VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

TRASOVATEĽNOSŤ OPTIMALIZOVANÉHO GENEROVANÉHO KÓDU

TRACEABILITY OF OPTIMIZED GENERATED CODE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **DANIEL GAVENDA**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. ZBYNĚK KŘIVKA, Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2021/2022

Zadání bakalářské práce

Student: **Gavenda Daniel**
Program: Informační technologie
Název: **Trasovatelnost optimalizovaného generovaného kódu**
Traceability of Optimized Generated Code
Kategorie: Překladače

Zadání:

1. Seznamte se s nástrojem MATLAB, jeho překladovým prostředím a s proprietárními nástroji používanými ve firmě Honeywell pro vývoj založený na modelech.
2. Analyzujte, jak nastavení překladového prostředí ovlivňuje průběh i výsledek překladu modelů do jazyka C se zaměřením na provedené optimalizace a s ohledem na aeronautické standardy.
3. Na základě konzultací navrhněte metodu pro analýzu a případně odstranění vybraných optimalizací za účelem lepší trasovatelnosti vygenerovaného kódu.
4. Metodu implementujte a ověřte její funkčnost na dílčích částech modelu a alespoň na jednom dodaném/zadaném modelu.
5. Implementovanou metodu zhodnoťte a navrhněte další vylepšení.

Literatura:

- Meduna, A.: Elements of Compiler Design. New York, US, Tailor & Francis, 2008.
- Grune, D.: Modern Compiler Design, 2. vydání. Springer, 2016.
- Dokumentace k produktům firmy MathWork. Dostupné na https://www.mathworks.com/ [cit. 2021-10-08]
- Standardy organizace RTCA dle doporučení konzultanta (např. DO-178C).

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Křivka Zbyněk, Ing., Ph.D.**
Konzultant: Dohnal Roman, Mgr., Honeywell
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 11. května 2022
Datum schválení: 25. října 2021

Abstrakt

Práca sa zaoberá vytvorením aplikácie v jazyku Python, ktorá zo zadaného modelu a kódu z neho vygenerovaného poskytne užívateľovi analýzu optimalizácií vykonaných pri automatickom generovaní kódu. Táto analýza bude užitočná pre modelových návrhárov vo firme Honeywell International v oddelení AeroSpace. Umožní im tak jednoduchšie lokalizovanie a odstránenie či upravenie častí modelu, ktoré boli optimalizované.

Abstract

The goal is to create an application that analyzes the optimizations applied during the translation of an input model into the corresponding generated code. This analysis will be provided to developers in order to show sources of potential automated review failures. In this case, the analysis reports should serve as a guide to a model designer to transform the model to avoid problematic optimizations.

Kľúčové slová

trasovateľnosť, automaticky generovaný kód, optimalizácie, vývoj založený na modeloch, analýza generovaného kódu, rozbor syntaktického stromu, Python, MATLAB, Stateflow

Keywords

traceability, automatically generated code, optimizations, model-based developement, syntax tree parsing, Python, MATLAB, Stateflow

Citácia

GAVENDA, Daniel. *Trasovateľnosť optimalizovaného generovaného kódu.* Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Zbyněk Křivka, Ph.D.

Trasovateľnosť optimalizovaného generovaného kódu

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Zbyňka Křivky, Ph.D. Ďalšie informácie mi poskytol pán Mgr. Roman Dohnal. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Daniel Gavenda
11. mája 2022

Poďakovanie

Rád by som sa poďakoval vedúcemu práce pánovi Ing. Zbyňkovi Křivkovi, Ph.D za odbornú pomoc a rady pri riešení tejto práce.

Rovnako by som sa chcel poďakovať konzultantovi pánovi Mgr. Romanovi Dohnalovi za trpezlivosť a ochotu odpovedať na všetky moje otázky.

Obsah

Kapitola 1

Úvod

Zadanie práce vzniklo v spolupráci s firmou Honeywell International, a.s. (ďalej len Honeywell), pod oddelením Aerospace. Toto oddelenie sa zaoberá vývojom softvéru založeným na modeloch, ktorý má široké využitie v letectve. To si vyžaduje, aby boli splnené požiadavky dané odporúčaniami popísané v dokumentoch [4] a [5]. Kvôli týmto odporúčaniam je potrebné, aby každý navrhnutý model prešiel revíziou. V rámci firmy bol vytvorený interný nástroj, ktorý prevádza revíziu automaticky. Avšak ten je limitovaný v prípade, že pri generovaní je kód optimalizovaný. Cieľom tejto práce je vytvoriť rozšírenie pre daný existujúci nástroj. Ak pôvodný nástroj prehlási model za neplatný, vytvorené rozšírenie sa pokúsi o analýzu vykonaných optimalizácií, ktoré mohli zabrániť automatickej revízii. O dôvode a mieste vykonania optimalizácií v modeli bude informovať návrhára modelu (ďalej len užívateľa). Na základe týchto informácii užívateľ bude môcť zmeniť model tak, aby mohol pôvodný interný nástroj vykonať automatickú revíziu. Súčasťou nej je aj verifikácia obojstrannej trasovateľnosti medzi kódom a modelom. Aplikácia podporuje podmnožinu blokov z knižnice blokov (Block library), ktorá je popísaná spolu s využívanými optimalizáciami v kapitole 2. Kapitola 3 sa bližšie zaoberá štruktúrou vstupných súborov – vygenerovaným kódom a modelovou reprezentáciou, a taktiež samotnou trasovateľnosťou. Za ňou nasleduje kapitola 4, ktorá popisuje architektúru aplikácie. Kapitola 5 špecifikuje spôsob analýzy kódu, modelu, algoritmus hľadania optimalizácií a kapitola 6 sa venuje overeniu funkcionality na dvoch sadách testovacích dát. Na vlastnej sade modelov a na sade reálnych modelov poskytnutých firmou. Posledná kapitola 7 sa venuje zhrnutiu výsledkov dosiahnutého riešenia a obmedzeniam analýzy optimalizácií.

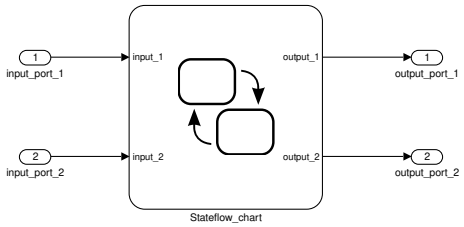
Kapitola 2

Nástroje pre návrh modelov a generovanie kódu

Modely, ktorých analýzou sa táto práca zaoberá sú vytvorené pomocou rozšírenia známej platformy MATLAB® nazývané Simulink® . Táto nadstavba ponúka grafické prostredie pre vývoj a simuláciu navrhnutých modelov. Každý model sa skladá z blokov. Bloky vykonávajú všetky výpočtové operácie v modeli. Sú prepájané signálmi – časovo premenná veličina, ktorá má hodnoty vo všetkých bodoch v čase [3]. Má svoj dátový typ, meno a rozmer.

Simulink® obsahuje širokú ponuku stavebných blokov pre modely, ale pre jednoduchosť je predmetom analýzy konkrétna skupina – Diagramy stavových automatov (angl. state–flow charts, ďalej len stavové diagramy) viď obr. 2.1. Stavové diagramy umožňujú modelovanie Mealy, Moore a kombinovaných automatov, rozhodovacích diagramov (angl. state–less charts) a pravdivostných tabuliek. Prvky tvoriace rôzne diagramy sa môžu v rámci modelu kombinovať a prepájať. To znamená, že napr. jeden blok môže obsahovať stavy, v ktorých sú definované pravdivostné tabuľky a sú prepojené rozhodovacími diagramami. Avšak samotné prvky umožňujú len riadenie toku programu a na vykonávanie príkazov pre manipuláciu dát stavové diagramy využívajú tzv. jazyk akcií (angl. action language). Jazyk akcií je dostupný v dvoch variantách. Buď ako podmnožina skriptovacieho jazyka MATLAB, alebo ako výrazy, bitové operácie a jednoduché syntaxové skratky jazyka C [1]. Ďalej Simulink® umožňuje simuláciu navrhnutého modelu s diskrétnym alebo spojitým časom. Aby bolo možné využiť modely v leteckých systémoch je potrebné ich preložiť na produkčný kód. Jazyk a prekladové prostredia sa môžu líšiť od požiadavok zákazníka. V tejto práci je využívaný jazyk C ako produkčný kód. Na jeho automatické generovanie sa využívajú moduly Embeded Coder a Real-Time Workshop. Skupina modulov, ktoré spolupracujú na preklade modelového návrhu na produkčný kód prinášajú radu optimalizácií.

Odporúčanie DO–178C [4] vyžaduje zdokumentované obojsmerné spojenia nazývané stopy (angl. traces) medzi požiadavkami na nízkej úrovni (automaticky generovaný kód) a požiadavkami na vysokej úrovni (model v Simulink®). Musí byť teda zabezpečené, že každá požiadavka na vysokej úrovni je implementovaná zdrojovým kódom a každý riadok zdrojového kódu má svoj účel (súvisí s požiadavkou). Dôsledkom týchto optimalizácií môžu byť pridané, alebo odobrané riadky, či zmena štruktúry generovaného kódu. To môže spôsobiť, že interný nástroj nebude schopný zdokumentovať všetky obojstranné spojenia medzi modelom a kódom. Následne preto vyhodnotí model ako neplatný.

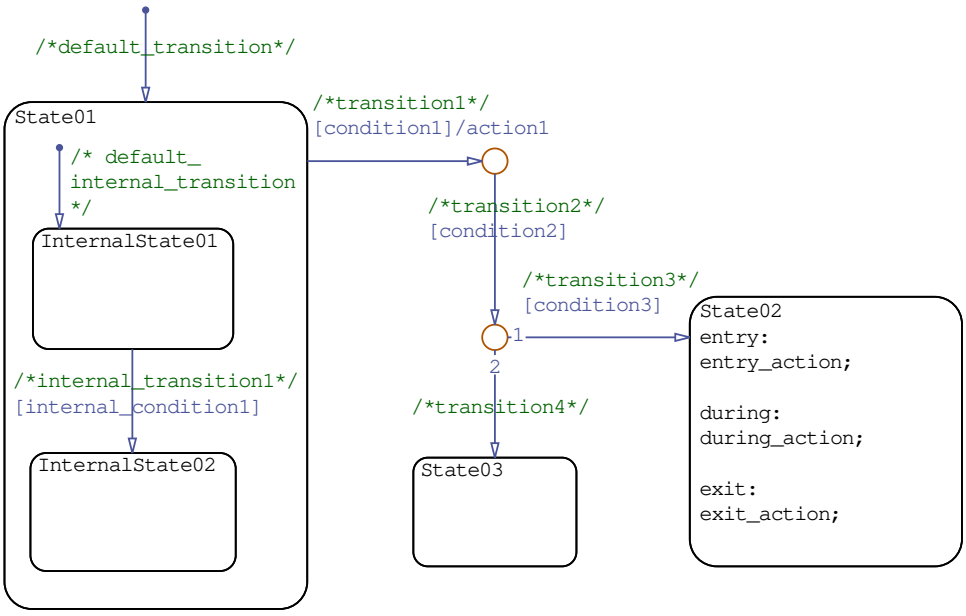


Obr. 2.1: Blok stavového diagramu v prostredí Simulink®

2.1 Popis vnútornej štruktúry stavových diagramov

Sekcia obsahuje detailný popis stavových diagramov a ich vlastností, ktorých využitie povoľuje interný nástroj. Medzi nepovolené patrí napr. dekompozícia paralelných stavov [6], akcie prechodových podmienok, prechody medzi úrovňami stavov a i. Všetky ďalej popisované stavové diagramy využívajú jazyk C pre popis akcií stavov, prechodov a sú realizované v diskrétnom čase.

Stavové diagramy sú zložené z neprázdnych množín prechodov a výstupov, množín stavov, pravdivostných tabuliek, vstupov a uzlov. Každý stavový diagram je na začiatku simulácie neaktívny. Pri prvej aktivácii, tzn. pri prvom vstupe do diagramu, nastáva proces inicializácie. Počas neho sa vykonajú všetky vstupné prechody na najvyššej vrstve viď `default transition` na obr. 2.2.



Obr. 2.2: Ilustračný stavový diagram

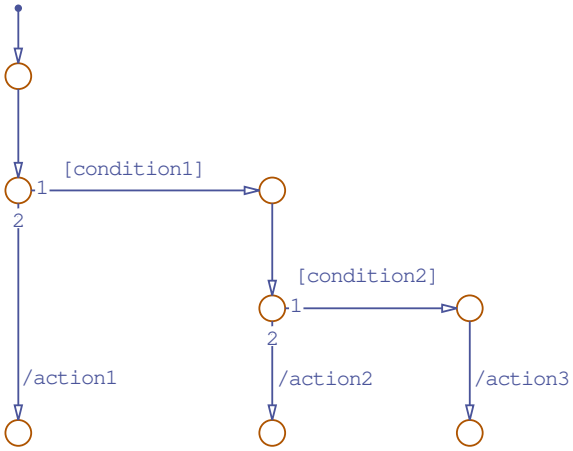
Stav (angl. state) popisuje režim udalosťami riadeného systému. Môže mať definované až tri akcie viď `State02` na obr. 2.2. Vstupnú akciu (angl. entry action): vykoná sa, keď sa daný stav označí za aktívny. Aktívnym stavom je stav, ktorého operácie sa v danom kroku vyhodnotia. Akcia "počas" (angl. during action), je spustená, ak bol daný stav aktívny v predchádzajúcom kroku a v aktuálnom kroku nie je splnená žiadna podmienka z množiny vychádzajúcich prechodov (VP) daného stavu. Posledná, výstupná akcia (angl. exit ac-

tion) je vykonaná pri odchode z aktívneho stavu. To znamená, že ak je počas vykonávania vnútornej logiky stavu splnená aspoň jedna podmienka z VP, výstupná akcia sa vykoná bezprostredne pred zmenou stavu. Každý stav môže ďalej obsahovať množinu vnorených stavov a ich prechodov, čím sa vytvárajú vnorené stavové diagramy na viacerých vrstvách.

Prechod (angl. transition) obvykle prepája dva stavy, môže slúžiť ako vstupný bod do stavového diagramu (angl. default transition), či vnoreného stavového diagramu viď `default internal transition` na obr. 2.2. Môže mať nastavenú podmienku (angl. guard), ktorej splnenie rozhoduje o tom, či sa zmení aktuálny aktívny stav. Ak nie je nastavená podmienka, zmena stavu nastane vždy pri vykonávaní vnútornej logiky aktívneho stavu, hovorí sa vtedy o nechránenom prechode (angl. unguarded transition). Ďalej môže obsahovať akciu prechodu (angl. transition action) Jej vykonanie je zaradené pred výstupnú akciu aktívneho stavu. To znamená, že musia byť splnené všetky podmienky prechodov medzi dvoma stavmi, aby sa akcia prechodu uskutočnila. Napr., cesta medzi stavmi `State01` a `State03` obsahuje prechody `transition1`, `transition2` a `transition4` viď obr. 2.2. Aby sa zrealizovala akcia prechodu `action1` nestačí splnenie podmienky `condition1`, ale je potrebné aj splnenie podmienky `condition2`. Inými slovami, ak nastane zmena stavu, tak sa vykonajú všetky akcie prechodov, ktoré boli na ceste zo stavu X do stavu Y. X a Y môžu byť jeden stav. Za zmienku stojí aj akcia podmienky prechodu (angl. condition action). Tie sa líšia od akcií prechodov v tom, že na ich spustenie stačí splnenie podmienky prechodu, ktorý prislúcha danej akcii, a nie všetkým prechodom na ceste medzi dvoma stavmi. Napr. ak by bola `action1` akcia podmienky prechodu a nie akcia prechodu, tak by na jej uskutočnenie stačilo splnenie podmienky `condition1`. Ďalej sa nimi nebude zaoberať, lebo ich výskyt nie je podporovaný interným nástrojom.

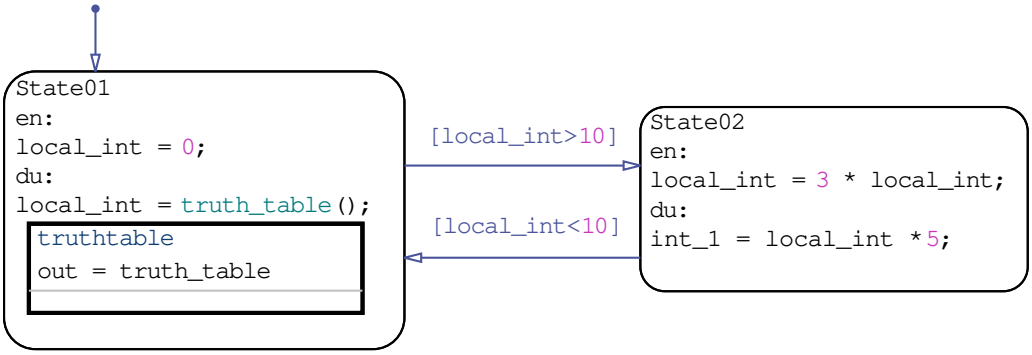
Uzol (angl. junction), slúži na prepájanie viacerých prechodov. Môže sa označovať ako stav, ktorý neobsahuje žiadnu vnútornú logiku. Prechody medzi uzlami sú vyhodnocované v jednom kroku. Napr. cesta medzi stavom `State01` a `State02` sa skladá z troch prechodov a dvoch uzlov. Prechody sú v smere zo stavu `State01` do stavu `State02`. Aby sa mohol prechod vykonať, musia byť splnené všetky podmienky prechodov na ceste medzi stavmi viď obr. 2.2. Pomocou uzlov sa tiež modelujú rozhodovacie diagramy (angl. state–less charts). Sú to také stavové diagramy ktoré obsahujú len uzly a prechody. Neuchovávajú informácie o vnútornom stave, nie sú vykonávané v čase a jediné akcie, ktoré obsahujú sú akcie prechodov viď obr. 2.3.

V prípade, že stav alebo uzol má aspoň dva odchádzajúce prechody, je každému prechodu pridaná vlastnosť poradie (angl. execution order) viď `transition3` a `transition4` na obr. 2.2, ktorá určuje v akom poradí budú podmienky prechodov vyhodnocované. Tým sa zaručí, že model stavového diagramu bude deterministický aj v prípadoch, že bude mať viacero odchádzajúcich trás s rovnakou podmienkou prechodu.



Obr. 2.3: Rozhodovací diagram

Pravdivostná tabuľka (angl. truth table) viď obr. 2.4, je prvok nezávislý od aktívneho stavu diagramu. Je modelovaná ako grafická funkcia. Jej hodnota alebo akcia je určená na základe definovaných vstupov funkcie a dát stavového diagramu, ktoré majú zastúpenie v jednotlivých podmienkach. Jej telo sa skladá, ako už bolo spomenuté z podmienok (angl. condition), ďalej z rozhodnutí (angl. decisions) a akcií (angl. actions) viď obr. 2.1. Každé rozhodnutie obsahuje ohodnotenia (angl. evaluations) pre každú definovanú podmienku. Ohodnotenie hovorí o tom, ako budú podmienky figurovať v rozhodnutí. Môže nadobúdať tri hodnoty. Buď podmienka v rozhodnutí vystupuje priamo – hodnota `(T)`, negovane – hodnota `(F)`, alebo nemá vplyv na dané rozhodnutie – hodnota `(-)`. Ak sú všetky ohodnotené podmienky daného rozhodnutia splnené, tak sa vykoná akcia, ktorá je pre dané rozhodnutie definovaná. Napr., je definovaná tabuľka, ktorá ma štyri rozhodnutia `Decision1` až `Decision4` viď tab. 2.1. Akcia `action2` sa vyhodnotí práve vtedy, keď nebolo vybrané žiadne predošlé rozhodnutie a neplatí prvá (`par1 > 10`) a druhá (`par2 < 200`) podmienka pretože obe majú ohodnotenie (`F`). Ak je definované rozhodnutie, v ktorom nefiguruje ani jedna z podmienok, viď `Decision4` v tab. 2.1, tak sa jedná o predvolené rozhodnutie (angl. default decision). Jeho akcia sa uskutoční vždy keď sa nevyberie žiadne z predošlých definovaných rozhodnutí.



Obr. 2.4: Stavový diagram s pravdivostnou tabuľkou

Condition	Decision1	Decision2	Decision3	Decision4	Action
$par1 > 10$	-	F	-	-	action1: $value = 1;$
$par2 < 200$	T	F	-	-	action2: $value = 2;$
$par1 != par2$	F	-	T	-	action3: $value = 3;$
Actions:	action1	action2	action3	action4	action4: $value = 4;$

Tabuľka 2.1: Ilustračná definícia pravdivostnej tabuľky

2.2 Dostupné optimalizácie

Optimalizácie môžu byť rozdelené do niekoľkých skupín. Prvé rozdelenie bude deliť optimalizácie podľa toho, či zmeny, ktoré sa ich prevedením vyskytnú v produkčnom kóde znemožňujú automatickú revíziu modelu. Ďalšie rozdelenie spočíva v tom, či prevedenie danej optimalizácie nastavuje užívateľ, alebo sú nevyhnutné pre výsledný produkt a každý vygenerovaný kód ich obsahuje. Skúmané boli tie optimalizačné parametre, ktoré sú súčasťou prekladových prostredí pre produkčné modely.

Parameter **I/O Storage Class** určuje spôsob, akým sú predávané vstupné a výstupné hodnoty modelu. Môže nadobúdať tri hodnoty. Pri nastavení `Auto` sú definované ako dátový typ štruktúra (`struct`) zvlášť pre vstupy a zvlášť pre výstupy viď obr. 2.5. Ich definícia je súčasťou hlavného hlavičkového súboru modelu. Pri nastavení hodnôt `ImportedExtern` viď obr. 2.7 alebo `ImportedExternPointer` viď obr. 2.6 sú vstupy a výstupy definované ako externé hodnoty alebo ukazovatele na dátový typ. Pre model na obr. 2.1 a postupne pre parametre optimalizácie budú vygenerované kódy vstupov a výstupov vyzerať nasledovne:

```
typedef struct {
    int32_T input_port_1;
    boolean_T input_port_2;
} state_flow_chart_ExternalInputs;

typedef struct {
    uint32_T output_port_1;
    boolean_T output_port_2;
} state_flow_chart_ExternalOutputs;
```

Obr. 2.5: Generovaný kód pre vstupy a výstupy s parametrom `Auto`

```
extern int32_T *input_port_1;
extern boolean_T *input_port_2;
extern uint32_T *output_port_1;
extern boolean_T *output_port_2;
```

Obr. 2.6: Generovaný kód pre vstupy a výstupy s parametrom `ImportedExternPointer`

```
extern int32_T input_port_1;
extern boolean_T input_port_2;
extern uint32_T output_port_1;
extern lstlisting output_port_2;
```

Obr. 2.7: Generovaný kód pre vstupy a výstupy s parametrom `ImportedExtern`

Parameter **Loop unrolling threshold** nadobúda hodnoty z množiny prirodzených čísel a určuje veľkosť rozmeru vektorového signálu, pri ktorej dochádza k zabaleniu priradení hodnôt do cyklu. Systémom predvolená hodnota rozmeru je tisíc, čo v praxi znamená, že je táto optimalizácia vypnutá.

```
signal_out[0] = signal_in[0];
signal_out[1] = signal_in[1];
signal_out[2] = signal_in[2];
signal_out[3] = signal_in[3];
signal_out[4] = signal_in[4];
```

Obr. 2.8: Priradenie hodnoty do vektorového signálu bez optimalizácie

```
for (int i = 0; i < 5; i++) {
    signal_out[i] = signal_in[i];
}
```

Obr. 2.9: Priradenie hodnoty do vektorového signálu s optimalizáciou

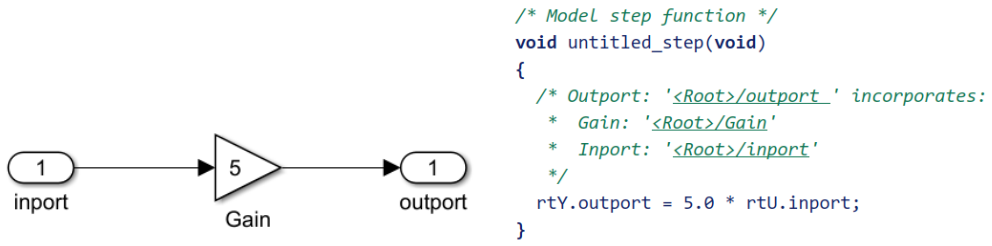
Optimalizácia **Logical to bitwise** pri zapnutí nahrádza vo výrazoch všetky logické operátory (`&&`, `||`, `!`) za ekvivalentné bitové operátory (`&`, `|`, `~`). V generovanom kóde sú reprezentované makrami definovanými ako (`AND`, `OR`, `NOT`).

Parameter **Zero Initialization**. Ak je táto optimalizácia zapnutá, vygeneruje sa kód, ktorý na inicializuje všetky signály v modeli na predvolenú hodnotu podľa dátového typu.

Parameter **Reuse buffers** ovláda používanie vyrovnávacej pamäti. Ak je povolená optimalizácia, tak vždy keď je to možné, je priradenie vnútorných dát blokov v modeli nahradené vstupnými a výstupnými signálmi modelu. Príklad, je daný model. Má jeden vstupný port s menom `in_port_01`, ktorý je pripojený k vstupu stavového diagramu s názvom `in_chart_01`. Analogicky má výstup z modelu `out_chart_01` a výstupný port `out_port_01`. Vnútri stavového diagramu je stav `state_01` so vstupnou akciou `out_chart_01 = in_chart_01;`. Pri zapnutej optimalizácii budú vnútorné dáta stavového diagramu v akcii nahradené vstupnými a výstupnými portami. V závislosti od nastavenia optimalizácie **I/O Storage class** môže priradenie vyzerať nasledovne: `out_port_01 = in_port_01;`.

Pri zapnutí optimalizácie **Inline parameters** sa nealokuje vnútorná pamäť pre parametre numerické blokov a vnútorný kód daného bloku je prenesený na vstup príslušnému pripojenému bloku. Nech je daný model, má vstupný port `inport` k nemu pripojený blok **Gain** (vstupný signál vynásobí zadaným parametrom a predá ho výstupu) s parametrom 4. Z neho ďalej vedie signál na výstupný port `outport`. Bez optimalizácie sa vygeneruje kód `gain_inner = inport * 5; outport = gain_inner;`. S optimalizáciou sa kód

zjednoduší na jediný riadok C kódu: `outport = 5.0 * inport;`, ako je ukázané na obrázku 2.10.

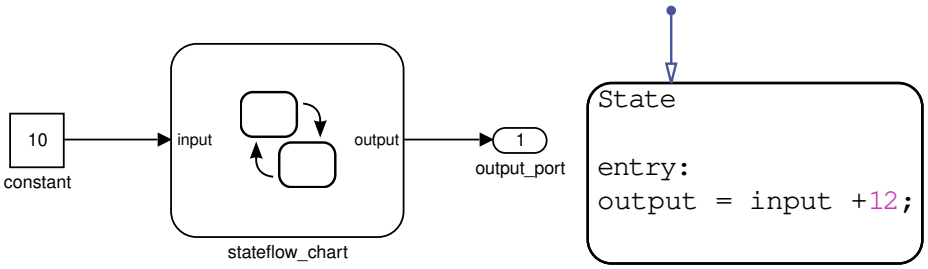


Obr. 2.10: Ukážka optimalizácie bloku Gain

2.3 Nesprávny návrh modelov vedúci optimalizáciám

MATLAB® a jeho moduly sa snažia, aby bol vygenerovaný kód minimálnou reprezentáciou modelu. Minimálna forma modelu by mala byť chápaná ako model, ktorý nemožno zjednodušiť pri zachovaní rovnakej funkčnosti ako pôvodne navrhnutý model. Táto sekcia sa venuje popisu niekoľkých častých chýb pri návrhu stavových diagramov, ktoré pri generovaní kódu vedú na optimalizáciu modelu na jeho minimálnu formu. Analýzou takto vzniknutých optimalizácií sa venuje táto aplikácia. Tieto optimalizácie patria do skupiny optimalizácií, ktoré sú nevyhnutné pre výsledný produkt a preto sú zahrnuté vo všetkých prekladových prostrediach.

Problém č. 1 táto optimalizácia sa môže vyskytnúť v prípadoch, ak je aspoň jeden zo vstupných signálov bloku stavového diagramu konštantný. To znamená, že je buď pripojený blok generujúci nemenlivý signál, alebo generátor kódu pri optimalizovaní vyhodnotil signál pripojený na vstup ako konštantný. Napr., je daný modelový návrh obsahujúci jeden blok stavového diagramu. Má jeden vstup, na ktorý je pripojený blok generujúci konštantný signál hodnoty 10 a jeden výstup na ktorý je pripojený výstupný port viď obr. 2.11. Stavový diagram ďalej obsahuje jeden stav, ktorý je pripojený k vstupnému predvolenému prechodu. Vo vstupnej akcii stavu je rovnica `output = input + C`, kde `C` môže byť literál alebo konštantou inicializovaná premenná. Nech je $C = 12$, potom rovnica definovaná v stave bude vyzerať: $output = input + 12$. Pretože optimalizácia **Inline parameters**, je schopná substituovať ľubovoľný literál namiesto premennej a vstup modelu je pripojený ku konštante, rovnica bude zmenená na $output = 10 + 12$. Ďalej bude rovnica zoptimalizovaná v rámci transformácie na minimálnu formu optimalizovaná na tvar $output = 22$. Síce rovnice $output = input + 12$ a $output = 22$ sa môžu javiť rovnaké, no nejde o funkčné ekvivalenty. Po prvé, chýba závislosť toku dát medzi symbolmi `output` a `input`. A po druhé, ekvivalencia pôvodnej a optimalizovanej rovnice je založená na predpoklade, že `input` je konštanta. To že sa ako konštanta vyskytuje v modeli, môže, ale nemusí znamenať, že sa bude ako konštanta vyskytovať aj vo vygenerovanom kóde.

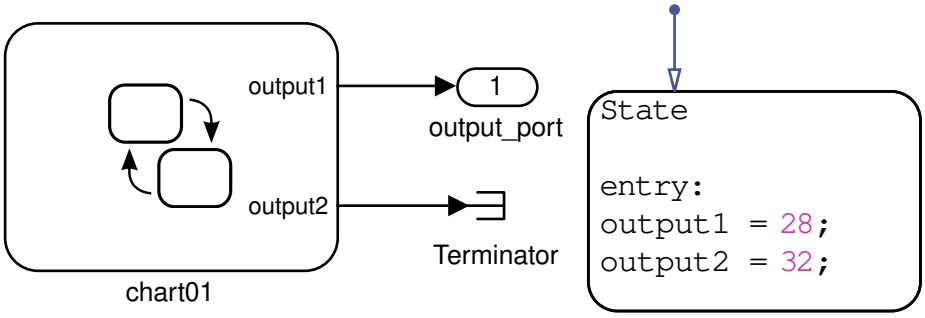



```
if (model_DWork.is_active_c3_model == 0U) {
    /* Entry: Chart */
    model_DWork.is_active_c3_model = 1U;

    /* Outport: '<Root>/Out1' */
    /* Entry Internal: Chart */
    /* Transition: '<S1>:4' */
    /* Entry 'State': '<S1>:3' */
    model_Y.output_port = 22;
  } else {
    /* During 'State': '<S1>:3' */
```

Obr. 2.11: Ukážka optimalizácie stavového diagramu s konštantným vstupom

Problém č. 2 pripojenie ukončovacieho bloku (angl. Terminator) na jeden z výstupov bloku stavového diagramu viď obr. 2.12. Nech je daný stavový diagram `chart01`. Má dva výstupy (`output1` a `output2`). Výstup `output1` je pripojený k výstupnému portu modelu a výstup `output2` k *Terminator* bloku. Problém nastáva keď sa Matlab pri generovaní kódu snaží odstrániť všetky bloky, ktoré nejakým spôsobom narúšajú minimálnu formu modelu. To znamená, že všetky závislosti toku dát na výstupe `output2` zaniknú. Napr., ak by stavový diagram `chart01` obsahoval stav `state` s priradením $output2 = 15$; tak by vo vygenerovanom kóde tento príkaz chýbal. Tým pádom nebude model plne implementovaný kódom a to naruší trasovateľnosť.

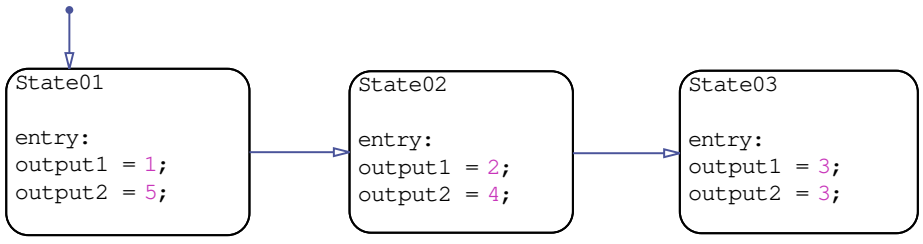



```
...
  /* Outport: '<Root>/output_port ' */
  /* Entry Internal: chart01 */
  /* Transition: '<S1>:4' */
  /* Entry 'State': '<S1>:3' */
  model_Y.output_port = 28;
} else {
...
```

Obr. 2.12: Ukážka optimalizácie výstupu pripojeného na ukončovací blok

Problém č. 3 nadmerná špecifikácia logických operácií. Môže vznikať, keď kód stavového diagramu obsahuje booleovské výrazy, ktoré možno redukovať na svoju podmnožinu pri zachovaní rovnakej logickej hodnoty. Tento problém sa vzťahuje na podmienky prechodov v stavových diagramoch a hlavne na definície pravdivostných tabuliek.

Problém č. 4 prepojenie stavov pomocou nechránených prechodov resp. pomocou prechodov bez špecifikovanej podmienky prechodu viď obr. 2.13. Takéto stavy nemôžu mať definovanú akciu počas, lebo by počas optimalizovania generovaného kódu boli odstránené. To by malo rovnaký dopad na trasovateľnosť ako v **Problém č. 2**. Ďalší problém tohto návrhu je absencia správnej štruktúry stavov. Tým, že sú všetky stavy priechodné v kombinácii s nechránenými prechodmi, sa zo stavového diagramu efektívne stane rozhodovací diagram. S jediným rozdielom, že jeho logika nie je vykonaná v jednom kroku, ale postupne jeden stav za druhým.



Obr. 2.13: Stavy zreťazené prechodmi bez podmienok

Posledný **problém č. 5** vzniká, keď aspoň jedna definovaná pravdivostná tabuľka vo vnútri stavového diagramu má viacero bezprostredne za sebou idúcich rozhodnutí, ktoré vedú na rovnakú akciu, alebo na rozdielne akcie, ktoré obsahujú rovnaké príkazy. Takto špecifikované rozhodnutia vedú k redukcii na minimálnu formu. To znova spôsobí, že vygenerovaný nebude plne implementovať model.

Condition	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**
`bool_1`	T	-	T	F	-	F
`bool_2`	T	T	-	F	F	-
`bool_3`	-	T	T	-	F	F
Actions:	act1	act1	act1	act2	act2	act2

Action
act1: `out = 1;`
act2: `out = 0;`

Tabuľka 2.2: Definícia pravdivostnej tabuľky, ktorá má viacero rozhodnutí vedúcich na rovnakú akciu

Kapitola 3

Štruktúra vygenerovaného kódu

Kapitola detailne popisuje štruktúru vygenerovaného kódu bez optimalizácií pre všeobecný stavový diagram, rozhodujúce diagramy a pravdivostnú tabuľku. Kód vygenerovaný z modelu má pevne danú štruktúru, vždy obsahuje modelovú krokovú funkciu, ktorá implementuje logiku modelu. Samostatnú funkciu pre každú definovanú pravdivostnú tabuľku a tiež môže obsahovať inicializačnú a terminačnú funkciu, ktoré spravujú pamäť modelu pri začiatku a ukončení práce. Štruktúra kódu stavového diagramu a pravdivostných tabuliek bola zostavená na základe dokumentácie Embedded coder [2], dokumentácie Simulink® [7] a na základe vykonaných experimentov na vlastnej sade testovacích modelov.

3.1 Generovanie kódu stavového diagramu

Implementácia kódu Stavového diagramu sa vždy nachádza vo vnútri krokovej funkcie modelu. Vždy začína vetvením pomocou `if` bloku `if (!chart_active) {activate_chart(); enter_first_state();} else { execute_state_logic();}` . Tým je model rozdelený na dve časti.

Prvá vetva obsahuje aktiváciu diagramu. To znamená, že zabráni ďalšiemu vykonaniu tejto vetvy počas behu modelu a vykoná vstupný prechod. Vstupný prechod sa od ostatných prechodov líši tým, že nemá zdrojový stav, a teda nemusí vykonávať logiku de-aktivácie a ukončovacie akcie. Vygeneruje sa kód pre definovanú akciu prechodu a za ňou nasleduje kód nastavenia aktívneho stavu, napr. (`chart_active_state = first_state;`). Potom sa vygeneruje vstupná akcia pre daný stav a následne sa tento proces opakuje pre každú úroveň zanorenia stavového diagramu v príslušnom vstupnom stave. Tiež bolo zo zadania dané, že vstupný prechod musí byť práve jeden a musí byť vždy uskutočniteľný. Z toho vyplýva že musí existovať cesta, ktorá nebude podmienená ani na jednom prechode.

V druhej vetve sa generuje kód pre vykonávanie stavovej logiky. Na základe počtu stavov sa vygeneruje príslušný blok pre výber aktívneho stavu. Pre jeden až dva stavy sa vygeneruje blok `if - else`, pre viac stavov blok `switch - case - default`. Pre každý stav sa vygeneruje fáza "počas" (angl. during phase). Táto fáza sa skladá z vygenerovania podmienok pre odchádzajúce všetky cesty z daného stavu, akcie "počas". A ak daný stav obsahuje vnorený stavový diagram, tak kódu vnútornej stavovej logiky. `if (transition_condition_01) { take_path_01(); ...} else {during_action(); internal_state_logic();}`. Každá cesta medzi dvoma stavmi môže obsahovať prechody a uzly. Čo z nej efektívne vytvára rozhodovací diagram, ktorý namiesto finálnej akcie vykonáva prechod medzi zdrojovým a cieľovým stavom. To znamená, vygenerovaný kód cesty bude mať rovnakú štruktúru ako

príslušný rozhodovací diagram. Zadanie zakazuje modelovanie rozhodovacích diagramov, ktoré využívajú iný návrhový vzor ako je `if - else` vetvenie. Preto vytváranie cyklov, či typy vetvení, ktoré sa generujú ako blok `switch` môže byť zanedbané. Kód samotného prechodu, je veľmi podobný štruktúre vstupného prechodu. Prechod medzi stavom *source* a *destination* môže byť popísaný nasledovne viď obr. 3.1.

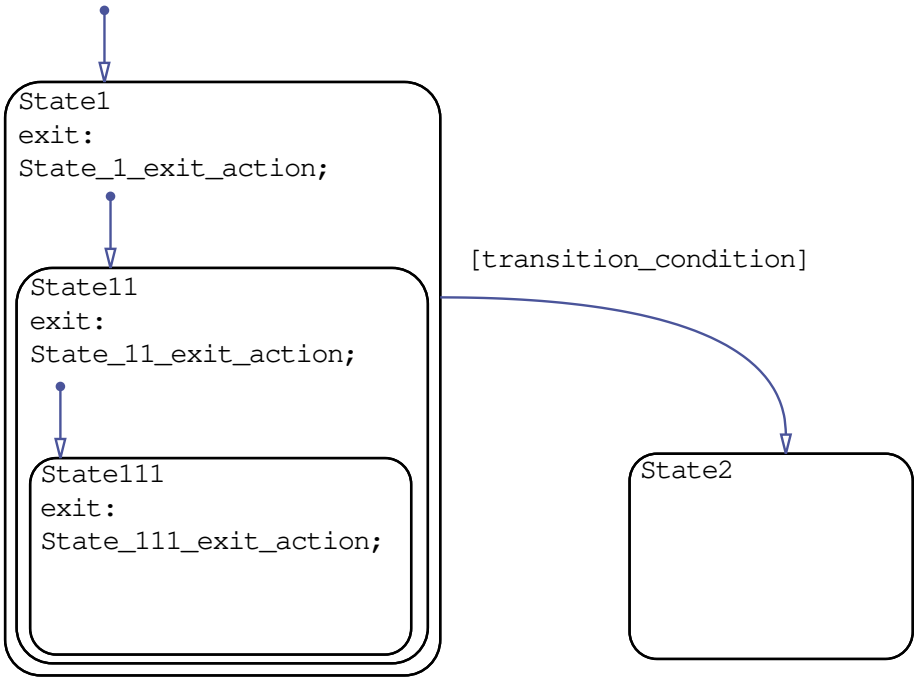
```
/* vo vygenerovanom kóde sú nižšie uvedené
funkcie nahradené za príslušné časti modelu */
exit_action(source);
internal_exits(source);
transition_action();
chart_active_state = destination;
entry_action(destination);
internal_entries(destination);
```

Obr. 3.1: Pseudo kód popisujúci časti vygenerované pre prechod medzi dvoma stavmi

Ak je definovaná, prvá sa vygeneruje výstupná akcia stavu, ktorý má byť v danom kroku de-aktivovaný. Pre každý stav, ktorý obsahuje vnorený stavový diagram, je vygenerovaný blok (`internal_exits(state)`). Tento blok obsahuje vetvenie, ktoré na základe aktívnych vnorených stavov vykoná ich akcie odchodu. Okrem toho sú vnorené diagramy na všetkých úrovniach de-aktivované. Napr., je daný stavový diagram viď obr. 3.3. Vygenerovaný odchod pre stav `State1` má nasledovnú formu. Ako prvá sa vygeneruje kód akcie odchodu `State_1_exit_action`, ďalej sa vygeneruje kontrola aktívneho vnoreného stavu `State11`, ktorý v prípade, že bude aktívny, bude obsahovať de-aktiváciu jeho vnoreného stavového diagramu obsahujúci stav `State111` viď útržok kódu na obr. 3.2.

```
...
State_1_exit_action;
/* State1 internal exit */
if (State1_active_state == State11) {
    State_11_exit_action;
    State1_active_state = INACTIVE;
    /* State11 internal exit */
    if (State11_active_state == State111) {
        State_111_exit_action;
        State11_active_state = INACTIVE;
    } else {
        State11_active_state = INACTIVE;
    }
} else {
    State1_active_state = INACTIVE;
}
...
```

Obr. 3.2: Stavový diagram s vnorenými stavmi obsahujúce ukončovacie akcie



Obr. 3.3: Stavový diagram s vnorenými stavmi obsahujúce ukončovacie akcie

Pre stav , ktorý obsahuje N úrovní zanorení stavových diagramov, je príslušne vygenerovaná vnútorná de-aktivačná logika pre každú nižšiu vrstvu voči aktuálnemu stavu. Aktuálnym stavom sa myslí stav, ktorému sa v danom čase generuje kód pre trasu medzi dvoma stavmi, kde tento stav figuruje ako počiatočný stav prechodu.

Nasleduje generovanie kódu pre akciu prechodu (angl. transition action) resp. akciu trasy. Trasa zo stavu *source* do stavu *destination* je usporiadaná množina prechodov, prepojených pomocou uzlov s počiatkom v stave *source* a končiac v stave *destination*. Z toho vyplýva, že akcia trasy bude obsahovať všetky akcie prechodov z jej množiny prechodov. Potom sa vygeneruje aktivácia cieľového stavu *destination*, vstupná akcia a logika vnorených vstupov rovnakým spôsobom ako bolo už vyššie popísané pre vstupný prechod.

Týmto je ukončená fáza generovania kódu pre odchádzajúce trasy zo stavu. Za ňou nasleduje posledná časť kódu stavového diagramu, ktorou je logika daného stavu. Jej kód sa vykonáva práve vtedy, keď je daný stav aktívny a nie je splnená žiadna podmienka na odchod. Obsahuje akciu "počas" (angl. during action), a stavovú logiku vnoreného stavového diagramu. Vnútorná stavová logika a stavová logika diagramu podľa rovnakých pravidiel.

3.2 Generovanie kódu pravdivostnej tabuľky

Štruktúra vygenerovaného kódu pravdivostnej tabuľky vzhľadom k modelovému popisu je oveľa priamočiarejšia ako stavový diagram. Kód všeobecnej tabulky sa dá jednoducho popísať. Každej pravdivostnej tabuľke (angl. truth table) TT v modeli odpovedá práve jedna funkcia vo vygenerovanom kóde, ktorá ju implementuje. Má x vstupných parametrov a návratovú hodnotu s dátovým typom (angl. data type) podľa výberu užívateľa (tiež môže mať definovaný návratový dátový typ `void`. Tým pádom jej výsledkom nebude hodnota, ale akcia ktorá sa vykoná). Nech TT, ktorá má množinu podmienok (angl. conditions) $C = \{condition_0, ..., condition_n\}$, rozhodnutí (angl. decisions) $D = \{decision_0, ..., decision_m\}$,

akcií $A = \{action_0, ..., action_m\}$ a na základe D a C je daná množina ohodnotení podmienok $E = \{evaluation_{0,0}, ..., evaluation_{m,n}\}$. Ohodnotenie podmienky prebieha spôsobom ako je popísane v sekcii 2.1. Potom pre TT bude vygenerovaný kód vyzerať nasledovne viď obr. 3.4:

```
/* funkcia pre pravdivostnu tabulku (TT)*/
static data_type truth_table(parameter_1, ..., parameter_x) {
    /* ak je definovaný návratový typ funkcie rozdielny od typu 'void' */
    data_type return_value;

    /* declarácia lokálnych premenných pre definované podmienky */
    boolean_T local_condition_var_0;
    ...
    boolean_T local_condition_var_n;

    /* nastavenie hodnot definovaných podmienok */
    local_condition_var_0 = condition_0;
    ...
    local_condition_var_n = condition_n;

    /* prvé rozhodnutie s reťazenými ohodnotenými podmienkami */
    if (evaluation_0_0 && ... && evaluation_0_n) {
        action_0;
    }

    ...

    /* posledné rozhodnutie s reťazenými ohodnotenými podmienkami */
    else if (evaluation_m_0 && ... && evaluation_m_n) {
        action_m;
    }
    /* ak je definovaný návratový typ funkcie rozdielny od typu 'void' */
    return return_value;
}
```

Obr. 3.4: všeobecný zdrojový kód pravdivostnej tabuľky

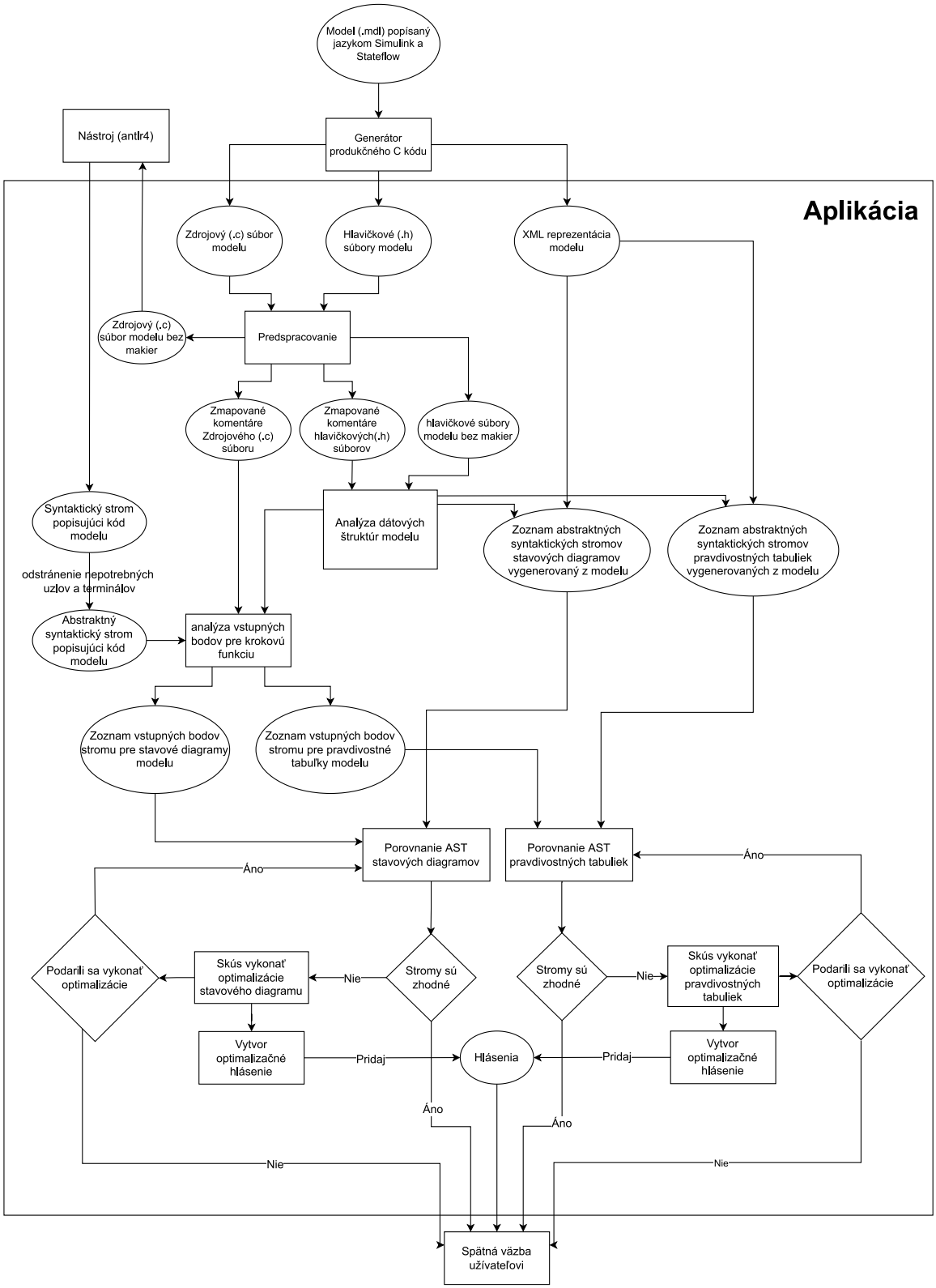
Kapitola 4

Architektúra aplikácie

Aplikácia bude na vstupe prijímať súbory, ktoré vzniknú generovaním produkčného kódu z modelu. Zdrojové (.c) a hlavičkové (.h) súbory modelu sa predspracujú. Na predspracovaných hlavičkových (.h) súboroch sa vykoná analýza vstupných, výstupných a vnútorných dát modelu. Zo zdrojového súboru (.c) sa pomocou externých nástrojov vytvorí syntaktický strom. Následným priechodom tohto stromu sa odstránením jeho nepotrebných uzlov a terminálov vytvorí abstrakčný syntaktický strom (ďalej len AST, angl. Abstract Syntax Tree).

Spracovanie pokračuje lokalizačnou fázou, kde sa v AST za pomoci sledovacích komentárov (angl. trace comments), ktoré sú zmapované počas fáze pred-spracovania a za pomoci získaných dát z modelu vyhľadajú vstupné body (uzly v AST) pre všetky definované stavové diagramy a v nich definované pravdivostné tabuľky.

Z XML reprezentácie modelu je pre každý zadefinovaný komponent (stavové diagramy a pravdivostné tabuľky) vytvorený AST (ďalej len referenčný strom). Ďalej sú referenčné stromy porovnávané s príslušnými AST získanými zo zdrojového kódu (ďalej len strom zdrojového kódu). Ak sú v porovnávaných stromoch nájdené rozdiely, tak je spustená optimalizačná fáza. Tá sa snaží o simuláciu optimalizácií na referenčnom strome na miestach, kde boli identifikované rozdiely so stromom zdrojového kódu. Ak sa optimalizácia podarí vykonať, tak je podľa nej zmenená štruktúra referenčného stromu. Vytvorí sa hlásenie o vykonaných optimalizáciách a znova sa vykoná porovnanie oboch stromov. Algoritmus sa snaží vykonávať optimalizácie do momentu, kým nie sú oba stromy zhodné. Ak aplikácia pri optimalizácii narazí na rozdiely medzi stromami, ktoré nevie vyriešiť, tak sa ukončí. Následne zozbierané hlásenia o optimalizáciách a rozdieloch stromov sú poskytnuté užívateľovi na štandardný výstup. Aplikácia ďalej umožňuje za pomoci externého nástroja vizualizáciu optimalizovaného AST, neoptimalizovaného AST a AST z vygenerovaného kódu. Architektúra aplikácie je znázornená na diagrame viď obr. 4.1. Elipsy predstavujú vstupné, výstupné a vnútorné dáta aplikácie. Obdĺžniky v diagrame predstavujú moduly, ktoré transformujú dáta a kosoštvorce naznačujú vetvenie. Všetky časti vo vnútri bloku **Aplikácia** sú autorské dielo.



Obr. 4.1: Diagram riadenia toku dát

Kapitola 5

Implementácia analýzy modelu, kódu a optimalizácií

Implementácia aplikácie je rozdelená do niekoľkých nezávislých častí, ktoré sú popísané ďalej v tejto kapitole.

5.1 Spracovanie vstupných súborov

Užívateľ zadá cestu k priečinku, ktorý by mal obsahovať model. Aplikácia prehľadá daný priečinok. Ak nájde práve jeden súbor s príponou `mdl`, zistí z neho názov modelu. Ak daný súbor nebol nájdený, alebo naopak ich bolo nájdených viacero, tak aplikácia skončí chybou.

V prípade úspechu opakovane prechádza vstupný priečinok. Tentokrát hľadá zdrojové súbory modelu v jazyku `C` a `XML` dokmuent reprezentujúci model. Prehľadávané súbory majú názvy v tvare: `model_name.c`, `model_name.h`, `model_name_private.h`, `model_name_data.h` a `model_name.xml`, kde `model_name` je názov modelu. Ak nie je nájdený aspoň jeden z troch hlavných súborov (`model_name.c`, `model_name.h` alebo `model_name.xml`), aplikácia nemôže ďalej pokračovať a skončí chybou. Výskyt hlavičkových súborov `model_name_private.h` a `model_name_data.h` nie je nevyhnutný, pri niektorých nastaveniach prekladových prostredí nemusia byť ani vygenerované. Získaný názov modelu a cesty všetkých nájdených súborov sú zabalené do objektu triedy `ModelInfo`.

5.2 Analýza vygenerovaného C kódu

Aby bolo možné spraviť rozbor zdrojových súborov popísaných jazykom `C`, je potrebné ich pred-spracovanie (angl. pre-processing). Túto úlohu má na starosti modul `preprocessor`. Pracuje tak, že na vstupe prijíma cestu k súboru, ktorý sa má pred-spracovať. Súbor s danou cestou otvorí, prečíta jeho obsah a rozdelí ho podľa podľa znakov konca riadku (`\r\n`). Takto rozdelený obsah súboru je uložený do vyrovnávacej pamäte (angl. buffer). Samotný algoritmus pred-spracovania funguje tak, že sekvenčne prechádza riadky uložené v pamäti, a ak daný riadok obsahuje práve jedno z direktív jazyka `C`, tak je nasledovným spôsobom transformované. Direktíva `if`, `elif`, `error` `pragma` a `line` nie sú spracovávané, lebo sa vo vygenerovaných súboroch nevyskytujú. `include` sa tiež ignoruje, lebo rozbor potrebných zahrnutých hlavičkových súborov je vykonaný nezávisle a zlúčenie zdrojových kódov potom nemá význam. A druhý dôvod je ten, že vygenerované kódy sú syntakticky, sémanticky správne a teda definície symbolov v hlavičkových súboroch nie sú potrebné.

Príkaz `define` slúži na definovanie makier. `preprocessor` makrá definuje, ale ďalej sa s nimi nepracuje, lebo pri analýze kódu majú názvy makier väčšiu výpovednú hodnotu ako hodnoty, ktoré definujú. Napr. názvy stavov stavových diagramov sú definované ako makrá, kde ich názov je v tvare `(názov modelu)_IN_(názov stavu)` a hodnota je poradie stavu v rámci jednej vrstvy. V prípade definovania viacerých stavových diagramov v rámci jedného modelu, alebo využívaní zanorenia stavových diagramov nastáva problém nejednoznačnosti. Viacerým stavom môže byť v kóde priradené rovnaké poradové číslo v rámci skupiny. Preto je jednoduchšie makrá neexpandovať.

Tým pádom sa definované makrá využívajú pre riadenie toku v príkazoch `ifdef` a `ifndef`. Direktíva (`else`, `endif`) sú spracovávané štandardným spôsobom. `preprocessor` okrem spracovania makier slúži na mapovanie komentárov zdrojového kódu. Všetky komentáre sú uložené v slovníku, kde kľúč je číslo riadku na ktorom sa vyskytujú a hodnota je text komentáru. Spracované súbory sú uložené v zložke s dočasnými súbormi.

Takto pripravené zdrojové súbory sú predané voľné dostupnému nástroju *(Another tool for language recognition 4)* (ďalej len `antlr4`) [10], aby vykonal rozbor. Nástroj funguje tak, že na základe poskytnutej gramatiky, ktorá je popísaná jazykom nástroja (jazyk `antlr4`), vygeneruje zdrojový kód pre zvolený cieľový jazyk. Vygenerovaný kód je potom schopný rozboru vstupného jazyka. Nástroj podporuje jazyk Python ako cieľový jazyk, tým pádom je možná priama integrácia nástroja do aplikácie. Ako vstupná gramatika to nástroja bola využitá voľne dostupná verzia gramatiky jazyka `C`, ktorá je založená na špecifikácii C11 [11]. Výstupom nástroja sú vygenerované zdrojové súbory jazyka Python. Menovite `CLexer`, `CParser` a `CVisitor`. `CLexer` slúži na vykonanie lexikálnej analýzy a vytvorenia prúdu lexémov. `CParser` prijíma na vstupe prúd lexémov vykoná syntaktickú analýzu a vytvorí syntaktický strom. `CParser` nevyužíva precedenčnú syntaktickú analýzu (PSA) pre spracovanie výrazov. To má za dôsledok, že vytvorený syntaktický strom obsahuje veľa zbytočných uzlov, ktoré mali význam len pri dodržaní správnej priority operácií. Syntaktický strom tiež obsahuje terminály (`(`, `)`, `;`, ...), ktoré nie sú potrebné pre analýzu optimalizácií. Preto aplikácia transformuje syntaktický strom vytvorený pomocou `antlr4` na AST, ktorý obsahuje len potrebné uzly.

Robiť preklad abstraktného stromu plnohodnotného jazyka C na AST je náročná výzva. Preto bola spravená analýza vygenerovaných kódov na testovacej sade modelov poskytnutej firmou Honeywell. Analýza bola vykonaná pomocou modulu `analyze_source_codes` a skúmala, ktoré typy uzlov syntaktického stromu jazyka `C` sú využívané syntaktickými stromami vygenerovaného kódu zo Simulink® modelov (ďalej len syntaktické stromy modelov). Využívaná gramatika má 103 rozličných typov uzlov syntaktického stromu. Analýza zistila, že 40 z nich sa v syntaktických stromoch modelov vôbec nevyskytuje. Celkové hlásenie z analýzy sa nachádza v prílohe B.

Na základe výsledku tejto analýzy boli vybrané typy uzlov syntaktického stromu, ktoré sa budú prekladať na AST) Preklad syntaktického stromu vykonáva modul `CVisitor`, jeho štruktúra bola vygenerovaná pomocou nástroja `antlr4`. Modul prechádza syntaktický strom a na základe typov uzlov poskladá AST.

Abstraktný syntaktický strom je popísaný modulom `AST`. Využíva objektovo-orientovaný prístup, kde listové uzly (trieda `Token`) využívajú implementáciu nástroja `antlr4`. Každý uzol, ktorý nie je listom obsahuje ukazovateľ na priameho predka. Toto previazanie je neskôr využívané pre jednoduchšie vykonávanie zmien v AST.

5.3 Analýza modelovej reprezentácie

Dátová štruktúra typu XML reprezentujúca model (ďalej len `XML` dokument) je spracovaná pomocou knižnice jazyka Python, `ElementTree`. Spracovávaný model môže obsahovať iné bloky z knižnice `Block library` ako tie, ktoré sú predmetom analýzy. Analýza `XML` dokumentu prebieha tak, že sa lokalizujú sa všetky uzly, ktoré obsahujú informácie o definovaných stavových diagramoch a uzly ktoré popisujú prepojenia diagramov s ostatnými blokmi v modeli. Na nájdenie uzlov sa využíva jazyk `XPath`, ktorý slúži pre adresovanie častí `XML` dokumentov pomocou jednoduchej syntaxe [8]. Stavový diagram je potomkom každého uzlu typu `SimulinkSubSystem`, ktorý má atribút `sfblocktype` s hodnotou `Chart` (ďalej len subsystém diagramu). Daná adresa v jazyku `XPath` vyzerá nasledovne: `.//SimulinkSubSystem[@sfblocktype='Chart']`. Samotný uzol stavového diagramu sa nachádza na adrese `.//Model/Syntax/StateflowState` relatívne voči subsystému diagramu a využíva rovnaký typ uzlu ako jeho stavy. Štruktúra stavových diagramov v `XML` dokumente sa zhoduje s tou, ktorá je popísaná v podkapitole 2.1. Stavový diagram je špeciálny typ stavu (uzol `StateflowState`), ktorý obsahuje dáta modelu (`StateflowData`), stavy (`StateflowState` s atribútom `type="OR_STATE"`), prechody (`StateflowTransition`), uzly (angl. junctions) (`StateflowState` s atribútom `type="CONNECTION_JUNCTION"`) a pravdivostné tabuľky nachádzajúce sa na relatívnej adrese (`Model/Syntax/TruthTable`) voči rodičovskému stavu.

Dáta načítané z `XML` dokumentu sú zabalené do objektov nasledovným spôsobom: Z uzlu typu `StateflowState`, ktorého priamy predok má typ `Syntax`, je vytvorený objekt triedy `StateflowChart`, ktorý reprezentuje stavový diagram. Má svoje meno, list stavov (trieda `StateflowState`) a list dát modelu (trieda `StateflowData`) a trasu k počiatočnému stavu (trieda `Path`). Z listu prechodov a uzlov (angl. junction) sú pre každý stav vytvorené listy ich odchádzajúcich trás (list objektov triedy `Path`). Objekty triedy `StateflowData`, obsahuje oproti uzlu (angl. node) v XML dokumente naviac názov vstupného alebo výstupného portu, ktorý je k nemu v modeli pripojený. Trieda `StateflowState` má rovnaké atribúty ako trieda `StateflowChart` s rozdielom, že neobsahuje list dát modelu. Triedy popisujúce pravdivostné tabuľky neponúkajú žiadnu pridanú hodnotu oproti ich štruktúre v `XML` dokumente.

Trieda `Path`, predstavuje stromovú štruktúru, ktorej uzly obsahujú prechody (objekty triedy `StateflowTransition`). List prechodov (angl. transitions), ktorý vznikne priechodom od koreňového uzla k ľubovoľnému listovému uzlu predstavuje trasu medzi dvoma stavmi.

5.4 Vytvorenie blokového úložiska

Táto podkapitola popisuje analýzu hlavičkových súborov modelu, dát modelu a vytvorenie objektovej reprezentácie blokového úložiska modelu. Pod názvom blokové úložisko sa rozumie štruktúra vstupných, výstupných dát modelu. Ich formát v kóde závisí od optimalizačného parametra **I/O storage class** viď podkapitolu 2.2. Implementácia aplikácie zatiaľ podporuje analýzu modelov, ktoré majú hodnotu parametru tejto optimalizácie nastavenú na *Auto*.

Na začiatku sa vykoná pred-spracovanie hlavičkového súboru, zmapujú sa komentáre. Následne sa vykoná rozbor kódu a zostaví sa z neho AST. Hlavičkový súbor obsahuje vždy aspoň tri definície štruktúr jazyka `C`. Prvá obsahuje vnútorné dáta modelu. Pre stavové diagramy sú vygenerované premenné, ktoré určujú aktívnosť diagramov a ich jednotlivých stavov. Taktiež obsahujú deklarácie premenných pre lokálne dáta stavových diagramov.

V prípade, že model obsahuje aspoň dva stavové diagramy, alebo aspoň jeden stav má vnorený stavový diagram, je potrebné rozlíšiť, ktoré premenné zodpovedajú ktorým diagramom resp. stavom. Tento problém pomáhajú riešiť trasovacie komentáre (angl. trace comments) v zdrojovom kóde. Komentáre, ktoré sa nachádzajú na rovnakom riadku ako deklarácie premenných popisujú blok modelu, kvôli ktorému bola daná premenná vygenerovaná. Majú tvar `<system>/block_name`. `system` je názov systému, ktorý je popísaný v modelovej hierarchii. Hierarchia systémov modelu je vždy vygenerovaná ako komentár na konci hlavného hlavičkového súboru modelu. Popisuje všetky systémy, ktoré sa nachádzajú v modeli a má tvar (`<symbolický názov> : názov systému v modeli`). Napr., pre koreňový systém modelu s názvom `example_model` bude vygenerovaný nasledujúci riadok: `'<Root>' : 'example_model'`.

Určenie pôvodu premennej prebieha v týchto krokoch. Nájde sa komentár zo slovníka komentárov podľa riadku kódu na ktorom sa vyskytuje symbol deklarácie. Pomocou modelovej hierarchie a príslušného komentára je možné jednoznačné určiť stavový diagram resp. stav, s ktorým daná premenná súvisí.

Analogicky sú spracované štruktúry popisujúce vstupné a výstupné porty. Ak model obsahuje konštantné alebo pred-vypočítané dáta, tak sa v hlavičkovom súbore nachádza ešte jedna štruktúra, ktorá ich popisuje. Všetky analyzované štruktúry sú spolu s názvom identifikátoru štruktúry zabalené do triedy `Storage` pre vstupné, výstupné, konštantné dáta a do triedy `ChartStorage` pre vnútorné dáta stavových diagramov resp. stavov.

Ďalej sú objektom všetkých stavových diagramov (`StateflowChart`) a pravdivostných tabuliek (`TruthTable`) priradené vstupné uzly v abstraktnom syntaktickom strome kódu. Znova sú využité vodiace komentáre. Kód implementácie každého stavového diagramu sa vo vygenerovanom kóde z modelu nachádza medzi komentármi, ktoré majú tvar:

`"Chart: '<system>/block_name'"` a `"End of Chart: '<system>/block_name'"`. Typ adresovania je zhodný s komentármi vyskytujúcimi sa v blokovom úložisku. Prvý riadok tela definície funkcie pre pravdivostnú tabuľku obsahuje komentár v tvare:

`Truth Table:truth_table_name`, kde `truth_table_name` je názov bloku pravdivostnej tabuľky v modeli.

5.5 Simulácia optimalizácií na AST vygenerovanom z modelu

V tomto bode sú spracované všetky vstupné dáta a aplikácia môže začať s generovaním AST z modelu. Zostavenie AST má na starosti modul `assembly`. Generovanie AST zo stavového diagramu (ďalej len strom diagramu) a pravdivostných tabuliek (ďalej len strom tabuliek) sa riadi štruktúrou generovaného kódu popísanej v kapitole 3. Strom diagramu je potom porovnávaný s príslušným uzlom z AST z vygenerovaného kódu (ďalej len strom kódu). Porovnávanie stromov má na starosti modul `match_ast`.

Funkcia, ktorá vykonáva porovnávanie stromov berie na vstupe dva uzly abstraktných syntaktických stromov. Prvý je referenčný (strom diagramu či tabuľky), podľa neho sa riadi algoritmus porovnávania a druhý je strom kódu. Algoritmus rekurzívne prechádza strom referenčného uzlu. Ak sa prehľadá celý strom, znamená to, že stromy sú zhodné. V opačnom prípade našiel nezhodu medzi stromami. Funkcia porovnávania stromov vracia objekt triedy `MatchReport`, ktorý buď obsahuje hlásenie o tom, že sú stromy zhodné, alebo obsahuje typ nezhody, hĺbku stromu, kde k nezhode došlo, nezhodujúce sa uzly a cestu od

koreňa referenčného stromu k uzlu, kde nastala nezhoda. Toto hlásenie je predané funkcii `missmatch_recovery`, ktorá sa snaží o vykonávanie optimalizácií na strome diagramu.

Typy optimalizácií, ktoré algoritmus vykonáva sú vyberané podľa typu nezhody, ktorá nastala pri porovnaní. Aplikácia podporuje šesť typov nezhôd:

- `LIT_MISS` – symboly dvoch literál-ov sú rozdielne. V tomto prípade nezhody nejde o optimalizovanie, ale vykonávanie zmien v strome diagramu či tabuľky, aby mohla nastať presná zhoda. Dokáže nahradiť numerické literály `0` a `1` príslušne za `false` a `true`, alebo pridať príponu numerickému literálu indikujúci dátový typ, napr. (`0.1 -> 0.1F`). Tieto zmeny je potrebné vykonávať z dôvodu, že aplikácia nevykonáva kontrolu na dátové typy vo výrazoch jazyka akcií (angl. action language) v modeli.

- `VAR_MISS` – premenné v stromoch nie sú zhodné. Táto nezhoda býva spôsobená optimalizáciou, kde premenné reprezentujúce dáta modelu, sú priamo nahradené premennými vstupných a výstupných portov. Alebo naopak bola lokálna premenná naviac v modeli. Optimalizácia je prevedená tak, že je príslušnej premennej nájdená alternatíva v blokovom úložisku modelu.

- `FUNC_CALL_SYMBOL` – symboly názvov funkcie pre uzly volania funkcie si nezodpovedajú. Simulink generuje systémové názvy funkciám pravdivostných tabuliek. Táto nezhoda je vyriešená pomocou namapovaných komentárov a vygenerovaných stromov tabuliek.

- `VAR_CONST_MISS` – uzol referenčného stromu očakával premennú alebo výraz a v kóde sa vyskytla konštanta. Algoritmus vyhľadá alternatívy za premenné nachádzajúce sa vo výraze a pokúsi sa o čiastočné vyhodnotenie výrazu.

- `BODY_LEN_MISS` – dĺžky tiel uzlov sú rozdielne. Pod telom môže byť chápaný napr. list príkazov vo vnútri definície funkcie, alebo vo vnútri cyklu `while`. Táto nezhoda môže znamenať, že vo vygenerovanom kóde sú niektoré uzly naviac, alebo tam naopak chýbajú alebo kombinácia oboch. Najskôr sa zistí, ktoré príkazy presne nezhodu zapríčiňujú a na základe toho, je určený spôsob simulácie optimalizácií. V tomto prípade, vie algoritmus odhaliť dva typy optimalizácií. Zatienenie príkazu priradenia. Napr., opakované zapísanie hodnoty do premennej, kde zápis poslednej hodnoty nie je ňou samou ovplyvňovaný. A druhý typ optimalizácie je rozbalenie tela uzlu typu (`IfStatement`), na základe vyhodnotenia podmienky ako konštantnej hodnoty.

- `STRUCT_MISS` – znamená typy porovnávaných uzlov sa nerovnajú. Môže byť spôsobené napríklad redukciou priradení typu (`A = A + 1;`) na unárnu operáciu (`A++;`). Ďalej všetkými optimalizáciami spomínanými pre typ `BODY_LEN_MISS` v prípadoch, kde nenastane rozdiel v dĺžke tiel.

Pravdivostné tabuľky, okrem vyššie spomenutých optimalizácií, obsahujú vlastné. Jedná sa o štrukturálne optimalizácie. Keďže štruktúra pravdivostných tabuliek je oproti stavovým diagramom veľmi jednoduchá, bolo možné pomocou experimentálnej metódy zistiť, ako vplýva jej definícia na optimalizácie vo vygenerovanom kóde. Výsledkom je metóda `assemble_optimized_table`, ktorá pomocou sady niekoľkých pravidiel dokáže generovať optimálnu štruktúru kódu pravdivostnej tabuľky.

Ak optimalizátor vykoná zmeny v stromoch tabuliek či stavových diagramoch, tak sa vytvorí optimalizačné hlásenie. Hlásenie sa vytvorí aj v prípade, že algoritmus vykonávajúci optimalizácie nevie na danú nezhodu reagovať. Na základe zoznamu neidentifikovateľných

optimalizácií je rozšírená implementácia porovnávania AST. Ak by bol daný algoritmus fungoval tak, ako bol popísaný vyššie viď obr. 4.1, tak by skončil po prvom stretnutí s optimalizáciami, ktoré by nevedel na-simulovať. Teraz algoritmus porovnávania stromov funguje, tak, že si vedie zoznam ciest uzlov, ktoré odpovedajú miestam v kóde, kde sa nachádzajú rozdiely nevyriešené optimalizačným algoritmom. Týmto spôsobom sa môže preskočiť porovnávanie uzlov, ktorých nezhody optimalizačný algoritmus nedokáže vyriešiť a môže byť tak spracovaný celý strom.

5.6 Formát výstupu aplikácie

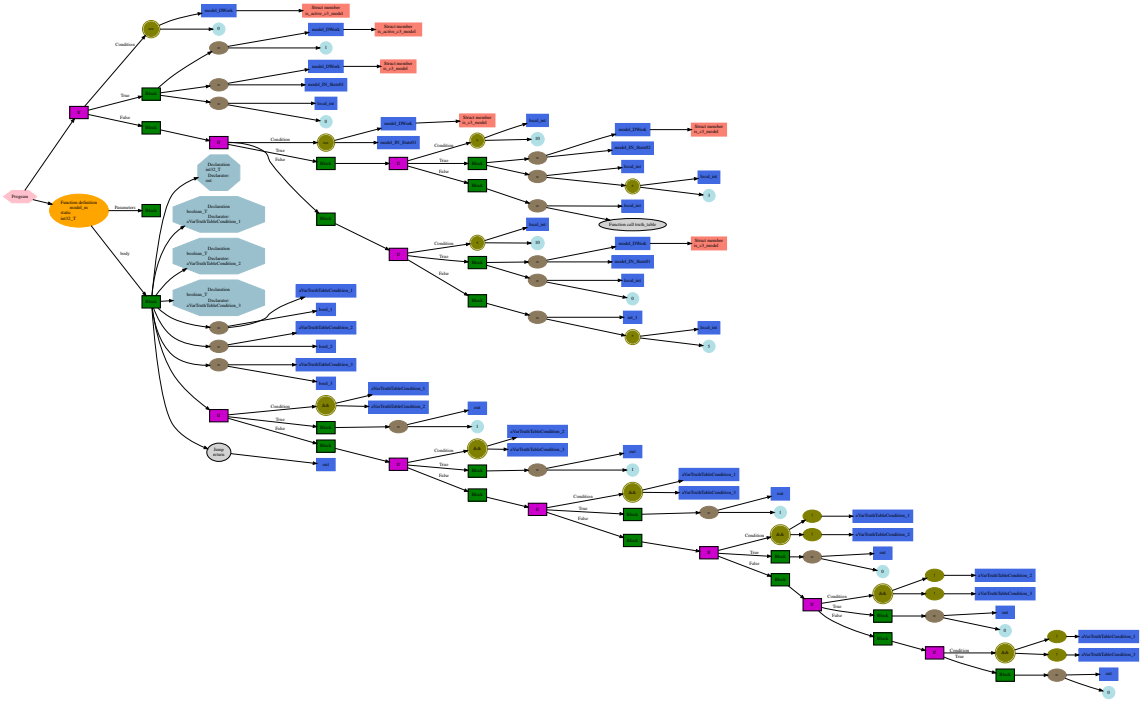
Užívateľ má možnosť si nechať vypísať dva typy hlásení. Prvé informuje užívateľa o tom aké sú rozdiely medzi finálnymi verziami stromov a druhé o tom aké optimalizácie boli vykonané na stromoch diagramov a pravdivostných tabuliek.

Hlásenie o rozdieloch v stromoch (angl. missmatch report) zobrazuje nasledujúce informácie:

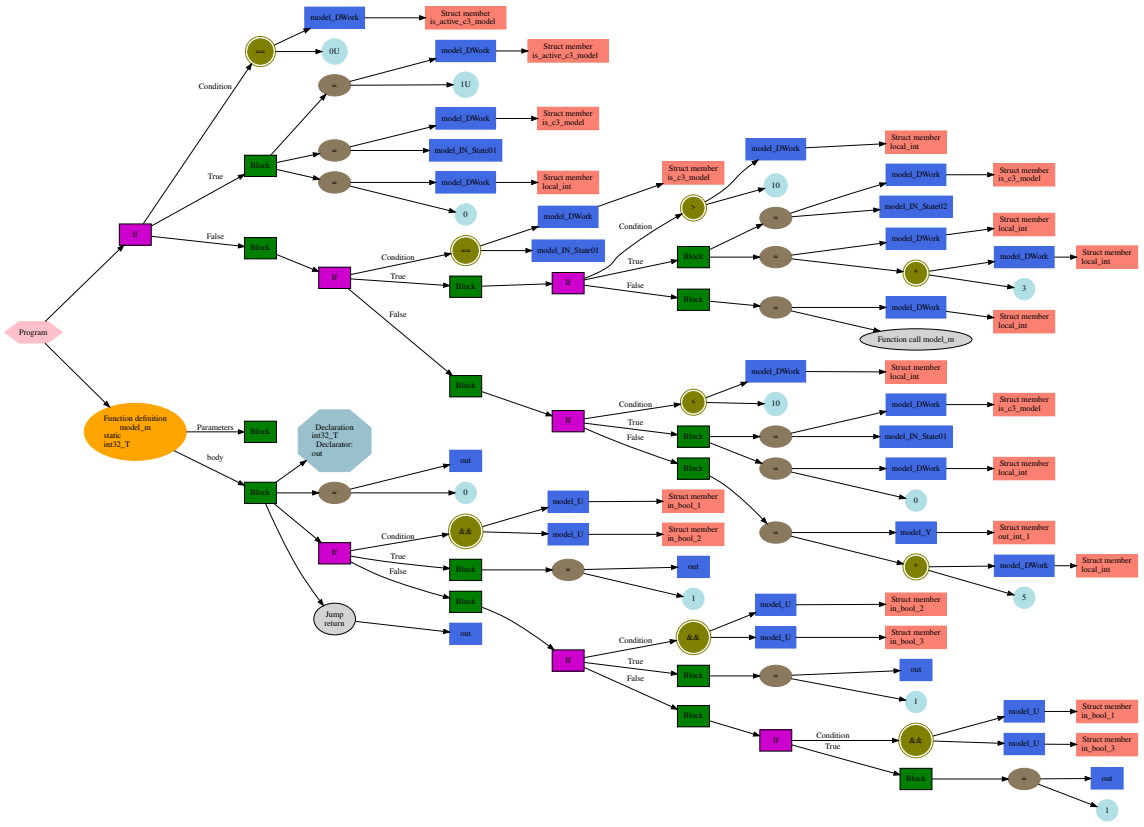
- Hĺbka stromu (angl. depth), v ktorej bola nezhoda nájdená
- Reťazcová reprezentácia uzlu referenčného stromu
- Reťazcová reprezentácia uzlu stromu vygenerovaného kódu
- Typ nezhody v systémovom tvare – (`INEQ_CODES.{missmatch}`), kde `mismatch` je názov atribútu typu nezhody. Napr., (`VAR_MISS`, `CONST_MISS`, ...).
- Cesta (angl. Path) medzi koreňom referenčného stromu a uzlom je zapísaná ako (`node_1/.../node_n`), kde `node` je v tvare (`class:attribute:index or len`). `class` je trieda uzlu, `attribute` je atribút uzlu, ktorý odkazuje na ďalší uzol na ceste. V prípade, že atribút uzlu má dátový typ list, `index` je index položky, ktorá odkazuje na ďalší uzol na ceste.

Hlásenie o vykonaných optimalizáciách (zmenách) na referenčnom strome obsahuje informáciu o tom, či danú optimalizáciu bolo alebo nebolo možné na-simulovať, slovný popis príčiny simulácie, v prípade, že ju bolo možné na-simulovať. Ďalej sa v ňom nachádza zdroj miesta v modeli. Túto informáciu vie aplikácia poskytnúť len v prípadoch, že boli optimalizácie priamo vykonané na kóde akcií modelu. V hlásení je zahrnutý aj typ dopadu optimalizácie na trasovateľnosť modelu na kód. Môže byť buď zanedbateľný (angl. negligible) alebo kritický (angl. critical). Tiež zobrazuje kód uzlu pred a po optimalizácii a na záver cestu medzi koreňom referenčného stromu a uzlu v ktorom bola vykonaná optimalizácia. Je popísaná rovnakým spôsobom ako v hlásení o rozdieloch v stromoch.

Okrem textových hlásení o rozdieloch v strome, či vykonaných optimalizáciách aplikácia ponúka vizualizáciu abstraktných syntaktických stromov. K dispozícii je zobrazenie troch typov AST. A síce vygenerovaný z modelu bezo zmien, optimalizovaný a AST zostavený popri rozbore z automaticky generovaného `C` Využíva sa voľne dostupný softvér **Graphviz** (angl. skratka pre *Graph Visualisation Software*). Obsahuje nástroje pre vykresľovanie orientovaných grafov a na ich popis využíva jazyk *dot* [9]. Zobrazovaný strom je vytvorený tak, že sa spojí počiatočný uzol stavového diagramu s uzlami pravdivostných tabuliek, ktoré sú v rámci neho definované pomocou koreňa s názvom `Program`. Príklad optimalizovaného (viď obr. 5.2) a ne-optimalizovaného (viď obr. 5.1) stromu jednoduchého stavového diagramu (viď obr. 2.4) s jednou definovanou pravdivostnou tabuľkou (viď obr. 2.2).



Obr. 5.1: Vizuálna reprezentácia neoptimalizovaného abstraktného syntaktického stromu modelu



Obr. 5.2: Vizuálna reprezentácia optimalizovaného abstraktného syntaktického stromu

Kapitola 6

Testovanie výslednej aplikácie

Testovanie aplikácie bolo rozdelené do dvoch fáz. Prvá fáza testovala jednotlivé časti aplikácie (analýza kódu, generovanie ne-optimalizovaných diagramov a pravdivostných tabuliek, optimalizovanie AST a i.) na vytvorenej sade testovacích modelov, ktoré sú súčasťou priloženého média v zložke `models`. Testovacia sada bola rozdelená na štyri skupiny podľa stavby diagramov.

Prvá skupina `simple_charts` obsahuje veľmi jednoduché stavové diagramy, ktoré testujú jednotlivé prvky generovania AST pomocou `XML` dokumentu reprezentujúceho dané diagramy. Menovite je testované správne zostavenie jednotlivých stavov a ich akcií. Testované sú jednoduché prechody bez použitia uzlov (angl. junctions), akcie prechodov, zložené výrazy a diagram s viacerými stavmi.

Druhá skupina `advanced_charts` obsahuje pokročilé konštrukcie stavových diagramov ako sú vnorené stavové diagramy, trasy medzi stavmi využívajúce uzly a pravdivostné tabuľky. Ďalej boli testované vlastnosti vnorených stavových diagramov (vnútorný vstup a vnútorný výstup). Tiež bol otestovaný výskyt viacerých stavových diagramov v jednom Simulink modeli.

Testy prvých dvoch skupín boli vykonávané na kódoch vygenerovaných s vývojárskym prekladovým prostredím (`dev`), v ktorom bolo povolené minimálne množstvo optimalizácií. To znamená, že všetky nastaviteľné optimalizácie popísané v podkapitole 2.2 boli vypnuté.

Na základe úspešných výsledkov testov oboch skupín, pre všetky modely aplikácia dokázala z modelu vytvoriť AST. Vytvorené AST z modelu sa zhodovalo zo zostaveným AST z vygenerovaného kódu.

Tretia skupina `optimized_charts` zahŕňa stavové diagramy, ktorých kód bol počas automatického generovania optimalizovaný. Táto skupina testovala bolo správne vykonávanie optimalizácií na stavových diagramoch.

Test na modeli `03_optimized_state_with_multiple_paths` odhalil nový typ nadmernej špecifikácie, viď podkapitola 2.3. Jedná sa modelovanie viac trás medzi dvoma stavmi. Tento typ optimalizácie aplikácia nebola schopná simulovať.

Posledná skupina `truth_tables` obsahuje, ako názov naznačuje, modely pravdivostných tabuliek (ďalej len PT). Testuje sa správnosť generovania definícií funkcií pre PT. Vzhľadom na to, že na generovanie a optimalizovanie PT je využívaný iný algoritmus ako pre stavové diagramy, je potrebné, aby boli otestované samostatne. Modely PT v testoch boli zámerne navrhnuté tak, aby obsahovali chyby designu. Napr., viacero rozhodnutí smerujúcich do jednej akcie, nadmerná špecifikácia podmienok a rozhodnutí či rôzne kombinácie poradia prázdnych akcií.

Testy prebehli takmer úspešne, zlyhal test jedného modelu, v ktorom bola vykonaná optimalizácia podmienky rozhodnutia na základe vyhodnotenia podmienok predchádzajúcich. Algoritmus generovania PT vykonáva "slepé" optimalizácie, a preto tento problém nedokáže vyriešiť.

Druhá fáza testovania spočívala vo využití sady **produkčných** modelov obsahujúcich stavové diagramy a pravdivostné tabuľky. Bolo testovaných **205** stavových diagramov s **28** pravdivostnými tabuľkami a bolo použitých **13** prekladových prostredí. Aplikácií sa poradilo celkovo odhaliť viac ako **3500** optimalizácií, z ktorých **561** bolo kritických pre trasovateľnosť. Ale aplikácia nebola bezchybná a v **21** prípadoch skončila chybou. Jednalo sa o chyby pri preklade syntaktického stromu na AST, spracovaní iných parametrov optimalizácie **I/O storage class** ako bolo pôvodne implementované, alebo zložité optimalizácie, ktoré nevedel algoritmus nasimulovať v konečnom počte iterácií.

Kapitola 7

Záver

Cieľom práce bolo vytvorenie aplikácie, ktorá na základe prijatého modelu a kódu z neho vygenerovaného vytvorí analýzu optimalizácií, ktoré boli zavedené do kódu počas generovania kódu. Aplikácia bola navrhnutá a implementovaná v jazyku Python. Je spustiteľná z príkazového riadku a hlásenia o optimalizáciách a rozdieloch v syntaktických stromoch zobrazuje vo forme textového zápisu. Ďalej poskytuje možnosť vizualizácie všetkých abstraktných syntaktických stromov, ktoré boli vytvorené počas analýzy.

Aplikácia bola otestovaná na dvoch sadách modelov. Na vlastnej sade viac ako štyridsiatich modelov a na sade produkčných modelov. Aplikácia bez problémov dokáže vytvoriť abstraktný syntaktický strom zo stavových diagramov aj pre náročné konštrukcie. Dokáže na-simulovať optimalizácie nielen pre čiastočné vyhodnocovanie výrazov, ale aj vyhodnocovanie zložitejších konštrukcií ako je vetvenie na základe podmienky (`if`) či štruktúru pravdivostných tabuliek. Avšak testovanie na sade produkčných modelov ukázalo nedostatky v algoritme na simuláciu optimalizácií, v prípadoch, kedy si dané simulácie vyžadovali širší kontext stromu než len uzly, v ktorých nastala nezhoda.

Pokračovaním tejto práce by potom mohlo byť rozšírenie o spätnú väzbu priamo v prostredí Simulink® spôsobom, že by aplikácia priamo vyznačila miesta v modeli, ktoré zapríčiňujú optimalizácie. Ďalej by mohol byť rozšírený algoritmus simulácie optimalizácií o prácu s kontextom celého abstraktného stromu a nie len jednotlivých uzlov. Tiež rozšírenie analýzy blokového úložiska modelu, aby dokázalo spracovať vstupné hlavičkové súbory aj pre nastavenie optimalizácie **I/O storage class** na hodnotu `ImportedExtern` a `ImportedExternPointer`.

Literatúra

[1] *Differences Between MATLAB and C as Action Language Syntax* [online]. The MathWorks, Inc. [cit. 2022-04-29]. Dostupné z: https://www.mathworks.com/help/stateflow/ug/differences-between-matlab-and-stateflow-action-language.html.

[2] *Embedded Coder® User's Guide* [online]. The MathWorks, Inc. [cit. 2022-05-09]. Dostupné z: https://www.mathworks.com/help/releases/R2017a/pdf_doc/ecoder/ecoder_ug.pdf.

[3] *Signal Basics* [online]. The MathWorks, Inc. [cit. 2022-04-04]. Dostupné z: https://www.mathworks.com/help/simulink/ug/signal-basics.html.

[4] *Software Considerations in Airborne Systems and Equipment Certification: RTCA/DO-178C*. RTCA Inc., 2011 [cit. 2022-04-27].

[5] *Software Considerations in Airborne Systems and Equipment Certification: RTCA/DO-331 Model Based Development and Verification Supplement to DO178C and DO-278A*. RTCA Inc., 2011 [cit. 2022-04-27].

[6] *Stateflow® User's Guide: Parallel (AND) State Decomposition* [online]. The MathWorks, Inc., 2017 [cit. 2022-05-01]. Dostupné z: https://www.mathworks.com/help/releases/R2017a/pdf_doc/stateflow/sf_ug.pdf.

[7] *Stateflow® User's Guide* [online]. The MathWorks, Inc., 2017 [cit. 2022-04-29]. Dostupné z: https://www.mathworks.com/help/releases/R2017a/pdf_doc/stateflow/sf_ug.pdf.

[8] Clark, J. a DeRose, S. *XML Path Language (XPath)*. Recommendation. W3C, November 1999 [cit. 2022-04-26].

[9] Gansner, E. R., Koutsofios, E. a North, S. *Drawing graphs with dot* [online]. User's Manual. January 2015 [cit. 2022-04-29]. Dostupné z: https://www.graphviz.org/pdf/dotguide.pdf.

[10] Parr, T. *ANother Tool for Language Recognition* [online]. [cit. 2022-04-04]. Dostupné z: https://www.antlr.org/index.html.

[11] Parr, T. *C 2011 grammar* [online]. [cit. 2022-04-04]. Dostupné z: https://github.com/antlr/grammars-v4/blob/master/c/C.g4.

Príloha A

Obsah priloženého média

```
Root
    models /* modely testovacej sady */
        advanced_charts
            01_advanced_big_chart_used_all_features
            02_advanced_chart_multi_level_state_nesting
            03_advanced_chart_multiple_charts_defined
            04_advanced_chart_junction_transitions
        optimized_charts
            01_optimized_chart_entry_action_constant_assignment
            02_optimized_chart_over_specified_transition_guard
            03_optimized_state_with_multiple_paths
        simple_charts
            01_simple_chart
            02_simple_chart_with_exit_action
            03_simple_chart_with_transition_actions
            04_simple_chart_with_extra_state
            05_simple_chart_unary_operation_in_action
            06_simple_chart_all_actions_specified
            07_simple_chart_with_truth_table
        truth_tables
            02_2_same_conditions_int
            03_2_same_conditions_no_operator_int
            04a_not_operator_boolean_cond
            04b_logical_operator_boolean_cond
            04c_logical_operator_boolean_cond_with_init
            04_without_operator_boolean_cond
            05a_all_same_cond_no_operator_boolean
            05b_all_same_cond_operator_boolean
            05c_all_same_cond_operator_boolean
            05_2_same_cond_no_operator_boolean
            06a_6_decisions_to_2_actions_set_true_set_false_bool
            06b_default_ret_value_actions
            06_6_decisions_to_2_actions_set_1_set_0
            07_6_decisions_to_2_actions_set_0_set_1000
            08_6_decisions_to_2_actions_set_1000_set_2
            09_6_decisions_to_2_actions_set_1000_set_2_with_default
            10_6_decisions_to_2_actions_set_1000_set_0_with_default
            11_decision_with_empty_action
```

```
        12_decision_with_empty_action_shuffled
        13_decision_with_different_empty_action
        14_truth_table_with_bool_ret
        15_truth_table_with_bool_ret_default
        16_truth_table_no_ret_empty_action
        17_truth_table_no_ret_empty_action_no_default
        18_bool_input_parameters
        19_mixed_input_parameters
        20_mixed_real_input_parameters
        21_mixed_real_input_parameters_with_ret
source /* zdrojové kódy */
    assembly.py
    AST.py
    error.py
    grammar
        CLexer.py
        CParser.py
        CVisitor.py
    main.py
    match_ast.py
    model_info.py
    parse.py
    preprocessor.py
    render_ast.py
    viz /* spustitelne súbory nástroja na vizualizáciu abstraktných
        syntaktických stromov */
        Graphviz
```

Príloha B

Hlásenie z analýzy automaticky generovaných kódov

Hlásenie je vo formáte `typ uzlu` : `kardinalita`. Kardinalita označuje počet výskytov uzlu daného typu naprieč celou testovacou sadou.

```
ExternalDeclarationContext : 60771
DeclarationContext : 162632
DeclarationSpecifiersContext : 192947
DeclarationSpecifierContext : 383321
TypeSpecifierContext : 1499918
TypedefNameContext : 1414447
TerminalNodeImpl : 19828385
FunctionDefinitionContext : 12784
DeclaratorContext : 148309
DirectDeclaratorContext : 274886
ParameterTypeListContext : 20970
ParameterListContext : 20970
ParameterDeclarationContext : 37148
DeclarationSpecifiers2Context : 19617
CompoundStatementContext : 206627
BlockItemListContext : 204736
BlockItemContext : 1274406
StatementContext : 1378151
ExpressionStatementContext : 1038395
ExpressionContext : 2595094
AssignmentExpressionContext : 5106339
UnaryExpressionContext : 6255143
PostfixExpressionContext : 5603668
PrimaryExpressionContext : 5603668
AssignmentOperatorContext : 1017597
ConditionalExpressionContext : 4103288
LogicalOrExpressionContext : 4103288
LogicalAndExpressionContext : 4138736
InclusiveOrExpressionContext : 4188681
ExclusiveOrExpressionContext : 4226322
AndExpressionContext : 4226322
EqualityExpressionContext : 4268957
RelationalExpressionContext : 4308559
ShiftExpressionContext : 4365063
```

```
AdditiveExpressionContext : 4375926
MultiplicativeExpressionContext : 4485499
CastExpressionContext : 5277348
SelectionStatementContext : 106064
TypeNameContext : 39802
SpecifierQualifierListContext : 1147517
LabeledStatementContext : 18584
ConstantExpressionContext : 14546
JumpStatementContext : 18705
ArgumentExpressionListContext : 124355
PointerContext : 22423
UnaryOperatorContext : 651060
TypeQualifierContext : 20608
InitDeclaratorListContext : 33509
InitDeclaratorContext : 33509
AbstractDeclaratorContext : 7282
DirectAbstractDeclaratorContext : 7282
IterationStatementContext : 2560
ForConditionContext : 2556
ForExpressionContext : 5112
StorageClassSpecifierContext : 32735
InitializerContext : 1051012
InitializerListContext : 29185
StructOrUnionSpecifierContext : 13333
StructOrUnionContext : 13333
StructDeclarationListContext : 13333
StructDeclarationContext : 591114
StructDeclaratorListContext : 84485
StructDeclaratorContext : 84485
```