VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

ZPĚTNÁ EXTRAKCE STATEFLOW DIAGRAMU Z KÓDU V JAZYCE C

REVERSE EXTRACTION OF STATEFLOW DIAGRAM FROM C CODE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **DANIEL GAVENDA**
AUTHOR

VEDOUCÍ PRÁCE **Ing. JAN FIEDOR, Ph.D.**
SUPERVISOR

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání diplomové práce

156463

Ústav:	Ústav inteligentních systémů (UITS)
Student:	**Gavenda Daniel, Bc.**
Program:	Informační technologie a umělá inteligence
Specializace:	Strojové učení
Název:	**Zpětná extrakce Stateflow diagramu z kódu v jazyce C**
Kategorie:	Algoritmy a datové struktury
Akademický rok:	2023/24

Zadání:

1. Seznamte se s formami reprezentace stavových strojů s ohledem na možnosti jejich strojového pracování.
2. Seznamte se s postupem, kterým MATLAB Embeded Coder překládá Stateflow diagramy do podoby kódu v jazyce C, včetně relevantních přepínačů ovlivňujících tento překlad.
3. Navrhněte metodu pro zpětný překlad kódu v jazyce C do interní reprezentace popisující Stateflow diagram vhodné pro potřeby porovnávání s původním Stateflow diagramem, která podporuje uživatelem vložený kód.
4. Implemenujte navrženou metodu.
5. Otestujte navrženou metodu na sadě vlastních modelů a reálných modelů dodaných firmou Honeywell.

Literatura:

- Baranov, Samary. (2018). Finite State Machines and Algorithmic State Machines.
- Dokumentace k produktům firmy MathWork. Dostupné na https://www.mathworks.com/ [cit. 2021-10-08]
- Standardy organizace RTCA dle doporučení konzultanta (např. DO-178C).

Při obhajobě semestrální části projektu je požadováno:
První tři body zadání.

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

Vedoucí práce:	**Fiedor Jan, Ing., Ph.D.**
Konzultant:	Mgr. Roman Dohnal
Vedoucí ústavu:	Hanáček Petr, doc. Dr. Ing.
Datum zadání:	1.11.2023
Termín pro odevzdání:	17.5.2024
Datum schválení:	6.11.2023

Abstrakt

Simulink diagramy majú široké využitie v priemysle, kde sa používajú na špecifikáciu riadiacich systémov. Konkrétnym blokom, ktorý sa v nich používa je Stateflow (stavový) diagram. Špecifikované systémy sú často kritické z hľadiska bezpečnosti, preto je otázka správnosti implementácie týchto systémov dôležitá. V tejto práci sa zameriame na návrh, popis a vytvorenie nástroja na spätnú rekonštrukciu Stateflow diagramu z optimalizovaného kódu v jazyku C za účelom preukázania funkčnej ekvivalencie medzi modelom a jemu odpovedajúcim kódom. Ďalej je vytvorená sada modelov reprezentujúcich modely so Stateflow diagramami a z nich vygenerované kódy pomocou viacerých optimalizácií. Na tejto sade bude vytvorený nástroj otestovaný, tiež je možné túto využiť pre evaluáciu iných analýz Stateflow a/alebo C kódu.

Abstract

Simulink diagrams are widely used in industry, where they are used to specify control systems. The specific block used in them is the Stateflow diagram. The specified systems are often critical from the point of view of security, therefore the question of the correctness of the implementation of these systems is important. In this work, we will focus on the design, description and creation of a tool for the reverse reconstruction of the Stateflow diagram from the optimized code in the C language in order to prove the functional equivalence between the model and its corresponding code. Next, a created set of models representing models with Stateflow diagrams and codes generated from them is created, using several optimizations. The created tool will be tested on this set, it can also be used to evaluate other Stateflow analyzes and/or C code.

Kľúčové slová

Simulink, Stateflow, optimalizácie, validácia prekladu, konečné stavové automaty, automatické generovanie kódu, spätná rekonštrukcia, Honeywell

Keywords

Simulink, Stateflow, optimizations, translation validation, finite state automata, automatic code generation, reverse reconstruction, Honeywell

Citácia

GAVENDA, Daniel. *Zpětná extrakce Stateflow diagramu z kódu v jazyce C*. Brno, 2024. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jan Fiedor, Ph.D.

Zpětná extrakce Stateflow diagramu z kódu v jazyce C

Prehlásenie

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jana Fiedora Ph.D. Další informace mi poskytl Mgr. Roman Dohnal ako zástupce firmy Honeywell International. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Daniel Gavenda
16. mája 2024

Poďakovanie

Rád by som sa poďakoval vedúcemu práce pánovi Ing. Janovi Fiedorovi, Ph.D za odbornú pomoc, rady pri riešení tejto práce. Rovnako by som sa chcel poďakovať konzultantovi pánovi Mgr. Romanovi Dohnalovi za ochotu odpovedať na všetky moje otázky.

Obsah

Kapitola 1

Úvod

Stateflow je súčasť nástroja Matlab určený na modelovanie a simuláciu diagramov stavového riadenia. Je využívaný v rôznych oblastiach ako napríklad automobilový a/alebo letecký priemysel a mnoho ďalších. Široká rada grafických konštrukcií, ktoré ponúka Stateflow umožňuje rýchlo, čitateľne a efektívne popísať komplexné systémy. Tento výskum sa uskutočnuje v spolupráci s firmou Honeywell International s.r.o., konkrétne s oddelením AeroSpace. Toto oddelenie sa zaoberá vývojom softvéru, ktorý je kritický na bezpečnosť (angl. safety-critical).

Návrh založený na modeloch (angl. model-based-design, skr. **MBD**) sa považuje ako správna cesta v tejto kategórií vývoja, pretože návrh modelov vytvára pre vývojára viac obmedzené a bezpečnejšie prostredie ako konvenčné ručné písanie kódu. Samotný nástroj Stateflow má však v tejto oblasti značné nedostatky. Je to preto, že v rámci dodávanej dokumentácie je definovaná sémantika podaná neformálnym spôsobom a tiež je definovaná len čiastočne [12]. Taktiež simulácia integrovaná v nástroji je schopná otestovať len malú podmnožinu vstupov a zložité systémy stavového riadenia môžu mať potenciálne neobmedzený počet konfigurácií.

Konkrétne požiadavky, ktoré sú kladené na softvér kritický na bezpečnosť sú obsiahnuté v rámci niekoľkých štandardoch a bližšie popísané v kapitole 2, kde je tiež spolu s existujúcimi nástrojmi zameranými na translačnú validáciu [5] automaticky generovaného kódu uvedená motivácia analýzy optimalizácií.

V kapitole 3 je detailný popis komponentov nástroja Stateflow, vymedzenie funkčné podmnožiny komponentov a popis sémantiky danej podmnožiny. Ďalej, v kapitole 4 sa nachádza popis použitých nástrojov na generovanie kódu a dopad dostupných nastaviteľných prepínačov generátoru kódu. Zvolenie internej reprezentácie a navrhnutie metódy pre spätnú rekonštrukciu opisujú kapitoly 5 a 6. Návrhu architektúry a implementácie navrhnutej metódy sa venuje kapitola 7. Výsledky testov sú obsiahnuté v 8 a o zhrnutí dosiahnutých výsledkov pojednáva kapitola 9.

Kapitola 2

Štandardy definujúce proces vývoja

Na to, aby mohol byť daný softvér certifikovaný na použitie v oblastiach, ktoré sú kritické pre bezpečnosť, musí spĺňať radu predpisov. Na vyriešenie tohto problému vznika rada štandardizovaných dokumentov.

Konkrétne našu problematiku popisujú dokumenty RTCA/DO-178C [8] a RTCA/D0-331 [6], ktorý je bližšie zameraný na vývoj založený na modeloch. V rámci nich sa nachádza odporúčaný postup pre všetky procesy v rámci céleho životného cyklu softvéru. Táto práca sa zaoberá konkrétnou častou vývoja, ktorou je takzvnaná *Verification of Outputs of Software Coding*. Tieto kontrolné a analytické činnosti zisťujú a hlásia chyby, ktoré môžu vzniknúť počas procesu tvorby zdrojového kódu softvéru. Medzi hlavné obavy patrí správnosť kódu s ohľadom na softvérové požiadavky, softvérovú architektúru a súlad so štandardmi softvérového kódu. Tieto kontroly a analytické činnosti majú za úlohu potvrdiť, že zdrojový kód tieto požiadavky spĺňa.

Ciele tejto časti sú popísané v rámci tabuliek **Table A-5** [9] resp. **Table MB.A-5** [7]. Berú sa do úvahy len ciele 1-6, lebo zvyšné sa zameriavajú na integračný proces, ktorý nie je predmetom tejto analýzy.

Ciele, ktoré popisujú verifikáciu výstupov z časti tvorby zdrojového kódu:

1. **Súlad s požiadavkami nízkej úrovne** (angl. Compliance with the low-level requirements): Cieľom je zabezpečiť, aby bol zdrojový kód presný a úplný s ohľadom na požiadavky nízkej úrovne a že žiadny zdrojový kód neimplementuje nezdokumentovanú funkciu. V tomto kontexte sú požiadavky nízkej úrovne vyjadrené návrhom modelovej reprezentácie.

2. **Súlad so softvérovou architektúrou** (angl. Compliance with the software architecture): Cieľom je zabezpečiť, aby zdrojový kód zodpovedal toku údajov a riadiacemu toku definovanému v softvérovej architektúre.

3. **Verifikovateľnosť** (angl. Verifiability): Cieľom je zabezpečiť, aby zdrojový kód neobsahoval žiadne konštrukcie *(štruktúry, statementy, ...)*, ktoré nie je možné overiť. Napríklad závislosť od externých zdrojov. Taktiež je potrebné, aby zdrojový kód bol testovateľný bez toho, že ho bolo potrebné meniť.

4. **Trasovateľnosť** (angl. Traceability) : Cieľom je zabezpečiť, aby boli všetky požiadavky na nízkej úrovni vyvinuté do zdrojového kódu. To znamená, že musí existovať

zobrazenie medzi jednotlivými stavebnými jednotkami modelového návrhu a časťami kódu. Keď sú požiadavky na nízkej úrovni vyjadrené modelovým návrhom, tento cieľ sa tiež zameriava na zabezpečenie toho, aby každá časť zdrojového kódu implementovala nejaký požiadavok na nízkej úrovni. Z toho vyplýva, že spomínané zobrazenie musí byť jednoznačné, a teda medzi komponentami kódu a modelu musí existovať funkčná ekvivalencia.

5. **Presnosť a konzistencia** (angl. Accuracy and consistency): Cieľom je určiť správnosť a konzistenciu zdrojového kódu vrátane využitia zásobníka, využitia pamäte, *fixed point* aritmetické pretečenie, aritmetika s pohyblivou desatinou čiarkou, spor o zdroje a obmedzenia, načasovanie vykonania v najhoršom prípade, spracovanie výnimiek, použitie neinicializovanej premennej, správa vyrovnávacej pamäte, nepoužívané premenné a korupcia dát v dôsledku konfliktov prerušenia (angl. interrupts) alebo plánovaním. Kompilátor (vrátane jeho možností), linker (vrátane jeho možností) a niektoré hardvérové funkcie môžu mať vplyv na najhorší prípad načasovania vykonávania a tento vplyv by sa mal posúdiť.

Výstupom splnenia každého z vyššie definovaných cieľov je požadované takzvané hlásenie softvérovej verifikácie (angl. Software Verification Results). Spomínané body sú splnené využitím automatického nástroja na generovanie kódu a ďalej vykonaním analýzy, ktorá potvrdí, že model a kód sú funkčne ekvivalentné. Zo zadania je potrebné zabezpečiť automatizáciu tohto procesu. Tiež je potrebné zmieniť, že pre softvér úrovne A, teda najväčšej úrovne kritickosti je vyžadované aby bol tento proces vykonaný nezávisle (angl. *with independence*). Pre splnenie tejto podmienky je možné použiť nástroj vytvorený inou technológiou ako nástroj čo daný kód vygeneroval. Tvorbe nástroja, ktorý automatizuje časť tejto analýzy, presnejšie rekonštrukciu modelu a kódu do zvolenej internej štruktúry sa venuje táto práca.

Zadanie ďalej vyžaduje zachovanie optimalizácií pri preklade, lebo minimalizujú potrebný čas na vykonanie úlohy a taktiež minimalizujú potrebnú pamäť. Aby bolo možné vytvoriť daný nástroj, potrebujeme mať znalosť o všetkých optimalizáciách, ktoré môžu nastať.

2.1 Súvisiace nástroje

Samotná firma Mathworks ponúka nástroj nazývaný *Simulink Code Inspector* (skr. SLCI) [11], ktorý automaticky porovnáva vygenerovaný kód s jeho zdrojovým modelom, aby splnil ciele kontroly kódu definované vyššie. Inšpektor kódu systematicky skúma bloky, stavové diagramy, parametre a nastavenia v modeli, aby určil, či sú funkčne ekvivalentné operáciám, operátorom a údajom vo vygenerovanom kóde. Poskytuje podrobnú analýzu trasovateľnosti medzi modelmi a kódmi. Generuje potrebné správy o splnení jednotlivých cieľov, ktoré môžu byť predložené certifikačným orgánom. Napriek tomu, že nástroj *SLCI* dokáže automaticky splniť všetky ciele, ktorým sa táto práca venuje, má niekoľko limitácií, ktoré ju robia nevhodným nástrojom pre danú úlohu. Hlavná limitácia je, že aby mohol byť tento nástroj spustený, je potrebné vygenerovať kód s použitím parametru `-SLCI`, ktorý má za dôsledok, že vypne všetky optimalizácie, ktoré prináša nástroj Embedded Coder [13]. Keďže má táto práca za cieľ rekonštruovať optimalizovaný kód, je tento nástroj pre potreby tejto práce nepoužiteľný.

Kapitola 3

Stateflow a jeho sémantika

Matlab Simulink je grafické prostredie vyvinuté firmou MathWorks, ktoré umožňuje návrh a simuláciu modelov dátového toku. Ponúka grafické rozhranie, ktoré umožňuje užívateľom vytvárať modely pomocou spájania jednotlivých blokov, ktoré reprezentujú rôzne komponenty (napr. filtre, násobičky, multiplexory, ...). Simulink modely sú blokové diagramy podobné IBD (Internal Block Diagram) diagramom zo *SysML* [1] s preddefinovanou a užívateľsky rozšíriteľnou sadou blokov s presne definovanou sémantikou (správaním), ktorá môže byť upravené podľa vstupu užívateľa (parametrizácia). Toto umožňuje nástroju Simulink vykonávať simuláciu správania celého systému v čase.

Matlab Stateflow je rožšírenie prostredia Matlab Simulink, ktoré poskytuje nástroje na modelovanie a simuláciu hierarchických stavových automatov. Je možné si ho predstaviť ako *State machine diagram* zo SysML [1], ktorý je rozšírený o nové konštrukcie, napr. pravdivostné tabuľky (*truth tables*). Skladá sa z viacerých komponentov a môžu byť rozdelené do 2 kategórií: s grafickou reprezentáciou a bez grafickej reprezentácie.

Popis jednotlivých komponentov Stateflow:

- **Stavový diagram** (angl. Flow Chart) tento blok sa používa v prostredí Simulink a vytvára izolovanú vrstvu medzi Simulink-om a Stateflow. Taktiež slúži ako rozhranie medzi týmito dvoma prostrediami. Umožňuje nastaviť množinu vstupných a výstupných signálov, ktoré ďalej komunikujú s jednotlivými funkčnými jednotkami prostredia Simulink.

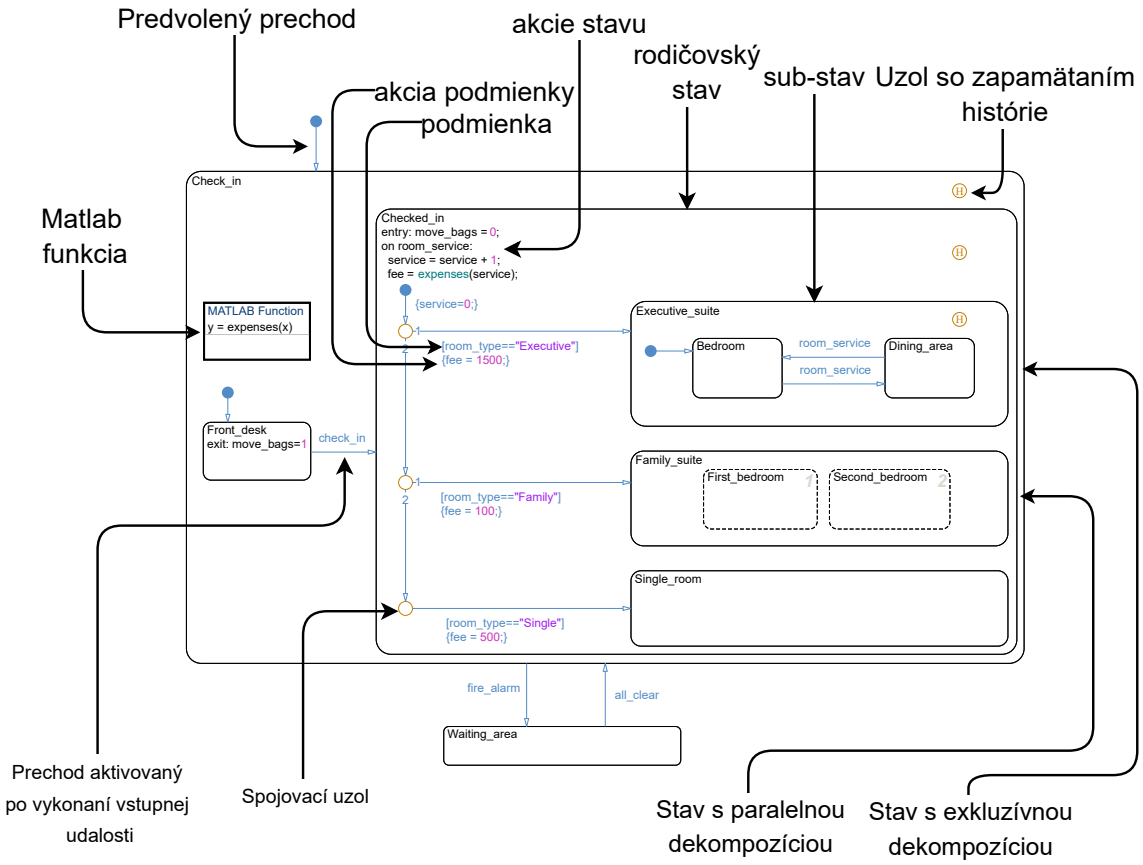
- **Stav** (angl. State), môže obsahovať až 3 akcie v príslušnom jazyku akcií (angl. action language). Vstupnú (entry), počas (angl. during) a výstupnú (angl. exit) akciu. Každý stav môže obsahovať všetky Stateflow komponenty, čím umožňuje vytvoriť hierarchickú štruktúru diagramov.

- **Prechod** (angl. transition) obvykle slúži na prepojenie dvoch stavov, môže slúžiť ako vstupný bod do stavového diagramu v ľubovoľnej úrovni a prepájať uzly. Môže obsahovať akcie a podmienku (angl. guard), ktorá určí kedy sa má daný prechod vykonať.

- **Spojovací uzol** (angl. Connective junction) slúži na prepojovanie prechodov, čím vytvára možnosť vetvenia. Tento uzol je možné analogicky prirovnať ku stavu bez vnútorných akcií, ktorého príchodzie a odchádzajúce prechody sa vyhodnotia v rámci jedného výpočtového kroku.

- **Uzol histórie** (angl. History junction) pridáva možnosť zapamätania si konfigurácie aktívnych pod-stavov v rámci rodičovského stavu. Pri prvej aktivácii sa vnorené stavy aktivujú bežne pomocou predvoleného prechodu. V prípade de-aktivácie rodičovského stavu, si uzol histórie zapamätá aktuálne aktívne vnorené stavy. Pri ďalšej aktivácii rodičovského stavu sa namiesto vykonania predvoleného prechodu aktivuje taký vnorený stav, ktorý bol posledný aktívny.
- **Grafické funkcie** (angl. Graphical function) je grafický element, ktorý pomáha zapuzdriť riadiacu logiku, iteratívne cykly a umožňuje ich znovupoužitie. V prípade, že v rámci grafických funkcií nie sú použité žiadne štruktúry, ktoré vedú na vytvorenie cyklov, tak sú pravdivostné tabuľky (angl. Truth Tables), funkčným ekvivalentom grafických funkcií.
- **Pravdivostné tabuľky** umožňujú definovať kombinatorickú logiku v tabuľkovom formáte.
- **Matlab funkcia** je ďalší grafický prvok, ktorý umožňuje implementovať algoritmy pomocou Matlab skriptu a tiež poskytuje volania niektorých vstavaných Matlab funkcií. Typicky sa využíva pre operácie zamerané na prácu s maticami, či dátovú analýzu a vizualizáciu.
- **Simulink funkcie** umožňujú znovu použitie simulink subsystémov.

Na obr. 3.1 je znázornený stavový diagram v modelovacom prostredí Simulink. Daný diagram slúži na ilustráciu popísaných komponentov daného grafického jazyka.



Obr. 3.1: Diagram znázorňujúci dostupné komponenty na modelovanie Stateflow diagramov

3.1 Vymedzenie Stateflow komponentov

Vzhľadom na to, že Stateflow diagramy sú využívané na modelovanie systémov, ktoré sú kritické s ohľadom na bezpečnosť, je dôležité určiť takzvanú *bezpečnú* podmnožinu komponentov prostredia Stateflow. Je nutné, aby odstránením daných komponentov Stateflow diagramy nestratili vyjadrovaciu silu. Komponenty sú zakazované hlavne z dôvodu, že často vedú na nesprávny návrh systému a tým otvárajú priestor pre chyby, alebo zapríčiňujú príliš náročnú analýzu generovaného kódu. Preto sú zavedené nasledovné obmedzenia:

- **Paralelná dekompozícia stavov** - povolená je len exkluzívna dekompozícia stavov. Vzhľadom na to, že aj tak nejde o pravý paralelizmus, tak je možné modelovať paralelné správanie pomocou zlúčenia paralelných stavov do jedného.
- **Udalosti** - v návrhoch sa nebudú vyskytovať udalosti ako aktivácie prechodov, nebudú sa vyskytovať ako podmienka na vykonanie akcie v stavoch a ani nemôžu byť v rámci diagramov nikde vyvolané. Nesprávna konštrukcia obsahujúca vyvolanie udalostí môže viesť na chybu pretečenia zásobníku (angl. Stack overflow).
- **Matlab funkcie** - nie je možné použiť v kombinácií s nastavením jazyka C ako jazyk pre akcie.

- **Simulink funkcie** - ich použitie je zakázané, v prípade, že je potrebné využiť nejaké vlastnosti Simulink prostredia, je možné tieto hodnoty propagovať priamo z modelu cez vstupné porty Stateflow diagramu.
- **Obmedzenie konštrukcií prechodov so spojovacími uzlami** - je zakázané vytvárať cykly, ktoré vedú v kóde na konštrukcie cyklov `while` a `for`. Ďalej je zakázané vytvárať prechody medzi stavmi, ktoré sú na inej úrovni hierarchie.
- **Grafické funkcie** - keďže je tiež zakázané vytvárať cykly medzi spojovacími uzlami, tak sú ekvivalentné k funkciám pravdivostných tabuliek a preto je ich používanie zakázané.
- **Uzly so zapamätaním konfigurácie vnorených stavov** - funkčnosť tohto komponentu je možné modelovať aj bez jeho použitia, tak ho nie je potrebné zahrnúť vo výslednej sade.

Okrem vymedzenia niektorých grafických komponent sa ďalej vyžaduje:

- Modely, v ktorých sa nachádza Stateflow diagram používajú **diskrétnu simuláciu s fixným krokom**
- Využíva sa výhradne jazyk C ako predvolený jazyk akcií
- Poradie vyhodnotenia prechodov je určené explicitne

3.2 Výpočtový krok stavového diagramu

Na základe vyššie definovaných obmedzení je teraz jednoduchšie definovať výpočtový krok stavového diagramu (ďalej len diagramu).

Pri prvej aktivácii diagramu sa nastaví diagram na aktívny a následne sa vykonajú predvolené prechody v predpísanom poradí. V prípade, že sa po aktivácii diagramu neaktivuje žiaden stav, spôsobí to chybu.

Ak je diagram aktivovaný a obsahuje aspoň jeden dosiahnuteľný stav, tak ostáva aktívny po celú dĺžku svojho životného cyklu.

Aktivovaný diagram vykoná svoj výpočtový krok každý krok modelovej simulácie. Ďalej je výpočet pre aktívny stav nasledovný. Vyhodnotia sa všetky odchádzajúce prechody podľa predpísaného poradia. Ak je aspoň jeden prechod splniteľný, tak sa daný stav deaktivuje, vykoná svoju výstupnú (angl. exit) akciu. Ak obsahuje vnorené stavy, vykoná to isté pre každý aktívny pod-stav. V prípade, že je vykonaný odchod zo stavu, ktorý obsahuje vnorené stavy, tak sa vynúti vykonanie a de-aktivácia všetkých aktívnych vnorených stavov. Ďalej sa vyhodnotia všetky akcie danej cesty prechodu, aktivuje sa cieľový stav a vykoná sa jeho vstupná (entry) akcia. Ak cieľový stav obsahuje aspoň jeden vnorený stav, tak sa aktivuje pomocou predvoleného prechodu na danej úrovni rovnakým spôsobom ako sa aktivuje diagram.

V prípade, že nie je ani jeden prechod splniteľný, tak sa v danom výpočtovom kroku aktívny stav nemení, ale sa vykoná sa jeho akcia počas (angl. during). Ak daný stav obsahuje aspoň jeden aktívny pod-stav, vykoná sa preň vyhodnotenie prechodov.

3.2.1 Výpočet pravdivostnej tabuľky

Každá pravdivostná tabuľka (ďalej len tabuľka) má zadaných N podmienok (angl. conditions). Podmienka je ľubovoľný booleovský výraz. Ďalej obsahuje M rozhodnutí (angl. decisions). Každé rozhodnutie obsahuje spôsob akým sa odpovedajúca podmienka vyhodnocuje. Hodnota (`T`) - znamená, že sa podmienka musí vyhodnotiť ako *pravda*, (`F`) - príslušná podmienka sa musí vyhodnotiť ako *nepravda* a (`-`) - na danej podmienke toto rozhodnutie nezávisí. Dané rozhodnutia sa vyhodnocujú v poradí zľava postupne D_1, D_2 a D_3 viď obr. 3.2. Ak je nejaké rozhodnutie splnené vyhodnotí sa príslušná akcie z tabuľky akcií, inak sa pokračuje ďalším rozhodnutím. Pravdivostné tabuľky je taktiež možné interpretovať ako logické formule viď 3.3.

Condition Table

	DESCRIPTION	CONDITION	D1	D2	D3
1		i32 > 10	T	-	-
2		f32 <= 0.1	T	F	-
3		i32 != 15 && f32 > 10.0	F	F	-
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE	A1	A2	A3

Action Table

	DESCRIPTION	ACTION
1		A1: y=false;
2		A2: y=true;
3		A3: y=!(i32 > 0);

Obr. 3.2: Definícia pravdivostnej tabuľky v prostredí Matlab

$$
\begin{array}{lll}
C_1 = i32 > 10 & D_1 = C_1 \wedge C_2 \wedge \neg C_3 & D_1 \implies A_1 \\
C_2 = f32 \leq 0.1 & D_2 = \neg C_2 \wedge \neg C_3 & \neg D_1 \wedge D_2 \implies A_2 \\
C_3 = i32 \neq 15 \wedge f32 > 10.0 & D_3 = true & \neg D_1 \wedge \neg D_2 \wedge D_3 \implies A_3
\end{array}
$$

$$
\begin{aligned}
\neg(C_1 \wedge C_2 \wedge \neg C_3) \wedge \neg(\neg C_2 \wedge \neg C_3) \wedge true &\implies A_3 \\
(\neg C_1 \wedge C_2) \vee C_3 &\implies A_3 \\
(i32 \leq 10 \wedge f32 \leq 0.1) \vee (i32 \neq 15 \wedge f32 > 10.0) &\implies y = i32 \leq 0;
\end{aligned}
$$

Obr. 3.3: Zápis pravdivostnej tabuľky z obrázku 3.2 v podobe logickej formule

Kapitola 4

Generovanie kódu zo Stateflow diagramov

Pomocou nástrojov *Embedded Coder*, *Stateflow Coder* a *Real-Time Workshop* je možné z modelových návrhov stavových diagramov automaticky vygenerovať kód v jazyku C, ktorý sa využíva v praxi vo vstavaných zariadeniach. Každý z týchto nástrojov tiež ponúka radu nastaviteľných parametrov, ktoré umožňujú nakonfigurovať kombinácie optimalizácií generovaného kódu. Z nich popíšeme len tie, ktoré majú dopad na spätnú automatickú rekonštrukciu v rámci zadania firmy Honeywell.

- **Predvolený chod parametrov** (angl. Default parameter behavior): Hovorí o tom, ako bude generátor kódu reprezentovať parametre numerických blokov. Hodnota *tunable* je predvolená a spôsobí to, že pre každý numerický blok ktorý obsahuje parameter vygeneruje explicitnú reprezentáciu daného parametru v C kóde. V prípade nastavenia hodnoty *inline* sa generátor kódu pokúsi o vloženie kódu nastavenia parametru priamo do výpočtu. Majme napríklad blok *Gain* z knižnice blokov Simulink, ktorý vykonáva operáciu násobenie skalárom hodnoty 3. Jednotlivé varianty kódu pre hodnoty *tunable* (riadky 3 a 4) a *inline* (riadok 5) je možné vidieť na obr. 4.1.

```
1 /* Hodnota optimalizačného parametru: tunable */
2 /* Hodnota parametru býva inicializovaná v
      samostatnom C súbore */
3 int_T gain_parameter = 3;
4 int_T gain_output = gain_input * gain_parameter;
5
6 /* Hodnota optimaliačného parametru: inline */
7 int_T gain_output = gain_input * 3;
```

Obr. 4.1: Varianty kódu pre typy optimalizácií parametru **Predvolený chod parametrov**

- **Operátor, ktorým budú reprezentované bitové a logické operácie** (angl. Operator to represent Bitwise and Logical Operator blocks) určuje typ operátoru, ktorý sa použije pre dané konštrukcie vo vygenerovanom kóde. Hodnota `Same as modeled` nijak nemodifikuje použité operátory v modeli. *Logical Operator* a *Bitwise Operator* naopak vynútia použitie príslušných operátorov vo vygenerovanom kóde `(!, &&, ||)`

resp. (`~, &, |`). Varianty vygenerovaného kódu pre hodnoty *Same as modeled* (riadok 2), *Logical Operator* (riadok 5) a *Bitwise Operator* (riadok 8) možno vidieť na obr. 4.2. Bitové operátory môžu byť okrem matematického zápisu reprezentované slovne pomocou definície príslušných makier (`NOT, AND, OR`).

```
1 /* Hodnota 'Same as modeled' */
2 output_bool = !((input_bool1 & input_bool2) ||
      input_bool3);
3
4 /* Hodnota 'Logical Operator */
5 output_bool = !((input_bool1 && input_bool2) ||
      input_bool3)
6
7 /* Hodnota 'Bitwise Operator */
8 output_bool = ~((input_bool1 & input_bool2) |
      input_bool3)
```

Obr. 4.2: Varianty kódu pri použití optimalizácie preferovaných operátorov pre realizovanie logických operácií

- **opätovné použitie pamäte** (angl. Buffer reuse) je skupina parametrov, ktorá sa špecializuje na minimalizovanie použitej pamäti, tým že použije alokovanú pamäť, ktorá sa v danom momente nevyužíva (hodnota, ktorú má v sebe uloženú už stratila platnosť) na uloženie hodnoty iného signálu. Príklad: majme jednoduchý simulink model, ktorý má dva vstupné porty, tie sú pripojené na výpočtový blok, ktorý realizuje logickú operáciu *And* a jej výstup je pripojený na výstupný port. Potom vygenerovaný kód bude vyzerať nasledovne, viď obr. 4.3, kde pri použití danej optimalizácie nie je potrebné v kóde vygenerovať premennú pre reprezentáciu bloku, lebo platnosť signálu výstupného portu začína až po vykonaní výpočtu numerického bloku.

```
1 /* Vygenerovaný kód bez použitia optimalizácie */
2 bool_T logical_and = input_port1 && input_port2;
3 bool_T output_port1 = logical_and;
4
5 /* Vygenerovaný kód s použitím optimalizácie */
6 bool_T output_port1 = input_port1 && input_port2;
```

Obr. 4.3: Varianty kódu optimalizácie opätovného použitia pamäte

- Ďalší parameter sa vzťahuje k použitiu a generovaniu vektorových signálov. Tento typ signálu umožňuje zadefinovať vektory o N dĺžke a zapúzdruje aritmetické operácie v rámci kódu akcií. Parameter **Prah rozbalenia cyklu** (angl. Loop unrolling threshold), má celočíselnú hodnotu, ktorá určuje pri akej šírke vektorového signálu sa priradenia týchto signálov generujú v cykloch. Jednotlivé varianty kódu pre zápis priradenia v modeli (riadok 1), ne-optimalizované priradenie v kóde (riadky 6 – 9) a zabalené vektorové priradenie (riadky 12 – 15) je možné vidieť na obr. 4.4.

```
1  /* Predpokladajme signál o šírke N */
2  /* Priradenie signálu v modeli */
3  out_vector1 = in_vector1 - 3;
4
5  /* Vygenerovaný kód bez optimalizácie */
6  out_vector1[0] = in_vector1[0] - 3;
7  out_vector1[1] = in_vector1[1] - 3;
8  /* ... */
9  out_vector1[N - 1] = in_vector1[N - 1] - 3;
10
11 /* Vygenerovaný kód s optimalizáciou */
12 int i = 0;
13 for (; i < N; i++) {
14     out_vector[i] = in_vector[i] - 3;
15 }
```

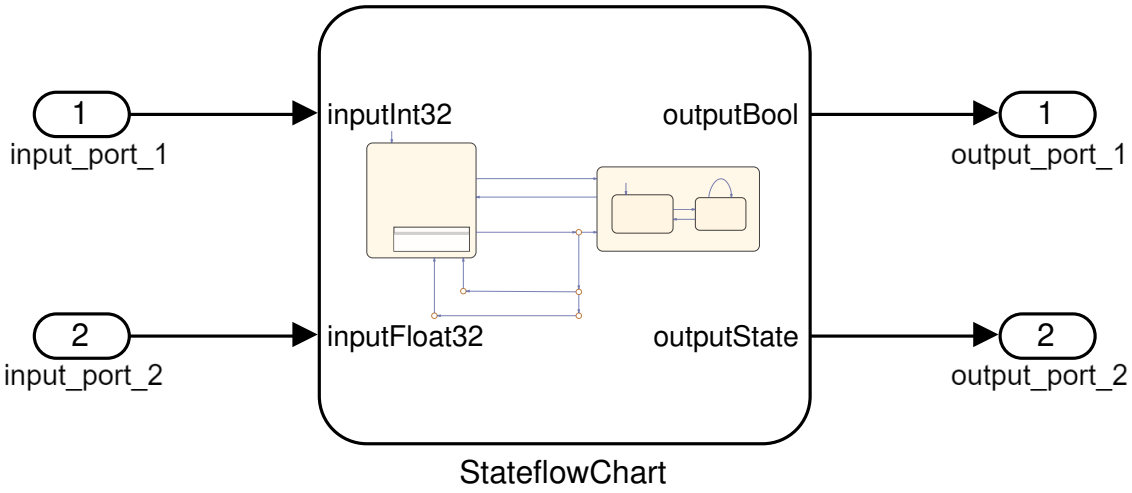
Obr. 4.4: Varianty kódu optimalizácie zabalenia vektorizovaného signálu do cyklu

- **Použitie funkcie *memcpy* na vektorové priradenie hodnoty** (angl. Use memcpy for vector assignment) používa celočíselný prah, ktorý špecifikuje počet bytov. V prípade, že daný vektor má väčšiu veľkosť ako zadaný prah, tak sa namiesto priradenia použije funkcia `memcpy`. Varianta kódu priradenia pre vypnutú optimalizáciu (riadky 3 – 6) a varianta s použitím funkcie (riadok 9) je možno vidieť na obr. 4.5.

```
1  /* Predpokladajme signál o šírke N s dátovým
       typom int32_T*/
2  /* Vygenerovaný neoptimalizovaný kód */
3  out_vector1[0] = in_vector1[0];
4  out_vector1[1] = in_vector1[1];
5  /* ... */
6  out_vector1[N - 1] = in_vector1[N - 1];
7
8  /* Vygenerovaný optimalizovaný kód */
9  memcpy(out_vector1, in_vector1, sizeof(int32_T)
       * N);
```

Obr. 4.5: Varianty kódu pre optimalizácie použitia funkcie memcpy

- Parameter **Typ vstupno/výstupného úložiska blokov** (angl. I/O storage class) dáva možnosť výberu, akým spôsobom budú reprezentované vstupné a výstupné signály modelu. Pri nastavení hodnoty *Auto* sú signály reprezentované ako dátový typ štruktúra (`struct`) v jazyku C. Ďalšie hodnoty *ImportedExtern* a *ImportedExternPointer* spôsobia, že vstupné a výstupné signály budú reprezentované externé hodnoty, alebo ukazovatele na dátový typ. Pre jednoduchý Simulink model, ktorý obsahuje Stateflow diagram pripojený na 2 vstupné a dva výstupné porty viď. obr 4.6. Jednotlivé varianty kódu pre hodnoty *Auto* (riadky 2 – 10), *ImportedExternPointer* (riadky 13–16) a *ImportedExtern* (riadky 19–22) je možno vidieť na obr. 4.7.



Obr. 4.6: Stateflow diagram komponent v prostredí Simulink

```c
/* Kód zodpovedajúci hodnote: Auto */
typedef struct {
    int32_T input_port_1;
    real32_T input_port_2;
} state_flow_chart_ExternalInputs;

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

/* Kód zodpovedajúci hodnote:
    ImportedExternPointer */
extern int32_T *input_port_1;
extern real32_T *input_port_2;
extern uint32_T *output_port_1;
extern boolean_T *output_port_2;

/* Kód zodpovedajúci hodnote: ImportedExtern */
extern int32_T input_port_1;
extern real32_T input_port_2;
extern uint32_T output_port_1;
extern boolean_T output_port_2;
```

Obr. 4.7: Varianty kódu s rôznymi hodnotami parametru I/O storage class

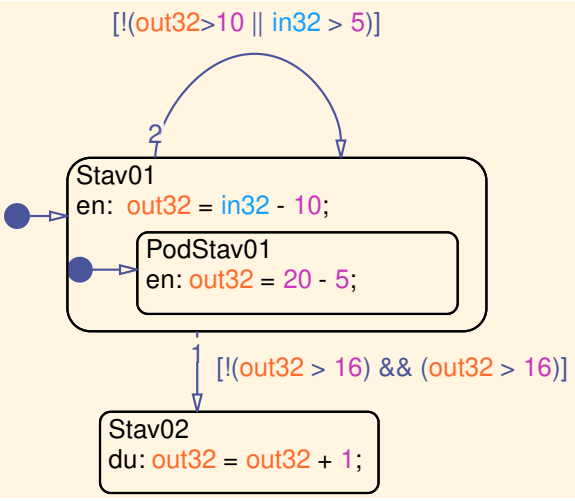
- **Inicializácia signálov na nulovú hodnotu** (angl. Signal Zero Initialization) určí, či sa pred začiatkom vykonávania výpočtu modelu majú všetky signály inicializovať na nulovú hodnotu. Typ tejto inicializácie je závislý od parametru (*Typ vstupno/výstupného* úložiska blokov). V prípade, že je zvolená hodnota parametru *Auto storage class*, tak je inicializácia vykonaná v dvoch krokoch. Najprv sa nastaví hodnota všetkých C štruktúr na nulovú pomocou vstavanej funckie jazyka C `memset`. To zahŕňa

vnútorné blokové úložisko, úložisko vstupných signálov a nakoniec výstupných signálov. Za touto inicializáciou nasleduje nastavenie nulových hodnôt pre jednotlivé polia štruktúr, resp. hodnoty externých hodnôt (v prípade, že parameter typ úložiska je nastavený na hodnoty *ImportedExtern* alebo *ImportedExternPointer*.

Je dobré poznamenať, že nástroje na generovanie kódu vykonávajú určité optimalizácie nezávisle od toho, aké parametre nástrojov na generovanie kódu sú nastavené. Tieto optimalizácie sa zameriavajú na minimalizáciu kódu. Je to napríklad odstránenie opakujúcich sa a nedosiahnuteľných častí diagramu. Stav sa nevygeneruje, v prípade, že do neho nevedie žiaden vykonateľný prechod. Konštantné výrazy sa vyhodnotia pri preklade. Ďalší preklad výrazov, napríklad: Konverzia aritmetických operácií delenia a násobenia na bitový posun v prípadoch, keď táto zmena nemá dopad na sémantiku daného výrazu.

Nech $sig1$ a $sig2$ sú signály v modeli, kde ich dátový typ je 32-bitové celé číslo. Potom bude akcia $sig1 = sig2 * 4$ vo výslednom kóde reprezentovaná ako $sig = sig2 << 2$. Ďalšia transformácia výrazov umožňuje preklad z $sig1 = sig1 + 2$ na $sig1+ = 2$ alebo naopak môže dôjsť k rozbaleniu aritmetického priradenia. Táto situácia nastáva v prípadoch, keď sú na pravej strane priradenia zložitejšie výrazy, alebo sa nástroj na generovanie kódu snaží zaručiť poradie vyhodnotenia v prípade neasociatívnych operácií.

Je daný jednoduchý Stateflow diagram viď. obr 4.8, do ktorého sú cielene zavedené konštrukcie, ktoré vynútia minimalizáciu generovaného kódu. Ako vidieť na útržku kódu 4.9 výraz vstupnej akcie stavu **PodStav01** je zjednodušený z `out32 = 20 - 5;` na `out32 = 15;`, to ilustruje vyhodnotenie konštantných výrazov v čase prekladu modelu na kód. Výraz v vstupnej akcii pre rodičovský stav **Stav01** nie je v kóde vygenerovaný, lebo priradenie do signálu `out32` je zatienené (angl. shadowed by) priradením vo vstupnej akcii v podstate **PodStav01**. Stav **Stav02** nebol vôbec vygenerovaný, lebo jediný prechod, ktorý do neho vstupuje má podmienku `!(out32 > 16) && (out32 > 16)`. Výraz tejto podmienky je nesplniteľný a generátor kódu to dokáže vyhodnotiť pri preklade. Na základe tejto informácie nie je daný prechod ani jeho cieľový stav zahrnutý vo výslednom kóde. Na tomto diagrame je ešte vidieť minimalizáciu podmienky cyklického prechodu na stave **Stav01**. Z kódu vyplýva, že na túto podmienku boli aplikované *De Morganove* zákony a má vo výsledku tvar `(out32 <= 10) && (out32 <= 5)`.



Obr. 4.8: Diagram znázorňujúci konštrukcie, ktoré vedú na predvolené optimalizácie

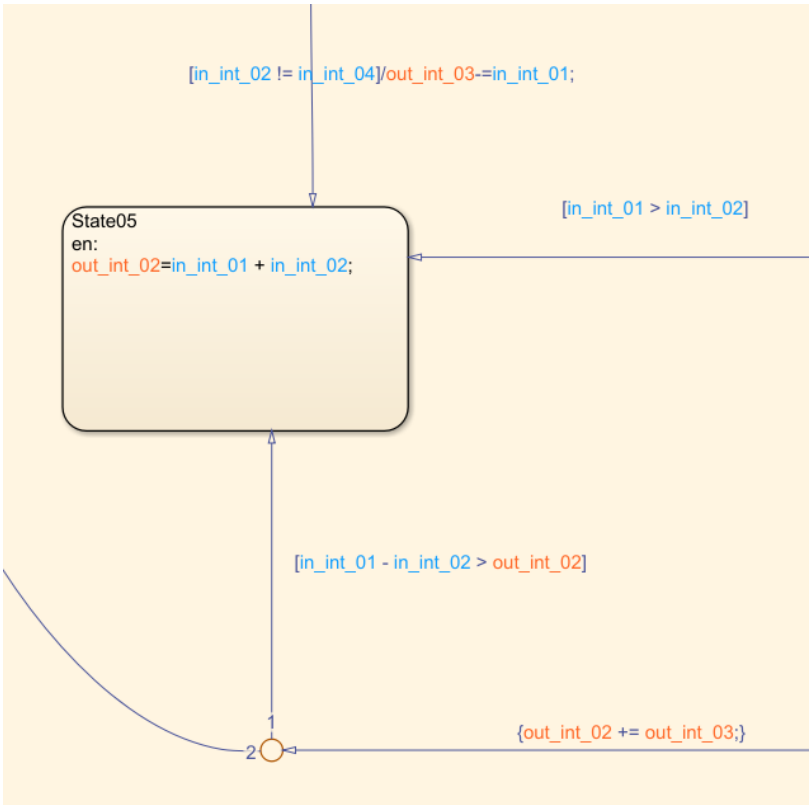
```
/* Model step function */
void model1_step(void)
{
  /* Chart: '<Root>/Chart' incorporates:
   *  Inport: '<Root>/In1'
   *  Outport: '<Root>/Out1'
   */
  /* Gateway: Chart */
  /* During: Chart */
  if (model1_DWork.is_active_c1_model1 == 0U) {
    /* Entry: Chart */
    model1_DWork.is_active_c1_model1 = 1U;

    /* Entry Internal: Chart */
    /* Transition: '<S1>:5' */
    model1_DWork.is_c1_model1 = model1_IN_Stav01;

    /* Outport: '<Root>/Out1' */
    /* Entry 'Stav01': '<S1>:4' */
    /* Entry Internal 'Stav01': '<S1>:4' */
    /* Transition: '<S1>:7' */
    /* Entry 'PodStav01': '<S1>:6' */
    model1_Y.Out1 = 15;
  } else if (model1_DWork.is_c1_model1 ==
      model1_IN_Stav01) {
    /* During 'Stav01': '<S1>:4' */
    if ((model1_Y.Out1 <= 10) && (model1_U.In1
        <= 5)) {
      /* Transition: '<S1>:17' */
      /* Exit Internal 'Stav01': '<S1>:4' */
      /* Entry 'Stav01': '<S1>:4' */
      /* Entry Internal 'Stav01': '<S1>:4' */
      /* Transition: '<S1>:7' */
      /* Entry 'PodStav01': '<S1>:6' */
      model1_Y.Out1 = 15;
    } else {
      /* During 'PodStav01': '<S1>:6' */
    }
  }

  /* End of Chart: '<Root>/Chart' */
}
```

Obr. 4.9: Ukážka vygenerovaného kódu z diagramu na obr. 4.8

Okrem minimalizácií výrazov, či odstránenie *mŕtveho* kódu sa pri preklade dejú optimalizácie typu: Extrakcia duplicitných častí kódu. Táto optimalizácia nastáva u väčších diagramov, ktoré využívajú komplexnú hierarchiu stavov, alebo vetvenie pomocou spojovacích uzlov. Je daný stavový diagram, ktorý má jeden stav bez odchádzajúcich prechodov viď obr. 4.10. Daný stav (*State05*) má 3 prichádzajúce prechody. To znamená, že vo výslednom kóde bude na 3 miestach test na podmienku daného prechodu, za ktorou bude nasledovať duplicitný kód aktivácie stavu a jeho vstupná akcia. Táto optimalizácia je prejavuje tak, že

namiesto danej časti kódu, v tomto prípade aktivácia stavu a vstupná akcia sú nahradené nastavením lokálneho prepínača. Ďalej je v kóde za vykonaním logiky daného stavového diagramu, pridaný kód, ktorého vykonanie je podmienené hodnotou daného lokálneho prepínača viď ilustračný útržok pseudo-kódu 4.11. Je potrebné uviesť, že táto optimalizácia nenastane v prípade, keď dané časti kódu nie sú zhodné. Príkladom je vstupný prechod s podmienkou (`in_int_02 != in_int_04`), ktorý obsahuje akciu prechodu. Keďže obsahuje kód naviac oproti ostatným prechodom tak sa neberie ako duplicitný a nie je na ňom táto optimalizácia vykonaná.

Nie je úplne jasné, čo je spúšťačom tejto optimalizácie. Ak je daný jednoduchý stavový diagram s rovnakou konštrukciou, tak daná optimalizácia nemusí nastať. Je pravdepodobné, že nástroje na preklad modelu obsahujú vnútorný prepínač, pri ktorom dané optimalizácie vykonávajú. Avšak, vnútorná implementácia použitých nástrojov nie je známa. Preto bola vykonaná analýza, ktorá testovala bod vykonania danej optimalizácie vzhľadom na počet použitých prvkov (stavy, prechody, spojovacie uzly) v stavových diagramoch. Z analýzy nebola zistená žiadna jednoznačná závislosť na vykonaní optimalizácie.



Obr. 4.10: Detail diagramu znázorňujúci stav s viacerými vstupnými prechodmi, ktoré vedú na optimalizáciu extrakcie duplicitného kódu

```
1
2  /* Vstupný bod diagramu */
3  if (!chart.isActive()) {
4      /* Aktivácia diagramu */
5      /* ... */
6  } else {
7      /* Vygenerovanie lokálneho prepínača */
8      boolean_T flag1 = false;
9      /* ... */
10     /* Výpočet diagramu na základe aktívneho
           stavu */
11     switch(chart.activeState()) {
12         /* ... */
13         if (in_int_01 > in_int02) {
14             /* Bez optimalizácie sa na tomto */
15             /* mieste generuje aktivácia */
16             /* stavu spolu so vstupnou akciou */
17             flag1 = true;
18         }
19         /* ... */
20     }
21
22     /* Extrahovaný duplicitný kód */
23     if (flag1) {
24         chart.setActiveState(State05);
25         State05.entryAction();
26     }
27 }
```

Obr. 4.11: Ukážka pseudo-kódu z diagramu na obr. 4.10

Tiež treba brať do úvahy kombináciu viacerých optimalizácii, práve tie spôsobujú najväčšie modifikácie do výslednej podoby kódu. vezmime si príklad z obr. 4.8, s tým, že je na prechode medzi stavmi **Stav01** a **Stav02** zmenená podmienka prechodu na `inBool == false`, kde `inBool` je vstupný signál daného stavového diagramu. V prípade, že bude na vstup diagramu pripojený blok generujúci konštantný signál, ktorý má vnútornú hodnotu `false` a sú nastavené optimalizačné parametre *Inline parameters* a *Buffer reuse*, tak dôjde k rovnakému odstráneniu daného prechodu a stavu **Stav02**. Rozdiel je však v tom, že v pôvodnom diagrame bola podmienka prechodu cielene nastavená, aby ju nástroj prekladu minimalizoval na konštantnú hodnotu. Podmienky tohto druhu sú považované za zlý návrh modelu, ktorý nie je akceptovaný. Avšak pri kombinácií optimalizácií to je problém, ktorý môže bežne nastať a nemusí byť jasne viditeľný pri návrhu modelu.

Nástroj Embedded Coder ponúka ďalej celú radu optimalizačných parametrov, ktorým sa táto práca nevenuje. Je nad rámec zadania tejto práce alebo cieľové architektúry, pre ktoré je daný kód generovaný, tieto vlastnosti nepodporujú. Je to napríklad použitie *SIMD* inštrukcií na optimalizovanie redukcií, paralelizácia `for` cyklov. Tieto optimalizácie nemajú zmysel používať, keď cieľový hardware nemá podporu vektorových inštrukcií, alebo obsahuje len jednu výpočtovú jednotku a preto nie je schopný vykonávať zdrojový kód paralelne.

Tiež sa v tejto práci nevenujeme optimalizáciám, ktoré vedú na minimalizáciu kódu. Tieto optimalizácie sú významné z pohľadu porovnávania vygenerovaného kódu a modelu na funkčnú ekvivalenciu. Avšak rekonštrukcia kódu z *minimálnej* podoby vedie na spočítateľne veľa validných riešení a ďalej bude v tejto práci ukázané, že pre preukázanie funkčnej ekvivalencie nie je potrebné kód rekonštruovať do podoby zhodnej s pôvodným vstupným modelom. Preto sa táto práca ďalej zaoberá popísanými výlučne nastaviteľnými optimalizáciami a optimalizáciou extrakcie duplicitného kódu.

Kapitola 5

Reprezentácie stavových strojov

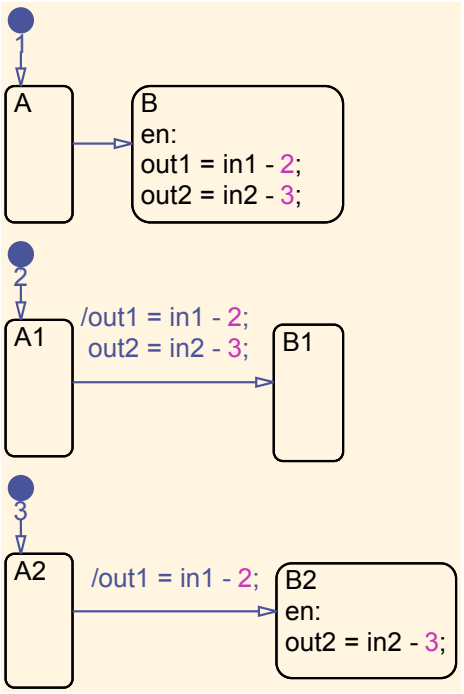
Táto kapitola sa venuje popisu vnútornej reprezentácii, do ktorej sa prekladá Stateflow diagram a rekonštruuje vygenerovaný C kód. Vnútorná reprezentácia sa využije na preukázanie funkčnej ekvivalencie. *Naivná* štruktúra reprezentujúca rekonštruovaný stavový diagram bude predvedená, spolu s jej nedostatkami. Následne sa predstaví štruktúra, ktorá sa použije aj v navrhnutom riešení.

Reprezentácia štruktúry na úrovni modelu (ďalej len štruktúra komponentov) vychádza z tabuľkovej reprezentácie konečných stavových automatov a je rozšírená tak, aby obsahovala všetky dodatočné informácie potrebné pre definíciu Stateflow diagramov. Samotná štruktúra bude implementačne realizovaná vhodným formátom pre strojové spracovanie. Skladá za z:

- **Model** obsahuje $1 - N$ stavových diagramov a $0 - M$ pravdivostných tabuliek.
- **Diagram** má svoj názov, $1 - K$ vstupných, $1 - L$ výstupných a $0 - Q$ lokálnych signálov. Ďalej obsahuje všetky príslušné stavy, prechody a spojovacie uzly.
- **Stav** má svoje akcie (vstupná, výstupná a akcia počas)
- **Prechod** obsahuje voliteľnú podmienku vykonania, akciu pri splnení podmienky a akciu pri splnení cesty.
- **Pravdivostná tabuľka** obsahuje $1 - P$ rozhodnutí, $1 - R$ podmienok a $1 - S$ akcií.
- **Signál** sa skladá z dátového typu, názvu a prípadne šírky vektoru.

V implementácií môže byť daná štruktúra reprezentovaná ľubovoľným objektovým formátom, ako sú (*xml*, *json*, a iné). Pre širokú podporu a lepšiu čitateľnosť formátu bude v implementácií zvolený formát *json*.

Avšak Stateflow je bohatý grafický jazyk a umožňuje zápis rovnakého výpočtu rôznymi spôsobmi. Majme stavové diagramy viď. obr 5.1. Ak by bola využitá štruktúra komponentov, tak by algoritmus porovnania označil jednotlivé stavové diagramy za rozdielne, lebo z pohľadu štruktúry komponentov sa jednotlivé počty líšia. Lenže ako je z príkladu zrejmé, výpočet každého zo znázornených diagramov sa vykoná v jednom kroku a priradenia sa vyhodnotia v rovnakom poradí. V skutočnosti je ďaleko viac možných konštrukcií, ktoré sú rozdielne, ale vykonávajú rovnaký výpočet. Môže sa k príkladom pridať ešte aj využitie výstupných akcií, akcií podmienok prechodov či využitie výstupných akcií z vnorených stavov. Preto je nutné použiť inú reprezentáciu, ktorá nebude stavové diagramy reprezentovať z pohľadu štruktúry modelu, ale z pohľadu výpočtu, ktorý daný model vykonáva.



Obr. 5.1: Ukážka príkladu rozdielnych diagramov, ktoré sú funkčne ekvivalentné

Táto štruktúra je získaná tak, že je pôvodný stavový diagram rozvinutý do stromu všetkých možných výpočtových ciest. V prípade, že pôvodný stavový diagram obsahuje cykly, daná reprezentácia bude viesť na nekonečne hlboký strom. Preto je zavedené obmedzenie, že každá cesta od koreňového uzlu, k listovému uzlu môže byť každý stav daného stavového diagramu navštívený nanajvýš 1-krát. Vezmeme strom prechodov na obr. 5.3 obsahujúceho všetky výpočtové cesty stavového diagramu z obr. 5.2. Na strome sú ďalej dokreslené šípky, ktoré ilustrujú cykly v danom stavovom diagrame. Cykly sú vždy vykonávané na uzloch akcie počas (uzly vyznačené zelenou farbou), ktoré sú listové a na uzloch vstupnej akcie (uzly vyznačené oranžovou farbou) taktiež tie ktoré sú listové.

Cykly na uzle akcie počas sú v diagrame vykonávané tak, že vetva s daného listového uzlu je previazaná na všetkých potomkov najbližšieho rodičovského uzlu, ktorý je iného typu ako *akcia počas*. V prípade, že sa jedná o uzly akcie počas stavov, ktoré neobsahujú žiadne vnorené stavy, tak existuje cyklus na samotnom uzle akcie počas. Tieto cykly sú v obrázku znázornené červenými šípkami.

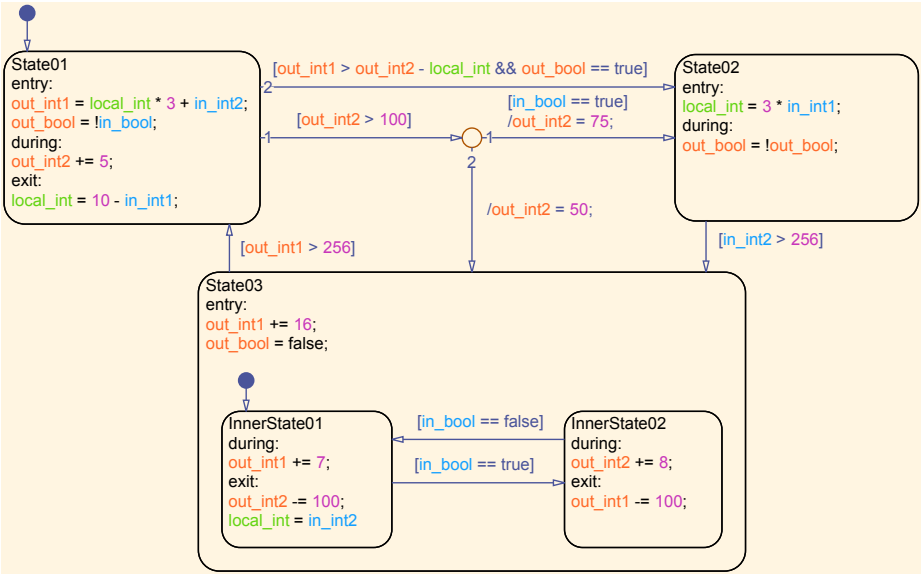
Cykly na listových uzlov vstupnej akcie sú tvorené tak, že sa na ceste rodičovských uzlov nájde uzol U_{en} vstupnej akcie, ktorý reprezentuje rovnaký stav. Potom hrany vedú od daného listového uzlu ku každému potomkovi uzlu U_{en}. V obrázku sú dané hrany vyznačené hnedou farbou pre uzly akcií vnorených stavov a modrou farbou pre stavy na najvyššej úrovne hierarchie (je vyznačený len jeden cyklus pre lepšiu čitateľnosť).

Uzol tohto stromu reprezentuje najmenšiu rozlíšiteľnú jednotku získanú z rekonštrukcie kódu resp. prekladu modelu. Sú to: *Inicializácia signálov* daného diagramu. Je to voliteľný uzol, ktorý sa vyskytuje v prípade, že daný diagram/model obsahuje aspoň jednu predvolenú inicializáciu vstupov/výstupov či lokálnych signálov. *Aktivácia diagramu* slúži ako vstupný bod pre stromy získané z kódu a obsahuje test aktívnosti a samotnú aktiváciu. Ďalej obsahuje uzly, ktoré reprezentujú modelové komponenty stavového diagramu: Vstupnú, výstupnú a akciu počas. Uzly prechodov, ktoré obsahujú podmienku prechodu a príslušné

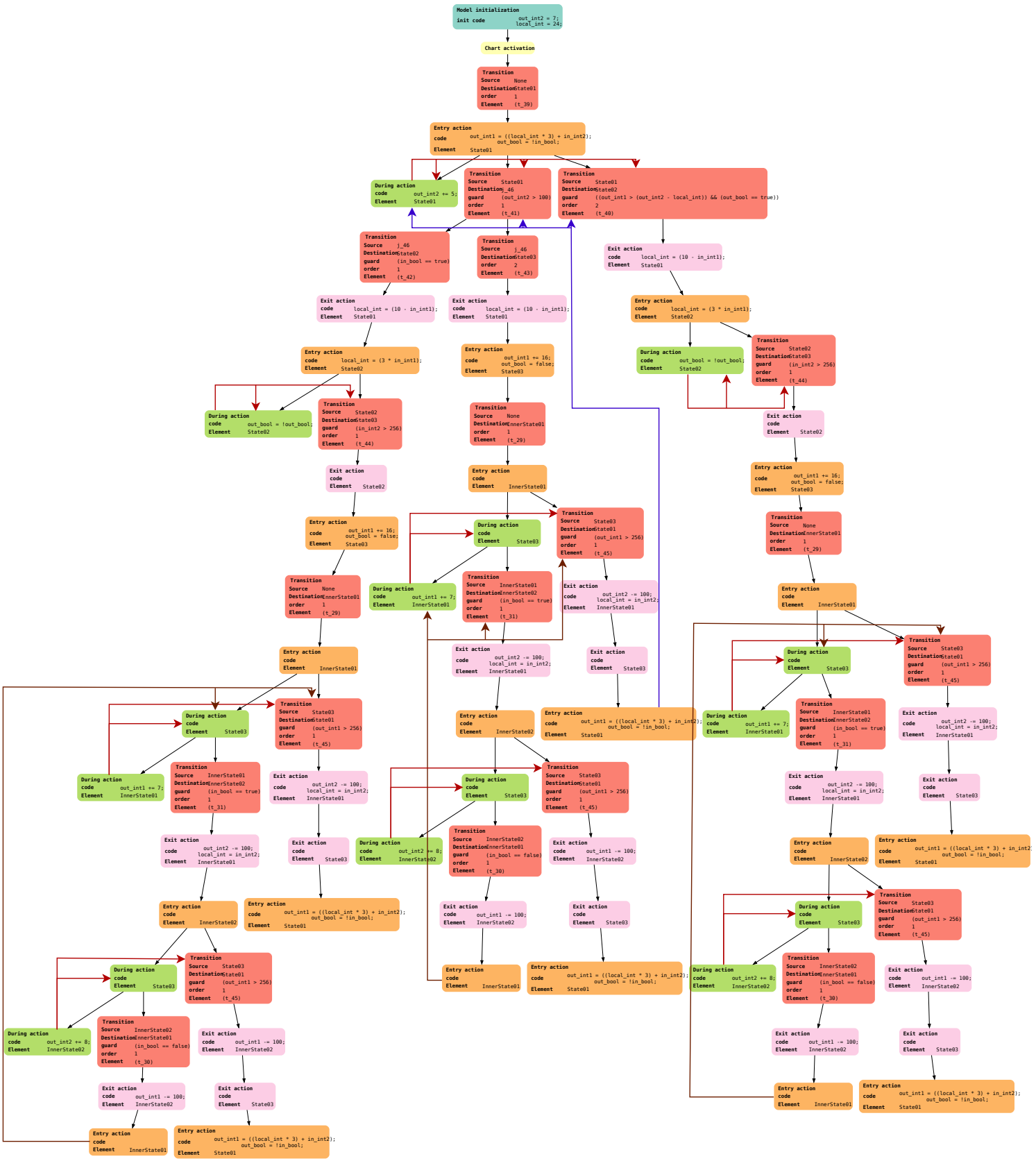
akcie. Strom z kódu naviac obsahuje uzol reprezentujúci aktiváciu stavu pri splnení cesty prechodov a informácie o deklaráciách lokálnych signálov či informácie o optimalizáciách, napr. uzly reprezentujúce extrakciu duplicitného kódu.

Ďalej bude táto reprezentácia ešte upravená takým štýlom, že všetky uzly, ktorých výpočet sa vykoná v rámci jedného výpočtového kroku sú zoskupené do jedného nadradeného uzlu. Toto umožní simuláciu optimalizácií na úrovni odstránenia *mŕtveho* kódu, nedosiahnuteľných stavov, prechodov a iné.

Táto výsledná upravená reprezentácia bude priamo využitá na vykonávanie porovnania vygenerovaného kódu a pôvodného modelu na funkčnú ekvivalenciu. V kapitole 8, kde je táto reprezentácia vidieť na príkladoch stromov na obrázkoch 8.4 a 8.5.



Obr. 5.2: Stavový diagram, ktorý bol použitý na vygenerovanie stromu prechodov z obrázku 5.3



Obr. 5.3: Ukážka stromu prechodov s naznačeným cyklením

Kapitola 6

Navrhnutá metóda spätnej rekonštrukcie

Ako spomenuté v predchádzajúcej kapitole, cieľom spätnej rekonštrukcie, a teda tejto práce je transformovať optimalizovaný generovaný kód do určenej internej reprezentácie, v ktorej bude jednoduché aplikovať ďalšie transformácie, ktoré umožnia preukázanie funkčnej ekvivalencie. Vzhľadom k tomu, že pri automatickom preklade stavového diagramu na C kód môže dôjsť k optimalizáciám (viď. kapitola 4) aj v prípadoch, keď sú všetky dostupné nastaviteľné prepínače nastavené na hodnotu, pri ktorej sa dané optimalizácie zakážu. Ako je spomenuté vyššie, nie je dosiahnuteľné a ani nevyhnutné zrekonštruovať kód do podoby zhodnej s modelom za účelom porovnania na funkčnú ekvivalenciu.

Nástroj na rekonštrukciu dostane na vstupe vygenerovaný kód z modelu. Vhodným nástrojom je kód načítaný a spracovaný. Je vykonané pred-spracovanie, syntaktická a sémantická analýza. Ich výsledkom je abstraktný syntaktický strom, reprezentujúci daný kód. Abstraktný strom je ďalej rekonštruovaný do štruktúry komponentov, ktorý je vykonávaný nasledovne. Na základe popisu sémantiky výpočtového kroku stavového diagramu na obmedzenej podmnožine komponentov v sekcii 3.2 môže byť určená základná štruktúra vygenerovaného kódu, ktorá bude odpovedať modelu na ktorom neboli vykonané žiadne optimalizácie. Na základe tejto štruktúry bude vytvorený algoritmus, ktorý prejde danú štruktúru kódu a rozdelí ju do zodpovedajúcich komponentov Stateflow diagramu.

Nech pre každý Stateflow diagram, ktorý má aspoň jeden dosiahnuteľný stav je vygenerovaná aktivácia diagramu viď. útržok pseudo-kódu 6.2. Tento uzol bude slúžiť ako vstupný bod do diagramu a preto je pri rekonštrukcií využitý na lokalizovanie diagramu v kóde. V `else` vetve je ďalej definovaná stavová logika. Podľa počtu stavov je daný výber stavov vygenerovaný ako `if` vetvenie pokiaľ diagram obsahuje 1 alebo 2 stavy. V opačnom prípade sa používa vetvenie pomocou konštrukcie `switch` viď. útržok kódu 6.2. V rámci logiky každého stavu je vygenerované vetvenie odchádzajúcich prechodov ako strom blokov `if`. V `else` vetve je potom vyhodnotenie `during` akcie a v prípade, že daný stav obsahuje jeden alebo viacero pod-stavov, je vygenerovaná stavová logika pre jeho pod-stavy.

Ak je cieľom prechodu stav, tak je kód vygenerovaný ako uzol `if` viď. útržok kódu 6.4. V prípade, že daný prechod nemá zadefinovanú podmienku prechodu, vygeneruje sa len telo daného bloku `if`. Najskôr sa vygeneruje kód pre odchádzajúcu (*exit*) akciu aktívneho stavu, potom v prípade, že daný stav má aspoň jeden pod-stav, vygeneruje sa odchádzajúca (*exit*) akcia všetkých aktívnych pod-stavov za ktorými nasleduje priradenie, ktoré má za účel deaktivovať každý aktívny pod-stav. Za odchádzajúcimi akciami sú ďalej generované

akcie podmienky a prechodu. Za týmito akciami nasleduje generovanie priradenia, ktoré reprezentuje aktiváciu cieľového stavu daného prechodu (ďalej len *priradenie prechodu*). Po ňom znova nasleduje kód pre vstupné akcie cieľového stavu a jeho pod-stavov. Z pohľadu rekonštrukcie je *priradenie prechodu* využívané na rozdelenie bloku kódu reprezentujúci logiku prechodu na časť pôvodného stavu a cieľového stavu. Ďalej nám toto priradenie umožňuje určiť či je v danom mieste v kóde cieľový stav, alebo uzol spojujúci viacero prechodov. Menšiu granularitu rozdelenia je možné dosiahnuť len v prípade, ak sú v čase generovania kódu do modelu vložené vodiace komentáre pre jednotlivé akcie a komponenty. Ďalej bude ukázané, že pre účel porovnávania bude postačovať mapovanie pomocou cesty medzi dvoma stavmi na výstupné akcie, akcie prechodov a na vstupné akcie. Kde cesta medzi je množina prechodov spojených uzlami.

V prípade, že destináciou prechodu je spojovací uzol, tak má vygenerovaný kód formu viď ilustráciu kódu 6.3. Z daného bloku je jednoznačne možné určiť akciu podmienky pre daný prechod. Telo vygenerovaného uzla `if` ďalej obsahuje vyhodnotenie všetkých odchádzajúcich prechodov z daného spojovacieho uzlu. Vyhodnotenie je realizované pomocou `if-elseif-else` vetvenia.

Spôsob a detaily rekonštrukcie sú bližšie popísané v kapitole 7. Rekonštruované komponenty sa ďalej využijú na zostavenie podoby stavového diagramu z kódu. Na nej je vykonaná simulácia prechodu diagramu, ktorá zostaví spomínaný strom rozvinutých prechodov daného diagramu.

Na druhej strane sa vykoná preklad modelu do reprezentácie štruktúry komponentov. Rekonštrukcia modelu na internú reprezentáciu je v princípe jednoduchšia. Model je na vstupe načítaný vo formáte `XML`. Z ktorej je následne rovnakou simuláciou prechodu diagramu ako kódová reprezentácia zostavený strom prechodov stavového diagramu, kde každá cesta od koreňa k uzlu obsahuje každý stav nanajvýš jedenkrát.

Výsledkom rekonštrukcie oboch častí je teda zvolená interná reprezentácia, ktorá bude slúžiť na porovnanie funkčnej ekvivalencie. Porovnanie pre funkčnú ekvivalenciu je síce nad rámec tohto zadania, ale môže byť vykonané nasledovne. Z modelu sa získajú všetky názvy stavebných blokov, ktoré sú prepojené so vstupmi resp. výstupmi nejakého stavového diagramu. Na základe tejto informácie sa vytvorí mapovanie mien reprezentácie z kódu na reprezentáciu zo stavového diagramu. Ak je toto mapovanie viacznačné, nástroj prehlási obe štruktúry za ne-ekvivalentné, podá hlásenie o mieste v štruktúre, ktoré danú viacznačnosť spôsobilo a skončí.

V opačnom prípade sa pokračuje jednoduchým testovaním oboch stromov prechodov na presnú zhodu, v tejto fáze nie sú vykonávané žiadne dodatočné transformácie stromu získaného z modelu. Ak porovnanie nájde presnú zhodu medzi štruktúrami, vypíše hlásenie dokazujúce zhodu a skončí. Ak sú prechodové stromy odlišné nástroj vykoná sadu predpísaných transformácií v pevne danom poradí:

1. Ak je nastavená optimalizácia *Parameter inlining*, tak je nahradený každý signál reprezentujúci parameter za jeho vnútornú hodnotu.

2. Nahradenie všetkých signálov, ktoré boli v rodičovskom uzle staticky vyhodnotené na konštantnú hodnotu.

3. Minimalizácia aritmetických výrazov, ktoré zahrňuje spočítanie konštantných výrazov, úprava výrazov, ktoré obsahujú operáciu bitového posunu na ich aritmetický ekvivalent $A = B << N$ sa zmení na $A = (B) * (2^N)$ resp. $A = B >> N$ sa zmení na $A = (B)/(2^N)$). Ďalej sa rozvinú všetky aritmetické priradenia: $A \mathrel{+}= B$ sa trans-

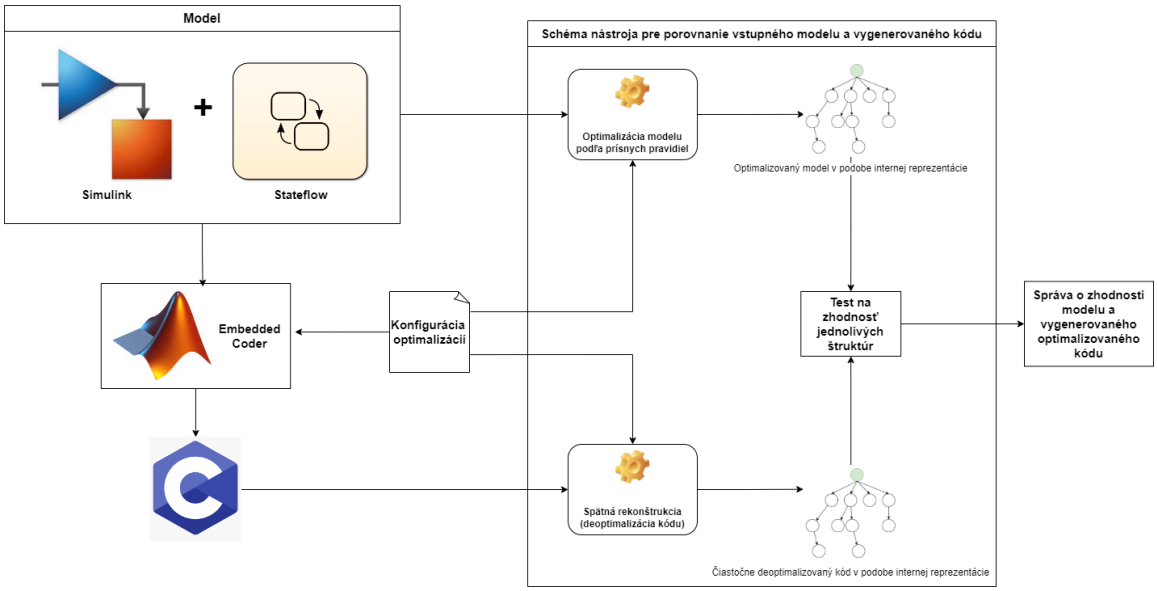
formuje na $A = (A) + (B)$, $A \ -= \ B$ sa transformuje na $A = (A) - (B)$ a analogicky pre všetky dostupné aritmetické priradenia dostupné v jazyku C.

4. Minimalizácia logických výrazov štandardnými postupmi: Quine–McCluskey algoritmická minimalizácia, aplikovaním axiómov booleovskej algebry apod.

5. Odstránenie uzlov, ktoré reprezentujú prechod stavového diagramu a ich podmienka prechodu je staticky vyhodnotená ako nesplniteľná resp. je podmienka prechodu vždy pravdivá, tak sú odstránené všetky uzly prechodov, ktoré majú vyššie číslo poradia vyhodnotenia.

6. Odstránenie tzv. *mŕtvych* priradení. Je to situácia, kde sa v jednom výpočtovom kroku diagramu nachádzajú 2 priradenia do rovnakého cieľového signálu a medzi nimi neexistuje kus kódu, ktorý by pristupoval k hodnote daného signálu.

Po transformáciách sa porovnanie zopakuje a nástroj podá príslušné hlásenie na základe zisteného výsledku.



Obr. 6.1: Diagram navrhnutej metódy spolu s aplikovaním transformácií na stromy a porovnaním oboch štruktúr

```
/* ... */
/* Ziačiatok kódu Stateflow diagramu */
if (sf_diagram.active == 0) {
    sf_diagram.active = 1;
    execute_default_transition();
} else {
    /* Vyhodnotenie logiky aktuálne aktívneho
        stavu */
    switch (sf_diagram.active_state) {
        case State01:
            eval_and_exec_ougoing_
                transitions(source);
            during_action(source);
            inner_state_logic(source);
            break;
        case State02:
            /* ... */
            break;
        /* ... */
        default:
            /* ... */
            break;
    }
}
/* Koniec kódu Stateflow diagramu */
/* ... */
```

Obr. 6.2: Štruktúra kódu pre aktiváciu diagramu

```
/* ... */
if (transition_condition) {
    condition_action(transition);
    /* if-elseif-else vetvenie prechodov, */
    /* ktoré vychádzajú z daného spojovacieho
        uzla*/
}
/* ... */
```

Obr. 6.3: Štruktúra kódu vyhodnotenia a vykonania prechodu, ktorý ústí v spojovacom uzle

```
/* ... */
if (transition_condition) {
    exit_action(source);
    internal_exits(source);
    condition_action(transition);
    transition_action(path);
    active_state = destination;
    entry_action(destination);
    inner_entry_actions(destination);
}
/* ... */
```

Obr. 6.4: Štruktúra kódu vyhodnotenia a vykonania prechodu, ktorý ústí v stave

Kapitola 7

Implementácia

Implementácia nástroja je vytvorená v jazyku Java21 kvôli jednoduchšej integrácií s existujúcim systémom, v ktorom bude využívaná. Popis je rozdelený do niekoľkých izolovaných celkov, ktorým sa ďalej venuje táto kapitola.

7.1 Spracovanie vstupu

Program prijíma na vstup zložku, ktorá musí obsahovať práve jeden model, buď vo formáte `slx`, alebo `mdl`. Ďalej je očakávaná pod-zložka, ktorá obsahuje vygenerovaný kód. Vygenerovaný kód je zložený z viacerých súborov. C zdrojový kód (v tvare `{názov_modelu}.c`), ktorý popisuje všetku logiku modelu, inicializáciu a výpočet pravdivostných tabuliek. Hlavný hlavičkový súbor obsahuje definície úložísk pre vstupy, výstupy, lokálne dáta, riadiace signály a parametre niektorých *Simulink* blokov. Pod-zložka s vygenerovaným kódom ďalej môže obsahovať voliteľné hlavičkové súbory generované na základe zvolenej konfigurácie a použitých modelovacích blokov:

- `{názov_modelu}_types.h` obsahuje definície extra typov napríklad definíciu štruktúry pre hodnoty blokových parametrov.
- `{názov_modelu}_private.h` definuje testy na správne veľkosti dátových typov pre zvolenú cieľovú architektúru, ktoré sa vykonajú v čase prekladu.
- `{názov_modelu}_data.h` obsahuje globálne definície hodnôt parametrov využitých v modeli. To sú napríklad parametre n-rozmerných vyhľadávacích tabuliek, generátorov konštantného signálu a iných výpočtových blokov, ktoré obsahujú vnútorné parametre.
- `{názov_modelu}_g.h` je tzv. *glue code*, ktorý vytvára rozhranie s cieľovým systémom.

S hlavičkovými súbormi `{názov_modelu}_private.h` a `{názov_modelu}_g.h` sa ďalej nepracuje, lebo nie sú potrebné pre rekonštrukciu. Taktiež sa využíva knižnica definovaných typov z nástroja *Real-Time Workshop* (rtwtypes.h). Viaže sa na konkrétnu verziu nástroja Matlab a vnútornú konfiguráciu generovania kódu. Tento súbor je súčasťou výsledného nástroja, tým pádom ho nie je potrebné dodávať s každým modelom zvlášť. Posledný súbor, ktorý je potrebný na vstupe je reprezentácia modelu vo formáte *json*. Model je reprezentovaný ako štruktúra komponentov, ktorá je popísaná v kapitole 5. Obsahuje všetky potrebné časti stavových diagramov: ich stavy, prechody a akcie. Všetky akcie sú načítané

ako reťazce textu, a teda bude potrebné na nich vykonať ďalšie spracovanie, aby boli použiteľné na porovnanie. V prípade, že tento súbor neexistuje vo vstupnej zložke, vygeneruje sa automaticky. Vytvorený nástroj v tomto prípade vyvolá program *Matlab* so skriptom `genericExport.m`. Tento skript využíva programové rozhranie nástroja *Stateflow*, kde sa môže dopýtať na všetky komponenty zahrnuté vo vstupnom modeli. Všetky nepotrebné komponenty sú odfiltrované a požadované sú transformované do *json* formátu.

Modul `ModelDataLoader`, ktorý je zodpovedný za načítanie vstupu, skontroluje existenciu všetkých povinných súborov a prípadne sa pokúsi danú reprezentáciu komponentov stavového diagramu vygenerovať. Treba dodať, že pre tento krok je potrebné mať na danom stroji nainštalované prostredie Matlab.

7.2 Analýza zdrojových súborov

Pre analýzu zdrojových súborov jazyka C je potrebné aspoň do istej miery vykonať kroky, ktoré robia prekladače jazyka C. Preklad začína fázou pred-spracovania (angl. preprocessing). Pred-spracovanie jazyka C definuje množinu direktív [2] (`#if`, `#ifdef`, `#elif`, `#else`, `#error`, `#pragma`, `#line`, `#include`, `#define` a iné). Štandardný zápis hlavičkových súborov v jazyku C využíva direktíva (`#ifndef`, `#define`) napríklad pre zabránenie viacnásobného importovania jedného súboru. Vzhľadom na to, že je dopredu známa množina všetkých zdrojových súborov, tak je možné tento krok pred-spracovania vylúčiť. Ďalej je možné vylúčiť spracovanie direktív `include` tým, že spracovanie zdrojových súborov bude vykonávané v pevne danom poradí tak, aby bola zachovaná závislosť definovaných symbolov. Poradie spracovania súborov je dané nasledovne: `rtwtypes.h`, `{model_name}_types.h`, `{model_name}.h` (hlavný hlavičkový súbor modelu), ďalej všetky ostatné dostupné hlavičkové súbory v ľubovoľnom poradí a nakoniec `{model_name}.c` súbor. Vzhľadom na to, že ostatné direktíva sa v zdrojových súboroch nevyskytujú, je pre fázu pred-spracovania potrebné implementovať len spracovanie `#define`. Príkaz `#define` má významné použitie pre stavové diagramy, lebo všetky konštanty, ktoré reprezentujú stavy sú v tvare: `#define {model_name}__IN__{state_name} {integer_state_constant}`. To umožní vytvoriť mapovanie uzlov kódu na názvy stavov a tým zjednoduší mapovanie medzi stavmi, ktoré sú rekonštruované z kódu a tými z modelu.

Pred-spracovanie prebieha nasledovne. Súbor sa načíta ako text a rozdelí do riadkov podľa znaku (`\n`). Ak daný riadok začína reťazcom `#define`, tak sa pre daný kľúč (názov modelu a stavu) a hodnotu (číselná reprezentácia) vytvorí záznam v pomocnej pamäti. Predspracovanie definície makier sa od štandardnej implementácie líši tým, že po analýze definovaných hodnôt nedochádza ďalej k nahradeniu všetkých výskytov v kóde.

Ďalšia fáza analýzy je syntaktická a sémantická analýza. Predpokladá sa, že generovaný C kód pomocou použitých nástrojov (Embedded coder apod.) je správny po syntaktickej aj sémantickej stránke. Avšak tieto analýzy je potrebné vykonať znovu, aby sa skonštruoval abstraktný syntaktický strom (angl. Abstract syntax tree, ďalej len AST). Na vykonanie lexikálnej a syntaktickej analýzy je použitý voľne dostupný nástroj *Another tool for language recognition ver. 4* (ďalej len `antlr4`). `Antlr4` je výkonný generátor syntaktického analyzátora na čítanie, spracovanie, interpretáciu alebo preklad štruktúrovaného textu alebo binárnych súborov. Je široko používaný na vytváranie jazykov, nástrojov a frameworkov. Z gramatiky vygeneruje syntaktický analyzátor, ktorý dokáže vytvárať a prechádzať derivačné stromy [4]. Nástroj dostane na vstup gramatiku, ktorá definuje tvar lexém a pravidiel pre analýzu zhora-nadol. Využíva vlastný formát gramatík `g4`. Táto práca využíva voľne dostupnú gramatiku [3] pre jazyk C, ktorá je založená na štandarde C11 . Prevzatá grama-

tika je ďalej čiastočne upravená. Lexémy logických operácií sú rozšírené o reťazce `AND`, `OR` a `NOT` ako je popísané v kapitole 4. Táto zmena gramatiky umožní bezproblémovú podporu rekonštrukciu pri použití parametru generovania kódu, ktorý zamení logické operátory za bitové, kde sú operátory reprezentované reťazcami. Ďalej sú v gramatike zmenené pravidlá pre zaobchádzanie s kódovými komentármi. V pôvodnej gramatike sa všetky komentáre ignorujú. Avšak ako bude neskôr ukázané, komentáre ktoré sú generátormi vložené zjednodušia rekonštrukciu niektorých častí diagramu. Preto je dané pravidlo upravené tak, aby namiesto zahadzovania komentárov sa dané lexémy ukladajú na skrytý kanál, z ktorého budú pri konštrukcií AST získané.

Takto upravená gramatika je predaná nástroju *antlr4*, ktorý z nej vygeneruje *rozpoznávač* daného jazyka (modifikovaný C11). *Rozpoznávač* je tvorený lexikálnou a syntaktickou analýzou a je implementovaný vo zvolenom programovacom jazyku (Java21), príjme na vstup textovú reprezentáciu kódu, skontroluje jeho správnosť a vytvorí derivačný strom.

Tento derivačný strom je veľmi *nafúknutý* a obsahuje priveľa uzlov, ktoré nie sú pre rekonštrukciu vôbec potrebné. Jednak sú to uzly ktoré reprezentujú terminály potrebné pre lexikálnu/syntaktickú analýzu, ale v stromovej štruktúre už nemajú význam (`(`, `)`, `;`, apod.). A ďalším dôvodom je prístup implementácie vytvoreného *rozpoznávača* jazyka. Nástroj *Antlr4* nepodporuje syntaktickú analýzu zdola-nahor. Aby bola zachovaná precedencia operátorov, je nutné daným štýlom navrhnúť gramatické pravidlá. To vo výsledku spôsobí, že vytvorený derivačný strom bude obsahovať veľa uzlov, ktoré vznikli za účelom zachovania tejto precedencie (viď. obr. 7.1).

Preto sa ďalej vykonáva preklad derivačného stromu na AST. Na preklad je využívaná vlastnosť *Antlr4* nazývaná *Visitor*, ktorá umožňuje spraviť prechod derivačným stromom tak, že na každom uzle je vyvolaný *callback*, ktorý daný uzol transformuje. Tieto *callbacky* sú zadefinované tak, aby transformovali derivačný strom na AST. Ďalej je pri prechode derivačného stromu overená definícia použitých dátových typov a tiež je vykonané spárovanie komentárov nasledovným spôsobom: Nech každý uzol v AST má definované číslo riadku n_u na ktorom sa nachádza v zdrojovom texte a index i_u ktorý určuje pozíciu v liste všetkých tokenov. Potom je s daným uzlom spárovaný každý komentár, ktorého číslo riadku je n_u a každý komentár ktorého číslo riadku n_k je menšie ako n_u, pre ktorý platí, že neexistuje uzol AST s indexom i_{u2}, takým, že $i_k < i_{u2} < i_u$ a zároveň neexistuje uzol s číslom riadku n_{u2} takým, že $n_k = n_{u2}$.

Štruktúra AST bola vytvorená na základe návrhu v [10]. Bola ďalej upravená reprezentácia niektorých uzlov za účelom zjednodušenia rekonštrukcie. Napríklad uzol reprezentujúci symboly premenných (uzol `Var`) je implementovaný tak, aby bol prístup k názvu premennej invariantný voči parametru *Typ vstupno/výstupného úložiska blokov*.

Pre zjednodušenie implementácie prekladu derivačného stromu bola vykonaná analýza, ktorá skúmala použité gramatické pravidlá vo všetkých vygenerovaných zdrojových kódov z modelov na vytvorenej sade modelov a tiež všetkých modeloch, ktoré sa využívajú vo firme Honeywell (výsledky analýzy sú dostupné na pribalenom médiu v súbore `node_analysis.txt`). Na základe tejto analýzy bolo možné vylúčiť určitú množinu typov uzlov, ktoré nástroje na generovanie kódu nevyužívajú.

Vzhľadom na to, ako je daná gramatika jazyka C postavená (niektoré uzly AST sa zostavujú počas viacerých volaní, napr. definícia funkcie) je sémantická analýza rozdelená na 2 prechody. V druhom prechode (teraz už zostrojeného AST) sa overí použitie symbolov, či je každý symbol v danom rámci (angl. scope) zadefinovaný. Symboly sú potom previazané tak, že každý uzol predstavujúci premennú (uzol `Var`) má odkaz na príslušný uzol deklarácie

(uzol `Decl`) resp. inicializácie (uzol `Init`) a symboly funkčných volaní (uzly `FcCall`) s príslušnou definíciou funkcie (uzol `FcDef`).

Obr. 7.1: Ukážka derivačného stromu, vygenerovaného z výrazu ‘$a + b$‘

7.3 Rekonštrukcia stavových diagramov

Implementácia sa snaží minimalizovať použitie komentárov, ktoré vkladajú nástroje generujúce kód. Tiež nevyužíva možnosť vkladania vlastných komentárov do stavových diagramov za účelom lepšej diskriminácie jednotlivých častí kódu. Je to hlavne z dôvodu, aby sa zabránilo modifikácii generovaného kódu. Ako bolo spomenuté v kapitole 2, jeden z cieľov hovorí o tom, že je potrebné, aby bol kód testovateľný bez toho, aby bolo potrebné vykonávať nejaké modifikácie.

Samotný algoritmus, ktorý realizuje rekonštrukciu zdrojového kódu do štruktúry komponentov stavového diagramu (ďalej len diagram z kódu) pracuje nasledovne: Ako vstup prijíma vytvorený AST v ktorom nájde uzol reprezentujúci modelovú krokovú (angl. step) funkciu. Ďalej pomocou získaných komentárov (jediný bod, v ktorom sú využité komentáre) lokalizuje všetky uzly (`If`), ktoré reprezentujú vstupné body pre stavové diagramy. Ďalej je z komentáru získaný názov stavového diagramu, ktorý je zhodný s názvom z modelu. To neskôr zjednoduší spárovanie diagramu z kódu a diagramu z modelu. Rekonštrukcia nájdených diagramov prebieha sekvenčne. Vytvorí sa pomocné mapovanie tak, že sa vo vstupnom uzle prehľadajú všetky priradenia a vyberú tie, ktoré sú v tvare
`active_state = state_constant` (ďalej len *priradenie prechodu*), kde `active_state` je premenná, ktorá určuje aktuálne aktívny stav v danej úrovni hierarchie stavov. Môže byť v tvare `is_{chart_alias}_{model_name}`, kde `chart_alias` je reťazec, ktorý je unikátny pre každý stavový diagram. Ak je `active_state` v tomto tvare, tak sa jedná o stavy, ktoré sú na najvyššej úrovni hierarchie, teda diagramu. Ďalší tvar je `is_{state_name}`,

kde `state_name` je názov nadradeného stavu. `state_constant` predstavuje konštantu definovanú pomocou makier a je spracovaná vo fáze pred-spracovania. Ako spomenuté vyššie má tvar `{model_name}__IN__{state_name}` a umožní získať názov aktivovaného stavu. Po vytvorení tohto mapovania je možné jednoznačne určiť všetky použité stavy a celú hierarchiu stavov. Pokračuje sa analýzou vstupného uzlu (`If`). Ako popísané v navrhnutej metóde (viď. obr. 6.2), overí sa aktivačná podmienka a aktivačné priradenie (tvar `{model_name}_DWork.is_active_{code_alias}_{chart_name} = 1;`) v kóde. Následne sa spracujú predvolené prechody (angl. default transitions), ktoré spracujú vstupný kód (list uzlov):

- Otestuj či je daný kód jednou z akcií, ktoré nadobúda stav. Ak áno, predaj riadenie metóde, ktorá rekonštruuje jednotlivé akcie stavov.
- Otestuj či je daný kód v tvare optimalizácie *Extrakcia duplicitných častí kódu*. Ak áno, predaj riadenie metóde, ktorá spracováva danú optimalizáciu.
- V kóde nájdi uzol vetvenia (`If` alebo `Switch`). Ak sa tam daný uzol nenachádza, skonči.
- Nech sa uzol vetvenia v kóde nachádza na indexe i, potom časť kódu na indexoch $< 0, i)$ je označená ako akcia podmienky.
- prejdi všetky vetvy nájdeného uzlu a každý pod-uzol rekurzívne zavolaj spracovanie podmienky.

Proces **spracovania akcií stavov** je značne komplikovanejší kvôli tomu, že sa nevyužívajú vložené komentáre do generovaného kódu. Daná metóda na vstup prijíma list uzlov AST, ktoré predstavujú kód. Najprv sa overí, či sa v kóde nachádza *priradenie prechodu*. Ak áno, kód je rozdelený na časť výstupnej (*exit*) akcie a vstupnej (*entry*) akcie ako bolo predstavené v navrhnutej metóde (viď. obr. 6.4). Kód výstupnej akcie sa pridá zdrojovému stavu danej cesty a ak sa v danom kóde nachádza uzol selekcie tak sa rekurzívne spracujú výstupné akcie vnorených stavov. Podobne sú spracované aj vstupné akcie. Príslušný kód vstupnej sa pridá cieľovému stavu a ak kód obsahuje uzol selekcie, tak je kód rozdelený a je zavolané rekurzívne spracovanie pre predvolené prechody vstupných stavov.

Spracovanie **extrakcia duplicitných častí kódu** pracuje tak, že pred započatím rekonštrukcie si vytvorí mapovanie všetkých uzlov, ktoré sa nachádzajú za uzlami, ktoré reprezentujú stavovú logiku. Ako ukázané na útržku kódu (4.11), extrahovaný kód je obalený uzlom `If`, kde je jeho vykonanie podmienené nastavením vygenerovaného príznaku. Mapovanie je vytvorené tak, že kľúčom je názov daného príznaku a hodnotou je extrahovaný kód.

De-optimalizácia je pri rekonštrukcii vykonaná nasledovne: vstupný kód sa overí, či obsahuje práve jeden uzol, je daný uzol priradenie konštanty `true` a názov premennej v priradení je validný kľúč spomínaného mapovania. Ak sú všetky podmienky splnené tak je riadenie predané späť spracovaniu prechodov s tým, že je pôvodný kód nahradený kódom z mapovania.

V prípade, že sa v danom kóde nenachádza *priradenie prechodu*, môže nastať niekoľko prípadov. V kóde sa vyhľadá uzol selekcie a overí sa jeho podmienka. Ak podmienka obsahuje test premennej, ktorá určuje aktívny stav tak to znamená, že sa jedná akciu počas (*during*) s vnorenou stavovou logikou. Ak podmienka neobsahuje test na danú premennú, jedná sa o vnorený prechod a je riadenie predané späť spracovaniu prechodov. Ak sa vyskytne situácia, že daný kód neobsahuje uzol selekcie tak nastane nejednoznačnosť. Kód

môže predstavovať akciu počas, alebo prechod, ktorý je cyklom (má rovnaký počiatočný aj koncový stav), nemá definovanú podmienku prechodu a má nastavenú akciu prechodu. Táto nejednoznačnosť sa rieši tak, že sa s oboma prípadmi zaobchádza, ako keby to bola akcia počas. Toto riešenie je možné použiť, lebo bez ohľadu na to, či je daný kus kódu klasifikovaný ako akcia prechodu alebo akcia stavu, tak je zachované poradie výpočtu.

Po spracovaní predvolených prechodov je spracovaný kód stavovej logiky. Ten sa nachádza v `else` vetve vstupného uzlu. Ako zmienené v navrhnutej metóde, podľa počtu stavov je reprezentovaná buď uzlom `If` alebo uzlom `Switch`. Každá vetva reprezentuje reprezentuje stav, jeho akciu počas (*during action*) a všetky jeho odchádzajúce prechody. Každý stav je identifikovaný na základe podmienky vykonania danej vetvy a využíva rovnaké symboly ako priradenie prechodu (`active_state == state_constant`). Všetky odchádzajúce prechody sú rekonštruované rovnakým algoritmom ako predvolené prechody.

Pri rekonštrukcii stavovej logiky treba dbať na to, že uzly selekcie sú generované tak, že uzly `If` vždy obsahujú vetvu `else` bez podmienky a uzly `Switch` obsahujú vetvu `default`. A teda je potrebné pre danú vetvu zistiť stav, ktorý reprezentuje. Zistenie prebieha nasledovne: nech A je množina všetkých stavov v danom kontexte (kontext predstavuje všetky stavy, ktoré majú spoločného predka v hierarchii stavov) a B je množina všetkých stavov získaných z vetiev uzla selekcie, ktoré majú zadefinovanú podmienku. Potom predvolená vetva odpovedá stavu, ktorý je daný rozdielom $A - B$ práve vtedy, keď daný je rozdiel jedno-prvková množina, inak rekonštrukcia končí chybou. Pre každý stav sa vykoná spracovanie prechodov.

7.4 Rekonštrukcia stavových diagramov z modelu

Rekonštrukcia stavových diagramov z modelu je priamočiara. Formát vstupných dát je už vo forme, ktorú popisuje štruktúra komponentov. Dáta sú preložené do štruktúry, ktorá zdieľa rozhranie s rekonštruovanými diagramami z kódu. Jediná vec, čo stojí za mienku je preklad reťazcov akcií do formy AST. Definícia symbolov premenných je získaná zo vstupných, výstupných a lokálnych signálov modelu. Všetky signály sú interpretované ako deklarácie premenných v AST, kde majú svoje meno, príslušný dátový typ a šírku signálu (odpovedá polu v jazyku C). Všetky tieto definície sú vložené do tabuľky symbolov, keď sa vykonáva sémantická analýza akcií modelu.

7.5 Rekonštrukcia pravdivostných tabuliek

Úloha rekonštrukcie pravdivostných tabuliek z kódu je proti rekonštrukcii stavových diagramov triviálna úloha. Na začiatku sa vyhľadajú všetky uzly funkčných volaní (`FcCall`), ktoré odpovedajú funkcii pravdivostnej tabuľke. Každý uzol funkčnej definície (`FcDef`), ktorý predstavuje pravdivostnú tabuľku je spracovaný nasledovne. Jej kód je rozdelený do troch častí: inicializácia, selekcia a vrátenie hodnoty. Posledná časť je voliteľná, lebo pravdivostné tabuľky môžu realizovať výpočet pomocou vedľajších efektov, kde menia pamäť signálov modelu. V rámci inicializácie je vytvorenie uzlu pre návratovú hodnotu, prípadne deklarácie dočasných premenných. Selekcia je realizovaná pomocou uzlov `If`, kde podmienka je daná agregáciou *podmienok* v danom rozhodnutí a telo uzlu je kód ktorý zodpovedá príslušnej akcii. Vrátenie hodnoty je tvorené jedným uzlom `Jump`, ktorý určuje návratovú hodnotu. Každá tabuľka je rekonštruovaná do listu štruktúr, kde je každá štruktúra definovaná uzlom podmienky, kódom akcie a množinou definovaných dočasných premenných.

7.6 Vytvorenie stromu prechodov a zoskupenie ich uzlov

Stavové diagramy sú preložené do štruktúry komponent a sú reprezentované objektami so spoločným rozhraním. To umožní definovať spoločný algoritmus, ktorý z danej štruktúry vytvorí strom prechodov rovnako pre diagram získaný z kódu, tak aj z modelu. Strom prechodov je vytvorený pomocou simulácie všetkých prechodov nad obomi štruktúrami komponentov. Algoritmus simulácie všetkých prechodov pracuje nasledovne: Ak existuje aspoň jedna inicializácia vstupného, výstupného alebo lokálneho signálu, vytvorí sa uzol strom prechodu (ďalej len uzol) typu inicializácia modelu. Za ním vždy nasleduje uzol aktivácie diagramu. Algoritmus postupuje podľa definovanej sémantiky výpočtového kroku stavového diagramu. Nájdu sa všetky predvolené prechody na úrovni hierarchie diagramu. Každý prechod je reprezentovaný jedným uzlom. V prípade, že prechody vytvárajú zložitejšie vetvenie, ktoré vzniká použitím spojovacích uzlov (*junctions*), tak sa dané vetvenie prejaví aj v strome prechodov. Samostatné spojovacie uzly nie sú v strome prechodov. Keď prechod skončí v stave, je pre daný stav vytvorený uzol ktorý reprezentuje vstupnú akciu daného stavu. Ak sa daný vstupný uzol už nachádza na danej ceste, tak v nej algoritmus ďalej nepokračuje. Inak sa pokračuje *navštívením* odchádzajúcich prechodov v danom poradí a za posledným prechodom nasleduje akcia počas. To znamená, že uzol typu vstupnej akcie má vždy $0 - N$ potomkov uzlov typu prechodu a jeden uzol typu akcie počas. Spracovanie pre odchádzajúce prechody prebieha rovnako ako pre predvolené prechody s rozdielom, že ak je dosiahnutý stav, tak je pred uzol vstupnej akcie vložený uzol výstupnej akcie stavu, z ktorého dané prechody odchádzajú. Ak stav, ktorému patrí uzol akcie počas obsahuje vnorené stavy, tak sú *navštívené* dané stavy navštívené a teda takýto uzol akcie počas obsahuje rovnakých potomkov ako uzol vstupnej akcie. To sú uzly odchádzajúcich prechodov a jeden uzol akcie počas vnoreného stavu. Algoritmus končí, keď každá cesta (list uzlov od ľubovoľného listového uzlu ku koreňu) obsahuje buď uzol vstupnej akcie každého stavu aspoň raz alebo jej listový uzol je typu akcie počas.

Stromy prechodov sú ďalej transformované tak, že jednotlivé uzly sú zoskupené do väčších celkov, kde jeden celok reprezentuje časť stavového diagramu, ktorá sa vykoná v rámci jedného výpočtového kroku. Rozdelenie nastáva vždy na uzle vstupnej akcie, ktorý má práve jeden uzol akcie počas ako priameho potomka. Uzlom prechodov je nastavená nová vlastnosť, ktorá indikuje či daná podmienka prechodu musí byť splnená alebo nesplnená, aby výpočet pokračoval danou vetvou.

Príklad: je daný strom prechodu s uzlom vstupnej akcie. Daný uzol má 4 potomkov (3 uzly prechodu a jeden uzol akcie počas). Nech je potomok, ktorý predstavuje uzol prechodu s poradím vykonania X označený U_{tX}. Po transformácii na strom prechodov so zoskupenými uzlami budú pred uzol U_{t3} pridané uzly U_{t1} a U_{t2} s nastavením príznaku, že ich podmienky prechodu musia byť nesplnené.

Kapitola 8

Testovanie

Najprv bude vykonané testovanie spätnej rekonštrukcie stavových diagramov na vlastnej a na dodanej sade modelov. Potom bude tiež otestované, že zvolenú reprezentáciu je možné použiť pre overenie funkčnej ekvivalencie.

8.1 Evaluácia rekonštrukcie stavových diagramov

Testovanie správnosti rekonštrukcie bolo testované na 2 testovacích sadách. Prvá testovacia sada bola vytvorená v rámci tejto práce a obsahuje 46 modelov. Z každého modelu je vygenerovaný kód s použitím 15 rôznych prostredí. Prostredie predstavuje určitú konfiguráciu nastaviteľných parametrov nástrojov na generovanie kódu, ktoré sú popísané v 4. To vo výsledku predstavuje viac ako 600 testovacích prípadov.

Jednotlivé prostredia sú inšpirované reálnymi prostrediami využívanými v praxi. Základné prostredie sa nazýva *baseline* a je to predvolené prostredie pre generovanie a neobsahuje žiadne optimalizácie okrem *buffer-reuse*. Ostatné prostredia sú rozdelené do 3 skupín. Každá skupina odpovedá nastaveniu parametra *I/O storage class*, po rade *Auto*, *ImportedExternPointer* a *ImportedExtern*. V každej skupine sa nachádzajú 4 rôzne konfigurácie, ktoré sú popísané v tabuľke 8.1. Ďalej boli použité ďalšie dve prostredia (`env31` a `env32`), ktoré nepoužívajú optimalizáciu *Buffer reuse*.

	Storage class	**Inline params.**	**Loop rolling**	**Log. operator**	**Zero init**	**Buffer reuse**
envx1	X	nie	áno	logický	áno	áno
envx2	X	nie	nie	bitový	nie	áno
envx3	X	áno	áno	logický	nie	áno
envx4	X	áno	nie	bitový	áno	áno
env31	Auto	nie	áno	logický	nie	nie
env32	Auto	áno	áno	logický	nie	nie

Tabuľka 8.1: použité optimalizačné parametre v rámci prostredia

Testovanie vo vytvorenej sade bolo vykonané nasledovne: Pre každý testovací prípad boli zrekonštruované stavové diagramy z kódu do štruktúry komponent, a následne boli z nich vytvorené stromy prechodov. Pre každú štruktúru bola ďalej vytvorená vizualizácia, na základe ktorej bola vykonaná ručná kontrola či rekonštruované štruktúry odpovedajú zdrojovému kódu. Rekonštrukcia sa podarila na 100%. V tabuľke 8.2 je možno vidieť výsledky rekonštrukcie do štruktúry komponentov.

Stĺpec **Model** značí názov modelu, **Konfigurácia parametrov** hovorí o tom, pomocou akého prostredia bol model generovaný a ostatné stĺpce (**Počet diagramov**, **Počet stavov**, **Počet spojovacích uzlov** a **Počet prechodov**) udávajú počet rekonštruovaných komponent v tvare: {počet zrekonštruovaných z modelu}({počet zrekonštruovaných z kódu})/{počet v originálnom modeli}.

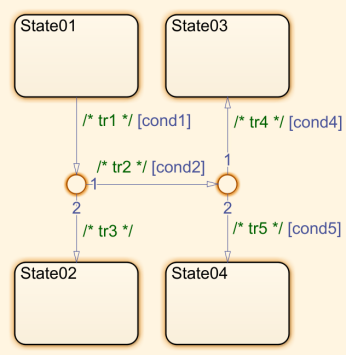
Výsledky (popísané v tabuľke 8.2) boli agregované tak, že ak boli počty rekonštruovaných komponentov jedného modelu pre rozličné prostredia generovania rovnaké, tak boli ich výsledky zlúčené do jedného riadku. Ak rozdielne konfigurácie nijak neovplyvnili počet komponentov, tak pre daný model je v stĺpci **Konfigurácia parametrov** hodnota *All configurations*. V opačnom prípade stĺpec obsahuje zoznam všetkých konfigurácií, ktorých sa dané hodnoty týkajú.

Ešte treba zmieniť, že základné verzie modelov vo vlastnej sade boli vytvárané tak, že vstupy a výstupy stavového diagramu sú priamo pripojené na vstupné a výstupné porty modelu. Ďalej sú v sade modely, ktorých meno je zakončené reťazcom *extra input*. Každý takýto model obsahuje niekoľko *Simulink* blokov za účelom vytvorenia zaujímavých propagácií.

Z výsledkov je možné vidieť, že počty komponentov vyššie popísaných *základných* modelov ostanú nezmenené nehľadiac na použité prostredia pri generovaní. Toto pozorovanie je aj intuitívne, lebo ako popísané v kapitole 4, dostupné optimalizačné parametre priamo neovplyvňujú stavové diagramy, ale *Simulink* bloky, ktoré sú s nimi prepojené. Často sú tieto optimalizácie propagované do štruktúry stavových diagramov. Ako vidieť na niektorých výsledkoch z modelov s pridanými *Simulink* blokmi (majú v názve *extra input*), tak pri použití optimalizácie napr. **Parameter inlining** sú počty rekonštruovaných komponentov z kódu ovplyvnené.

Taktiež je u niektorých *základných* modeloch vidieť, že počet zrekonštruovaných komponentov z modelu a z kódu sa líši. Toto nie je chyba, ako sa mohlo na prvý pohľad zdať. Po manuálnej revízii bolo zistené, že dané počty odpovedajú realite. Tieto rozdiely sú zapríčinené spôsobom návrhu daného modelu, kde niektoré prípady spôsobia, že vo vygenerovanom kóde menej či dokonca viac komponentov (viď. výsledky modelu *SingleChildState*, kde sú v kóde zlúčené vnorené stavy, alebo *SwitchTransition*, kde sú pridané spojovacie uzly a niektoré prechode čisto z estetického hľadiska, atď.).

Zaujímavý je prípad, pri ktorom vo vygenerovanom kóde vznikne viac prechodov ako v modeli. Nech je daný jednoduchý model (viď. obr 8.1), spojovacie uzly sú zľava označené *uzol1* a *uzol2*. Nastane situácia takzvaného *backtrackingu* v prípade, že sú splnené podmienky *cond1* a *cond2* a nesplnené podmienky *cond3* a *cond4* tak sa riadenie vráti z *uzlu2* do *uzlu1* a vykoná sa prechod s vyšším číslom poradia, v tomto prípade *tr3*. V kóde je táto konštrukcia potom reprezentovaná tak, že z *uzlu2* vedie priamo prechod bez podmienky do *State02* a tým vznikne prechod naviac oproti modelu. Popísanú konštrukciu obsahuje model *JunctionChart*.



Obr. 8.1: Konštrukcia stavového diagramu, ktorá vedie na vytvorenie viac prechodov v kóde, ako je v modeli definované

Model	Konfigurácia parametrov	Počet diagramov	Počet stavov	Počet spojovacích uzlov	Počet prechodov
AdvancedTruthTables extra input	All configurations	2(2)/2	5(5)/5	0(0)/0	8(8)/8
AdvancedTruthTables	All configurations	2(2)/2	5(5)/5	0(0)/0	8(8)/8
BasicStateflowWithLocals extra input	All configurations	1(1)/1	10(10)/10	0(0)/0	14(14)/14
BasicStateflowWithLocals	All configurations	1(1)/1	10(10)/10	0(0)/0	14(14)/14
BasicStateflow extra input	All configurations	1(1)/1	10(10)/10	0(0)/0	14(14)/14
BasicStateflow	All configurations	1(1)/1	10(10)/10	0(0)/0	14(14)/14
ExampleModel	baseline	1(1)/1	5(5)/5	1(1)/1	10(10)/10
JunctionChart extra input	baseline, ENV01, ENV02, ENV11, ENV12, ENV21, ENV22, ENV31	1(1)/1	8(8)/8	5(4)/5	24(25)/24
JunctionChart extra input	ENV03, ENV04, ENV13, ENV14, ENV23, ENV24, ENV32	1(1)/1	8(8)/8	5(4)/5	24(24)/24
JunctionChart	All configurations	1(1)/1	8(8)/8	5(4)/5	24(25)/24
LargeStateflow extra input	baseline, ENV01, ENV11, ENV21, ENV31	1(1)/1	22(21)/22	8(7)/8	48(49)/48
LargeStateflow extra input	ENV02, ENV03, ENV04, ENV12, ENV13, ENV14, ENV22, ENV23, ENV24, ENV32	1(1)/1	22(21)/22	8(9)/8	48(58)/48
LargeStateflow	baseline, ENV01, ENV11, ENV21, ENV31	1(1)/1	22(21)/22	8(7)/8	48(49)/48
LargeStateflow	ENV02, ENV03, ENV04, ENV12, ENV13, ENV14, ENV22, ENV23, ENV24, ENV32	1(1)/1	22(21)/22	8(9)/8	48(58)/48
LargeTruthTable extra input	All configurations	1(1)/1	10(10)/10	1(3)/1	19(42)/19
LargeTruthTable	All configurations	1(1)/1	10(10)/10	1(3)/1	19(42)/19
MixedChartModel extra input	All configurations	1(1)/1	2(0)/2	0(0)/0	2(2)/2
MixedChartModel	All configurations	1(1)/1	2(0)/2	0(0)/0	2(0)/2
MultiLevelNesting extra input	baseline, ENV01, ENV02, ENV03, ENV04, ENV11, ENV12, ENV13, ENV14, ENV21, ENV22, ENV23, ENV24,	1(1)/1	9(9)/9	0(0)/0	14(14)/14
MultiLevelNesting extra input	ENV31 ENV32	1(1)/1	9(9)/9	0(0)/0	14(10)/14
MultiLevelNesting	All configurations	1(1)/1	9(9)/9	0(0)/0	14(14)/14
MultiStateflows extra input	All configurations	3(3)/3	14(14)/14	0(0)/0	20(20)/20
MultiStateflows	All configurations	3(3)/3	14(14)/14	0(0)/0	20(20)/20
OverSpecifiedBranching	All configurations	1(1)/1	3(3)/3	1(1)/1	7(7)/7
OverSpecifiedCondition	All configurations	1(1)/1	2(2)/2	0(0)/0	3(3)/3
SimpleChart02	All configurations	1(1)/1	3(3)/3	0(0)/0	5(5)/5
SimpleHierarchySf extra input	All configurations	1(1)/1	4(4)/4	0(0)/0	5(5)/5
SimpleHierarchySf	All configurations	1(1)/1	4(4)/4	0(0)/0	5(5)/5
SingleChildState extra input	All configurations	1(1)/1	4(2)/4	0(0)/0	6(4)/6
SingleChildState	All configurations	1(1)/1	4(2)/4	0(0)/0	6(4)/6
SmallChartWithOptimization	All configurations	1(1)/1	2(2)/2	0(0)/0	3(3)/3
StateflowAdvancedHierarchy extra input	baseline, ENV01, ENV02, ENV11, ENV12, ENV21, ENV22, ENV31	1(1)/1	9(9)/9	0(0)/0	14(14)/14
StateflowAdvancedHierarchy extra input	ENV03, ENV04, ENV13, ENV14, ENV23, ENV24, ENV32	1(1)/1	9(7)/9	0(0)/0	14(9)/14
StateflowAdvancedHierarchy	All configurations	1(1)/1	9(9)/9	0(0)/0	14(14)/14
StateflowSingleState extra input	All configurations	2(2)/2	2(0)/2	1(0)/1	6(0)/6
StateflowSingleState	All configurations	2(2)/2	2(0)/2	1(0)/1	6(0)/6
StateflowWithSimpleHierarchy extra input	All configurations	1(1)/1	4(4)/4	0(0)/0	5(5)/5
StateflowWithSimpleHierarchy	All configurations	1(1)/1	4(4)/4	0(0)/0	5(5)/5
StateflowWithUnaryOperation	All configurations	1(1)/1	2(2)/2	0(0)/0	3(3)/3
SwitchTransition extra input	All configurations	1(1)/1	5(5)/5	6(0)/6	15(9)/15
SwitchTransition	All configurations	1(1)/1	5(5)/5	6(0)/6	15(9)/15
TruthTable01	All configurations	1(1)/1	2(2)/2	0(0)/0	3(3)/3
TruthTable02	All configurations	1(1)/1	2(2)/2	0(0)/0	3(3)/3
TruthTable03	All configurations	1(1)/1	2(2)/2	0(0)/0	3(3)/3
TruthTable04	All configurations	1(1)/1	2(2)/2	0(0)/0	3(3)/3
TruthTable05	All configurations	1(1)/1	2(2)/2	0(0)/0	3(3)/3
TruthTable06	All configurations	1(1)/1	2(2)/2	0(0)/0	3(3)/3
TruthTable07	All configurations	1(1)/1	2(2)/2	0(0)/0	3(3)/3
TruthTableCalls extra input	All configurations	1(1)/1	2(2)/2	1(1)/1	5(5)/5
TruthTableCalls	All configurations	1(1)/1	2(2)/2	1(1)/1	5(5)/5

Tabuľka 8.2: Výsledky rekonštrukcie stavových diagramov na vlastnej sade modelov

Ďalej boli testy vykonané na sade modelov dodaných firmou Honeywell. Tieto modely boli vybrané z produkčného prostredia s cieľom maximalizovať pokrytie jednotlivých používaných konštrukcií rovnako ako veľkosť vytváraných stavových diagramov. Bolo vytvorených približne 4000 testovacích prípadov z modelov, ktoré obsahujú stavové diagramy a bola vytvorená tabuľka výsledkov ôsmich z nich viď. tabuľka 8.3. Názvy modelov sú nahradené za `HONxx`, aby nevznikol žiaden únik informácií. Na prvý pohľad je hneď zrejmé, že reálne používané stavové diagramy sú väčšie z pohľadu počtu použitých komponentov. Tiež je ďalej vidieť, že väčšina ilustrovaných diagramov má správanie ako stavové diagramy z vytvorenej sady, ktoré sú označené *extra input*. Je to tým, že v praxi sú modely zložené hlavne zo *Simulink* blokov. A pri použitých prostrediach generovania dochádza ku zlučovania týchto systémov.

Model	**Konfigurácia parametrov**	**Počet diagramov**	**Počet stavov**	**Počet spojovacích uzlov**	**Počet prechodov**
HON01	baseline, ENV01, ENV02, ENV11, ENV12, ENV21, ENV22, ENV31	3(3)/3	43(42)/43	7(8)/7	72(76)/72
HON01	ENV03, ENV04, ENV13, ENV14, ENV23, ENV24, ENV32	3(3)/3	43(37)/43	7(6)/7	72(65)/72
HON02	All configurations	1(1)/1	12(12)/12	2(2)/2	17(17)/17
HON03	baseline, ENV01, ENV11, ENV21, ENV31	1(1)/1	122(122)/122	6(6)/6	140(142)/140
HON03	ENV02, ENV03, ENV04, ENV12, ENV13, ENV14, ENV22, ENV23, ENV24, ENV32	1(1)/1	122(118)/122	6(6)/6	140(137)/140
HON04	baseline, ENV01, ENV11, ENV21, ENV31	1(1)/1	24(23)/24	8(7)/8	51(50)/51
HON04	ENV02, ENV03, ENV04, ENV12, ENV13, ENV14, ENV22, ENV23, ENV24, ENV32	1(1)/1	24(23)/24	8(9)/8	51(58)/51
HON05	All configurations	3(3)/3	31(28)/31	0(0)/0	34(31)/34
HON06	baseline, ENV01, ENV02, ENV03, ENV04, ENV11, ENV12, ENV13, ENV14, ENV21, ENV22, ENV23, ENV24,	1(1)/1	24(24)/24	2(0)/2	34(25)/34
HON06	ENV31 ENV32	1(1)/1	24(24)/24	2(0)/2	34(21)/34
HON07	baseline, ENV01, ENV02, ENV11, ENV12, ENV21, ENV22, ENV31	1(1)/1	10(10)/10	0(0)/0	16(16)/16
HON07	ENV03, ENV04, ENV13, ENV14, ENV23, ENV24, ENV32	1(1)/1	10(9)/10	0(0)/0	16(14)/16
HON08	baseline, ENV01, ENV02, ENV11, ENV12, ENV21, ENV22, ENV31	2(2)/2	31(31)/31	0(0)/0	39(39)/39
HON08	ENV03, ENV04, ENV13, ENV14, ENV23, ENV24, ENV32	2(2)/2	31(30)/31	0(0)/0	39(36)/39

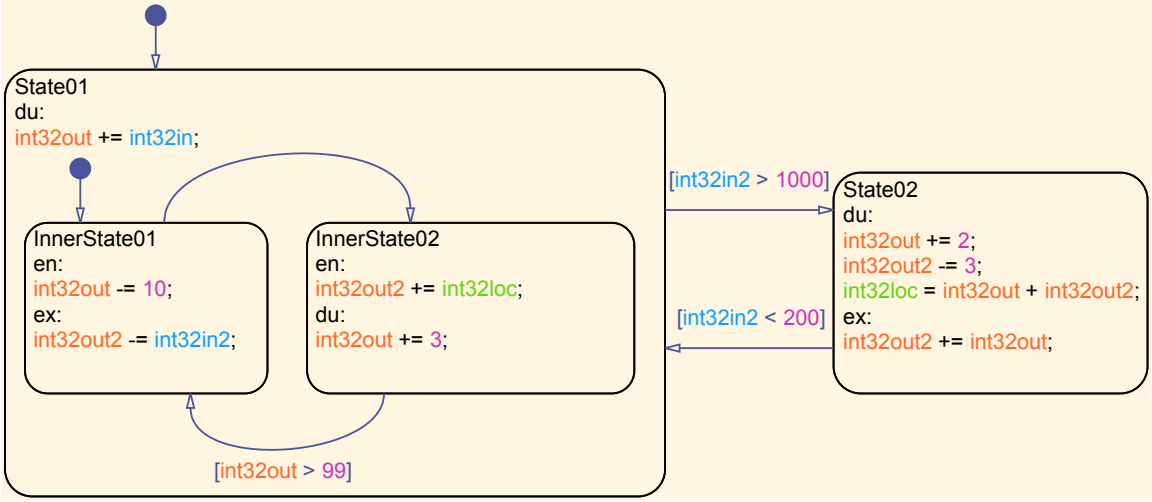
Tabuľka 8.3: Vybrané výsledky rekonštrukcie stavových diagramov zo sady modelov poskytnutej firmou Honeywell

8.2 Evaluácia porovnania na funkčnú zhodu

Táto práca sa síce nezaoberá priamo porovnaním kódu a modelu vzhľadom na funkčnú ekvivalenciu, ale pre ilustráciu a dôkaz, že zvolené štruktúry sú ďalej použiteľné pre túto úlohu je demonštrované na jednoduchom príklade. Je vytvorený jednoduchý model s 4 stavmi, jednou úrovňou hierarchie a bez použitia spojovacích uzlov (viď. obr. 8.2). Stavový diagram je priamo pripojený na vstupné a výstupné porty modelu. Kód v rámci akcií stavového diagramu je tiež zámerné navrhnutý tak, aby pri generovaní C kódu neboli vykonané žiadne optimalizácie. Následne je vygenerovaný kód pomocou prostredia *baseline*. Model a vygenerovaný kód sú zrekonštruované do podoby štruktúry komponentov pomocou vytvoreného nástroja. Tá je ďalej v niekoľkých krokoch transformovaná na stromy prechodu rozdelené do výpočtových krokov pre kód (viď obr. 8.4) a model (viď obr. 8.5). Už len pri pohľade na vizualizácie oboch stromov je zjavné, že až na niekoľko málo detailov sú stromy zhodné. Rozdiely medzi uzlami stromov sú názvy symbolov premenných a v kódovom strome sú vo výstupných akciách vnorených stavov explicitné de-aktivácie v prípade, že je vykonaný odchádzajúci prechod z nadradeného stavu. Pri porovnaní je otestovaná existencia tohto výrazu a je ďalej ignorovaný. Názvy premenných sú mapované tak, že ak sa názvy nezhodujú názov daného symbolu premennej v modelovom diagrame je nahradený názvom *Simulink* blokom, ku ktorému je daný vstup/výstup priamo pripojený. V tomto príklade to môžu byť len vstupné alebo výstupné porty modelu. Po aplikovaní týchto zmien sú stromy zhodné a je možné aplikovať naivný algoritmus, ktorý prechod *zhora-dolu* oboch stromov súčasne a vykoná porovnanie všetkých ich prvkov na zhodu.

Popísaný algoritmus je spustený na stavový diagram (viď. obr. 8.2) a jeho vygenerovaný kód prehlási, že sú oba stromy zhodné, čo je vzhľadom na to ako stromy vyzerajú zrejmé. Ďalej je spravený jednoduchý negatívny test tak, že sa zavedie malá zmena do stavové diagramu či vygenerovaného kódu a testuje sa, či algoritmus dokáže tieto zmeny odhaliť.

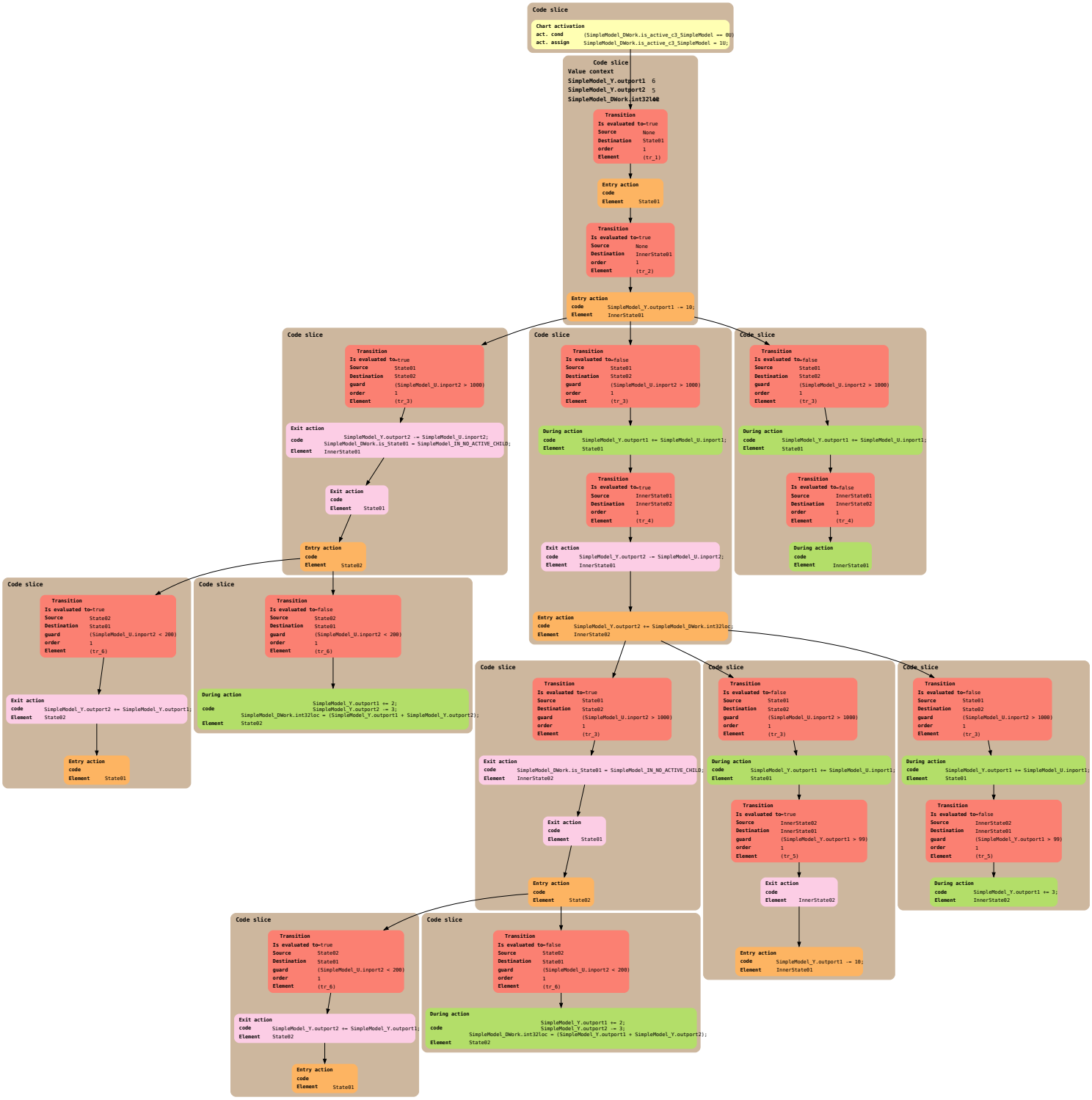
V danom stavovom diagrame je v stave *InnerState02* zmenená akcia počas (*during action*) z `in32out += 3;` na `int32out += 10;`. Následne sa spustí porovnanie a vytvorí výstup viď obr. 8.3. Z výstupu vidieť, že algoritmus správne klasifikoval nezhodný kód a model a tiež správne určil miesto nezhody. Tento algoritmus porovnania je súčasťou vytvoreného nástroja a je možné dané experimenty otestovať.



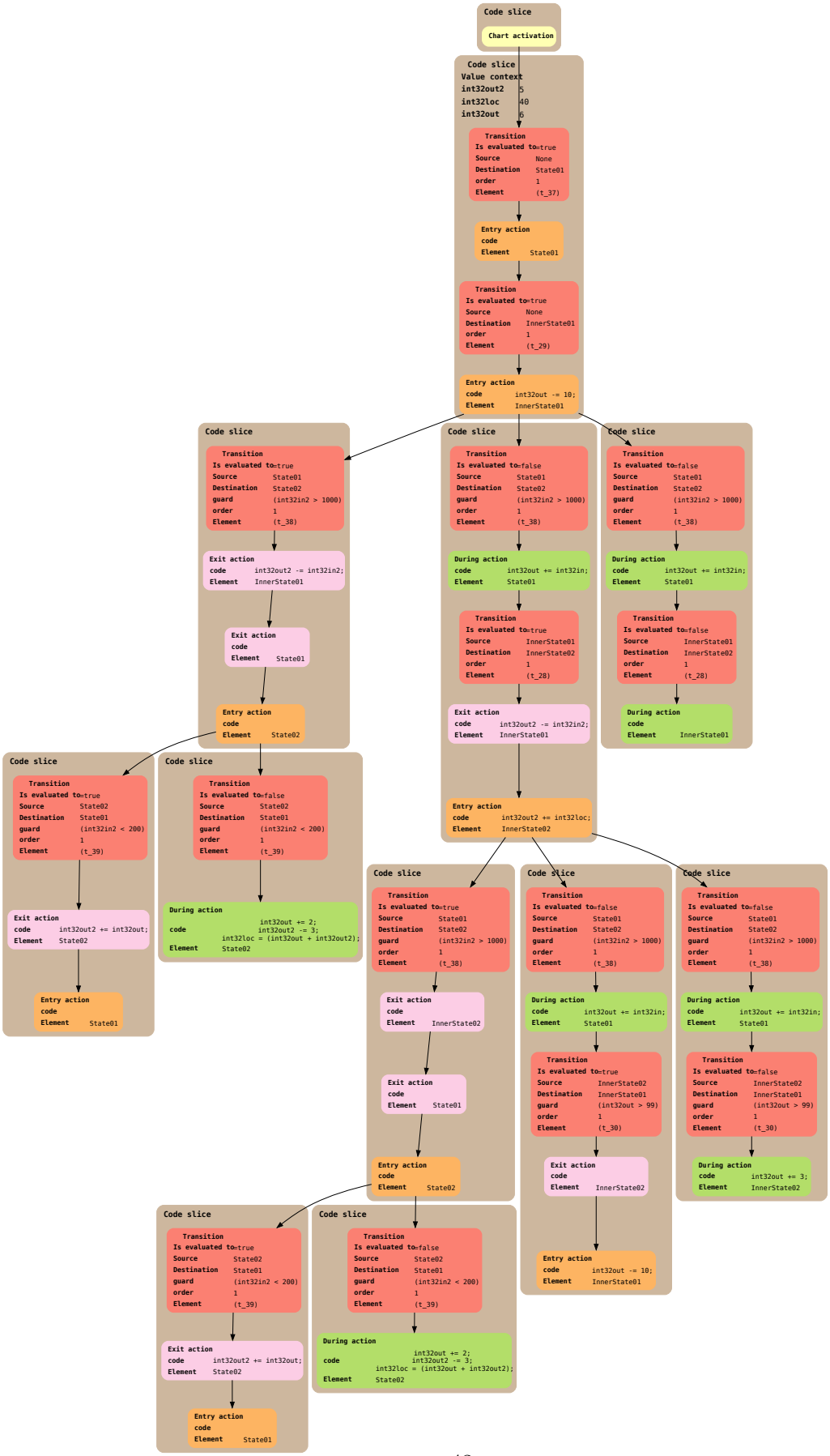
Obr. 8.2: Stavový diagram použitý pre porovnanie na funkčnú ekvivalenciu

```
Code and model are not equivalent
Mismatch occurred in element: Code[InnerState02] Model[InnerState02]
    caused by:
Action - During action
Mismatched occurred in action Code[SimpleModel_Y.outport1 += 3;] Model[int32out += 10;]
    caused by:
Constant mismatch: Code[3] Model[10]
```

Obr. 8.3: Výstup nástroja pri porovnaní nezhodného kódu a modelu



Obr. 8.4: Ukážka stromu prechodov vytvoreného zo stavového diagramu z obr. 8.2



Obr. 8.5: Ukážka stromu prechodov vytvoreného zo stavového diagramu z obr. 8.2

Kapitola 9

Záver

Cieľom práce bolo vytvorenie nástroja, ktorý na vstupe príjme model v prostredí *Simulink*, ktorý obsahuje *Stateflow* stavový diagram, z modelu vygenerovaný kód a zrekonštruuje ich do zvolenej štruktúry, ktorá bude vhodná na preukázanie funkčnej ekvivalencie medzi modelom a daným kódom. Nástroj bol navrhnutý a implementovaný v jazyku Java21, je spustiteľný z príkazového riadku a umožňuje niekoľko výstupov: štruktúru komponentov diagramov, stromy prechodov a stromy prechodov zoskupené do jedného výpočtového kroku z modelu aj z kódu vo formáte *json*. Ďalej umožňuje vytvorenie vizualizácie rekonštruovaného stavového diagramu z kódu, 3 typy stromov (strom prechodov, zoskupený strom prechodov a jednoduchý strom prechodov) vo formáte vektorových obrázkov *svg*. Nástroj bol testovaný na vytvorenej vlastnej sade modelov, ktorá obsahuje viac ako 600 testovacích prípadov a na sade reálnych modelov využívaných v praxi, ktoré boli poskytnuté firmou Honeywell. Táto sada obsahuje približne 4000 testovacích prípadov. Ďalej bol ukázaný demonštračný príklad, že určené výsledné štruktúry sú použiteľné pre porovnanie na funkčnú ekvivalenciu.

Nástroj dokáže bez problémov analyzovať a rekonštruovať kód a modely nehľadiac na použité optimalizačné parametre generovania, ktoré predpisuje zadanie. Taktiež správne pracuje pri použití zložitejších konštrukcií ako je vetvenie pomocou spojovacích uzlov, zanorenie stavov, či akcie na prechodoch.

Zadanie práce je špecifické a jedná sa o riešenie konkrétneho problému vo firme Honeywell. Na integrácii vytvoreného riešenia do existujúceho systému sa pracuje. Avšak na to, aby mohla byť táto práca použitá v *safety-critical* prostredí je najprv potrebné, aby bol výsledný nástroj riadne kvalifikovaný. Túto kvalifikáciu vykonajú pracovníci v danej firme.

Aj ked bolo zadanie špecifické pre firmu Honeywell, pri návrhu a implementácii nástroja neboli využité žiadne interné zdroje firmy Honeywell. Preto je možné prácu ďalej využiť ako predlohu pre implementáciu vlastného nástroja, ktorý bude robiť porovnanie funkčnej ekvivalencie medzi kódom a modelom. Tiež je možné rozšíriť sadu povolených komponentov *Stateflow*, či pridať podporované optimalizačné parametre, ktoré ponúka nástroj *Embedded Coder* a *Real-Time Workshop*.

Literatúra

[1] Group, O. M. *Systems Modeling Language (SysML) Specification*. 2.0-beta-2. Object Management Group (OMG), apríl 2024. Dostupné z: https://www.omg.org/spec/SysML.

[2] International Organization for Standardization. *ISO/IEC Standard 9899:201X(E): Programming Languages - C*. Standard 9899. International Organization for Standardization, 2011. 159 - 177 s.

[3] Parr, T. *C11 Grammar for ANTLR4* [https://github.com/antlr/grammars-v4/blob/master/c/C.g4]. [cit. 2024-05-12].

[4] Parr, T. *The Definitive ANTLR 4 Reference*. Dallas, TX: Pragmatic Bookshelf, 2014. ISBN 978-1-934356-99-9.

[5] Pnueli, A., Siegel, M. a Singerman, E. Translation validation. In: Steffen, B., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998. ISBN 978-3-540-69753-4.

[6] RTCA, I. Model-Based Development and Verification Supplement to DO-178C and DO-278A. In: *RTCA/DO-331*. Radio Technical Commission for Aeronautics, 2011 [cit. 2024-01-20]. Dostupné z: https://www.rtca.org/.

[7] RTCA, I. Model-Based Development and Verification Supplement to DO-178C and DO-278A. In: *RTCA/DO-331*. Radio Technical Commission for Aeronautics, 2011, s. Table MB.A–5 [cit. 2024-01-20]. Dostupné z: https://www.rtca.org/.

[8] RTCA, I. Software Considerations in Airborne Systems and Equipment Certification. In: *RTCA/DO-178C*. Radio Technical Commission for Aeronautics, 2011 [cit. 2024-01-20]. Dostupné z: https://www.rtca.org/.

[9] RTCA, I. Software Considerations in Airborne Systems and Equipment Certification. In: *RTCA/DO-178C*. Radio Technical Commission for Aeronautics, 2011, s. Table A–5 [cit. 2024-01-20]. Dostupné z: https://www.rtca.org/.

[10] Thain, D. *Introduction to Compilers and Language Design*. 2. vyd. 2023. ISBN 979-8-655-18026-0. Dostupné z: http://compilerbook.org.

[11] The MathWorks, I. *Simulink® 's Code Inspector*. 1 Apple Hill Drive, Natick, MA 01760-2098: The MathWorks, Inc., 1997–2023 [cit. 2024-01-20]. Dostupné z: https://www.mathworks.com/help/pdf_doc/slci/slci_ug.pdf.

[12] The MathWorks, I. *Stateflow® User's Guide*. 1 Apple Hill Drive, Natick, MA 01760-2098: The MathWorks, Inc., 1997–2023 [cit. 2024-01-20]. Dostupné z: https://www.mathworks.com/help/pdf_doc/stateflow/stateflow_ug.pdf.

[13] The MathWorks, Inc.. *Embedded Coder® User's Guide*. The MathWorks, Inc., 2011–2024. Natick, MA: The MathWorks, Inc. Dostupné z: https://www.mathworks.com/help/pdf_doc/ecoder/ecoder_ug.pdf.