sa pýtame na nasledujúci stav. Medzi ďalšie operátory patrí **U** (**U**ntil), **F** (**F**innaly) a **G** (**G**lobally)

CTL - Computation Tree Logic

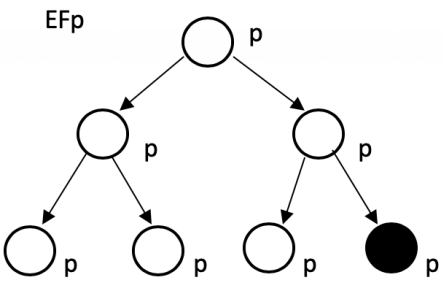
Je logika, ktorá pracuje na úrovni stromov, ktorá dokáže zabezpečiť aby niečo platilo vo všetkých stavoch alebo existuje aspoň jeden stav, pre ktorý to platí. K tomu nám pomáhajú kvantifikátory ∀ a ∃. Pomocou spojenia týchto kvanfikátorov a temporálnych operátorov z LTL nám vznikne CTL - logika výpočtového stromu. Takýto strom sa skladá zo stavov a hrán, ktoré vedú k ďalším stavom. Počet stavov a teda úrovní daného stromu je nekonečný. Jednotlivé kombinácie kvantifikovaných a temporálnych operátorov môžeme vidieť na obrázkoch 2.3, 2.4, 2.5 a 2.6.



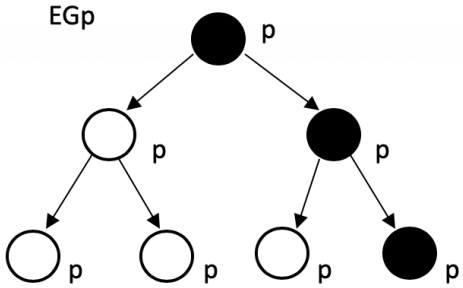
Obr. 2.3: Formula AFp



Obr. 2.4: Formula AGp



Obr. 2.5: Formula EFp



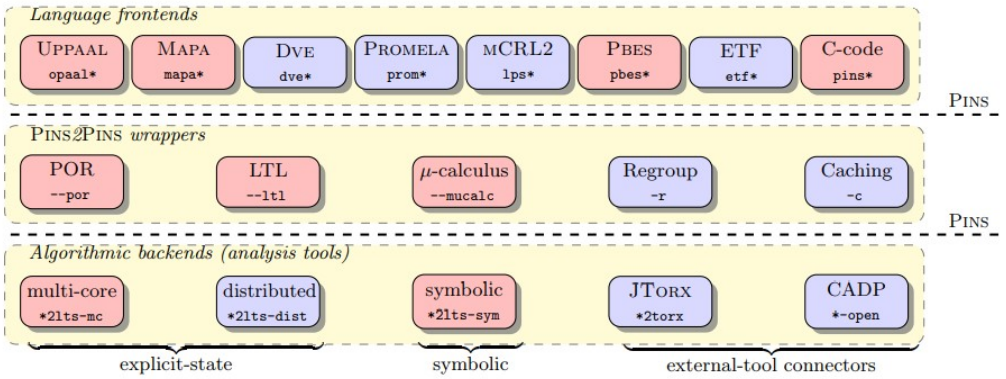
Obr. 2.6: Formula EGp

2.4.7 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, obr. 2.7), ktorého základom je definícia stavového vektora, počiatočný stav, funkcia NextState a funkcie označovania [5].



Obr. 2.7: PINS architektúra (obrázok bol prevzatý zo zdroja [5])

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

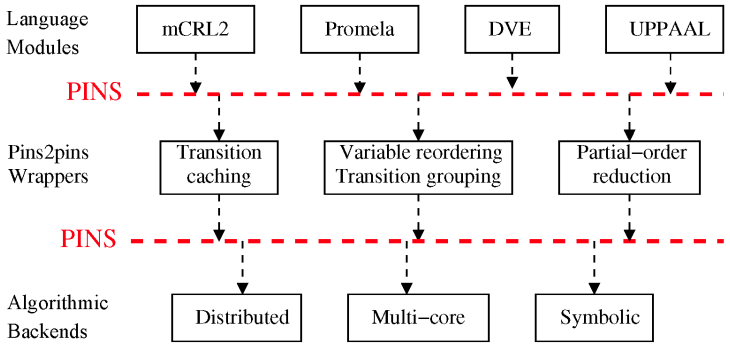
PINS2PINS

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

- jazykové moduly
- algoritmy kontroly modelov

Umožňuje však aj vytvorenie modulov PINS2PINS (obr. 2.8), 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. 2.8: PINS2PINS (obrázok bol prevzatý zo zdroja [5])

Kapitola 3

Interpreter UNITY

Táto kapitola obsahuje popis postupu vytvárania interpretera pre verifikačný nástroj UNITY, prieskum dostupných knižníc na verifikáciu CTL logiky a ich základné funkcie a vlastnosti.

3.1 Základné informácie

V predchádzajúcej kapitole sme sa dozvedeli základné pojmy interpretera, knižníc, ktoré sú k dispozícií a samotného programu UNITY. Teraz sme všetky tieto znalosti uviedli do teoretického plánu pre verefikačný nástroj pre UNITY. Čo sme si predstavovali pod pojmom verifikačný nástroj? Išlo o nástroj, ktorý mal kontrolovať, či daný program spĺňa všetky spomenuté vlastnosti programu (FP, Unless, Ensures, Leads-to). K dosiahnutiu potrebných výsledkov bolo za potrebu vytvoriť pre verifikačný nástroj interpreter, ktorý daný program načítal a spracoval. Vrátené dáta boli vyhodnotené na základe použitej knižnice zo zbierky nástrojov, ktoré poskytuje LTSmin.

3.2 Úprava jazyku UNITY

Pre ciele tejto práce bolo potrebné zmeniť niektoré syntaktické prvky a vlastnosti jazyku UNITY:

- Náhodné hodnoty neinicializovanej premennej
- Syntax always seckcie
- Syntax program seckie

3.2.1 Náhodné hodnoty neinicializovanej premennej

Ako už vieme ak je deklarovaná premenná ale nie je inicializovaná podľa vlastností UNITY by mala táto premenná nadobudnúť náhodnú hodnotu, avšak pri riešení niektorých úloh by táto vlastnosť mohla dané výsledky programu znehodnotiť. Problém nastával v rozsahu náhodných čísiel. Ak bola daná premenná deklarovaná ako integer nevedeli sme presne určiť najsprávnejšie riešenie pre tento rozsah čísiel. Preto sme túto vlastnosť upravili nasledovne. V implementácií bolo potrebné každú deklarovanú premennú inicializovať. Týmto sme sa úplne vyhli náhodne generovaným hodnotám pre deklarované premenne.

3.2.2 Syntax always sekcie

Sekcia always slúži na vytvorenie tzv. transparentných premenných, ktoré sú definované názvom premennej na ľavej strane a na pravej strane sú jednotlivé výrazy. Toto priradenie bolo definované znakom rovnosti `=`. V našom prípade, aby parser interpretera nemal problém rozlišovať medzi operátormi `=>`, `<=`, `!=`, `==` a znakom rovnosti sme znak rovnosti zmenili na `:=` tak isto ako tomu je v assign sekcii.

3.2.3 Syntax program sekcie

V ukážke programu máme sekciu program, tá obsahuje program-name, názov programu, ktorý sa môže vynechať, lenže aby bol program kompletný zo strany syntaxe bola táto zložka povinná.

3.3 Postup pri vytváraní interpretera

3.3.1 Proces fungovania

Interpreter je jadro celého verifikačného nástroju pre programovací jazyk UNITY. Obsahuje tieto časti:

Spracovanie vstupného textu - telo programu

Interpreter vstupný text prečíta, rozdelí na tokeny, ktoré parser spracuje, skontroluje syntax a vyhodnotí či daný program neobsahuje chybu. Výsledkom parseru bude vygenerovanie abstraktného syntaktického stromu - AST.

AST

Vygenerovaný abstraktný syntaktický strom obsahuje celé telo programu. Koreň stromu je program a listy sú sekcie programu, tie obsahujú všetky priradenia danej sekcie. Algoritmus, ktorý tento strom prechádza vygeneruje cieľový kód pre zvolenú knižnicu.

Cieľový kód

Tento kód musí spĺňať všetky vlastnosti zvolenej knižnice. Každá z knižníc LTSmin využíva odlišný paralelný programovací jazyk alebo zápis programu

do špecifického modelu. Preto je potrebné cieľový kód upraviť podľa syntaktických pravidiel zvolenej knižnice. Výsledkom kódu je program sformalizovaný podľa zvolenej knižnice.

Vykonanie kódu

Knižnice obsahujú nástroje, ktoré vygenerovaný kód vykonajú. Výsledkami sú vlastnosti programu.

3.3.2 Skúmanie dostupných knižníc LTSmin

Pri výbere bolo viacero možností knižníc, ktoré zvoliť. Hlavnou podmienkou ale bolo aby knižnica spĺňala temporálnu logiku CTL.

DiViNe

Jedným z príkladov bol model checker DiViNe, ktorý je efektívny aj na veľmi náročné logické formule. DiViNe nie je náročný na výpočtovú silu a preto je vhodný na používanie na jednom počítači alebo sieti výpočtových staníc. Model môže byť vytvorený pomocou DVE modelovacieho jazyka. Avšak DiViNe je hlavne zameraný na temporálnu logiku LTL aj keď z istej časti dokáže pracovať s CTL nie je pre nás vhodný model checker, pretože aj zapisovanie do jazyku DVE nie je jednoduché a dostupná dokumentácia neobsahuje dostatok informácií.

opaal

Tento model checker zaručuje ciele, pre ktoré sa oplatí používať ako je napr. Rapid prototyping a Easy to learn. Je písaný v jazyku Python, čo by nám niekoľko vecí uľahčilo ale jeho hlavnými obmedzeniami sú, že dokáže pracovať

iba s časovými automatmi a je implementovateľný iba na Ubuntu 12.04 a Ubuntu 14.04.

SpinS

SpinS je model checker, ktorý bol pôvodne vytvorený z knižnice SpinJa, písaná v jazyku JAVA. Obe tieto knižnice sú pôvodne vytvorené z model checker-u Spin, ktorým je základom paralelný programovací jazyk Promela. Tento jazyk je jednoduchý a vysoko efektívny na logiku LTL. Hlavnou podmienkou práce bolo aby daná knižnica vedela pracovať na úrovni logiky CTL. To sa vďaka tomuto jazyku dá jednoducho docieliť. V prípade, že každé jedno priradenie z jazyku UNITY vložíme do nekonečného cyklu, bežiaceho vlákna zaručíme všetky potrebné vlastnosti pre logiku CTL a zároveň jazyku UNITY. Ak uvažujeme nad tým, že UNITY program sa skladná z množiny priradení a každé toto priradenie vložíme do samostatného vlákna, ktoré sa bude vykonávať nepretržite v istom okamihu sa zaručene dostaneme do FP a dosiahneme požadovaných výsledkov. Ibaže pri overovaní vlastnosti nám nastali komplikácie a z tohoto dôvodu sme danú knižnicu nevyužili na overovanie vlastností, ale iba testovanie a debugovanie programov písaných v Promela.

3.4 Promela - Spin

3.4.1 Promela

Je to verifikačný programovací jazyk zameraný na tvorbu modelov pre Spin model checker. Jeho základnými vlastnosťami sú:

- asynchrónne vykonávanie programu

- zdieľané premenné
- komunikačné kanále (channels)

Jeho úlohou je overiť logiku paralelných systémov. Na to slúži Spin, ktorý daný program vykoná. Nasledujúca tabuľka (3.1) zobrazuje dátové typy, ktoré Promela podporuje.

Name	**Size (bits)**	**Usage**	**Range**
bit	1	unsigned	0..1
bool	1	unsigned	0..1
byte	8	unsigned	0..255
mtype	8	unsigned	0..255
short	16	signed	-215..215 - 1
int	32	signed	-215..215 - 1

Tabuľka 3.1: Dátové typy

Popis jazyku Promela

Operátory a priradenia sú veľmi podobné ako v jazyku C alebo JAVA. Premenné musia byť definované typovo a jednotlivé priradenia oddelené bodkočiarkou.

```
int x = 10;
int a, b = 1, 2;
```

Taktiež môžu byť definované globálne premenné, alebo funkcie podobne ako v jazyku C za pomoci `define`.

```
#define N 10
#define sum(a, b) ((a) + (b))
```

Ak chceme takúto funkciu definovať viacriadkovo použijeme `inline`.

```
inline sum(a,b) {
    a + b
}
```

V Promela je možné vytvoriť proces `proctype`, ktorý je buď definovaný ako funkcia a následne spustená v `init` alebo priamo funkciu napísať a priamo ju pustiť za pomoci `active`.

```
byte n=0;
active proctype P() {
    n = 1;
    printf("Process P, n = %d\n", n);
}
proctype Q() {
    n = 2;
    printf("Process Q, n = %d\n", n);
}
init {
    run Q();
}
```

Je tiež možné vykonávať atomické operácie za pomoci `atomic` do ktorého stačí obaliť operácie, ktoré chceme vykonávať atomicky.

```
active proctype P() {
    atomic {
        n = 1;
        printf("Process P, n = %d\n", n);
    }
```

```
}
```

Cykly sa vykonávajú za pomoci `do`, ktoré obsahujú podmienky a v prípade, že sú `true` vykonajú sa priradenia. Každá nová podmienka a priradenie musí byť oddelené `::`. Takýto cyklus bude pokračovať nepretržite pokiaľ nebude zastavený pomocou `break`.

```
do
:: condition -> statement
...
:: a >= b -> max = a
od
```

Podmienené operácie uvedené v `if` sa vykonávajú nedeterministicky.

```
if
:: condition -> statement
...
:: a >= b -> max = a; branch = 1
:: a <= b -> max = b; branch = 2
fi;
```

Objekty sa vytvárajú nasledovne.

```
typedef MyStruct
{
    short Field1;
    byte  Field2;
}

init {
```

```
    MyStruct x;
    x.Field1 = 1;
}
```

3.4.2 Spin

Je verifikačný nástroj, ktorý podporuje dizajn a verifikáciu asynchrónnych procesov. Verifikačné modely sú zamerané na dokazovanie správnosti iterácií procesov [9]. V stručnosti nám Spin slúžil ako verifikátor pre jazyk Promela. Pomocou neho sme boli schopný testovať a debugovať jednotlivé programy. Taktiež nám vedel poskytnúť základné informácie o danom testovanom algoritme ako je napr. počet stavov, procesov a premenných použitých v programe. V našom prípade nám to pomohlo či je daný program napísaný správne a efektívne. Príklad programu:

```
mtype = {red, yellow, green};
mtype light=green;
init {
    do
    :: if
    :: light==red -> light=green
    :: light==yellow -> light=red
    :: light==green -> light=yellow
    fi;
    od
}
```

Daný program bolo možné spustiť príkazom `./spin tlight.pml`. Kde `tlight.pml` je súbor s programu s koncovkou `pml` známu pre Promela. Avšak

v tomto prípade sa program zacyklí a to z dôvodu, že program bol spustený v tzv. simulation mode. Pre plné využitie model checker-a je potrebné program spustiť v tzv. exhaustive verification mode, kedy využijeme všetky schopnosti model checker-a. K takémuto spusteniu sme mali dve možnosti:

```
spin -a tlight.pml
gcc -o pan pan.c
pan
```

alebo

```
spin -search tlight.pml
```

V prvom prípade nám Spin vytvoril súbor `pan.c` v jazyku C, ten sme museli následne skompilovať príkazom `gcc` a vráti nám spustiteľný súbor `pan`. V druhom prípade nám príkaz vráti priamo to isté ako súbor `pam` a tým je výsledok verifikácie programu. Ak pri verifikácií nastal akýkoľvek error vytvorí sa súbor `trail`, ktorý popisuje všetky informácie o prípadnom deadlocku alebo vzniknutej chybe. Do súboru `trail` je možné nahliadnuť príkazom `spin -t -p -l tlight.pml`. Ak by program obsahoval príkaz `printf` vydeli by sme niekoľko veľa výstupov.

```
...
printf("The light is now %e\n",light);
...
```

Aby sme tomuto zabránili a otestovali výstupy bolo potrebné obmedziť počet krokov vykonaných v programe a to príkazom `spin -u40 tlight.pml`, ktorý vykoná iba 40 krokov.

3.5 Výsledné použité knižnice

Po celkovom skúmaní knižníc sme dospeli k záveru, že ani jeden zo spomínaných model checher-ov nevyhovuje nášmu cieľu. Preto týmto skúmanie neskončilo a výsledné použité knižnice boli S2N, NuSMV a programovací jazyk Promela s pomocou Spin.

3.5.1 NuSMV

NuSMV je výsledkom reengineeringu a reimplementácie symbolického model checker-u SMV. NuSMV bol aktualizovaný v troch častiach oproti SMV:

- NuSMV má viacej funkcií, napr. CTL a LTL verfikáciu, ktoré zvyšujú schopnosť používateľa spolupracovať so systémom.
- Systémová architektúra NuSMV je vysoko modulárna a otvorená. Ďalšou vlastnosťou je, že v NuSMV môže užívateľ riadiť a prípadne meniť poradie vykonávania niektorých systémových modulov.
- Kvalita implementácie sa výrazne zvyšuje. NuSMV je veľmi robustný a dobre zdokumentovaný systém, ktorého kód je (relatívne) ľahko modifikovateľný.

NuSMV dokáže spracovať súbory napísané v jazyku SMV a umožňuje konštrukciu modelu s rôznymi metódami, analýzou dosiahnuteľnosti, kontrolou modelu CTL, výpočtom kvantitatívnych charakteristík modelu a generovaním protikladov [8]. Základné príkazy pre NuSMV:

- `NuSMV -int [filename].smv` - načíta daný súbor do interaktívneho shell-u
- `go` - prečíta a inicializuje NuSMV pre verifikáciu a simuláciu

- `pick_state [-v] [-r | -i]` - vyberie stav z množiny stavov
  - `-v` vypíše vybraný stav
  - `-r` vyberie náhodný stav
  - `-i` stav si vyberieme sami
- `simulate [-r | -i] -k steps` - od aktuálneho stavu vygeneruje počet stavov zadaných prepínačom `-k`
  - `-r` každý nový vygenerovaný stav zvolí náhodne
  - `-i` každý nový vygenerovaný stav si vyberieme sami
- `show_traces -v` - vypíše všetky vykonané kroky
- `reset` - resetuje celý systém
- `quit` - ukončí shell

Keďže NuSMV podporuje verifikáciu CTL zvolili sme si práve tento model checker na overovanie vlastností jazyku UNITY. Bude hlavnou súčasťou verifikátora v našej aplikácií. Avšak NuSMV vie pracovať iba s jazykom SMV, ktorý je na zápis UNITY programu príliš zložitý. Preto použijeme nástroj S2N na transformáciu z jazyka Promela na jazyk SMV.

3.5.2 S2N

Po skúmaní verifikačných nástrojov v zbierke LTSmin sme nenašli žiaden nám vyhovujúci nástroj. Väčšina týchto nástrojov nespĺňala požiadavku aby takýto verifikačný model checker dokázal overovať CTL logiku. Po hlbšom pátraní sme ale narazili na jeden nástroj, ktorý dokáže z modelu spísanom v

jazyku Promela resp. Spin vytvoriť CTL model. Tento nástroj sa volá S2N - Spin to NuSMV.

Spin a NuSMV sú dva najviac rozšírené model checker-y. Spin používa jazyk Promela, ktorý je určený na modelovanie synchronizácie a koordinácie procesov. Promela má syntax podobnú jazyku C, čo uľahčuje vytváranie čítanie modelov. NuSMV poskytuje jazyk na opis stavových prechodných systémov. Spin a NuSMV sú veľmi pôsobivé a úspešné vo svojich oblastiach. Zosúladenie ich výhod by bolo užitočné pre modelovanie. Preto vznikol nástroj S2N, ktorý dokáže preložiť modely v jazyku Promela do jazyka NuSMV. Toto spojenie prináša veľa výhod. Na jednej strane je modelovanie zložitého systému v NuSMV tvrdou prácou, ktorá je náchylná na chyby. Teraz máme pre model NuSMV jazyk vyššej úrovne. Na druhej strane S2N funguje tak, že rozširuje systém Spin o schopnosť kontrolovať CTL a ďalšie vlastnosti NuSMV [7].

S2N podporuje väčšinu základných funkcií jazyku Promela. Ako sú napr. polia, všetky dátové typy, do a if, proctype a aj komunikáciu pomocou kanálov. Avšak nepodporuje inline, typedef a printf, ktoré sme pre naše účely aj tak nepotrebovali. Všetky dostupné vlastnosti S2N nám vyhovovali na transformovanie Promela na NuSMV.

Kapitola 4

Implementácia

Kapitolu sme zamerali na opis implementácie webovej aplikácie, časti interpretera - verifikátoru UNITY a dostupných knižníc S2N, NuSMV.

4.1 Aplikácia

Prostredie pre vytvorenie samotnej aplikácie sme si zvolili webovú stránku. Táto stránka okrem aplikácie je uvádzaná aj ako prezentačná stránka našej diplomovej práce. Obsahuje všetky základné informácie o práci, postupe a zdrojov. Hlavná časť stránky je zameraná samotnej aplikácií teda verifikátoru pre UNITY. Aplikácia sa skladá z interpretera a knižníc S2N, NuSMV. Na zápis programu UNITY sme použili jednoduchý code editor, ktorý sa nachádza na podstránke Aplikácia, ktorú môžete vidieť na obrázku 4.1.

4.1.1 Back-end

Pre server-side sme si zvolili jazyk GO, známy aj ako Golang. Je to typovo orientovaný, kompilovateľný programovací jazyk, ktorý navrhol Google. Golang je syntakticky veľmi podobný jazyku C. Samotný jazyk je pôvodne na-

vrhnutý na vývoj webových aplikácií, avšak dá sa použiť rôzne. Nás osobne veľmi zaujal a tak sme si ho vybrali na vývoj aplikácie.

Web framework

Golang sám o sebe sa dá použiť bez akýchkoľvek framework-ov na vývoj webových aplikácií. Avšak takáto implementácia je obzvlášť zložitá z dôvodu rôznych modulárnych knižníc. Preto sme sa rozhodli použiť už existujúci web framework Buffalo. Tento framework je jednoduchý a pri tom výkonný na tvorbu web-u.

4.1.2 Front-end

Použili sme HTML5/CSS3 s pomocou React komponentov. Jednotlivé podstránky sú tvorené z HTML súborov, ktoré obsahujú React komponenty, tie nám zaručujú real-time rendering.

React

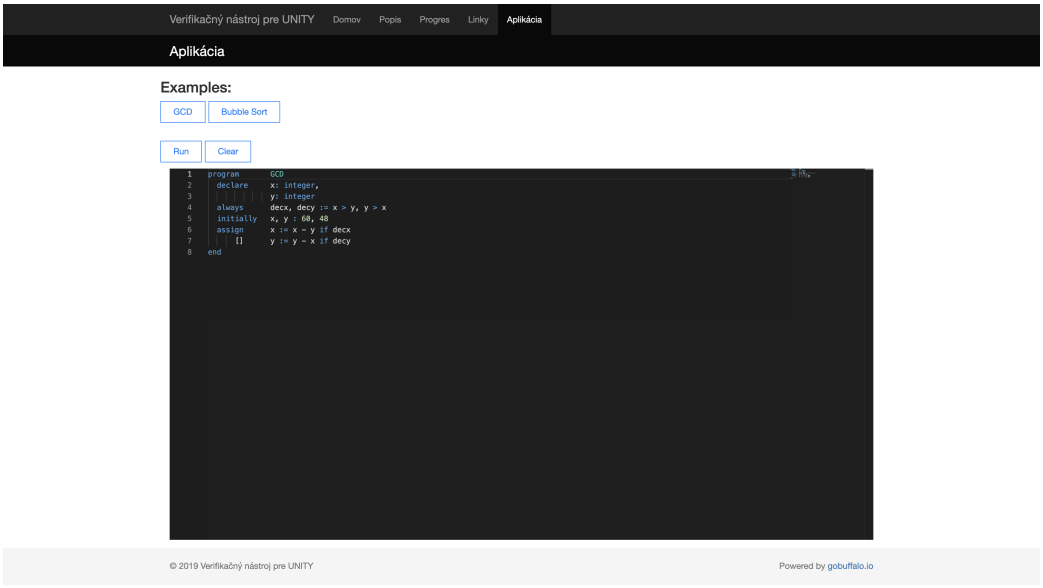
Je JavaScript-ová knižnica, ktorá dokáže v reálnom čase vykresľovať dané komponenty bez akéhokoľvek refreshu stránky. Časti stránky obsahujú dva zložitejšie komponenty: časovú os postupu práce a code editor.

4.1.3 Server - Heroku

Web framework Buffalo je už od začiatku vytvorenia nového projektu pripravený na nahratie na server. K tomu používa systém Heroku. Heroku je cloud-ová služba na správu webových aplikácií. Preto sme túto možnosť využili a celú aplikáciu sme nahrali na Heroku server. Aplikácia je neustále dostupná na tomto LINKU.

Zdrojové súbory aplikácie

Pre prípad potreby sme celú aplikáciu uložili na stránke GITHUB. Je potrebné mať nainštalovaný Golang a preň framework Buffalo. Samotný server sa spustí príkazom `buffalo dev`.



Obr. 4.1: Podstránka Aplikácia

4.2 Implementácia interpretera

Náš interpreter obsahuje lexikálnu analýzu, syntaktickú analýzu a generátor cieľového kódu.

4.2.1 Lexikálna analýza

Táto časť interpretera je najdôležitejšia, pretože načítava vstupný text programu UNITY. Text sa zadáva do code editor-u. Editor obsahuje dve hlavné tlačidlá RUN a CLEAR. RUN spustí samotný interpreter a CLEAR vyčistí obsah code editor-a.

Keď sa celý proces spracovania textu začne vytvorí sa trieda `Unity` (obr. 4.2), ktorá je jadrom interpretera. Táto trieda má niekoľko funkcií ale najdôležitejšou je funkcia `Parse`, ktorá využíva funkcie `Next` a `Scan`. Spolupráca `Next` a `Scan` nám vracia token-y, ktoré predstavujú buď čísla, slová (premenné, časti programu) alebo symboly (operátory).

`Parse` prechádza text postupne po sekciách programu a zapisuje jednotlivé premenné do lokálnej premennej `Variables`. Tieto premenné sú deklarované v sekcii **declare** programu UNITY. Ak program obsahuje aj sekciu **always**, tak jej transparetné premenné sa tiež pridajú k `Variables`. Následne prechádzame do sekcie **initially**. Okrem toho, že `Parse` kontroluje správnu syntax textu, zároveň kontroluje či dané premenné sú správne deklarované a počas vykonávania sekcie **initially** overuje, že tieto premenné boli aj správne inicializované. **Assign** sekcia je špeciálna tým, že obsahuje príkazy, ktoré predstavujú chod celého programu. Je to časť, kde sa využívajú všetky deklarované a inicializované premenné aby dosiahli nejakého výsledku. Preto sa jednotlivé príkazy pridávajú do lokálnej premennej `Body`. `Body` je tzv. array of dictionaries (pole slovníkov), ktoré okrem sekcii **assign** obsahuje aj ostatné sekcie. Kľúčom pre tieto slovníky sú názvy sekcií, hodnotou sú priradenia danej sekcie. Táto premenná sa ďalej používa v syntaktickej analýze

```
type Unity struct {
    Program   string
    Input     string
    Index     int
    Look      string
    Token     string
    Kind      int
    Position  int
    Variables map[string]interface{}
    Body      map[string]map[string]interface{}
    Tree      Node
}
```

Obr. 4.2: Trieda Unity

4.2.2 Syntaktická analýza

Syntaktická analýza má jednu dôležitú úlohu a to z celého programu UNITY vytvoriť abstraktný syntaktický strom. AST musel obsahovať všetky sekcie aby sme dokázali vygenerovať cieľový kód. Analyzátor využíva lokálnu premennú `Body`. Postupne prechádza každú sekciu a vytvára z nej AST. Avšak pri niektorých sekciách to bolo obzvlášť náročné.

K vytvoreniu AST nám pomáhala trieda `Node` (obr. 4.3). Obsahujúca niekoľko premenných, ktoré nám hovorili čo daný `Node` predstavuje. Napr. ak premenná `Statement` obsahovala nejakú hodnotu, tak to vždy bol `string`, hovoriaci čo sa bude diať s jeho podstromami. Mohla obsahovať `=, -, +, *, /, <, >, <=, >=, ==, !=`.

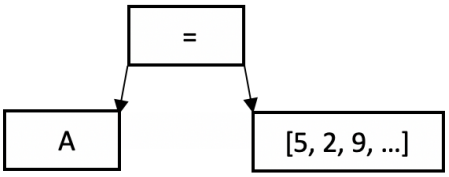
```
type Node struct {
    Root      bool
    Name      string
    Statement string
    Section   string
    Value     interface{}
    Ref       *Node
    Nodes     []*Node
}
```

Obr. 4.3: Trieda Node

Sekcia **always** môže obsahovať už zložitejšie výrazy napr. `decx, decy := x > y, y > x`. Premenná `decx` predstavuje výraz `x > y`. Takýto výraz sme museli rozdeliť na tri časti: podstrom kde `Statement` sa rovnal operandu `>` a jeho listy. Ľavým listom bola hodnota `x` a pravým listom hodnota `y`. Na vytvorenie podstromu v sekcii **always** sme používali pomocnú funkciu `makeAlwaysNode`.

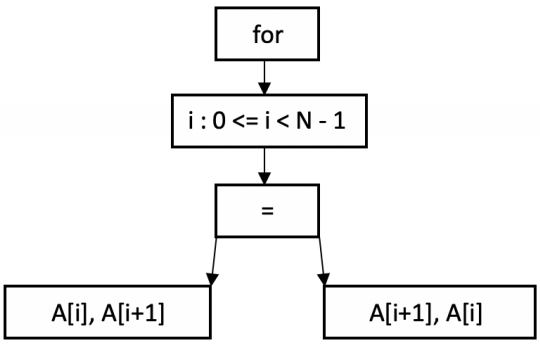
Initially obsahuje aj kvantifikované priradenia a výrazy. Príkladom je `<|| k: 0 <= k < N :: A[k] = rand() >`. Toto priradenie hovorí, že sa má

pole `A` naplniť náhodnými hodnotami. Ľavá strana od znaku `::` predstavuje for-cyklus od `0` po `N` a pravá strana samotné priradenie náhodnej hodnoty pre daný index poľa. Keď sme chceli priradenie zapísať do stromu museli sme pole A vytvoriť a inicializovať (obr. 4.4). Ľavú stranu sme inicializovali funkciou `forParserLeft`, ktorá nám vrátila iteráciu od `0` po `N`. Pole `A` sme potom vytvorili za pomoci `forParserRightInitially`.



Obr. 4.4: Initially - Pole A

Ak išlo o sekciu **assign** tá obsahovala príkazy - množinu priradení. Priradenia sme museli rozdeľovať na bežné a kvantifikované. Bežné sme zapisovali ako rovnosť ľavej a pravej strany, avšak pravá strana mohla obsahovať podmienku `if`. Podmienka bola priradená do `Node` ako referencia (`Ref`). Kvantifikované priradenia sme vytvorili ako for-cyklus so svojimi jednotlivými priradeniami (obr. 4.5).



Obr. 4.5: Kvantifikované priradenie v AST

4.2.3 Generátor cieľového kódu

V tomto okamihu sme už mali vytvorený AST, ktorý sme museli následne transformovať do cieľového kódu. Pre vytvorenie kódu sme využili funkciu `MakePromela`.

Na prechádzanie stromu sme použili algoritmus DFS (Depth-first search). Postupne sme zapisovali do lokálnej premennej `pom` všetky sekcie programu. Deklarované premenné sme definovali ako globálne premenné, pričom sme ich museli inicializovať v časti `init`. Nástroj S2N nepodporuje inicializácie globálnych premenných. Ako sme už spomínali bolo potrebné aby sa všetky priradenia vykonávali paralelne a preto sme každé priradenie z množiny priradení sekcie **assign** definovali ako `active proctype` s názvom `process_index_procesu`. Pre zabezpečenie nekonečného vykonávania sme priradenia vložili do cyklu za pomoci `do`. Musel ale obsahovať podmienku `:: else -> skip`, ktorá finálne zabezpečila nekonečné vykonávanie každého priradenia. Keďže sa priradenia vykonávali paralelne a používali rovnaké premenné nastávala situácia, že sa niektoré priradenia premenných vykonávali zle a nadobúdali zlé hodnoty. Museli sme preto použiť atomické operácie.

Výsledný kód bol zapísaný v jazyku Promela do súboru `program.pml`. Ukážku môžete vidieť na obr. 4.6.

```
int x;
int y;

init {
    x = 60;
    y = 48;
}
active proctype process_0() {
    do
    :: x > y →
        atomic {
            x  =  x - y
        }
    :: else → skip
    od
}
active proctype process_1() {
    do
    :: y > x →
        atomic {
            y  =  y - x
        }
    :: else → skip
    od
}
```

Obr. 4.6: GCD v jazyku Promela

4.3 Verifikácia programu

Po vložení programu UNITY do code editor-u a stlačení tlačidla RUN sa program spracuje. Výsledkom procesu je súbor `program.smv`. Tento súbor sa vygeneruje za použitia nástroja S2N, ktorý upraví súbor `program.pml`.

Pre verifikáciu programu sme použili model checker NuSMV. Pretože NuSMV používa svoj vlastný interaktívny shell nebolo možné proces verifikácie zahrnúť do vytvorenej aplikácie.

NuSMV je dostupný v zložke projektu `bin`, kde sa nachádza aj nástroj S2N. Výstupný súbor `program.smv` si buď stiahnete po vykonaní aplikácie alebo ho môžete nájsť v zložke `public/out/`.

Pre spustenie verifikátoru NuSMV je potrebné vykonať príkaz `bin/NuSMV`

`-int public/out/program.smv` z koreňu projektu. Následne sa spustí interaktívny shell, kde už začína verfikácia vlastností programu UNITY.

Ukážku shell-u po simulovaní troch krokoch môžete vidieť na obr. 4.7 a 4.8.

```
NuSMV > go
WARNING *** Processes are still supported, but deprecated.      ***
WARNING *** In the future processes may be no longer supported. ***

WARNING *** The model contains PROCESSes or ISAs. ***
WARNING *** The HRC hierarchy will not be usable. ***
NuSMV > pick_state -r
NuSMV > simulate -r -k 3
********  Simulation Starting From State 1.1   ********
NuSMV > show_traces -v
    <!-- ################## Trace number: 1 ################## -->
Trace Description: Simulation Trace
Trace Type: Simulation
  -> State: 1.1 <-
    tmp = 0sd32_0
    N = 0sd32_0
    a[0] = 0sd32_0
    a[1] = 0sd32_0
    a[2] = 0sd32_0
    a[3] = 0sd32_0
    a[4] = 0sd32_0
    p_initial.pc = 1
    p_process_0.pc = 1
    p_process_1.pc = 1
    p_process_2.pc = 1
    p_process_3.pc = 1
  -> Input: 1.2 <-
    _process_selector_ = p_initial
    running = FALSE
    p_process_3.running = FALSE
    p_process_2.running = FALSE
    p_process_1.running = FALSE
    p_process_0.running = FALSE
    p_initial.running = TRUE
  -> State: 1.2 <-
    tmp = 0sd32_0
    N = 0sd32_5
    a[0] = 0sd32_0
    a[1] = 0sd32_0
    a[2] = 0sd32_0
    a[3] = 0sd32_0
    a[4] = 0sd32_0
    p_initial.pc = 2
    p_process_0.pc = 1
    p_process_1.pc = 1
    p_process_2.pc = 1
    p_process_3.pc = 1
```

Obr. 4.7: Prvá časť simulácie NuSMV

```
-> Input: 1.3 <-
  _process_selector_ = p_process_0
  running = FALSE
  p_process_3.running = FALSE
  p_process_2.running = FALSE
  p_process_1.running = FALSE
  p_process_0.running = TRUE
  p_initial.running = FALSE
-> State: 1.3 <-
  tmp = 0sd32_0
  N = 0sd32_5
  a[0] = 0sd32_0
  a[1] = 0sd32_0
  a[2] = 0sd32_0
  a[3] = 0sd32_0
  a[4] = 0sd32_0
  p_initial.pc = 2
  p_process_0.pc = 6
  p_process_1.pc = 1
  p_process_2.pc = 1
  p_process_3.pc = 1
-> Input: 1.4 <-
  _process_selector_ = p_process_0
  running = FALSE
  p_process_3.running = FALSE
  p_process_2.running = FALSE
  p_process_1.running = FALSE
  p_process_0.running = TRUE
  p_initial.running = FALSE
-> State: 1.4 <-
  tmp = 0sd32_0
  N = 0sd32_5
  a[0] = 0sd32_0
  a[1] = 0sd32_0
  a[2] = 0sd32_0
  a[3] = 0sd32_0
  a[4] = 0sd32_0
  p_initial.pc = 2
  p_process_0.pc = 7
  p_process_1.pc = 1
  p_process_2.pc = 1
  p_process_3.pc = 1
```

Obr. 4.8: Druhá časť simulácie NuSMV

Kapitola 5

Výsledky

Verikikačný nástroj sme použili na dva programy. Prvý je GCD (Greatest common divisor) a druhý Bubble sort. Pre oba tieto programy budeme chcieť verifikovať vlastnosti programu, t. j. FP a leads-to.

Na overenie vlastností sme použili nástroj NuSMV. Príkazom `bin/NuSMV -int public/out/program.smv` sme sa dostali do interaktívneho shell-u. Nasledujúce príkazy v shell-i nás doviedli k výsledkom, z ktorých sme dokázali FP a leads-to overiť. Príkaz `show_traces -v` nám vrátil všetky vykonané kroky. Pre overenie vlastností sme uviedli iba niektoré časti krokov.

```
NuSMV > go
NuSMV > pick_state -r
NuSMV > simulate -r -k 3
NuSMV > show_traces -v
```

5.1 GCD

GCD je jednoduchý algoritmus na výpočet najväčšieho spoločného deliteľa.

```
program       GCD
    declare   x: integer,
              y: integer
    always    decx, decy := x > y, y > x
    initially x, y : 60, 48
    assign    x := x - y if decx
        []    y := y - x if decy
end
```

Po vykonaní 38 krokoch hodnoty x a y nadobudli rovnakú hodnotu. To znamená, že spoločný deliteľ bol nájdený.

```
-> State: 1.22 <-
x = 0sd32_12
y = 0sd32_48
.
-> State: 1.26 <-
x = 0sd32_12
y = 0sd32_36
...
-> State: 1.39 <-
x = 0sd32_12
y = 0sd32_12
...
-> State: 1.41 <-
x = 0sd32_12
y = 0sd32_12
```

Od kroku 1.39 sa hodnoty nezmenili, teda **nastáva FP**. Pre overenie

vlastnosti leads-to sme použili nasledujúci výraz.

$$x \geq y \rightarrow y \geq x$$

Výraz platil vo všetkých krokoch teda **leads-to je splnené**.

5.2 Bubble sort

Bubble sort je algoritmus, ktorý slúži na utriedenie jednorozmerného poľa čísel. Jeho princípom je posúvať neutriedené čísla od najmenšieho po najväčšie. Využíva výmenný triediaci algoritmus, ak je jedno číslo väčšie ako to druhé tak im zamení miesto v poli.

V našom konkrétnom príklade má pole veľkosť N, kde N je 5, a čísla nadobúdali náhodnú hodnotu. UNITY program v sekcii **assign** vytvorí $N-1$ priradení, ktoré kontrolujú či sa dané dve čísla môžu zameniť.

```
program       BubbleSort
    declare   N: integer,
              a: array [0..N] of integer
    initially N: 5 []
              <|| k: 0 <= k < N :: a[k] = rand() >
    assign    <[] i: 0 <= i < N - 1 ::
                  a[i], a[i + 1] = a[i + 1], a[i]
                  if a[i] > a[i + 1] >
end
```

Po spustení programu nám verfikátor vygeneroval pole `a`.

```
a = [62, 72, 9, 83, 59]
```

Na začiatok sme simulovali 6 krokov.

```
// tu môžeme vidieť inicializáciu poľa
-> State: 1.4 <-
tmp = 0sd32_0
N = 0sd32_5
a[0] = 0sd32_62
a[1] = 0sd32_72
a[2] = 0sd32_0
a[3] = 0sd32_0
a[4] = 0sd32_0
.
// pole bolo inicializované v kroku 1.7, po opakovaní príkazu
    simulate -r -k 3
-> State: 1.7 <-
tmp = 0sd32_0
N = 0sd32_5
a[0] = 0sd32_62
a[1] = 0sd32_72
a[2] = 0sd32_9
a[3] = 0sd32_83
a[4] = 0sd32_59
```

Pre ďalšie výsledky sme simulovali program o ďalších 10 krokov.

```
// premenná tmp, ktorá slúži na usporiadanie poľa nadobúda
    hodnotu 83, čo znamená, že proces triedenia sa už začal
-> State: 1.17 <-
tmp = 0sd32_83
N = 0sd32_5
a[0] = 0sd32_62
a[1] = 0sd32_72
```

```
a[2] = 0sd32_9
a[3] = 0sd32_59
a[4] = 0sd32_83
```

Pre ďalšie výsledky sme simulovali program o 20 krokov.

```
// pole sa už začína utriedovať
-> State: 1.29 <-
tmp = 0sd32_72
N = 0sd32_5
a[0] = 0sd32_62
a[1] = 0sd32_9
a[2] = 0sd32_59
a[3] = 0sd32_72
a[4] = 0sd32_83
.
-> State: 1.37 <-
tmp = 0sd32_62
N = 0sd32_5
a[0] = 0sd32_9
a[1] = 0sd32_62
a[2] = 0sd32_59
a[3] = 0sd32_72
a[4] = 0sd32_83
```

Pole bolo utriedené po vykonaní 68 krokoch, každý ďalší krok sa už pole nezmenilo a **nastáva FP**, kedy sa už žiadne vstupné hodnoty nezmenia.

```
-> State: 1.69 <-
tmp = 0sd32_62
N = 0sd32_5
```

```
a[0] = 0sd32_9
a[1] = 0sd32_59
a[2] = 0sd32_62
a[3] = 0sd32_72
a[4] = 0sd32_83
.
-> State: 1.77 <-
tmp = 0sd32_62
N = 0sd32_5
a[0] = 0sd32_9
a[1] = 0sd32_59
a[2] = 0sd32_62
a[3] = 0sd32_72
a[4] = 0sd32_83
```

Pre overenie vlastnosti leads-to sme použili nasledujúci výraz, kde A je pole `a`.

$$A[i] \rightarrow \langle \exists j : j \geq i :: A[i] \geq A[j] \rangle$$

V krokoch 1.29, 1.37 a 1.69 môžeme vidieť, že existuje také $A[i]$, ktoré je väčšie ako $A[j]$. A keďže tento výraz platí pre niekoľko krokov programu vlastnosť **leads-to je splnená**.

Kapitola 6

Záver

Vývoj aplikácie bol pre nás veľmi prínosný z hľadiska skúseností. Niektoré technológie sme použili dokonca prvý krát. V živote a našich prácach to býva tak, že všetko ide vylepšiť, toto je tiež ten prípad. Interpreter by sa dal v budúcnosti vylepšiť o vyspelejšiu kontrolu syntaxe, správnosti zápisu jednotlivých sekcií programu a zvládnutie zložitejších operácií s kvantifikovanými priradeniami a výrazmi. Tie sú obzvlášť zložité na implementáciu. Avšak navzdory všetkým možným budúcim vylepšeniam sme s našou prácou výsostne spokojný a nebojíme sa povedať, že sa zdarila. Všetky ciele, ktoré sme si stanovili sme aj dokázali dosiahnuť. Dúfam, že naša diplomová práca prinesie do oblasti paralelného programovania chuť a vytrvalosť sa v tejto oblasti vzdelávať, prípadne pomôže všetkým budúcim študentom pri ich diplomových prácach v tejto oblasti.

Úprimne verím, že sa Vám naša práca páčila a jej čítanie Vám prišlo pútavé a poučné.

Literatúra

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

[2] Lawrence C. Paulson. Mechanizing UNITY in Isabelle, 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, Tom van Dijk. LTSmin: High-Performance Language-Independent Model Checking, 2015

[6] Michal Šuster. Intepreter UNITY, 2006

[7] Yong Jiang, Zongyan Qiu. S2N: Model Transformation from SPIN to NuSMV, 2012

[8] A. Cimatti, E. Clarke, F. Giunchiglia, M. Roveri. NuSMV: A New Symbolic Model Verifier, 1999

[9] Gerard J. Holzmann. The Model Checker SPIN, May 1997

[10] Vladimír Kvasnička. Jiří Pospíchal, Matematická logika, 2006

[11] Martin Baláž. Dynamické Kripkeho štruktúry pre dobre založenú sémantiku, 2002