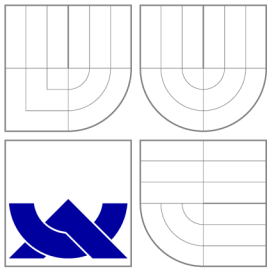
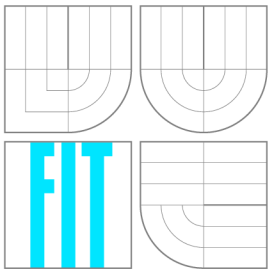


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PŘEKLADAČ GRAFU TOKŮ DAT DO LOGIKY BITOVÝCH VEKTORŮ

A BIT-VECTOR COMPILER FOR DATA-FLOW GRAPHS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — TOMÁŠ SUŠOVSKÝ
AUTHOR

VEDOUCÍ PRÁCE — Ing. ALEŠ SMRČKA, Ph.D.
SUPERVISOR

BRNO 2016

Zadání bakalářské práce/18543/2015/xsusov01

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2015/2016

Zadání bakalářské práce

Řešitel: **Sušovský Tomáš**
Obor: Informační technologie
Téma: **Překladač grafu toků dat do logiky bitových vektorů**
A Bit-Vector Compiler for Data-Flow Graphs
Kategorie: Překladače

Pokyny:

1. Seznamte se v současnosti používanými grafy toku dat (GTD). Nastudujte informace o nástrojích pro verifikaci využívajících teorii bitových vektorů a polí (např. nástroje boolector STP). Seznamte se s formáty SMT-LIB a BTOR, které tyto nástroje zpravidla využívají.
2. Navrhněte překladač ze zvoleného formátu GTD do formátu SMT-LIB nebo BTOR. Diskutujte možnosti optimalizace generovaného kódu pro dosažení co možná nejvyšší rychlosti verifikace výsledného kódu daným nástrojem.
3. Implementujte navržený překladač v jazyce C/C++ nebo Python. Překladač implementuje pro Gnu/Linux i MS Windows.
4. Otestujte správnost generovaného kódu s využitím výše uvedených verifikačních nástrojů. Na základě výsledků experimentů vylepšete a srovnejte navržené optimalizace. Použití demonstrujte na příkladech.

Literatura:

- Nástroj boolector a formát BTOR, http://fmv.jku.at/boolector/
- Nástroj STP, https://sites.google.com/site/stpfastprover/
- Knihovna a formát SMT-LIB, http://www.smtlib.org/

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Smrčka Aleš, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 18. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Cílem této bakalářské práce je vytvořit a implementovat nástroj pro překlad modelů grafů toků dat do formátu SMT-LIB. Práce navazuje na projekt HADES výzkumné skupiny VeriFIT Fakulty informačních technologií Vysokého učení technického v Brně. V řešení bylo použito překladače vytvářejícího z původního grafu objektový model. Objektový model je možné převést do zápisu ve formátu SMT-LIB a přidat do něj aserce požadovaných vlastností systému. Pro ověřování vlastností závisejících na změnách systému je použita metoda rozbalování smyček s uživatelem zadanou hranicí maximálního počtu rozbalení. Možnosti vytvořeného nástoje jsou demonstrovány na sadě modelů grafů toků dat pokrývající všechny prvky vstupního jazyka VAM a jejich kombinace. Výsledek této práce představuje nové možnosti pro zpracování grafů toků dat ve formátu VAM a jejich verifikaci.

Abstract

The principal goal of this bachelor thesis is to design and implement a tool for compiling data-flow graph models to SMT-LIB format. This thesis builds on the research project HADES developed by VeriFIT research group of the Faculty of Information Technology, Brno University of Technology. The solution uses compiler for generating object model from original graph. Object model can be converted to a SMT-LIB format description including assertions of the desired system properties. Loop unrolling method (with user defined boundary for unrollment) is used for verification of system properties depending on changes in state of model. Capabilities of the developed tool are demonstrated on set of data-flow graphs models. Models cover usage of all elements defined in VAM language (input format) and their combinations. Result of this thesis presents new ways of processing data-flow graphs in VAM format and their verification.

Klíčová slova

Formální verifikace, SMT, grafy toků dat, VAM, překladače

Keywords

Formal verification, SMT, data-flow graphs, VAM, compilers

Citace

Tomáš Sušovský: Překladač grafu toků dat do logiky bitových vektorů, bakalářská práce, Brno, FIT VUT v Brně, 2016

Překladač grafu toků dat do logiky bitových vektorů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Aleše Smrčky Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Tomáš Sušovský
18. 5. 2016

Poděkování

Děkuji vedoucímu této bakalářské práce Ing. Aleši Smrčkovi Ph.D. za aktivní odbornou pomoc a podněty při řešení této práce.

Obsah

Kapitola 1

Úvod

Cílem této bakalářské práce je představit nástroj Gravector pro překlad modelů grafů toků dat, zapsaných v jazyce VAM 2.5, do formátu SMT-LIB 3.3. Přeložený model bude možno verifikovat pomocí SMT solverů. Práce navazuje na projekt HADES (Hazard Detection System) [12] výzkumné skupiny VeriFIT [17] Fakulty informačních technologií Vysokého učení technického v Brně. Spolu s projekt HADES a programy vytvořenými v rámci studentských diplomových prací navazujících na projekt HADES (interaktivní simulátor pro grafy toku dat dfsim – bakalářská práce Davida Kovaříka [20] a překladač jazyka VHDL pro potřeby formální verifikace – bakalářská práce Jiřího Matyáše [28]) tvoří Gravector rámec nástrojů pro práci s grafy toků dat ve formátu VAM (jejich simulaci a verifikaci).

Kapitola 2 popisuje základy teorie grafů a grafy toků dat, ze kterých se v této práci dále vychází. Dále obsahuje popis formátů používaný pro zápis grafů toků a podrobný popis formátu VAM, který je v této práci použit jako výchozí formát 2.5. V podkapitole 2.5.1 jsou popsány jednotlivé prvky vyskytující se v tomto jazyce a jejich zápis.

Kapitola 3 popisuje pricipy formální verifikace a nástoje, které se k ní používají. Jejich použití k verifikaci hardware a jejich využití v rámci projektu HADES 3.7.

Kapitola 4 popisuje specifikaci požadavků na výsledný program implementovaný v rámci této práce, návrh jeho architektury, rozhraní, samotnou implementaci a ovládání výsledného programu.

Kapitola 5 popisuje testování implementace překladače. Podkapitola 5.1 popisuje jednotlivé fáze testování projektu a vytvořené testy. Sada ukázkových modelů vytvořených pro experimentování obsahuje jak jednoduché příklady obsahující jednotlivé prvky jazyka VAM, jejich kombinaci a složitější modely jako je model 8-bitového procesoru TinyCPU nebo 32-bitový procesor DLX5. V podkapitole 5.2 jsou uvedeny specifikace platforem využitých pro vývoj a testování.

V závěru této práce jsou popsány výsledky zjištěné prováděním experimentů nad testovací sadou modelů. Dále je v závěru zhodnoceno využití nástroje Gravector v rámci dalších nástrojů pracujících s jazykem VAM a možnosti budoucího rozšíření.

Kapitola 2

Grafy toku dat

Tato kapitola se věnuje popisu teorie grafů, grafům toku a jejich použití ve formátech pro popis grafů toku dat a výběrem vhodného formátu grafů toku dat pro použití v rámci implementace překladače grafů toku dat do logiky bitových vektorů.

2.1 Teorie grafů

Pojem grafu byl zaveden Leonhardem Eulerem v roce 1736. Jedná se o model, který reprezentuje objekty a vztahy mezi nimi.

Definice 1 *Jednoduchý graf G je uspořádaná dvojice (V; E) , kde V je neprázdná množina vrcholů a E je nějaká množina dvouprvkových podmnožin množiny V. Prvkům E říkáme hrany.*

[21]

Definice 2 *Orientovaný graf je graf 1, jehož hrany jsou uspořádané dvojice.*

Definice 3 *Kružnice (cyklus) je graf 1 C = (V, E), kde V =*$\{v_1, \ldots, v_n\}$ *a E =*$\{e_1, \ldots, e_n\}$ *a platí:* $e_i = (v_i, v_i + 1)$*, kde i = 1, . . . , n - 1 a* $e_n = (v_n, v_1)$.

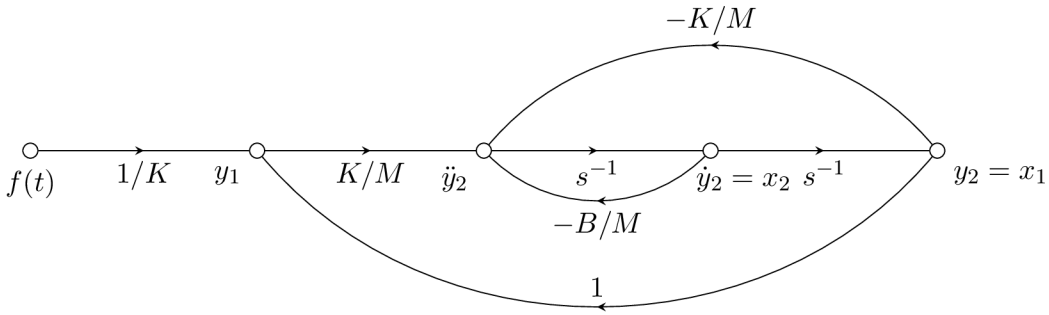
Definice 4 *Orientovaný acyklický graf (directed acyclic graph – DAG) je graf bez orientovaných kružnic[3].*

2.2 Grafy toků

Grafy toků jsou orientované grafy 2 sloužící k modelování systémů (elektrotechnických/hardwareových, či softwareových), kde vrcholy reprezentují prvek systému (funkční prvek, nebo stavovou proměnnou) a hrany přechody mezi nimi, přes které dochází k přenosu informace (v tzv. tocích).

Grafy signálových toků

Grafy signálových toků (*angl. signal flow graphs* – SFG), také označovány jako *Masonovy grafy*, jsou specializovaným druhem orientovaných grafů, ve kterých vrcholy reprezentují systémové proměnné a hrany funkční spojení dvojice vrcholů. SFG se nejčastěji používají k modelování elektrotenických systémů, jako jsou zesilovače nebo digitální a analogové filtry.



Obrázek 2.1: Příklad grafu signálových toků znázorňující Masonovo pravidlo.

Grafy toků řízení

Grafy toků řízení (*angl. Control flow graphs* – CFG) jsou orientované grafy, popisující všechny možné průchody při vykonávání programu. Vrcholy reprezentují základní bloky, které jsou tvořeny posloupností příkazu, pro které je vstupní bod řízení běhu programu na prvním příkazu, provádí se sekvenčně v pořadí posloupnosti a na posledním příkazu je výstupní bod řízení programu (větvení, skoková instrukce).

Grafy toků řízení jsou využívány překladači pro statickou analýzu nad základními bolky a následné optimalizace a generování kódu [35].

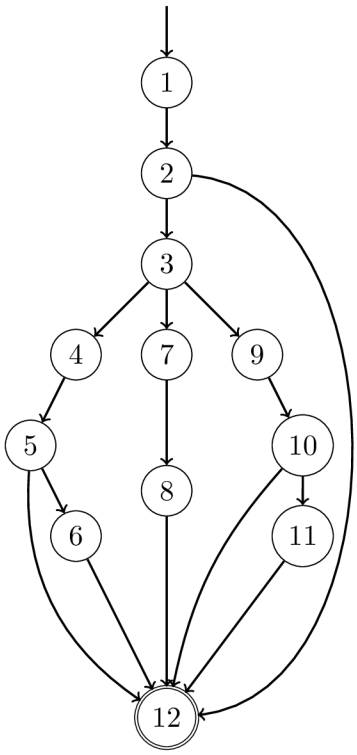
Grafy toků dat

Grafy toků dat (*angl. Data flow graphs* – DFG) jsou orientované grafy, kde uzly reprezentují určitou funkcionalitu a hrany představují spojení dvou uzlů, přes které dochází k přenosu (toku) dat ze zdrojového uzlu (producenta) do cílového uzlu (konzumenta) [26]. Grafy toků dat umožňují vytvořit distribuovaný model výpočtu bez řízení soustředěného do jednoho bodu (např. vykonávání jediné instrukce v daném čase). Provedení výpočtu na funkčních uzlech (*angl. firing*) může probíhat asynchronně, jakmile jsou na vstupech správná (aktuální) data (modely dále využívané dále v této práci mají provádění výpočtu synchronizované vůči paměťovým blokům, což zajišťuje konzistenci dat přicházejících na vstupy funkčních uzlů). Hlavní výhodou grafů toků dat oproti ostatním způsobům modelování pralelních systémů je jejich kompaktnost a možnost přímé interpretace.

2.3 Binární rozhodovací diagramy

Binární rozhodovací diagramy (*angl. Binary decision diagrams*) [29] jsou datovou strukturou využívanou k reprezentaci funkcí booleovy algebry. K reprezentaci funkce využívá uspořádání v kořenovém stromu (acyklickém orientovaném grafu s jedním význačným vrcholem – kořenem – který nemá žádné předchůdce). BDD strom obsahuje rozhodovací vrcholy a terminální vrcholy (listové vrcholy, vrcholy bez potomků) dvou druhů – 0-terminální a 1-terminální. Každý rozhodovací uzel má dva potomky – jeden pro vyhodnocení hodnoty rozhodovacího uzlu na 0 a druhý na 1. Potomkem může být další rozhodovací vrchol nebo terminální vrchol představující vyhodnocení zadané booleovské funkce (pro hodnoty proměnných určených cestou od kořene stromu po terminál). Binární rozhodovací diagram umožňuje provádět operace přímo na zkomprimované reprezentaci a nemusí se předtím

```
begin
s1;
if(s2)
  case(s3)
    when(s4)
      if(s5)
        s6;
      endif;
    when(s7)
      s8;
    when(s9)
      if(s10)
        s11;
      endif;
  endcase;
endif;
s12;
```



Obrázek 2.2: Příklad grafu toku řízení uvedeného pseudokódu.

dekomprimovat. BDD se využívá hlavně při formální verifikaci 3 a také v CAD (Computer-aided design) systémech při syntéze obvodů (převodu z RTL 2.5 popisu na návrh implementace logickými hradly).

2.4 And-inverter graph – AIG

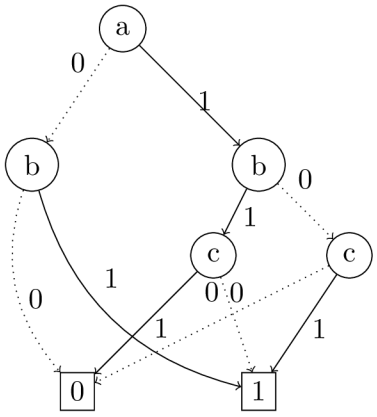
AIG (*And-inverter graphs*) jsou orientované acyklické grafy 4 sloužící k popisu struktury implementace funkční logiky číslicových obvodů. Využívá dekompozici složitějších obvodů na systém využívající pouze dvouvstupá hradla AND a invertory. Při převodu na odpovídající booleovskou funkci tedy využívá logický součin a negaci. AIG je využíván systémy pro syntézu sekvenčních a kombinačních obvodů a formální verifikaci. Příkladem programů využívající AIG je program ABC [9] nebo AIGER [6]. Hlavní výhodou je kompaktní reprezentace, vhodná i pro reprezentaci velkých obvodů, nevýhodou naopak může být ztráta některých informací o původním obvodu při zpětném převodu z AIG na původní obvod (rozdělení hradel a invertorů na původní komponety).

2.5 VAM

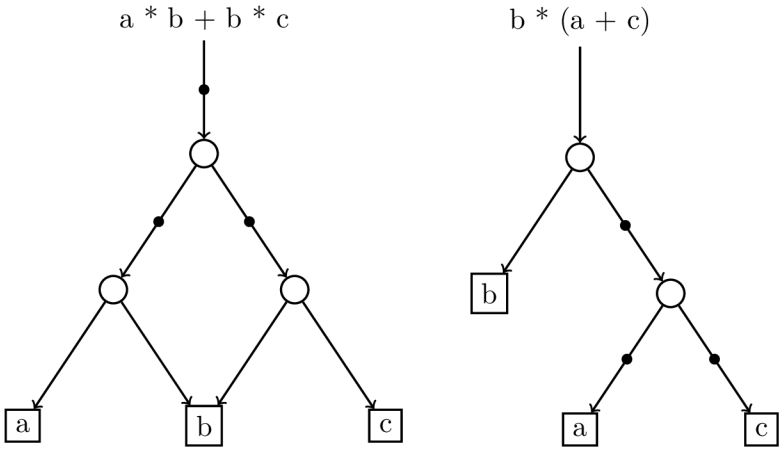
Variable-Assignment-Model (VAM)[13] je jazyk pro popis struktury datových toků, reprezentujících RTL (Register-transfer level) model mikroprocesoru. VAM byl vyvinut výzkumnou skupinou VeriFIT [17] v roce 2012 v rámci projektu HADES [12] pro interní potřeby zápisu modelů mikroprocesorů, statické analýzy a vytváření derivovaného modelů pro parametrizovanou formální verifikaci. Model v jazyce VAM je tvořen sítí paměťových a funkčních

Booleovská funkce $f(a, b, c)$

a	b	c	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



Obrázek 2.3: Příklad binárního rozhodovacího diagramu funkce f definované pravdivostní tabulkou.



Obrázek 2.4: Příklad dvou AIG grafů znázorňujících dvě různé grafové reprezentace zápisu funkce f(a, b, c) = b * (a + c).

uzlů a orientovaných hran mezi nimi – signálů, reprezentujících datové toky mezi nimi o pevně daném směru a bitové šířce. Každý uzel má sadu svých vstupních a výstupních signálů, nicméně z důvodu analýzy datových toků je podle standardu jazyku VAM zakázano vytvářet rychlé smyčky 3 mezi funkčními uzly bez paměťového uzlu.

Zápis v jazyce VAM využívá syntaxi podobnou jazyku LISP [36] a v rámci jednoho textového souboru popisuje jeden model. Zápis modelu obsahuje základní informace o modelu (jméno, stupně pipeline [19]), deklaraci signálů, deklaraci paměťových uzlů a definici funkčních uzlů. Paměťové uzly mohou být deklarovány jako registr (paměťové uložiště s pevnou velikostí), nebo adresovatelné paměťové uložiště (registrový soubor) obsahující více registrových buněk (o stejné velikosti), ke kterým je možno přistupovat pomocí portu a adresy.

Příklad struktury zápisu modelu v jazyce VAM:

```
(model
    jmeno_modelu
    (pipeline-stages <seznam stupnu pipeline>)
    (sig ...)
    ...
    (reg ...)
    ...
    (mem ...)
    ...
    (fnode ...)
    ...
)
```

Kde:

- *(sig ...)* reprezentuje zápis jednotlivých signálů,
- *(reg ...)* reprezentuje zápis jednotlivých registrů,
- *(mem ...)* reprezentuje zápis jednotlivých adresovatelných paměťových uložišť (registrových souborů) a
- *(fnode ...)* reprezentuje zápis jednotlivých funkčních uzlů. V rámci jednodušších modelů, nevyužívajících zřetězené zpracování, je možné deklaraci stupňů pipeline vynechat.

Příklad jednoduchého modelu (bez pipeline):

```
(model
  test-bitwise-and-8-bit
  (sig v_fun_in_1 8 (vt data ))
  (sig v_fun_in_2 8 (vt data ))
  (sig v_fun_out 8 (vt data ))

  (fnode fun_bitwise_and (input v_fun_in_1 v_fun_in_2)
        (output v_fun_out)
    (assign
        (:= v_fun_out (& v_fun_in_1 v_fun_in_2) )
        )
```

```
  )

  (fnode v_fun_in_1_init (input)
       (output v_fun_in_1)
    (assign
       (:= v_fun_in_1 (_ 127 8 ))
       )
  )

  (fnode v_fun_in_2_init (input)
       (output v_fun_in_2)
    (assign
       (:= v_fun_in_2 (_ 69 8) )
       )
  )
)
```

Paměťové uzly (registry) jsou v rámci modelu synchronizovány (implicitním hodinovým signálem), tak aby v každém kroku mohla být na vstupu zpracována nová hodnota z logiky funkčních uzlů, u kterých není v rámci modelu zohledněno zpoždění. Adresovatelné paměťové uzly (registrové soubory) mají zpoždění definováno pro každý přístupový port zvlášť (mohou mít tedy jiná zpoždění pro čtení a zápis).

2.5.1 Komponenty VAM

Bližší popis jednotlivých prvků používaných v jazyce VAM pro popis RTL modelu procesoru.

Signály – *(sig)*: Syntaxe zápisu prvku signál:

```
(sig identifikator_signalu
    bitova sirka
    (vt <seznam typu signalu>)
    (dir smer)
    (inputs <seznam identifikatoru pametovych uzlu>)
    (stage identifikator stupne pipeline)
    (connections
           <seznam dvojic <identifikator uzlu, typ pripojeni>>
    )
    (meta ...)
)
```

Kde:

- `identifikator_signalu` je identifikátor signálu spojujícího dva funkční nebo paměťové uzly, je unikátní v rámci ostatních signálů.
- `bitova sirka` určuje velikost datového toku reprezentovaného signálem v bitech.
- `vt` (value type) určuje druh datového toku, může nabývat hodnot: 'data', 'address', 'control', nebo 'unknown'.

- `dir` (direction) je volitelný parametr pro specifikaci použití složitějších datových toků – může nabývat hodnot: 'ro' (pouze ke čtení), 'wo' (pouze pro zápis), nebo 'rw' (pro čtení i zápis).
- `inputs` je volitelný parametr obsahující seznam identifikátorů paměťových uzlů, které mají vliv na hodnotu datového toku reprezentovaného signálem.
- `stage` je volitelný parametr obsahující identifikátor jednoho stupně pipeline, do kterého patří datový tok reprezentovaný signálem.
- `connections` je volitelný parametr obsahující dvojice (identifikátor paměťového uzlů a typ připojení), popisující informaci o připojení signálu.
- `meta` je volitelný parametr obsahující další nepovinné informace a komentáře.

Příklad:

```
(sig v_sig_1
       8
       (vt data)
       (dir rw)
       (connections (reg_pc q) )
    (meta (comment "pc_Q0" ) )
)
```

Registry – *(reg)*: Syntaxe zápisu prvku registr:

```
(reg identifikator_registu
    bitova sirka
    (d identifikator singalu pripojeneho na port D)
    (q identifikator signalu pripojenych na port Q)
    (we identifikator signalu pripojeneho na port WE)
    (stall identifikator signalu pripojeneho na port STALL)
    (clr identifikator signalu pripojeneho na port CLEAR)
    (rst identifikator signalu pripojeneho na port RESET)
    (stage identifikator stupne pipeline)
    (meta ...)
)
```

Kde:

- `identifikator_registu` je identifikátor registru unikátní v rámci ostatních paměťových uzlů.
- `bitova sirka` určuje velikost paměti registru bitech.
- `d` obsahuje identifikátor signálu, připojeného na vstupní port D registru (signál pro přenos vstupní hodnoty pro zápis).
- `q` obsahuje identifikátor signálu, připojeného na výstupní port Q registru (signál pro přenos hodnoty uložené v registru).
- `we` obsahuje identifikátor signálu, připojeného na vstupní port WE registru (signál pro povolení zápisu nové hodnoty ze vstupního portu D).

- `stall` (volitelný) obsahuje identifikátor signálu, připojeného na vstupní port STALL registru (signál pro vkládání přerušovacích stavů pro prevenci hazardů).
- `clr` (volitelný) obsahuje identifikátor signálu, připojeného na vstupní port CLEAR registru (signál nulující obsah registru).
- `rst` (volitelný) obsahuje identifikátor signálu, připojeného na vstupní port RESET registru (signál pro ansynchronní nulování obsahu registru).
- `stage` je volitelný parametr obsahující identifikátor jednoho stupně pipeline, do kterého patří registr.
- `meta` je volitelný parametr obsahující další nepovinné informace a komentáře.

Příklad:

```
(reg reg_pc 8 (d v_sig_0) (q v_sig_1)
    (meta "state" (init "0" ) )
)
```

Registrové soubory – *(mem)*: Syntaxe zápisu prvku registrový soubor:

```
(mem identifikator_registorveho_souboru
    bitova sirka bunky
    bitova sirka adresy
        pocet pametovych bunek
    (ports
        (port identifikator_portu
            (dir smer)
            (timing zpozdeni)
            (en identifikator povolovaciho signalu)
            (addr identifikator adresoveho signalu)
            (data <seznam identifikatoru datovych signalu>)
            (control <seznam identifikatoru ridicich signalu>)
            (stage identifikator stupne pipeline)
            (meta ...)
        )
        (port ...)
        ...
    )
    (rst identifikator signalu pripojeneho na port RESET)
    (meta ...)
)
```

Kde:

- `identifikator_registroveho_souboru` je identifikátor registrového souboru unikátní v rámci ostatních paměťových uzlů.
- `bitova sirka bunky` určuje velikost jednotlivých adresovatelných buněk paměti v bitech.

- `bitova sirka adresy` určuje velikost adresy pro adresování jednotlivých buněk paměti v bitech. Maximální počet adresovatelných buněk je omezen na 2^N buněk, pro N-bitovou adresu.
- `pocet pametovych bunek` určuje skutečný počet buněk (registrů) v registrovém souboru. Počet buněk musí být větší než 1 a menší, nebo roven 2^N, pro N-bitovou adresu.
- `ports` sekce popisující jednotlivé přístupové porty patřící registrovému souboru:
  - `port` unikátní identifikátor přístupového portu (unikátní v rámci daného registrového souboru).
  - `dir` směr datového toku na daném portu, může nabývat hodnot: 'ro' (pouze ke čtení), 'wo' (pouze pro zápis), nebo 'rw' (pro čtení i zápis).
  - `timing` zpoždění při přístupu přes daný port do paměti. Udává se v počtu hodinových cyklů a může být nulový.
  - `en` obsahuje identifikátor povolovacího signálu, připojeného na daný port.
  - `addr` obsahuje identifikátor signálu, připojeného na daný port, přenášejícího hodnotu adresy pro přístup do paměti.
  - `data` obsahuje seznam identifikátorů signálů přenášejících data z nebo do paměti přes daný port. Pro porty sloužící k zápisu do paměti by měl být připojen jen jeden datový signál.
  - `stage` je volitelný parametr obsahující identifikátor jednoho stupně pipeline, ke kterému náleží daný port.
  - `control` je volitelný parametr obsahující identifikátory dalších řídících signálů připojených na daný port (bez další specifikace využití těchto signálů).
- `rst` (volitelný) obsahuje identifikátor signálu, připojeného na vstupní port RESET registru (signál pro ansynchronní nulování obsahu registrového souboru).
- `meta` je volitelný parametr obsahující další nepovinné informace a komentáře.

Příklad:

```
(mem regs 16 3 8
    (ports
        (port r0
            (dir ro)
            (timing 0)
            (en v12)
            (addr v14)
            (data v18 v158 v164)
            (stage id))
        (port r1
            (dir ro)
            (timing 0)
            (en v20)
            (addr v22)
            (data v24 v159 v165)
```

```
            (stage ex))
        (port w1
            (dir wo)
            (timing 1)
            (en v32)
            (addr v34)
            (data v38)
            (stage wb))
    )
)
```

Funkční uzly – *(fnode)*: Syntaxe zápisu prvku funkční uzel:

```
(fnode identifikator_funkcniho_uzlu
    (input <seznam identifikatoru vstupnich signalu>)
    (output <seznam identifikatoru vystupnich signalu>)
    (stage identifikator stupne pipeline)
    (assign
        (:= identifikator vystupniho signalu
               vyraz pro prirazeni
        )
    )
    (meta ...)
)
```

Kde:

- `identifikator_funkcniho_uzlu` je identifikátor funkčního uzlu unikátní v rámci ostatních funkčních uzlů.
- `input` obsahuje seznam identifikátorů signálů připojených na vstup funkčního uzlu.
- `output` obsahuje seznam identifikátorů signálů připojených na výstup funkčního uzlu.
- `stage` je volitelný parametr obsahující identifikátor jednoho stupně pipeline, ke kterému náleží daný funkční uzel.
- `assign` obsahuje identifikátor výstupního signálu a výraz, jehož hodnota je zapisována na daný výstupní signál.
- `meta` je volitelný parametr obsahující další nepovinné informace a komentáře.

Výraz použitý ve funkčních uzlech pro přiřazení mohou být:

1. Konstanta
2. Identifikátor vstupního signálu
3. Aplikace operátoru na další výraz, nebo výrazy

Příklad:

```
(fnode f128
    (input ex_regA_q ex_regB_q ex_op_q)
    (output ex_alu_d)
    (stage ex)
    (assign
        (:= ex_alu_d_1 (?
               (== ([..] ex_op_q 0 3) 5)
               (+ ex_regA_q ex_regB_q) 0)
        )
    )
)
```

2.5.2 Operátory ve VAM

Přehled jednotlivých operátorů [a] používaných v jazyce VAM:

Operátor	Operace	Zápis
:=	přiřazení hodnoty signálu	(:= identifikátor_signálu *výraz)*
_	integerová konstanta	(_ hodnota bitová_šířka)
!	logická negace	(! *výraz)*
||	logický součet	(|| *výraz_a výraz_b)*
&&	logický součin	(&& *výraz_a výraz_b)*
?	ternární oprátor	(? *výraz_podmínky výraz_true výraz_false)*
~	bitová inverze	(~ *výraz)*
|	bitový součet	(| *výraz_a výraz_b)*
&	bitový součin	(& *výraz_a výraz_b)*
^	bitová nonekvivalence	(^ *výraz_a výraz_b)*
«	bitový posuv vlevo	(« *výraz_hodnoty výraz_počtu_posuvů)*
»	bitový posuv vpravo	(« *výraz_hodnoty výraz_počtu_posuvů)*
».	bitový aritmetický posuv vpravo	(« *výraz_hodnoty výraz_počtu_posuvů)*
==	rovnost	(== *výraz_a výraz_b)*
!=	nerovnost	(!= *výraz_a výraz_b)*
<	bezznaménkové menší než	(< *výraz_a výraz_b)*
>	bezznaménkové větší než	(> *výraz_a výraz_b)*
<=	bezznaménkové menší nebo rovno	(<= *výraz_a výraz_b)*
=>	bezznaménkové větší nebo rovno	(>= *výraz_a výraz_b)*
<.	znaménkové menší než	(<. *výraz_a výraz_b)*
>.	znaménkové větší než	(>. *výraz_a výraz_b)*
<=.	znaménkové menší nebo rovno	(<=. *výraz_a výraz_b)*
=>.	znaménkové větší nebo rovno	(>=. *výraz_a výraz_b)*
-.	přetypování na znaménkovou hodnotu	(-. *výraz)*
.-	přetypování na bezznaménkovou hodnotu	(.- *výraz)*
[]	indexování	([] *výraz výraz_indexu)*
[.]	extrakce bitu	([.] *výraz konstanta_indexu)*
.	konkatenace vektorů	(. *výraz_vektor_1 výraz_vektor_2)*
^^	bezznaménkové rozšíření vektoru	(^^ *výraz_vektor konstanta_počtu_bitů)*
^^.	znaménkové rozšíření vektoru	(^^. *výraz_vektor konstanta_počtu_bitů)*
[..]	exktrakce vektoru	([..] *výraz_vektor konst_index_od konsta_index_do)*
–	negace	(– *výraz)*
+	sčítání	(+ *výraz_a výraz_b)*
-	odčítání	(- *výraz_a výraz_b)*
*	násobení	(* *výraz_a výraz_b)*
/	bezznaménkové dělení	(/ *výraz_a výraz_b)*
%	zbytek po bezznaménkovém dělení	(% *výraz_a výraz_b)*
!!	abstraktní logický operátor	
~~	abstraktní bitový operátor	
<>	abstraktní relační operátor	
][	abstraktní vektorový operátor	
-+	abstraktní aritmetický operátor	

[a] Abstraktní operátory vyjadřují zjednodušení posloupnosti více operátorů, kdy v rámci analýzi nezáleží na konkrétních aplikovaných operátorech, z operandů pouze potřebuje určit typ (velikost) výsledku.

switch	switch-case podmínika	(switch *výraz_podmínky* (*case_hodota_podmínky_1 výraz_hodnoty_1)* (*case_hodota_podmínky_2 výraz_hodnoty_2)* ... (*výraz_defaultní_hodnoty))*
cond	podmínka	(cond (*výraz_podmínky_1 výraz_hodnoty_1)* (*výraz_podmínky_2 výraz_hodnoty_2)* ... (*výraz_defaultní_hodnoty))*

Kapitola 3

Formální verifikace

Tato kapitola se věnuje popisu principů formální verifikace, potřebných k pochopení použití výsledků získaných touto bakalářskou prací.

3.1 Formální verifikace

Definice 1 *Verifikace je proces ověřování, zda daný systém (reálný systém, nebo model systému) splňuje požadavky specifikace.*

Definice 2 *Formální verifikace je proces dokazování platnosti nebo neplatnosti správnosti (splnění formálních požadavků specifikace) modelu vůči modelovanému systému s využitím formálních matematických metod.*

Definice 3 $M \models \varphi$
M je model systému a φ je původní systém.

V oboru informačních technologií se můžeme setkat jak s formální verifikací software (zdrojových kódů programů), tak hardware (kombinačních a sekvenčních obvodů). Nástroje provádějící formální verifikaci dokazují platnost nebo neplatnost správnosti podle formální specifikace s využitím formálních matematických metod poskytujících formální důkaz. Formální verifikace narozdíl od testování nebo simulování dokazuje správnost pro všechny možné stavy systému [25].

Tři hlavní metody využívané při formální verifikaci:

- *Ověřování ekvivalence* (*angl. equivalence checking*) – proces dokazování ekvivalence systému vůči jeho specifikaci (vykazuje stejné chování). Ekvivalence chování systému a jeho specifikace je v takovém případě vyjádřena jako booleovská funkce, jejíž splnitelnost je dále ověřována pro dokázání platnosti ekvivalence. Programy pro provádění ověřování ekvivalence využívají k ověřování nejčastěji binární rozhodovací diagramy 2.3 (případně AIG 2.4), nebo SAT solvery 3.2, pro dokázání splnitelnosti získané booleovské funkce ekvivalence v konjunktivní normálové formě.

- *Ověřování modelu*(*angl. model checking*) – proces dokazování zda systém splňuje nebo nesplňuje zadanou vlastnost (podle specifikace). V případě, že systém nesplňuje vlastnost, pokračuje dokázáním protipříkladu – dokázání příkladu chování systému, které zadanou vlastnost porušuje. Příkladem ověřované vlastnosti (pro softwareové systémy) může být absence deadlocků v programu, pokud by nebyla tato vlastnost splňena,

mohlo by docházet k nežádoucímu vzájemnému čekání mezi vlákny programu a jeho zablokování nebo selhání. Pro ověření je nutné vyjádřit model systému i specifikaci požadavků ve formální podobě, ze které je možno převést problém platnosti vlastnosti na úlohu provedení důkazu splnitelnosti logické funkce (vyjadřující splnění vlastnosti) pro vytvořený logický model.

- *Dokazování vět* (*angl. theorem proving*) – proces převedení systému a jeho vlastností na výraz v logice a nalezení důkazu platnosti vlastnosti.

Hlavní kroky nutné pro provedení verifikace

1. Specifikace požadavků správnosti (vlastnosti systému)
2. Vytvoření modelu systému
3. Verifikace požadovaných vlastností nad modelem systému

3.2 SAT

Splnitelnost (*angl. Boolean Satisfiability Problem* nebo zkráceně *Satisfiability* – SAT) [33] je problém řešící otázku zda existuje řešení, pro které zadaný výraz booleovské logiky bude vyhodnocen jako pravdivý – je splnitelný (`satisfiable`). V opačném případě musí být dokázano, že daný výraz je kontradikcí (bude pro všechny možné vstupy vyhodnocen jako nepravdivý) a tedy nesplnitelný (`unsatisfiable`)) – neexistuje žádné řešení, pro které by byl vyhodnocen jako pravdivý.

SAT je problém patřící do třídy složitosti NP-úplných (NPC) problémů (byl prvním problémem vůbec, pro který bylo dokazáno, že se jedná o NP-úplný problém). NP (Nedeterministicky Polynomiální) problémy jsou problémy, které je možné teoreticky vyřešit v polynomiálním čase, pokud by byl použit k řešení počítač schopný v každém kroku provést neomezené větvení řešení a dále hledat řešení současně ve všech větvích (takový počítač existuje pouze jako teoretický model – nedeterministický Turingův stroj). Prakticky jde výslené řešení takovýchto problému v polynomiálním čase ověřit, ale ne nalézt. NP-úplný problém je NP problém, pro který platí, že každý NP problém na něj může být redukován v polynomiálním čase.

Programy řešící SAT pomocí rozhodující procedury označujeme jako *SAT-solvery*, patří mezi ně například GRASP, WalkSAT nebo Lingeling.

3.3 SMT

SAT modulo teorie (*angl. Satisfiability Modulo Theories* – SMT) [25] rozšiřuje SAT problém o splnitelnost výrazů predikátové logiky prvního řádu s rovností a prvky z různých teorií predikátové logiky určitého univerza (jako jsou lineární aritmetika, teorie polí, bitové vektory ad.). Mezinárodní iniciativa SMT-LIB [3] zaměřující se na výzkum a vývoj v oblasti SMT vydává standard formátu SMT-LIB (aktuální verze standardu 2.5 [2]), jazyku pro zápis SMT, pro využití v programech řešících SMT (SMT-solvery 3.4), a definující sémantiku v rámci SMT. SMT-LIB dále vyvíjí řadu nástrojů a knihoven pro práci s SMT a sadu benchmarků pro testování účinnosti a efektivity programů řešících SMT.

3.4 SMT-Solvery

SMT-solvery jsou programy řešící splnitelnost problému SMT [**?**]. Jako vstupní formát je SMT-solvery využiván SMT-LIB zmíněný v předchozí podkapitole. SMT-solvery se každý rok poměřují v soutěži International Satisfiability Modulo Theories Competition [14], kde se na sadě benchmarků zaměřených na měření správnosti a rychlosti v jednotlivých logikách využívaných v SMT-LIB.

Mezi významné SMT-solvery patří:

- Z3 [15] – *"State of art"* SMT-solver vyvíjený divizí Microsoft Research společnosti Microsoft. Z3 je vyvíjen s free software licencí MIT license a jeho zdrojový kód je volně dostupný v github repozitáři projektu[30]. Z3 nabízí binding pro programovací jazyky C, C++, Java, Python, OCaml a jazyky využívající platformu .NET, což přináší možnost využívat objekty a funkce programu Z3 v rámci software třetích stran.
- Boolector [32] – efektivní smt-solver pro teorii bitových vektorů bez kvantifikátorů s rozšířením o teorii polí. Boolector je vyvíjen v rámci Institute for Formal Models and Verification na Univerzitě Johanna Keplera v Linci.
- CVC4
- Yices
- SMT-RAT
- VeriT
- Beaver

V rámci praktické časti této práce byly zvoleny nástroje Z3 a Boolector jako referenční SMT-solvery.

3.5 Static Single Assignment Form

Static Single Assignment Form (SSA) [7] je forma transformovaného kódu využívána překladači při vytváření *mezijazyku* (*angl.* intermediate representation). Aby kód slňoval SSA musí pro něj platit, že každé proměnné je přiřazena hodnota pouze jednou a každá proměnná je definovaná před svým použitím. Transformace spočívá v přidání *pseudo-přiřazení* a *pseudo-použití*, kdy je původní proměnná, které je v kódu vícekrát přiřazována hodnota, rozdělena na více proměnných (verzí původní proměnné) a použití původní proměnné je nahrazeno za použití její odpovídající verze. Problémem tohoto převodu je větvení kódu, které se při převodu řeší analýzou CFG 2.2 a přdáním Φ-funkce na začátek základního bloku. Do nové verze proměnné se přiřadí výsledek Φ-funkce a dále se s ní pracuje. Převedení kódu na SSA umožňuje překladačům provádět mnoho optimalizací, mezi které patří propagace konstant, odstraňování mrtvého kódu a odstraňování zbytečné redundance.

Příklad jednoduchého převodu kódu na SSA formu:

Původní kód:

```
int x = 5;
int y = 10 + x;
int z~= x + y;
x = 2 * x + z;
z~= 5 * x + z;
return x + z;
```

Kód převedný na SSA:

```
int x_1 = 5;
int y_1 = 10 + x_1;
int z_1 = x_1 + y_1;
int x_2 = 2 * x_1 + z_1;
int z_2 = 5 * x_2 + z_1;
```

3.6 Rozbalování smyček

Rozbalování smyček (*angl. loop unrolling* nebo *loop unwinding*) je optimalizační metoda pro transformaci cyklů za účelem zvýšit rychlost vykonání programu (za cenu vyšších paměťových nároků). Optimalizace spočívá ve vykonání více kroků cyklu v jednom průchodu, což sníží režii na kontrolu podmínky a instrukce skoku na návěští začátku cyklu. Vedlejším efektem je také lepší využití cache.

V rámci formální verifikace je rozbalování smyček využito při ohraničeném ověřování modelů (bounded model checking 3.1). Jako model systému je využit jeho přechodový systém (např. graf toků řízení 2.2), ze kterého je výpočet rozbalen do stromové struktury výpočtu. Stromová struktura výpočtu se analyzuje a hranici tvoří počet rozbalení původního výpočtu (odpovídá počtu nově přidaných stavů). Hranice se využije při hledání protipříkladu, který by vyvrátil platnost ověřované vlastnosti. Pro verifikaci SAT-solvery je tedy třeba převést zápis modelu systému na logický výraz a vytvořit negaci výrazu odpovídajícího platnosti ověřované vlastnosti. Pokud SAT-solver vyhodnotí jako splnitelný výraz odpovídající modelu a zároveň výraz negace platnosti vlastnosti, byla nalezena chyba. Řešení poskytnuté SAT solverem je použito jako protipříklad pro vyvrácení platnosti ověřované vlastnosti systému.
Převod modelu pro SAT vyžaduje:

1. Zjednodušení toku řízení (nahrazení příkazů ovlivňující chod řízení `break` a `continue` za konkrétní `goto`, převod smyček for a do while na while, ad.)
2. Rozbalení všech smyček.
3. Převod na SSA 3.5.
4. Převod na soustavu rovnic.

Pro správné rozbalení smyčky musí být známá hranice (bound) po kterou se má rozbalit. Pokud je možné z modelu vypočítat potřebnou hranici (např. smyčky s pevně stanoveným počtem iterací), je možné smyčku zcela rozbalit. Pokud nelze hranici určit staticky (např. podmínka pro počet iterací závisí na vstupním parametru získaném dynamicky) musí být stanovena hodnota počtu maximálního počtu rozbalení smyčky (např. paramterem programu provádějícím ověřování modelu). Zcela rozbalená smyčka může být v posledním kroku rozbalení opatřena asercí neplatnosti podmínky pro provedení další iterace. Pokud má smyčka v sobě zanořené další smyčky, je možné je v rámci rozbalení sloučit.

Příklad rozbalování smyčky na rozbalení cyklu `while`:

Původní smyčka před rozbalením:

```
while(condition) {
  do();
}
```

První rozbalení smyčky:

```
if(condition) {
  do();
    while(condition) {
      do();
    }
}
```

Druhé rozbalení smyčky:

```
if(condition) {
  do();
  if(condition) {
    do();
    while(condition) {
      do();
    }
  }
}
```

V případě, že je možné určit maximální možné rozbalení smyčky, je smyčka zcela rozbalena a původní cyklus `while` se v kódu již vůbec nevyskytuje:

```
if(condition) {
  do();
  if(condition) {
    do();
    if(condition) {
      do();
      assert(!condition);
    }
  }
}
```

Příklad rozbalování smyčky na rozbalení cyklu `for`:

Původní smyčka před rozbalením:

```
for(int i = 0; i < 100; i++) {
  do(i);
}
```

Rozbalení smyčky na pět kroků v jedné iteraci. Výsledný program vykoná pouze 20 iterací namísto původních 100, postačí tedy pouze pětina podmíněného větvení a skokových instrukcí oproti původnímu kódu:

```
for(int i = 0; i < 100; i += 5) {
  do(i);
  do(i + 1);
  do(i + 2);
  do(i + 3);
  do(i + 4);
}
```

Příkladem verifikačního programu využívajícího rozbalování smyček pro ohraničené ověřování modelu je program CBMC [22] [24] [23], který se v roce 2014 umístil na prvním místě v mezinárodní soutěži ve verifikaci software Competition on Software Verification (SV-COMP'14) [4]. V posledním ročníku SV-COMP'16 [5] se přes velkou konkurenci (včetně programů Predator-HP [37] a Forester [27] vyvíjených v rámci výzkumné skupiny VeriFIT [17] Fakulty informačních technologií Vysokého učení technického v Brně) umístil na třetím místě v kategorii Floats.

3.7 Verifikace v projektu HADES

Projekt HADES [12] výzkumné skupiny VeriFIT se zaměřuje na automatizovanou verifikaci návrhů mikroprocesorů se zřetězenou linkou (se zaměřením na aplikačně-specifické mikroprocesory). Projekt využívá pro verifikaci kombinaci více metod mezi něž patří analýza toků dat, statická analýza pro vyhledávání možných hazardů a dynamická analýza pomocí parametrizace systému. Pro zápis modelů grafů toků dat zkoumaných mikroprocesorů využívá k tomu vytvořený jazyk VAM 2.5. Návrh mikroprocesorů se zřetězenou linkou přináší řadu problémů, které souvisí se zřetězeným zpracováním instrukcí na více úrovních zřetězené linky současně. Pokud více instrukcí vykonávaných mikroprocesorem současně (v různých úrovních zřetězené linky) provádí čtení nebo zápis na uložiště (do registrů nebo paměti) může dojít k tzv. *hazadru*, riziku že při zpracování instrukcí zřetězenou linkou nebude dodrženo pořadí operací tak jak bylo uvedeno v kódu programu a data, se kterými pracují instrukce, nebudou správná – výsledek výpočtu tedy nebude správný. Ošetření hazardů spočívá v *pozastavení zřetězené linky* (*angl. pipeline stall* nebo také *bubble*), kdy je zpracování operací pozastaveno o jeden takt. Rozlišujeme tři základní druhy hazardů: datové, strukturální a řídící.

Konkrétně se projekt zaměřil na ověření absence datových hazardů: *RAW* (Read-after-Write) hazardů [10] a později také *WAW* (Write-after-Write) a *WAR* (Write-after-Read) hazardů [11].

Definice 1 *Read after Wrie (čtení po zápisu) hazard nastane pokud instrukce čte data z místa, kde dřívější instrukce data zapisovala. V kódu operace zápisu předcházela operaci čtení, ale při zpracování instrukcí ve zřetězené lince dojde k zápisu až po přečtení (instrukce provádějící čtení získá neplatná data).*

Definice 2 *Write after Read (zápis po čtení) hazard nastane pokud instrukce zapisuje data na místo, odkud dřívější instrukce data četla. V kódu operace čtení předcházela operaci zápisu, ale při zpracování instrukcí ve zřetězené lince dojde ke čtení až po zápisu (instrukce provádějící čtení získá neplatná data).*

Definice 3 *Write after Write (zápis po zápisu) hazard nastane, pokud dvě instrukce zapisují data na místo, ale není mezi nimi žádná instrukce čtení (jinak by došlo k RAW nebo WAR hazardu). V kódu byla provedena nejdříve první operace zápisu a poté druhá operace zápisu, ale při zpracování instrukcí ve zřetězené lince dojde k provedení zápisů v opačném pořadí (po provedení obou instrukcí budou daném místě zapsaná neplatná data).*

Postup při detekci RAW hazardu se skládá z pěti kroků:

1. Analýza datových toků pro rozlišení jednotlivých stupňů zřetězené linky.
2. Kontrola konzistence správné implementace pro každý stupeň zřetězené linky.
3. Statická analýza datových cest instrukcí pro učení podmínek, které mohou vést ke vzniku RAW hazardu.
4. Vytvoření parametrizovaného systému modelujícího vzájemné působení mezi instrukcemi s možným konfliktem.
5. Analýza vytvořeného parametrizovaného systému.

Kapitola 4

Návrh a implementace překladače

Tato kapitola popisuje specifikaci požadavků na výsledný produkt této bakalářské práce – překladač grafu toků dat do logiky bitových vektorů, návrh řešení (jeho architekturu a jednotlivé komponenty, ze kterých se skládá, jejich kompozici a rozhraní přes, které spolu komunikují ve výsledném programu) a konkrétní popis implementace výsledného programu.

4.1 Specifikace požadavků

Cílem této práce je vytvořit program, který bude překládat grafy toků dat, zapsané ve formátu VAM, do formátu SMT-LIB. Generovaný kód potřebuje splňovat normy standardu SMT-LIB v2 pro verifikování SMT-Solvery. Ověřování výsledků bude prováděno dvojicí vybraných SMT-Solverů (dále *referenční SMT-Solvery*. Pro možnost automatizace celého procesu genrování přeloženého SMT-LIB kódu a jeho verifikaci solvery je potřeba optimalizovat generovaný kód pro dosažení co nejvyšší rychlosti verifikace.

Navržený překladač musí splňovat tyto požadavky:

- Určit zda je vstupní soubor ve formátu VAM validní
- Pro nevalidní vstupní soubor určit sémantické a syntaktické chyby
- Pro validní vstupní soubor generovat validní výstup ve formátu SMT-LIB v2
- Validní výstup ve formátu SMT-LIB v2 musí být ověřitelný pomocí SMT-Solverů
- Generovat výstup co nejlépe optimalizovaný pro rychlost verifikace SMT-Solvery
- Multiplatformní podporu pro operační systémy GNU/Linux a Microsoft Windows
- Minimalizovat systémové nároky pro použití ve virtualizovaném prostředí
- Nabídnout uživateli možnost vizualizace zpracovávaných grafů toků dat
- Zahrnovat testovací sadu ověřující správnou funkčnost pro všechny konstrukce jazyka VAM a kompatibilitu s referenčními SMT-Solvery
- Poskytovat vhodně dokumentovanou implementaci pro možnost budoucího rozšíření a využití s dalšími nástroji pracujícími s grafy toků dat

4.2 Architektura

Výsledkem této práce bude jeden spustitelný program – `Gravector`.

Gravector přijímá vstupní data (graf toku dat zapsaný ve formátu VAM) od uživatele, která čte ze zvoleného souboru, nebo ze standardního vstupu. Výstupní data (graf toku dat přeložený do formátu SMT-LIB v2, vygenerovaný podle parametrů zadaných uživatelem) generuje a zapisuje do zvoleného souboru, nebo na standardní výstup. Popřípadě generuje vizualizaci zadaného grafu ve formátu jazyka pro popis grafů DOT. Uživatel řídí proces překladu pomocí zadání argumentů při spuštění programu.

Samotný program překladače se skládá ze tří základních modulů – vstupní funkce main, objektového modelu grafu toku dat a parseru.

- `Vstupní funkce main` – stará se o zpracování parametrů zadaných uživatelem, vytvoření objektů potřebných pro překlad grafu a řízení chodu programu.
- `Objektový model grafu` – obsahuje jednotlive objekty reprezentující vrcholy a hrany grafu. Poskytuje také rozhraní pro práci s grafem (metody pro generování zápisu ve výstupním formátu SMT-LIB v2 nebo vizualizace).
- `Parser` – stará se o lexikální a syntaktickou analýzu vstupního grafu a vytvoření objekotvého modelu.

4.3 Objektový model grafu toků dat

Objektový model grafu toků dat je tvořen objektem obsahujícím kolekci všech objektů reprezentujících prvky grafu (hrany a vrcholy). U grafu toků dat patří prvky do těchto tříd:

–Hrany:

–Sigály spojující dva vrcholy.

–Vrcholy:

–Funkční bloky (Functional nodes) – reprezentují bezstavové uzly s obecně N vstupními hranami a obecně M výstupními hranami (v ukázkových grafech se pracuje prakticky výhradně s funkčnimi uzly s jediným výstupním signálem). Pro každý výstupní signál obsahuje výpočetní logiku reprezentovanou stromovou strukturou operatorů, které se aplikují pro výpočet další hodnoty ze vstupů. Funkční bloky v rámci modelu pracují bez zpoždění (jsou synchronizovány s paměťovými bloky na frekvenci hodinového signálu).

–Paměťové bloky

–Registry (Register nodes) – stavové uzly s pevnou bitovou šířkou (N >= 1 bitů) pro uchování hodnoty a sadou připojených signálů – datových (D a Q) a řídících (WRITE ENABLE, CLEAR, STALL, RESET). Zápis do registrů probíhá bez zpoždění a je synchronizován s hodinovým signálem

–Registrové soubory (Memory nodes) – stavové uzly, které obsahují obecně N adresovatelných paměťových buněk o stejné bitové šířce (jejich počet je větší než 1 a zároveň menší než druhá mocnina bitové šířky adresy). Obsahují obecně N přístupových portů (pro čtení, zápis, nebo čtení a zápis), každý se svou sadou připojených datových a řídících signálů (včetně signálů pro adresování).

Všechny prvky mají společnou rodičovskou třídu "Prvek grafu"(Graph element), která poskytuje základní rozhraní pro práci s prvky všech podtříd. Rozhraní obsahuje metody pro získání unikátního identifikátoru v rámci grafu, získání jména prvku a jména s prefixem typu prvku, metodu pro získání (výpočet) výstupní hodnoty při aktuální konfiguraci grafu a metodu pro vygenerování SMT-LIB kódu daného prvku (deklaraci či definici prvku ve formátu SMT-LIB). Při generování SMT s rozbalením smyček na více cyklů prvky signálů generují kód pro všechny rozbalené cykly (použití spravných vstupních hodnot pro daný cyklus je zajištěno assertací v rámci SMT kódu), a také paměťové bloky mohou mít rozgenerovány kód uložené hodnoty, pro každý cyklus, pokud je to potřeba.

4.4 Překladač

Překladač se stará o zpracování vstupního souboru ve formátu VAM, jeho lexikální a syntaktickou analýzu a vytvoření odpovídajícího objektového modelu grafu.

Bázová abstraktní třída Parser popisuje základní rozhraní parserů – metodu parse, a také jednotlivé vyjímky, pro lexikální, syntaktické a sémantické chyby, které mohou být vyvolány při překladu.

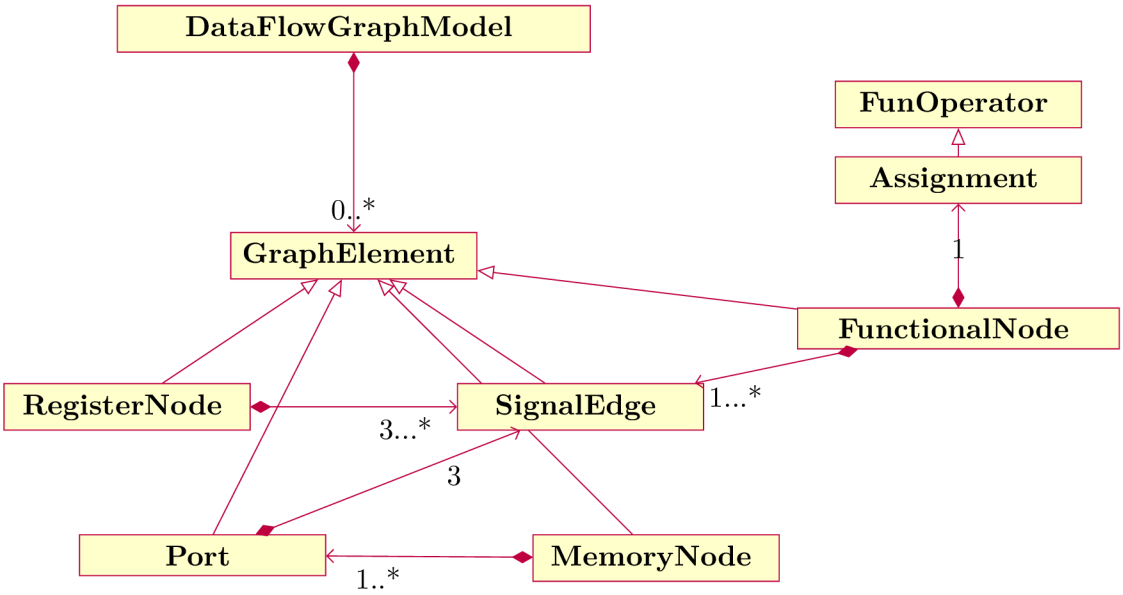
Z bázové abstraktní třídy dědí třídy parserů, které překládají jednotlivé vstupní formáty – pro formát VAM tedy existuje VAMParser. Rozšíření o podporu dalších formátu popisujících grafy toků dat je tedy možné přidáním třídy implementující rozhraní základního Parseru pro zpracování daného formátu.

VAMParser provádí lexikální a syntaktickou analýzu v rámci jednoho průchodu vstupním řetězcem. Lexikální symboly, tokeny, dále zpracovává při syntaktické analýze a vytvaří abstraktní syntaktický strom. Z abstraktního syntaktického stromu generuje výsledné objekty prvků grafu a ukládá do svých vnitřních struktur, tak aby zachoval původní hierarchii. Kontextové informace zahrnující více prvků grafu (jako je například propojení jednotlivých prvků spojujícími signály) se provádí včetně kontroly signálů až na konci analýzy, kdy již jsou zpracovány všechny prvky grafu. Výsledkem překladu je jeden objekt modelující zpracovaný graf, obsahující kolekci všech jeho prvků provázaných v zadané hierarchii. Pro každý nový prvek grafu vytváří unikátní identifikátor, pomocí kterého je možné daný prvek adresovat v kolekci všech prvků. Protože jména prvků jsou unikátní jen v rámci své kategorie (v grafu může existovat signál a registr označený stejným jménem), využívají prvky v rámci modelu jména prefixovaná zkratkou své třídy [1], která již zajišťují unikátnost v rámci grafu. Níže je uveden diagram tříd znázorňující třídy modelu grafu toků dat4.1.

4.5 Další komponenty

Kromě generování kódu ve formátu SMT-LIB pro verifikaci nabízí Gravector také možnost generovat vizualizaci zadaného grafu toků dat. Pro vizualizaci je vytvořen objektový model grafu toků dat, z něhož je následně vytvořen zápis grafu v jazyce DOT[18]. DOT graf je možné programem `dot` (nástroj ze sady graphviz[8]) převést na obrázek(jpg, bmp, png, svg) nebo do formátu pdf, či postscript. Pro převod do DOT je použita knihovna Boost Graph Library [34].

[1] prefixy: `"sig_"`, `"fun_"`, `"reg_"`, `"mem_"`



Obrázek 4.1: Diagram tříd znázorňující třídy modelu grafu toků dat

4.6 Implementace

Pro implementaci překladače byl zvolen jazyk C++ (standard C++14), který umožňuje reliazaci objektového návrhu bez ztráty výkonu při použití objektů. C++ také umožňuje dobré škálování i při překladu velkých grafů, které obsahují velké možství prvků, pro které musí být vytvořeny odpovídající objekty a následně propojeny pomocí ukazatelů. Využity jsou standardní knihovny C++ STL a knihovny Boost [1], které poskytují množství datových struktur a implementací algoritmů a nemusí se tedy znovu implementovat od začátku v rámci projektu. Pro reprezentaci hodnot bitových vektorů (prvků v objektovém modelu grafu toků dat) je využita struktura `boost::dynamic_bitset`, která má výhodu proti třídě `bitset` v možnosti vytvoření bitového vektoru s dynamicky získanou velikostí (jako je tomu u vytváření prvků při parsování vstupního grafu) a lepší sadu metod pro manipulaci s jednotlivými bity vektoru než `std::vector<bool>`.

4.6.1 Kompozice objektového modelu

Objekty prvků grafu toků dat jsou vytvořeny při parsování vstupního grafu Parserem a následně jsou propojeny pomocí ukazatelů, jež reprezentují relace mezi prvky a jsou použity v metodách, kde prvek využívá ostatní s ním propojené prvky. V instanci třídy `DataFlowGraphModel` vytvořené Parserem jako výsledek překladu vstupního grafu jsou všechny prvky elementů uloženy v kolekci (`std::vector<GraphElement *> elements`, která umožňuje přístup ke všem prvkům pomocí indexu – interního `id` prvku, který je prvku přiřazen při překladu a je v něm uložen. Kromě hlavní kolekce obsahuje model kolekce ukazatelů na prvky rozdělené podle kategorií:

- `std::vector<SignalEdge *>` signalEdges
- `std::vector<RegisterNode *>` registers
- `std::vector<MemoryNode *>` memories

- `std::vector<FunctionalNode *>` functionalNodes

Dále obsahuje model kolekci řetězců textových zápisů asercí zadaných uživatelem.

4.6.2 Přidání asercí uživatelem

Uživatel má možnost zadat, které vlastnosti se mají na modelu přeloženém do formátu SMT-LIB ověřovat pomocí aserčních meta-značek. Meta-značku uživatel zapíše do VAM zápisu k příslušnému prvku, jehož vlastnosti chce ověřovat. V současné verzi jsou dostupné dvě aserční-meta značky `meta (assert)`, pro jednoduchou aserci, a `meta (assert_foreach)`, pro aserci v rámci rozbalení cyklů. Aserce se do meta-značky zapisují ve formátu SMT-LIB a je nutné (v současné verzi) požít prefixovaná jména prvků. Aserce se zpravidla váží k prvklm u kterých jsou uvedeny, u asercí pro rozbalení cyklů je možné použít aliasy `$this`, `$previous` a `$next` pro porovnání mezi rozbalením vůči předcházejícím a následujícím ierací (odpovídají aplikaci správné verze výstupu na prvku).

Příklad jednoduché aserce:

```
(meta (assert "!= sig_v_fun_out (_ bv0 8)"))
```

Příklad aserce pro rozbalení:

```
(meta
  (assert_foreach "and
    (bvult $previous $this)
    (bvult (bvmul $this (_ bv2 8)) $next)"
  )
)
```

4.6.3 Generování SMT-LIB kódu

Pokud je vytvořen validní objektový model grafu, může být vygenerován odpovídající SMT-LIB kód. Generování začíná zápisem informací o použité logice (konkrétně se využívá logika `QF_AUFBV` – uzavřené formule bez kvantifikátorů pro teorii bitových vektorů, teorií polí bitových vektorů a nedefinovaných funkcí) a verzi SMT-LIB (použita verze: smt-lib-version 2.0 podporovaná referenčními SMT-solvery). Dále jsou deklarovány všechny signály (a v případě rozbalení cyklu jejich další verze). Poté jsou definovány paměťové uzly - nejrpve všechny registry a poté registrové soubory, je přidán signál pro reprezentaci uložené hodnoty v registru (pro danou iteraci). Posledními převedenými prvky grafu jsou funčkní uzly, které se definují pouze jednou (struktura jejich realizace se v čase nemění) a před samotnou definicí jsou deklarovány funkce pro použití na místě abstraktních operátorů (jejich implementace není známá v době překladu – o jejich definici se již postará SMT-solver). Při generování funkce funkčního uzlu se prochází jeho operátory přizazení, obsahující stromovou strukturu operátorů, která se postupně převádí pomocí průchodu DFS (Depth-first search – průchod do hloubky). Správné propojení prvků je zajištěno asercí hodnot na vstupech signálů (platí za přiřazení) – asertuje hodnotu signálů na stejnou hodnotu jaká je na prvku, ke kterému je připojen na výstup (získá ji apliakcí – hodnota funkčního nebo paměťového uzlu pro signály v určitém stupni rozbalení smyčky). Posledním krokem je připojení uživatelem zadaných asercí do generovaného kódu (pro aserce `asert_foreach`

dojde k rozgenerování a nahrazení aliasů za funkce verzí konkrétních prvků). Uživatelem zadané vlastnosti jsou ověřovány ve vlastním `scope` pomocí provedení příkazu `(push 1)`, před vložením nové aserce a provedením příkazu `(pop 1)` pro odebrání splněné aserce z interního zásobníku SMT-solveru (ukládá deklarace a definice). Před ukončením zápisu jsou ještě přidány příkazy pro SMT-solver: `(check-sat)`, `(get-model)` a `(exit)`.

4.7 Ovládání programu

Uživatel programu Gravector zpravidla musí provést čtyři kroky pro dosažení svého cíle – verifikace vlastností systému.

1. Vytořit zápis modelu systému ve formátu grafů toků dat (v jazyce VAM, nebo jiném formátu při použití rozšíření o parser pro daný jazyk).
2. Opatřit model meta-značkami pro aserci žádaných vlastností (meta značky assert a assert_foreach).
3. Spustit program gravector s příslušnými parametry ovlivňujícími generování výstupního kódu ve formátu SMT-LIB.
4. Použít SMT solver na vygenerovaný kód ve formátu SMT-LBI pro provedení verifikace.

Přehled argumentů příkazové řádky pro spuštění programu Gravector:

Argumenty příkazové řádky programu Gravector			
`-h`	`[--help]`		vypíše nápovědu k programu
`-f`	`[--file]`	filename	vstupní soubor filename pokud není zadán, program čte vstup ze standardního vstupu
`-o`	`[--output]`	filename	výstupní soubor filename pokud není zadán, program zapisuje na standardní výstupu
`-c`	`[--cycles]`	cycles_count	rozbalení smyček na cycles_count iterací
`-g`	`[--graphviz]`		mód generování vizualizace grafu generuje výstup ve formátu DOT
`-v`	`[--version]`		vypíše verzi programu Gravector
`-d`	`[--debug]`		debugovací mód vypisuje pomocné informace během překladu

Příklad spuštění:

```
gravector -f test.vam -o test.smt -c 10
```

Vygenerovaný SMT-LIB kód (pokud byl vstupní model v souboru test.vam validní) je uložen do souboru test.vam a může být dále použit jako vstup pro SMT solvery.

Příklad použití pro SMT-solver Z3 3.4:

```
z3 -smt2 -file:test.smt
```

Příklad použití pro SMT-solver Boolector 3.4:

```
boolector -smt2 -m -i -smt2-model test.smt
```

Kapitola 5

Testování

Testování je nezbytným krokem při vývoji software. Testování může pomoci při odhalení chyb a ověření funkčnosti jejich oprav. Slouží také k ověření správné funkčnosti na připravených testovacích případech. V této kapitole je popsán postup jakým byl program Gravector, při svém vývoji testován, a technologie využité při testování. V závěru kapitoly jsou uvedeny platformy a verze software využitého při vývoji a testování.

5.1 Fáze testování

Testovní provedené na projektu Gravector obsahovalo tyto tři fáze:

1. *jednotkové testování* (*angl. unit testing*) – ověření správné funkčnosti jednotlivých komponent (tříd a jejich metod) využitých v projektu, izolovaně od zbytku systému. Bylo provedeno na začátku implementac, ještě před propojením všech komonent, pomocí testovacího frameworku `Catch` (C++ Automated Test Cases in Headers) [31], který umožňuje jednoduché použití vložením jednoho C++ hlavičkového souboru (catch.hpp).

2. *integrační testování* (*angl. integration testing*) – ověření správné integrace systému (propojení komponent a jejich vzájemné komunikace). Návrh architektury programu Gravector umožnil provést jen několik integračních testů a přejít k systémovému testování.

3. *systémové testování* (*angl. system testing*) – závěrečná fáze testování ověřující fungování programu Gravector jako celku s využitím sady připravených modelů ve formátu VAM. Přidáváním modelů zaměřených na různé specifické případy, které mohou nastat ve vstupním grafu toků dat se odhalily některé nedostatky, které nebyly zachyceny jednotkovými a integračními testy a mohly být následně opraveny.

Sada testovacích modelů zahrnuje modely vytvořené pro ověření správného překladu všech operátorů podporovaných v jazyce VAM, jejich kombinaci, jejich kombinaci s paměťovými bloky a komplexními příklady velkých grafů toků dat (modely procesorů `TinyCPU` a `DLX5` převzetými z projektu HADES). Správný překlad celé sady připravených modelů demostruje funkčnost programu Gravector a jeho možnosti. Sada testovacích modelů také poslouží uživatelům programu Gravector jako zdroj příkladů pro pomoc s vytvářením vlastních modelů.

5.2 Platformy použité pro testování

Program Gravector byl vyvíjen na operačním systému Fedora 22 64-bit s využitím software:

- g++ (GCC v5.3.1)
- knihovna BOOST v1.57
- Z3 v4.4.0
- Boolector v2.2.0

Pro testovací a demonstrační účely byl dále zřízen VPS kontejner (droplet) u služby DigitalOcean[16]. Droplet obsahuje OS Fedora 23 64-bit a software:

- g++ (GCC v5.3.1)
- knihovna BOOST v1.58
- Z3 v4.4.1
- Boolector v2.2.0

Na platformě windows byl překlad proveden pomocí nástroje mingw-w64.

Kapitola 6

Závěr

Cílem této bakalářské práce bylo navrhnout a implementovat překladač grafů toků, zapsaných v jazyce VAM, do formátu SMT-LIB. Vyvinutý překladač má rozšířit možnosti analýzy grafů toků dat a přinést nové možnosti jejich verifikace (představuje odlišný přístup než předchozí práce v rámci projektu HADES).

Správná funkčnost překladače byla demonstrována na sadě ukázkových modelů. Sada obsahuje několik kategorií modelů (modely pro ověření správného překladu operátorů, paměťových prvků, kombinaci funkčních a paměťových prvků a modelech jednoduchých mikroprocesorů) a nachází se v příloze této technické zprávy. K testování výstupu překladače byly využity SMT-Solvery Z3 a Boolector.

Další rozšíření výsledného programu bude možné přídáním podpory dalšího vstupního jazyku přidáním nového parseru, který bude vytvářet objektový model kompatibilní s původním řešením. Nově podporovaným jazykem by mohl být jazyk vytvořený jako dialekt jazyka DOT se zaměřením na vizualizaci grafů toků dat. Další úpravy mohou nastat po analýze zpětné vazby uživatelů a mohou vést k přidání nebo úpravě meta-značek přidaných do formátu VAM pro specifikaci vlastností systému, které se mají ověřovat.

Literatura

[1] *Boost C++ libraries [online].* [Online; navštíveno 10.5.2016].
URL http://www.boost.org/

[2] Barrett, C.; Fontaine, P.; Tinelli, C.: *The SMT-LIB Standard Version 2.5 [online].* 2015, [Online; navštíveno 25.11.2015].
URL http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.5-r2015-05-28.pdf

[3] Barrett, C.; Fontaine, P.; Tinelli, C.: *The Satisfiability Modulo Theories Library (SMT-LIB) [online].* www.SMT-LIB.org, 2016, [Online; navštíveno 10.5.2016].

[4] Beyer, D.: *Status Report on Software Verification (Competition Summary SV-COMP 2014) [online].* 2014, [Online; navštíveno 10.5.2016].
URL http://www.sosy-lab.org/~dbeyer/Publications/2014-TACAS.Status_Report_on_Software_Verification.pdf

[5] Beyer, D.: *Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016)[online].* 2016, [Online; navštíveno 10.5.2016].
URL http://www.sosy-lab.org/~dbeyer/Publications/2016-TACAS.Reliable_and_Reproducible_Competition_Results_with_BenchExec_and_Witnesses.pdf

[6] Biere, A.: *The AIGER And-Inverter Graph (AIG) Format Version 20071012 [online].* [Online; navštíveno 12.5.2016].
URL http://fmv.jku.at/aiger/FORMAT

[7] Bilardi, G.; Pingali, K.: *Algorithms for Computing the Static Single Assignment Form. J. ACM*, ročník 50, č. 3, Květen 2003: s. 375–425, ISSN 0004-5411.

[8] Bilgin, A.; Ellson, J.; Gansner, E.; aj.: *Graphviz - Graph Visualization Software [online].* [Online; navštíveno 10.5.2016].
URL http://www.graphviz.org

[9] Brayton, R.; Mishchenko, A.: *Scalably-Verifiable Sequential Synthesis [online].* [Online; navštíveno 12.5.2016].
URL
http://www.eecs.berkeley.edu/~alanmi/publications/2007/tech07_pss.pdf

[10] Charvát, L.; Smrčka, A.; Vojnar, T.: Using Formal Verification of Parameterized Systems in RAW Hazard Analysis in Microprocessors. In *2014 15th International Microprocessor Test and Verification Workshop*, Dec 2014, ISSN 1550-4093, s. 83–89.

[11] Charvát, L.; Smrčka, A.; Vojnar, T.: Microprocessor Hazard Analysis via Formal Verification of Parameterized Systems. In *Proceedings of the 15th International Conference on Computer Aided Systems Theory (EUROCAST 2015)*, The Universidad de Las Palmas de Gran Canaria, 2015, ISBN 978-84-606-5438-4, s. 193–194.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=10767

[12] Charvát, L.; Smrčka, A.; Vojnar, T.: *HADES (Hazard Detection System) [online]*. FIT VUT v Brně, 2012-01-24 [cit. 2008-11-28], [Online; navštíveno 10.5.2016].
URL http://www.fit.vutbr.cz/research/groups/verifit/tools/hades

[13] Charvát, L.; Smrčka, A.; Vojnar, T.: *Variable-Assignment-Model (VAM) – Structure and Language [online]*. FIT VUT v Brně, 2012-01-24 [cit. 2008-11-28], [Online; navštíveno 10.5.2016].
URL http://www.fit.vutbr.cz/research/groups/verifit/tools/hades/download/language.txt

[14] Conchon, S.; Déharbe, D.; Heizmann, M.; aj.: *SMT-COMP 2016 [online]*. 2016, [Online; navštíveno 10.5.2016].
URL http://smtcomp.sourceforge.net/2016/

[15] De Moura, L.; Bjørner, N.: *Z3: An Efficient SMT Solver*. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, Berlin, Heidelberg: Springer-Verlag, 2008, ISBN 3-540-78799-2, 978-3-540-78799-0, s. 337–340.
URL http://dl.acm.org/citation.cfm?id=1792734.1792766

[16] DigitalOcean Inc.: *DigitalOcean [online]*. [Online; navštíveno 12.3.2016].
URL https://www.digitalocean.com/

[17] Fakulta informačních technologií VUT v Brně: *Výzkumná skupina automatizované analýzy a verifikace - VeriFIT [online]*. [Online; navštíveno 10.5.2016].
URL http://www.fit.vutbr.cz/research/groups/verifit/

[18] Gansner, E.; Koutsofios, E.; North, S.: *Drawing graphs with dot [online]*. 2006, [Online; navštíveno 10.5.2016].
URL http://www.graphviz.org/Documentation/dotguide.pdf

[19] Hennessy, J. L.; Patterson, D. A.: *Computer architecture: a quantitative approach (5th ed.)*. Elsevier, Morgan Kaufmann, 2011, ISBN 978-0-12-383872-8.

[20] Kovařík, D.: *Interaktivní simulátor pro grafy toku dat*. Bakalářská práce, Fakulta informačních technologií VUT v Brně, 2015.

[21] Kovář, P.: *Teorie grafů [online]*. 2012-06-12 [cit. 2008-11-28], [Online; navštíveno 10.5.2016].
URL http://mi21.vsb.cz/sites/mi21.vsb.cz/files/unit/skriptum_teorie_grafu_rozsirene_interaktivne.pdf

[22] Kroening, D.: *CBMC homepage [online]*. [Online; navštíveno 10.5.2016].
URL http://www.cprover.org/cbmc/

[23] Kroening, D.: *CBMC: Bounded Model Checking for ANSI-C [online]*. 2010, [Online; navštíveno 10.5.2016].
URL http://www.cprover.org/cbmc/doc/cbmc-slides.pdf

[24] Kroening, D.; Clarke, E.; Yorav, K.: *Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking* . In *Proceedings of DAC 2003*, ACM Press, 2003, ISBN 1-58113-688-9, s. 368–371.

[25] Křena, B.; Vojnar, T.: *Automated formal analysis and verification: an overview. International Journal of General Systems*, ročník 42, č. 4, 2013: s. 335–365.

[26] Lattuada, M.; Ferrandi, F.: *Modeling pipelined application with Synchronous Data Flow graphs*. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*, July 2013, s. 49–55.

[27] Lengál, O.; Habermehl, P.; Holík, L.; aj.: *Forester—Tool for Verification of Programs with Pointers [online]*. [Online; navštíveno 10.5.2016].
URL http://www.fit.vutbr.cz/research/groups/verifit/tools/forester

[28] Matyáš, J.: *Překladač jazyka VHDL pro potřeby formální verifikace*. Bakalářská práce, Fakulta informačních technologií VUT v Brně, 2015.

[29] Meinel, C.; Theobald, T.: *Algorithms and Data Structures in VLSI Design: OBDD - Foundations and Applications*. Springer, 1998, ISBN 3-540-64486-5.

[30] Microsoft Research: *The Z3 Theorem Prover [online]*. [Online; navštíveno 10.5.2016].
URL https://github.com/Z3Prover/z3

[31] Nash, P.: *Catch [online]*. [Online; navštíveno 20.10.2015].
URL https://github.com/philsquared/Catch

[32] Niemetz, A.; Preiner, M.; Biere, A.: *Boolector 2.0*. JSAT*, ročník 9, 2015: s. 53–58, [Online; navštíveno 10.5.2016].*
URL https://satassociation.org/jsat/index.php/jsat/article/view/120

[33] Nieuwenhuis, R.; Oliveras, A.; Tinelli, C.: Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T). J. ACM*, ročník 53, č. 6, Listopad 2006: s. 937–977, ISSN 0004-5411.*

[34] Siek, J.; Lee, L.-Q.; Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002, ISBN 0201729148.

[35] Smrčka, A.: ITS - Testování se znalostı kódu, datové toky [online]. [Online; navštíveno 10.5.2016].
URL https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/ITS-IT/lectures/3-dataflow.pdf

[36] Steele, G. L., Jr.: Common LISP: The Language (2Nd Ed.). Newton, MA, USA: Digital Press, 1990, ISBN 1-55558-041-6.

[37] *Vojnar, T.; Dudka, K.; Peringer, P.; aj.: Predator [online]. [Online; navštíveno 10.5.2016].*
URL http://http://www.fit.vutbr.cz/research/groups/verifit/tools/predator

Přílohy

Seznam příloh

Příloha A

Obsah CD

- `/`
  - `src` ..... zdrojové soubory
  - `bin` ..... adresář pro přeložený spustitelný binární soubor programu
  - `obj` ..... adresář pro objektové soubory pro překladu programu
  - `doc` ..... adresář s dokumentací projektu
    - `tz_src` ..... adresář se zdrojovými soubory technické zprávy
    - `xsusov01.pfd` ..... PDF verze této technické zprávy
  - `test` ..... adresář obsahující modely pro testovací účely
    - `operators` ..... příklady modelů pro testy jednotlivých operátorů
    - `memory_blocks` ..... příklady modelů pro testy paměťových bloků
    - `simple` ..... příklady modelů s kombinací funkčních a paměťových bloků
    - `complex` ..... příklady složitějších modelů
    - `out` ..... adresář pro ukládání výsledků testů
  - `Makefile` ..... skript pro překlad programu
  - `README.md` ..... stručný popis popis programu a jeho ovládání
  - `test.sh` ..... skript pro spuštění testovací sady

Příloha B

Parametry spuštění programu

Spuštění programu:
`gravector [-h] [-f FILENAME] [-o FILENAME] [-c CYCLES_COUNT] [-g] [-v] [-d]`

`-h`, `[--help]`	vypíše nápovědu k programu
`-f`, `[--file]` FILENAME	vstupní soubor FILENAME (pokud není zadán, program čte vstup ze standardního vstupu)
`-o`, `[--output]` FILENAME	výstupní soubor FILENAME pokud není zadán, program zapisuje na standardní výstupu
`-c`, `[--cycles]` CYCLES_COUNT	rozbalení smyček na CYCLES_COUNT iterací
`-g`, `[--graphviz]`	mód generování vizualizace grafu generuje výstup ve formátu DOT
`-v`, `[--version]`	vypíše verzi programu Gravector
`-d`, `[--debug]`	debugovací mód vypisuje pomocné informace během překladu