
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**
**BRNO UNIVERSITY OF TECHNOLOGY**

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**
**FACULTY OF INFORMATION TECHNOLOGY**

**ÚSTAV INFORMAČNÍCH SYSTÉMŮ**
**DEPARTMENT OF INFORMATION SYSTEMS**


# STATICKÁ ANALÝZA A VYHODNOCENÍ V JAZYKU C

**STATIC ANALYSIS AND EVALUATION IN C LANGUAGE**

**BAKALÁŘSKÁ PRÁCE**
**BACHELOR'S THESIS**


**AUTOR PRÁCE** — **VLADIMÍR UŽÍK**
**AUTHOR**

**VEDOUCÍ PRÁCE** — **doc. Dr. Ing DUŠAN KOLÁŘ,**
**SUPERVISOR**



**BRNO 2020**

**Vysoké učení technické v Brně**
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2019/2020

# Zadání bakalářské práce


Student: **Užík Vladimír**
Program: Informační technologie
Název: **Statická analýza a vyhodnocení v jazyku C**
**Static Analysis and Evaluation in C Language**
Kategorie: Překladače

Zadání:

1. Nastudujte problematiku statické analýzy zdrojových kódů jazyka C. Seznamte se s existujícími nástroji a knihovnami.
2. Navrhněte aplikaci, která ze zdrojového kódu získá stav periferních registrů po provedení vybraných konfiguračních funkcí pro inicializaci pinů a hodin, které jsou volány při inicializaci aplikace na mikrokontroleru. Uvažujte i volání vnořených funkcí.
3. Navrženou aplikaci implementujte - jako vstup uvažte SDK balíček obsahující kód, který má být analyzován. Parametry aplikace se předají pomocí příkazové řádky.
4. Popište podmnožinu jazyka C, která je aplikací podporována i s případnými omezeními.
5. Otestujte aplikaci na sadě testovacích příkladů a porovnejte získané výstupy s hodnotami registrů získaných při spuštění na reálných mikrokontrolerech.
6. Zhodnoťte přínos své práce, diskutujte možná rozšíření a případné nedostatky.

Literatura:

- Aho, Sethi, Ullman - Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986. ISBN 0-201-10088-6
- Další dle pokynů vedoucího.

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

Vedoucí práce: **Kolář Dušan, doc. Dr. Ing.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 28. května 2020
Datum schválení: 21. října 2019

# Abstrakt

Cieľom práce je tvorba nástroja na analýzu zdrojového kódu v jazyku C za účelom zistenia hodnôt registrov perférnych zariadení. Výsledkom je nástroj regcheck ktorý je schopný určiť konečný stav týchto registrov ako aj kroky k nemu vedúce. Nástroj je otestovaný na 2 mikrokontroléroch.



# Abstract

Aim of this thesis is to produce a tool which performs static analysis on C language source code. Result is application regcheck which determines final state of registers used by peripherals and steps leading to that state. Application was tested on two microcontrollers.



# Kľúčové slová

statická analýza, jazyk C, mikrokontrolér, MCU, NXP

# Keywords

static analysis, C language, microcontroller, MCU, NXP


# Citácia

UŽÍK, Vladimír. *Statická analýza a vyhodnocení v jazyku C*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Dr. Ing Dušan Kolář,

# Statická analýza a vyhodnocení v jazyku C

## Prehlásenie

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doc. Dr. Ing Dušana Koláře. Další informace mi poskytol kozultant za firmu NXP, pán Petr Hradský. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Vladimír Užík
28. mája 2020


## Poďakovanie

Chcel by som poďakovať pánovi docentovi Kolářovi za možnosť pracovať na tejto téme ako aj jeho poznatky k jej spracovaniu. Ďalej by som chcel poďakovať firme NXP za zapožičanie fyzických mikrokontrolérov použitých k testovaniu, ďalej pánu Petrovi Hradskému za rýchlu a bezproblémovú komunikáciu a pánu Petrovi Dohnalovi za trpedzlivosť pri testovaní nástroja v reálnom nasadení.

# Obsah

# Kapitola 1

# Úvod

Vývojár je len človek a ako taký bude vždy zdrojom takzvaných „ľudských“ chýb. Pre predstavu, zástupcami tohto druhu môžu byť nepozornosť, nesprávna interpretácia dokumentácie, alebo prehliadnutie súvislostí, ktoré nemusia byť na prvý pohľad jasné. Neoceniteľnou pomocou v tomto smere sú rôzne statické analyzátory, ktoré nám dokážu napovedať, či sa spoliehame na nedefinované chovanie, alebo meníme logiku funkcie natoľko, že nie je naďalej schopná spĺňať svoju zodpovednosť voči zvyšku systému. Často podobné chyby môžu byť odchytené ešte vo fázi vývoja, kedy je ich oprava rýchlejšia a tým pádom aj lacnejšia – aj vďaka týmto nástrojom sú vývojári schopný dodávať o niečo bezpečnejší kód o niečo efektívnejšie.

Regcheck, nástroj tvorený v rámci tejto bakalárskej práce, je jedným z kategórie statických analyzátorov popísaných v predošlom odseku. Konkrétne jeho zameraním je analýza zdrojového kódu určeného pre mikrokontroléry vyvíjané v rámci firmy NXP Semiconductors. Počas analýzy určuje, aké hodnoty budú uložené na špecifických adresách – to ide využiť na kontrolu či jednotlivé periférie boli správne inicializované, prípadne či zmena v zdrojovom kóde nemá nechcené vedľajšie účinky.

Kapitola 2 sa venuje statickým analyzátorom ako takým a porovnáva niekoľko analyzátorov zameraných na analýzu zdrojového kódu v jazyku C. Ďalej v kapitole 3 je popísaný medzijazyk – často použitý na modelovanie zdrojového kódu, pričom až tento model je predmetom analýzy. Kapitola 4 pojednáva o nástrojoch potrebných k práci s mikrokontrolérmi pre ktoré je nástroj určený. V kapitole 5 je priblížený obecný návrh architektúry nástroja regcheck ako aj alternatívy, ktoré pri návrhu boli zvažované. Konkrétna implementácia návrhu je opísaná v kapitole 6 vrátane sekcie o použitej technológii a informáciach o jednotlivých moduloch z ktorých sa regcheck skladá. Kapitola 7 sa venuje grafickému rozhraniu regcheck-gui, ktoré robí prácu s nástrojom regcheck intuitívnejšiu. Popis testovania a výkonnosti nástroja je možno nájsť v kapitole 8. Posledná kapitola, kapitola 9 je záver – táto kapitola obsahuje zhrnutie práce ako takej ako aj smer ktorým sa nástroj regcheck môže uberať v budúcnosti.

# Kapitola 2

# Statická analýza

Táto sekcia zodpovedá otázky, čo je statická analýza a aké vlastnosti zdrojového kódu dokáže popísať. Taktiež na konci sekcie je porovnanie 10 vybraných nástrojov. Sekcia čerpá primárne z [4].

Za nástroj statickej analýzy je možné považovať všetko, čo analyzuje zdrojový kód bez jeho spustenia. To, či beh programu je nutnosť pre správnu funkciu nástroja, je totiž jedna z najdôležitejších vlastností, ktorá ho delí od dynamickej analýzy (akou je napríklad detekcia chybnej práce s pamäťou ponúkaná nástrojom *valgrind*[1]). Chyby práce s pamäťou, detekcia degradácie výkonu, ale aj napríklad generovanie testov z behu programu kde sú hodnoty premenných nahradené symbolmi (angl. symbolic execution) sú prístupy analýzy ktoré nejdú dosiahnuť staticky pre netriviálne prípady. Preto je vhodné nástroje statickej a dynamickej analýzy používať spolu.

Statická analýza je jedna z foriem verifikácie zdrojového kódu. A ako taká má svoje limity a je označená ako nerozhodnuteľný problém. Cieľom však nie je kompletná analýza – aj čiastočná analýza je schopná odhaliť radu chýb a prispieť k zlepšeniu kvality vyvíjaného produktu. Druhá, pre užívateľa nevýhodná, strana kompromisu čiastočnej analýzy pozostáva z existencie falošných poplachov (ďalej *false positives*) a nezachytenia skutočných nálezov (*false negatives*).

- *false positives* sú nálezy ktoré boli chybne identifikované ako problémy. Dôvodov môže byť niekoľko - nález je validný v kontexte analyzovaného kódu, už nie v kontexte kombinácie analyzovaného kódu a hardwaru na ktorom kód beží (napríklad čítanie neinicializovanej hodnoty, pričom hodnota je inicializovaná hardwarom), chyba v logike nástroja použitého na analýzu a ďalšie. Rôzne nástroje často ponúkajú možnosti ako potlačiť niektoré diagnostiky, jednak lokálne pre istý úsek kódu, jednak globálne ako konfiguračnú možnosť nástroja.

- *false negatives* sú absencie nálezov, ktoré sú reálne problémy. Príčiny môžu byť nedokonalosť nástroja (v tomto prípade s tragickejšími následkami ako pri *false positives*) prípadne neschopnosť extrahovať z kódu dostatok informácii pre úspešnú detekciu. V tomto prípade je niekedy možné kód dodatočne anotovať – v prípade jazyka C napríklad pomocou kľúčového slova `__attribute__`[2] (jeho primárny účel je síce možnosť efektívnejšej optimalizácie procesu kompilácie, statické analyzátory ho ale dokážu využiť podobne, prípadne ponúknuť vlastné prostriedky anotácie).

[1] https://valgrind.org/

[2] https://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html

Väčšina nástrojov sa snaží vyvažovať medzi počtom *false postitives* a *false negatives*. V skutočnosti často je možné dedukovať účel nástroja z pomeru týchto 2 veličín. Napríklad analyzátory zamerané na identifikáciu možných bezpečnostných dier budú na strane minima *false negatives* aj za cenu väčšieho počtu *false postitives*. Naopak analyzátory zamerané na kontrolu formátovania zdrojového kódu sa môžu v prípade nejasnosti rozhodnúť neupozorniť na chybu a pripustiť niektoré *false negatives*.

## 2.1 Typy statických analýz

Existuje niekoľko druhov nástrojov zameriavajucích sa na analýzu rôznych aspektov kódu. V tejto podsekcií priblížim 9 z nich. Popísaná funkcionalita sa môže prekrývať a niektoré nástroje kombinujú kontroly za účelom väčšej efektivity.

### 2.1.1 Kontrola správnosti práce s typmi

Jedna z najpoužívanejších typov analýz – pri kompilovaných jazykoch ako C/C++ vynucovaná počas kompilácie ako súčasť jedného z prechodov kompilátorom. Typová kontrola predchádza celým radom chýb od priradenia reťazca do typu integer po preklep v názve pri prístupe k položke v štruktúre.

### 2.1.2 Kontrola dodržania štýlu kódu

Známe taktiež ako anglický termín *linting*. Obecne tieto nástroje kontrolujú aspekty ako správne riadkovanie, pomenovanie premenných, konštrukcie ktoré sú náchylné na chybu, používanie *deprecated* funkcií, popísanie účelu funkcie pomocou *docstringu* a podobne. Mnoho z vynucovaných konvencií je stále predmetom diskusie a preto tieto nástroje majú často rozšírené možnosti konfigurácie. Niektoré kompilátory ako napríklad *gcc* začali tieto kontroly zahŕňať do svojho procesu (v prípade *gcc* v podobe `-Wall` parametru, ktorý vyústi v indikáciu nedefinovanej možnosti v konštrukcií `switch`).

### 2.1.3 Podporné analýzy

Nástroje ktorých účel je uľahčovať vývoj aplikácie pre programátora. Často sú súčasťou IDE vo forme funkcionality ako `nájdi všetky volania funkcie` alebo `premenuj premennú`, prípadne za analýzu sa dá považovať aj generovanie UML diagramu hierarchie dedičnosti tried.

### 2.1.4 Formálna verifikácia

Analýza používajúca matematický aparát a techniky formálneho dokazovania. Tradične je cieľom dokázať prítomnosť niektorej žiadanej vlastnosti (definícia hodnoty pred použitím) prípadne absenciu vlastnosti (dereferencia *NULL* pointeru). Príkladom môže byť časť kódu z obrázku 2.1.

Nález môže byť vrátený v podobe protipríkladu, ktorý dokazuje, akým spôsobom bola špecifikácia porušená, ako je vidieť na obrázku 2.2.

```
in_buf = malloc(in_buf_size);
if (in_buf == NULL) {
    return -1;
}
out_buf = malloc(out_buf_size);
if (out_buf == NULL) {
    return -1;
}
```

Obr. 2.1: Príklad programu obsahujúceho možné neuvoľnenie alokovanej pamäte

```
Violation of property "allocated memory should always be freed":
    line 2: in_buf != NULL
    line 5: out_buf == NULL
    line 6: function returns (-1) without freeing in_buf
```

Obr. 2.2: Protipríklad na potenciálne neuvoľnenie pamäti z 2.1

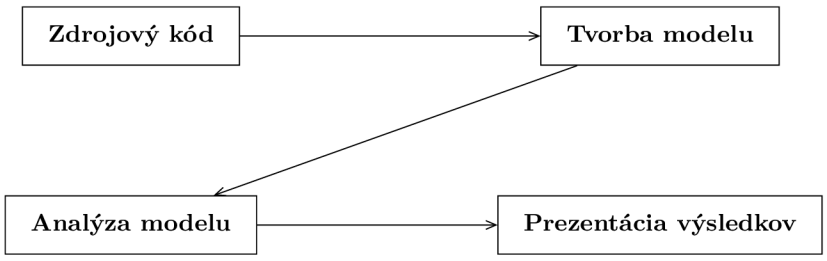
### 2.1.5 Hľadanie chybových stavov

Tento typ statickej analýzy detekuje vzory, ktoré sú známe tým, že môžu viesť k neočakávaným stavom. Často tieto nástroje obsahujú pravidlá (takzvané idiómy), ktoré popisujú situácie vedúce k týmto chybám.

### 2.1.6 Hľadanie bezpečnostných chýb

Princíp je podobný ako pri chybových stavoch z predchádzajúceho odstavca. Tieto nástroje však produkujú viac *false positives*, keďže pri bezpečnosti nie je vhodné riskovať možné neodhalenie skutočnej chyby.

## 2.2 Tvorba modelu pre statickú analýzu




Obr. 2.3: Diagram zobrazujúci princíp práce statického analyzátoru

V tejto sekcii sa venujem popisu práce statického analyzátoru so zreteľom na objasnenie potreby modelu pri analýze. Tvorba modelu je jedna z prvých krokov ktoré nástroj vykoná. Vhodný model zefektívni prácu so zdrojovým kódom, zjednoduší implementáciu rôznych kontrol prípadne pravidiel, umožní zanedbať informácie nezaujímavé z pohľadu analýzy.

Jedným z popísaných modelov je abstraktný syntaktický strom popísaný v sekcií o medzijazyku 3, konkrétne v podsekcií 3.3.

## 2.3 Porovnanie statických analyzátorov

V práci [1] sa autori zaoberali porovnaním známych nástrojov na statickú analýzu. Ich porovnanie použijem ako zdroj v tejto podsekcii. Porovnanie bolo uskutočnené na dátach poskytnutých [21][3]. Jedná sa o syntetickú sadu testov zameranú práve na testovanie statických analyzátorov C/C++. Testy pozostávajú predovšetkým z nízko-úrovňových chýb, ako pretečenie dátových typov, pretečenie bufferu, chyby alokovania pamäte a podobne, organizované do kategórii podľa typov a podtypov. Konkrétne sa v sade nachádza 638 prípadov rozdelených do 9 typov a 51 podtypov. Každý test pozostáva z 2 variant, pozitívneho nálezu a jeho negatívneho protiprípadu – spolu 1276 prípadov. Bolo porovnaných spolu 10 nástrojov.

### 2.3.1 Clang

Clang ako projekt je zložený z viacerých knižníc (library based architecture). Jedna z nich, *libanalysis*, je zameraná na statickú analýzu. Knižnica implementuje path-sensitive (predikáty vedúce k danej vetve sú brané v úvahu), interprocedurálnu analýzu založenú na princípe nahradenia hodnôt premenných za symboly (angl. symbolic execution). K tejto knižnici sú dostupné 2 nástroje – *Scan-Build* a *CodeChecker*[5].

Utilita *Scan-Build* vykoná statickú analýzu popri kompilácii projektu. Utilita prepíše premenné `CC` a `CXX` s myšlienkou ich nahradenia za „falošný“ kompilátor – ten použije buď `gcc` alebo `clang` (v závislosti od užívateľovej platformy) na kompiláciu, pričom v druhom kroku nad výsledkom kompilácie spustí statickú analýzu. Výstupom analýzy sú HTML súbory, každý nájdený efekt má vlastný súbor. K tomu je vygenerovaný `index.html`, ktorý slúži ako rozcestník [6].

*CodeChecker* je webová aplikácia postavená nad infraštruktúrou nástroja *Clang*. Má podporu pre iteratívnu analýzu (analyzuje iba súbory ktoré boli zmenené od posledného behu), API založené na frameworku *Thrift*[4]. Aplikácie potrebuje pre správne fungovanie minimálne clang verzie 3.6, odporúčane aspoň 3.8. Aplikácia je mienená ako náhrada nástroja *Scan-Build* [7].

### 2.3.2 Cppcheck

Open-source nástroj so zameraním na produkciu čo najmenšieho počtu *false positives*. K nástroju je možné stiahnuť rošírenie ktoré vynúteni dodržiavania štandardu MISRA 2012 (podpora však nie je kompletná, k dispozícii je zoznam[5] implementovaných bodov). Nástroj umožňuje definíciu vlastných pravidiel [9]. Príklad definície pravidla je možné vidieť na obrázku 2.4.

### 2.3.3 Flawfinder

Analyzátor zameraný na detekciu možných bezpečnostných chýb. Zásadná časť jeho návrhu je jednoduchosť – nebere v úvahu tok dát ani tok programu. K svojej práci používa pred-

[3] dáta sú dostupné na https://github.com/regehr/itc-benchmarks
[4] https://thrift.apache.org/
[5] http://cppcheck.sourceforge.net/misra.php

```
<rule version="1">
    <pattern>
        if \( ([!] )*?(strlen) \( \w+? \) ([>] [0] )*?\) {
    </pattern>
    <message>
        <id>StrlenEmptyString</id>
        <severity>performance</severity>
        <summary>
            Using strlen() to check if a string is empty
            is not efficient.
        </summary>
    </message>
</rule>
```

Obr. 2.4: XML definícia pravidla pre cppcheck, prevzaté z [10]

pripravenú databázu funkcií, ktoré nesú rôzne riziká (napríklad `strcpy` a riziko pretečenia alokovanej pamäte), a robí textové porovnanie oproti analyzovanému kódu. K nálezom ponúka CWE (common weakness enumeration) klasifikáciu a priradenie priority. Výstup je možno dostať textovo, prípadne v CSV alebo HTML. Nástroj je napísaný v jazyku *Python*, pričom podporuje obidve hlavné verzie interpretru 2.7 aj 3+ [11].

### 2.3.4 Flint++

Fork nástroja Flint pôvodne vyvíjaný facebookom (vývoj pôvodného nástroja ustal v prospech nového analyzátoru *Infer*, ktorý bude popísaný neskôr). Fork je napísaný v C++ (narozdiel od jeho predchodcu, napísaného v jazyku D), nepotrebuje žiadne závislosti (nástroj však potrebuje kompilátor, ktorý podporuje aspoň C++11) a je multiplatformový [12].

### 2.3.5 Frama-C

Frama-C nie je nástroj sám o sebe, ale platforma pre rôzne analyzátory, ktoré dokážu použiť výsledky iných analyzátorov ako svoj model. Frama-C je na rozdiel od väčšiny zameraná na korektnosť – dovoľuje užívateľovi vytvoriť špecifikáciu pre analyzovaný zdrojový kód a dokázať, že špecifikácia platí, prípadne že je porušená a pokiaľ je to možné aj s protipríkladom. V dobe tvorby tejto práce je Frama-C dodaná so 14 pluginmi, medzi ktoré patrí analýza možných hodnôt pomocou abstraktnej interpretácie, deduktíva analýza a v neposlednom rade identifikácia mŕtveho kódu [13].

### 2.3.6 Facebook Infer

Vznikol zo startupu Monoidics v roku 2013, v tom čase používajúci najmä *separačnú logiku* (angl. *separation logic*). Viac o princípoch použitých v separačnej logike je možné nájsť v [22]. Od vtedy bol pridaný modul na abstraktnú interpretáciu a lintovací modul. Nástroj sa zameriava predovšetkým na prácu s pamäťou, dodržiavanie štandardov písania kódu a detekciu nedostupných API volaní. Infer je dostupný open-source a je napísaný v jazyku OCaml [15].

### 2.3.7 OCLint

Prevažne lintovací nástroj používajúci abstraktný syntaktický strom ako model. Medzi jeho funkcionality patrí aj dynamické nahranie pravidiel. Nástroj je založený na *LibTooling* ktorá spadá pod Clang. Nástroj je open-source, napísaný v jazyku C++ [19].

### 2.3.8 Sparse

Názov je odvodený od súslovia sémantický parser. Nástroj bol vyvinutý pre potreby linuxového jadra – obsahuje anotácie, ktoré nesú sémantickú informáciu ako do ktorého adresového priestoru smeruje daný pointer alebo ktoré zámky daná funkcia zamyká/uvoľňuje [23].

### 2.3.9 Uno

Nástroj sa primárne zameriava na 3 typy chýb uvedených dole. Vďaka tejto úzkej špecializácii sa snaží zachovať nízky počet *false positives* a vysokú informačnú denzitu pri výstupe [24].

- použitie neinicializovanej premennej
- referencovanie NULL pointra
- prístup mimo hraníc poľa

### 2.3.10 Porovnanie výkonnosti

Z [1] som vybral spolu 5 metrík pre porovnanie výkonnosti jednotlivých analyzátorov. V definícii miery robustnosti sa vyskytuje pojem *robustne spracovaných* – to znamená, že prípad mal správne detekovanú pozitívnu variantu a zároveň pri negatívnej variante nevznikol falošný poplach. Nasleduje popis metrík a tabuľka porovnávajúca 9 nástrojov popísaných v predchádzajúcej sekcií, viz 2.5. Pre kompaktnosť sú výsledky zarovnané na 2 desatinné miesta.

- miera detekcie $DR = \frac{počet\ detekovaných\ pozitív}{počet\ pozitív\ spolu}$
- miera false positives $FPR = \frac{počet\ detekovaných\ negatív}{počet\ negatív\ spolu}$
- miera produktivity $PR = \sqrt{DR * (100 - FPR)}$
- miera robustnosti $RDR = \frac{počet\ robustne\ spracovaných\ prípadov}{počet\ prípadov\ spolu}$
- čas $T = doba\ behu\ nástroja\ v\ sekundách$

| Nástroj | DR [%] | FPR [%] | PR [%] | RDR [%] | Čas [s] |
|---|---|---|---|---|---|
| Clang | 35.84 | 10.95 | 56.49 | 25.67 | 13.55 |
| Frama-C | 27.86 | 5.79 | 51.23 | 22.38 | 14.43 |
| Oclint | 44.13 | 52.74 | 45.67 | 5.01 | 23.95 |
| Cppcheck | 20.81 | 0.78 | 45.44 | 20.03 | 2.83 |
| Infer | 9.70 | 1.41 | 30.92 | 8.29 | 47.30 |
| Uno | 5.48 | 0.16 | 23.39 | 5.32 | 50.80 |
| Flawfinder | 2.97 | 2.97 | 16.98 | 0.16 | 1.15 |
| Sparse | 1.56 | 0.00 | 12.49 | 1.56 | 3.97 |
| Flint++ | 1.10 | 1.10 | 10.43 | 0.16 | 0.27 |

Obr. 2.5: Tabuľka výsledkov benchmarku, zoradená podľa produktivity

Prvé miesto patrí nástroju Clang čo nie je prekvapivé, stojí za ním veľká komunita čo je často priamoúmerné k použiteľnosti nástroja. V prípade nástroja OClint miera false positives prekonala mieru detekcie, pričom nástroj stále dosiahol vysokú mieru produktivity. V tomto prípade sa ukazuje, že miera robustnosti môže byť vhodnejší indikátor efektivity nástroja.

# Kapitola 3

# Medzijazyk

Sekcia čerpá z [8]. Česky mezijazyk, skratka IR. IR je reprezentácia zdrojového kódu, ktorá je voliteľnou súčasťou procesu kompilácie zdrojového kódu. Vzniká ako produkt scanneru, je konzumovaná generátorom kódu. Často je vstupom rôznych nadstávb ktoré IR analyzujú, transformujú a optimalizujú – pričom výsledky sú reflektované ako modifikácia IR. Za súčasť IR sa považujú aj pomocné štruktúry ako tabuľka symbolov, tabuľka virtuálnych metód a ďalšie.

## 3.1 Rozdelenie IR na základe dátových štruktúr

Podľa typu použitých dátových štruktúr je možné IR rozdeliť na 3 kategórie – grafové, lineárne a hybridné. V nasledujúcich podsekciách priblížim jednotlivé varianty.

### 3.1.1 Grafové

Jedná sa o stromové štruktúry, prípadne po optimalizáciách ako CSE [1] orientované acyklické grafy (ďalej DAG).

Uzly predstavujú syntaktické prvky jazyka ako priradenie a volanie funkcie. Hrany spájajú operácie s operandmi – k operácii volania funkcie priraďujú jej názov. Uvedený príklad je zámerne triviálny, v reálnej implementácii by informácia o názve funkcie bola obsiahnutá v metadátach uzlu alebo v prípade DAG by mohla ukazovať priamo na uzol s jej definíciou.

Spôsob prechádzania stromu má vplyv na vlastnosti jazyka (alebo opačne, je nutné zvoliť správny prechod na základe požadovaných vlastností) – príkladom nech je vlastnosť vyhodnocovania výrazov zľava do prava ktorú je možné získať postorder prechodom.

Príkladom grafových IR sú AST (abstraktný syntaktický strom) a CFG (control flow graph).

### 3.1.2 Lineárne

IR je sekvencia operácii, ktorá sa má vykonať v poradí, v akom je definovaná. Predošlá definícia však nezahŕňa podporu pre skoky – tie sú implementované iným mechanizmom, napríklad špeciálne štítky (anglicky label), podobne ako v jazyku assembler.

Operácia zahŕňa operačný kód/identifikátor inštrukcie a 0-n operandov, v závislosti od konkrétnej inštrukcie a obecnej špecifikácie IR.

Príkladom lineárnej reprezentácie sú n-adresné kódy.

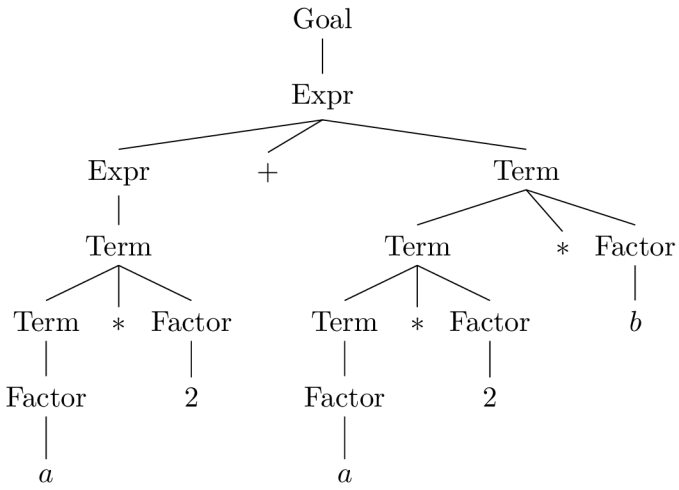
[1] Common subexpression elimination - eliminácia spoločných (rovnakých) výrazov

### 3.1.3 Hybridné

Špecifikácia hybridných IR sa snaží skombinovať grafové a lineárne IR tak, aby zachovala ich pozitívne vlastnosti a obišla ich slabé stránky. Príkladom môžu byť bloky kódu, ktoré sa vykonávajú lineárne pospájané do grafovej štruktúry.

## 3.2 Derivačný strom

Derivačný strom je grafická reprezentácia použitých derivácii v rámci gramatických pravidiel jazyka k dosiahnutiu daného reťazca [14]. Za povšimnutie stojí, že strom obsahuje veľké množstvo uzlov relatívne k veľkosti vstupného výrazu (konkrétne 22 uzlov pre výraz z obrázku 3.1).




Obr. 3.1: Derivačný strom pre výraz $a * 2 + a * 2 * b$

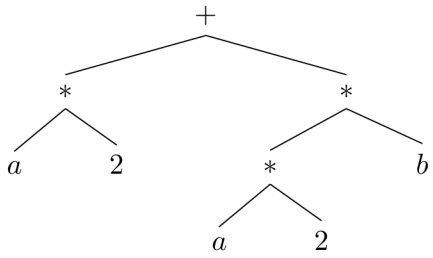
Táto reprezentácia je pre väčšinu kompilátorov neefektívna – obsahuje až príliš veľa informácii, čo sa prejavuje v pamäťovej aj výpočtovej náročnosti. V praxi je preto možné vidieť ich zmenšené formy (napríklad *Abstraktný syntaktický strom* popísaný v ďalšej subsekcií).

## 3.3 Abstraktný syntaktický strom

Anglicky *abstract syntax tree*, skratka AST. Vytvorený ako zefektívnenie derivačného stromu. Strom na obrázku 3.2 má oproti tomu na 3.1 iba 9 uzlov. Odstránené boli všetky nepotrebné uzly ako napríklad neterminál *Factor*. Teb bol nahradený terminálom, ktorý by bol výsledkom jeho derivácie.

## 3.4 Interpretácia abstraktného syntaktického stromu

V tejto sekcii sú uvedené pojmy, ktoré sú ďalej použité v kapitole 6 (Implementácia). Konkrétne je tu objasnené, v akom poradí je možné spracovať uzly stromu v podsekcií 3.4.1 a pomocná štruktúra, ktorá drží informácie nazbierané počas prechodu stromom popísaná v podskekcií 3.4.2.




Obr. 3.2: Abstraktný syntaktický strom pre výraz z 3.1

### 3.4.1 Typy prechodov stromu

Prechod stromu je postupnosť všetkých uzlov stromu, v ktorej sa žiaden uzol nevyskytuje viac ako raz (za predpokladu že všetky uzly v strome sú unikátne). Akt prechodu transformuje grafovú štruktúru stromu na lineárnu štruktúru jednotlivých uzlov. Existujú 3 typy prechodov – pre-order, in-order, post-order.

#### Pre-order

Spracuje predaný uzol, potom pokračuje na potomkov. Pri prechode stromom je teda prvý spracovaný koreňový prvok stromu (angl. root element). Tento prechod je okrem iného použitý k tvorbe prefixovej notácie (operátor predchádza operandy). Z abstraktného stromu na obrázku 3.2 by teda vznikol výraz `+*a2**a2b`. Implementáciu tohto prechodu je možné vidieť na obrázku 3.3.

```
def preorder(node):
    if (node == null):
        return
    visit(node)
    preorder(node.left)
    preorder(node.right)
```

Obr. 3.3: Pseudokód algoritmu pre pre-order prechod

#### In-order

Najprv spracuje ľavého potomka. Po spracovaní ľavej strany stromu spracuje predaný uzol, potom pokračuje na pravého potomka. Pri zobecnení na n-árny strom (ktorým abstraktný syntaktický strom je) by bolo potrebné definovať aká množina potomkov tvorí ľavú stranu, pričom ostatný potomkovia tvoria pravú stranu. Produkuje infixovú notáciu ktorá sa tradične používa v artimetike (tzn. `a * b`). Príklad implementácie je na obrázku 3.4.

#### Post-order

Spracuje ľavého potomka, pravého potomka, nakoniec predaný uzol. Post-order je typ prechodu stromom, ktorý je primárne použitý k interpretácii abstraktného syntaktického stromu. Hlavný dôvod je, že je nutné spracovať operandy predtým, ako je možné spracovať uzol s operátorom. Implementácia prechodu je naznačená na obrázku 3.5.

```
def inorder(node):
    if (node == null):
        return
    inorder(node.left)
    visit(node)
    inorder(node.right)
```

Obr. 3.4: Pseudokód algoritmu pre in-order prechod

```
def postorder(node):
    if (node == null):
        return
    preorder(node.left)
    preorder(node.right)
    visit(node)
```

Obr. 3.5: Pseudokód algoritmu pre pre-order prechod

### 3.4.2 Tabuľka symbolov

Symbol v kontexte tabuľky symbolov je identifikátor ktorý je použitý pre referencovanie entity použitej v zdrojovom kóde. Tento identifikátor je často unikátny aspoň v rámci typu entity na ktorý odkazuje. Môže to byť názov funkcie, premennej, štítku (labelu) a tak ďalej. Držia sa o ňom informácie ako napríklad dátový typ, v prípade funkcie počet a dátové typy jej argumentov, dátový typ návratovej hodnoty a podobne.

Počas prechádzania stromu je vhodné ukladať niektoré časti dát, najmä týkajúce sa symbolov. V opačnom prípade pri strete už definovaného symbolu by interpreter musel v strome znova vyhľadať uzol s definíciou, čo by predĺžilo čas nutný k interpretácii. Za zmienku stojí rozpustiť dáta z tabuľky symbolov do stromu, napríklad uzol referencujúci daný identifikátor by obsahoval aj referenciu na jeho definíciu ktorá by obsahovala všetky potrebné dáta.

Kľúčové vlastnosti pre tabuľku symbolov sú:

- rýchly prístup k dátam (časté používanie symbolov je základným kameňom väčšiny imperatívnych jazykov)
- možnosť efektívneho zväčšenia dátovej štruktúry (neefektivita odhadu počtu symbolov v dobe tvorby tabuľky)

S uvedenými vlastnosťami korešponduje hashovacia tabuľka. Vyhľadanie reťazca je O(1), zväčšenie tabuľky znamená rozšírenie listu možných výsledkov hashovacej funkcie a redistribúciu niektorých elementov.

# Kapitola 4

# Vývojové dosky firmy NXP

V tejto kapitole sa venujem balíkom potrebných k vývoju na mikrokontroléroch, vývojovému prostrediu *MCUXpresso* a nástrojom používaným k uľahčeniu práce s nadstavovaním hodín a pinov na mikrokontroléri.

## 4.1 MCUXpresso Software development kit

Balík pre vývoj softwaru, skratka SDK. Je to vrstva medzi implementáciou užívateľa a hardwarom. Obsahuje obslužné rutiny potrebné pre správnu inicializáciu hardwaru, podporu pre debugger a ďalšie. Pre výrobcu hardwaru je často vhodné dodať túto vrstvu spolu so špecifickým SDK jednak z dôvodu zvýšenia komfortnosti vývoja, ale aj spoľahlivosti produktu ako celku. Druhý aspekt je posunutie niektorých kontrol z hardwarovej vrstvy do softwarovej z dôvodu zníženia nákladov na výrobu hardwaru [2].

Popis MCUXpresso SDK preložený z [17].

*Vytvorený ako softwarový framework a referenčná príručka pre vývoj aplikácií postavených na NXP mikrokontroléroch. Užívateľ si dokáže prispôsobiť SDK na základe výberu konkrétneho mikrokontroléru a voliteľných softwarových komponent. Každé MCUXpresso SDK zahŕňa software na produkčnej úrovni obsahujúci predintegrovaný real-time operačný systém (RTOS), ovládače pre periférie, zapínateľné softwarové technológie (napr. middleware), referenčný software a ďalšie.*

### 4.1.1 Súbor popisujúci periférie

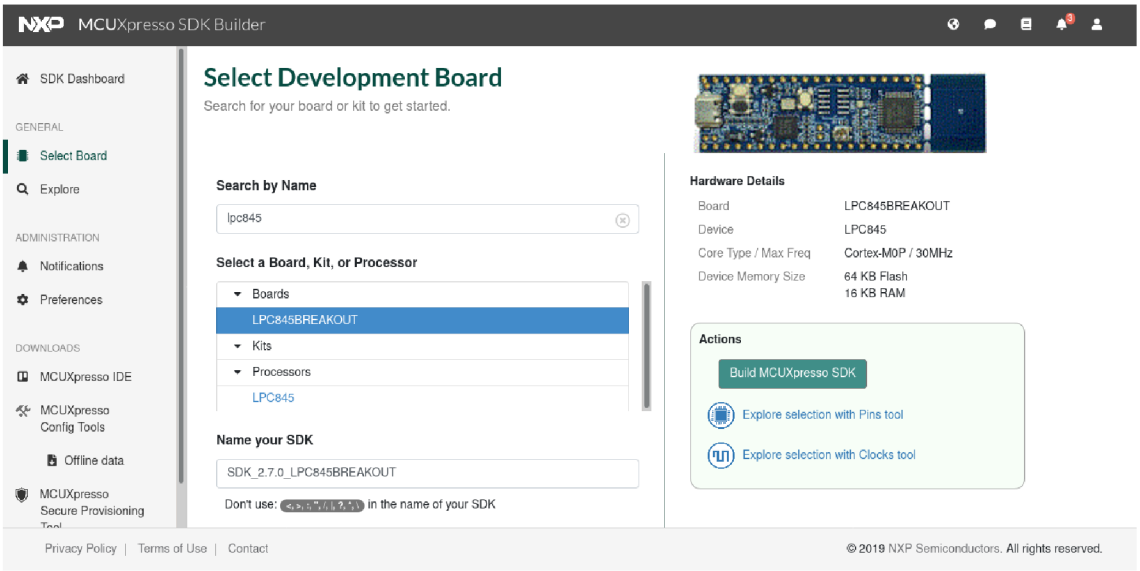
Jedná sa o súbor vo formáte XML s názvom odpovedajúcim mikrokontreléru, ku ktorému sa viaže. Obsahuje elementy popisujúce periférie, ktoré sú súčasťou mikrokontroléru, registre, ktoré dané periférie ovládajú, a podobne. Nasleduje zoznam elementov ktoré sú spracované v implementácii nástroja (ďalej rozobrané v podsekcii 6.3.3). Zanorenie elementov udáva ich vzájomný vzťah v hierarchií XML súboru, tzn. periféria obsahuje registre, ktoré obsahujú bitové polia (angl. bitfields).

- `peripheral` popisuje jednu periférne zariadenie. Periférne zariadenie je funkcionalita mikrokontroléru, medzi ne patria časovače, analógovo-digitálne prevodníky a ďalšie. Element obsahuje informácie ako jej meno, popis, na akej adrese začína blok registrov obsluhujúcich danú periférne zariadenie, ktoré prerušenia obsluhuje, a podobne.

- `register` popisuje jeden register, zväčša o veľkosti 16 alebo 32 bitov. Zápisom do registru ide kontrolovať funkcionalitu periférneho zariadenia, pod ktorú register spadá. Konkrétny postup, ako zmeny v registri ovplyvňujú periférne zariadenie, je obsiahnutý v dokumentácii k mikrokontroléru. Súčasťou elementu je meno a popis registru, jeho východzia hodnota, posun od začiatku adresového priestoru vyhradeného pre periférne zariadenie a podobne.
- `field` časť registra. V niektorých prípadoch register slúži ako kontajner pre niekoľko nadstavení, ktoré spolu môžu a nemusia súvisieť – tieto nadstavenia sa anotujú ako field-y. Týmto spôsobom je register dokumentovaný s granularitou na samotné bity. Podobne ako register a periférne zariadenie, obsahuje názov a popis. V niektorých prípadoch obsahuje aj preklad hodnoty na jej intuitívnejší popis (napríklad RUN / STOP). Field korešponduje k bitovému poľu v jazyku C.

## 4.2 SDK Builder

Firma NXP vytvorila dedikovanú webovú stránku venovanú SDK, kde je možné zvoliť vývojovú dosku, platformu, na ktorej sa bude vyvíjať, integrované vývojové prostredie, ktoré sa bude používať, a podobne. Po navolení je možné SDK stiahnuť formou súboru vo formáte zip. Ďalej je možné stiahnuť dokumentáciu k SDK API a dokumentáciu k SDK ako takému. V dobe tvorby tejto práce sa *Builder* nachádza na adrese `https://mcuxpresso.nxp.com/en/select`. Vyžaduje prihlásenie sa NXP užívateľským účtom. Na obrázku 4.1 je možné vidieť jeho užívateľské rozhranie.

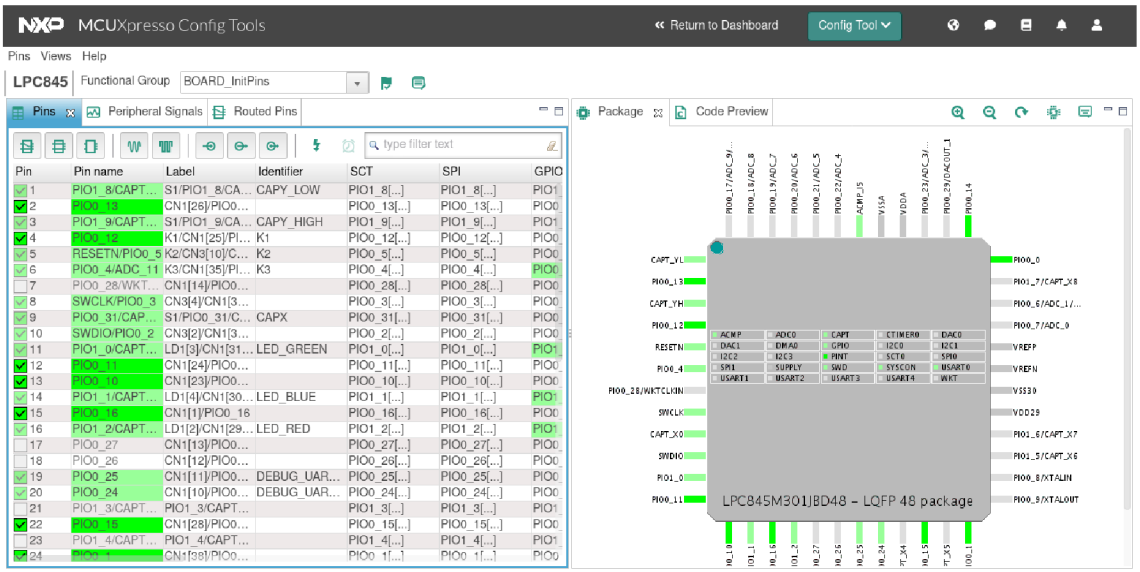



Obr. 4.1: SDK Builder po navolení dosky LPC845

## 4.3 Nástroj pre konfiguráciu pinov

Predstavuje intuitívny spôsob definície prepojení periférii s mikročipom. Nástroj je dostupný z webu aj z MCUXpresso IDE. Na webe ho možno nájsť pod možnosťou *Explore*

*selection with Pins tool*. Zvolenú konfiguráciu je možno exportovať ako stiahnuteľný zip súbor alebo zobraziť ako zdrojový súbor, ktorým je nutné prepísať východzí. V dobe písania je nutné zapnúť zobrazenie záložky s vygenerovaným kódom v menu, v záložke *View*. Nástroj neumožňuje zobraziť iba zmeny oproti súčasnému stavu. Piny sú podfarbené, je možné vidieť nevalidné konfigurácie podfarbené červenou farbou – v tomto prípade nebude obslužný kód vygenerovaný [18]. Rozhranie nástroja je viditeľné na obrázku 4.2.




Obr. 4.2: Konfigurácia pinov dosky LPC845

## 4.4 Nástroj pre konfiguráciu hodín

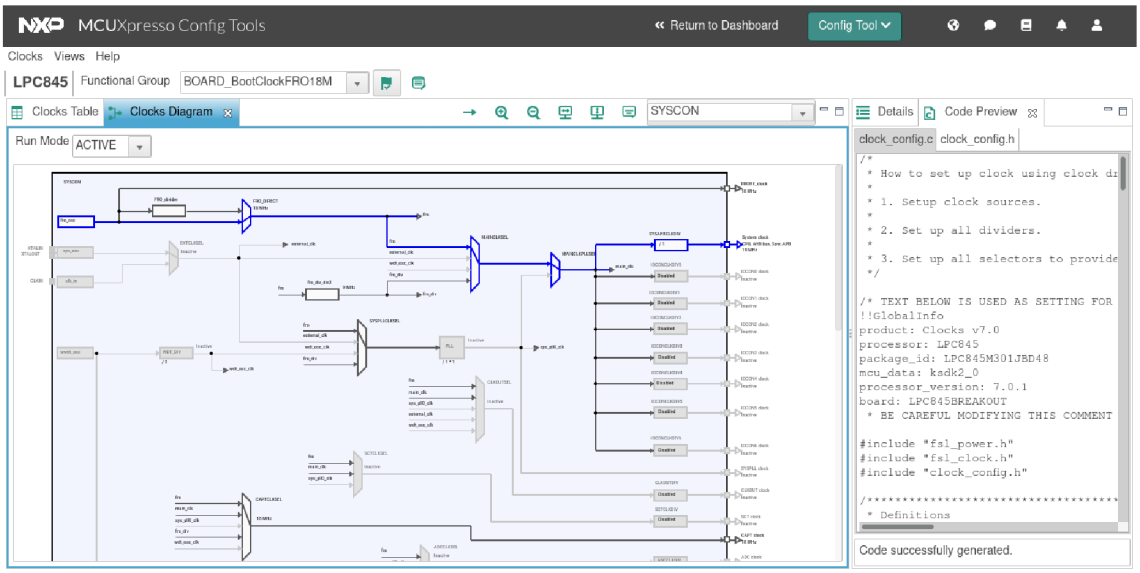
Veľmi podobný nástroju pre konfiguráciu pinov popísanom v predchádzajúcej podskecií. Nástroj obsahuje záložku s interaktívnym diagramom. V diagrame je vyznačený tok signálu užívateľom zvolených hodín. Po zmenách konfigurácie hodín sa prepočíta výsledná frekvencia a je zobrazená užívateľovi. Je možný aj opačný prístup, kedy užívateľ zadá cieľovú frekvenciu a nástroj sa ju snaží vhodnými voľbami dosiahnuť prípadne aproximovať [18]. Príklad rozhrania je na obrázku 4.3.

## 4.5 MCUXpresso

*MCUXpresso* je integrované vývojové prostredie (ďalej IDE) od firmy NXP. Je postavené nad existujúcim prostredím *Eclipse*[1] vlastneným Eclipse Foundation. Obsahuje všetky nástroje popísané v predchádzajúcich 3 podsekciách.

Výhoda prostredia je debugovacie rozhranie prispôsobené pre mikrokontroléry. Príkladom môžu byť anotácie podčastí registrov (tzv. bitfieldov[2]) – majú názvy podľa ktorých sa dajú identifikovať, interval na ktorej podčasti dátového typu registru sa nachádzajú,

[1] https://www.eclipse.org/
[2] https://en.cppreference.com/w/cpp/language/bit_field




Obr. 4.3: Konfigurácia hodín dosky LPC845

prípadne ich hodnoty sú interpretované ak to dáva zmysel – viz 4.4. Užívateľ si pri niektorých registroch môže všimnúť absenciu anotácii pre niektoré intervaly – to často značí že hodnota je rezervovaná, napriek tomu je vhodné konzultovať dokumentáciu k danému mikrokontroléru.

| Názov | Interval | Hodnota |
|---|---|---|
| POR | [0] | DETECTED |
| EXTRST | [1] | DETECTED |
| WDT | [2] | NOT_DETECTED |
| BOD | [3] | NOT_DETECTED |
| SYSRST | [4] | DETECTED |

Obr. 4.4: Demonštrácia anotácie registru SYSRSTSTAT (system reset status)

# Kapitola 5

# Návrh

Zadanie bakalárskej práce vyžaduje nástroj, ktorý analyzuje zdrojový kód napísaný v jazyku C s ohľadom na stavy registrov po vykonaní užívateľom zvolenej funkcie. Jeho výstup má obsahovať zoznam registrov obsluhovaných danou funkciou a ich hodnoty po jej vykonaní. Odvodil som, že k dosiahnutiu žiadanej funkcionality je nutná schopnosť spustiť ľubovoľnú funkciu (s podmienkou, viz ďalšia veta) obsiahnutú v dodanom zdrojovom kóde, pričom daná funkcia musí operovať nad rovnakým stavom programu ako jej spustenie na cieľovom zariadení. Zadanie obsahuje 2 zjednodušenia – a) množina očakávaných funkcií je obmedzená na tie, ktoré konfigurujú piny a hodiny, a b) stav programu v dobe pred spustením funkcie bude dodaný ako súčasť konfigurácie nástroja.

Opodstatnenie prvého zjednodušenia a zároveň prvá komplikácia je nasledujúca. Tým, že sa jedná o obslužný program pre mikrokontrolér, použité konštrukcie počítajú s meniacim sa stavom hardwaru – prerušenia, cykly čakajúce na zmenu registru a podobne. Obmedzenie množiny možných kategórií funkcií zároveň obmedzuje množinu stavov, ktoré je nutné riešiť (rozšírenie množiny na ľubovoľnú funkciu je však priestor na rozšírenie funkcionality a ponúka nové problémy na riešenie).

Druhú požiadavku, požiadavku na dodanie stavu programu, považujem za potrebnú. Časť stavu, ako napríklad stav pamäte, nie je možný získať zo zdrojového kódu samotného. Táto informácia je obsiahnutá v špecifikácii daného zariadenia. V iných prípadoch, ako stav globálnych premnenných, by teda bolo nutné spustiť funkciu od jej skutočného počiatku, to znamená interpretovať funkciu *main* až po koniec definície danej funkcie. Prakticky by to spravilo prvé zjednodušenie bezpredmetným (podobne ako v prvom prípade je to však námet na zlepšenie závisiaci na implementácii rozšírenia z predchádzajúceho odstavca).

Ďalšie zjednodušenie je použitie medzijazyka (popísaného v sekcií 3) ako vstupu pre interpretáciu – existujú nástroje, ktoré sú schopné zdrojový kód pretransformovať do abstraktného syntaktického stromu popísanom v podsekcií 3.3. Vďaka takto predpripravenému modelu nie je nutné riešiť lexikálnu analýzu, parsovanie podľa gramatiky a podobne. Nevýhoda je strata kontroly nad týmto procesom, čo má za následok, že pri akejkoľvek chybe v tejto vrstve bude nástroj odkázaný na obsluhu chybových stavov v rámci použitého nástroja na preklad.

## 5.1 Požiadavky na nástroj

Aká je motivácia za požiadavkou na tento nástroj? Súčasná situácia vyžaduje niekoľko operácii k otestovaniu, či sú dané registre správne inicializované. Je nutné zdrojový kód

skompilovať, nahrať na cieľové zariadenie a následne spustiť nástroje, ktoré skontrolujú správnosť inicializácie. Tento postup je časovo náročný a vyžaduje fyzickú prítomnosť mikrokontroléru v blízkosti stroja, ktoré vykonáva testovanie.

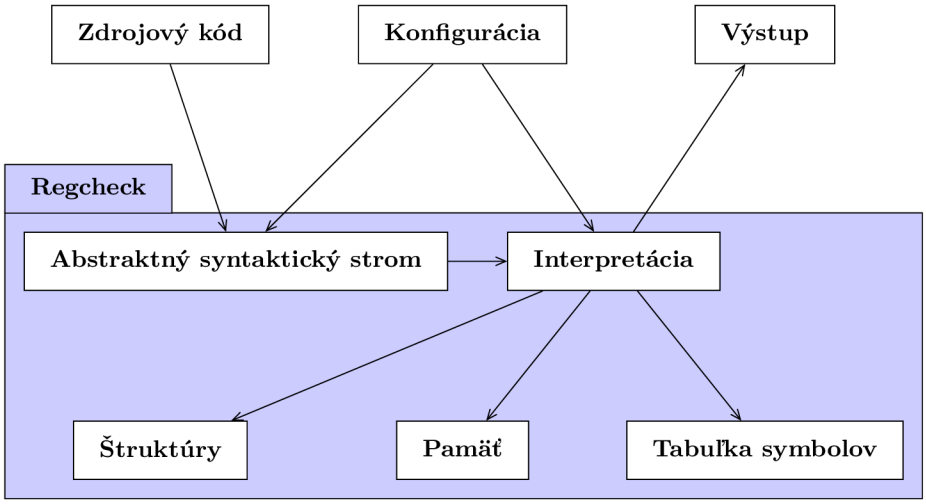
Statická analýza kódu je jedna z vecí, ktorá odchytí niektoré druhy chýb a skráti veľkosť času potrebnú k odhaleniu, kde sa chyba potenciálne môže nachádzať. Výsledný nástroj rozhodne nenahradí testovanie na cieľovom mikrokontroléri, nakoľko kompletná emulácia mikrokontroléru je náročná na implementáciu aj údržbu.

## 5.2 Návrh architektúry

Vstupom nástroja je konfiguračný súbor a zdrojové kódy, očakávaným výstupom je informácia sumarizujúca, ktoré registre boli obslúžené danou funkciou a aká je ich výsledná hodnota. K tomu je nutné niekoľko modulov:

- modul, ktorý bude prechádzať abstraktný syntaktický strom, interpretovať jeho časti, podľa potreby volať ostatné moduly
- modul s tabuľkou symbolov, ako je popísaný v podsekcií 3.4.2
- modul simulujúci dynamickú pamäť cieľového zariadenia
- modul pre prácu so `struct` a `union`

Mimo modulov popísaných vo výčte bude výsledný nástroj obsahovať obecné moduly ako modul implementujúci rozhranie príkazovej riadky, modul na generovanie výstupu a podobne. Na obrázku 5.1 je znázornená architektúra pomocou UML diagramu.




Obr. 5.1: Diagram znázorňujúci architektúru nástroja Regcheck

## 5.3 Popis konfigurácie

Nástroj je ovládaný kombináciou konfiguračných možností a konfiguračného súboru. Tento spôsob oddeľuje konfiguráciu interpretru od obecnej konfigurácie ako typ a destinácia výstupu nástroja. Konfiguračný súbor je vhodné uložiť ako súčasť projektu, ku ktorému sa

vzťahuje. Textové súbory taktiež ponúkajú väčšie možnosti formátovania, čo zlepšuje intuitivitu tvorby konfigurácie.

Ako formát konfiguračného súboru volím INI. Formát je natoľko používaný, že väčšina jazykov bude obsahovať rutiny schopné jeho rozparsovania. Kľúčovými prvkami tohto formátu sú sekcie – názov sekcie je uvedený v hranatých zátvorkach. V sekciách sú definované hodnoty vo formáte kľúč, hodnota. Sekcie slúžia ako menné priestory a dovoľujú duplikovať názvy kľúčov v rámci jedného konfiguračného súboru. Príklad formátu je zobrazený na obrázku 5.2.

INI súbor má aj svoje limitácie, nedovoľuje importovanie ostatných súborov (v tomto prípade by bol nutný šablonovací systém), znovupoužitie niektorých hodnôt alebo vnorené sekcie (v prípade požiadavky na túto funkcionalitu by som volil jeden z komplexnejších formátov ako YAML).

```
[params]
SystemCoreClock=uint32_t, 12000000

[files]
directory=examples/LPC845_Project
files=board/boards/clock_config.c, drivers/fsl_clock.c,
device_file=LPC845.xml
cpp_args=-DCPU_LPC845M301JBD48, -Iboard, -Idrivers, -Idevice, -ICMSIS
```

Obr. 5.2: Príklad konfigurácie vo formáte INI

## 5.4 Popis výstupu

Výstup je ponúkaný v parsovateľnom formáte CSV – formát, kde stĺpce sú oddelené čiarkou a riadky znakom nového riadku. Z užívateľského pohľadu musí byť jasné, ktoré registre boli menené a na akú hodnotu. Jedným z prostriedkov, ako to dosiahnuť, je prekladať adresy registrov na identifikátory, ktoré sú definované v špecifikácií daného mikrokontroléru (napríklad adresa 0x40048038 v LPC84X mikrokontroléri korešponduje k registeru SYSRSTSTAT spadajúci pod kategóriu SYSCON). Ďalší je deliť výstup na jednotlivé položky bitového poľa ako naznačené v tabuľke 4.4.

## 5.5 Zavrhnuté alternatívy návrhu

Sekcia je venovaná alternatívam návrhu, ktoré hoci boli zvažované, neboli vybrané pre implementáciu. Bližšie dôvody vedúce k zavrhnutiu daných myšlienok sú popísane v ich respektívnych podsekciách.

### 5.5.1 Emulátor

Použitie emulátora má nespornú výhodu – objekt testovania je výsledný firmware (tým pádom sú v istej miere testované aj procesy podieľajúce sa na jeho tvorbe). Emulátor ma však aj niekoľko nevýhod:

- Náročnosť vývoja emulátora
- Pre jednotlivé modely mikrokontrolérov sú potrebné špecifické emulátory (zaujímavé riešenie by mohol byť modulárny emulátor)
- Potreba verifikácie emulátoru oproti jeho predlohe (jedným z riešení by mohla byť sada testov púšťaných ako na mikrokontroléri tak aj na jeho emulácii a porovnanie výsledkov)

V prípade NXP majúcich niekoľko desiatok mikrokontrolérov by bol vývoj emulátorov neefektívny (a vývoj emulačnej platformy by pravdepodobne prekračoval rozsah bakalárskej práce). Príklad služby zaujímajúcej sa touto problematikou je https://docs.jumper.io/.

### 5.5.2 Plug-in do kompilátora

Uvedený spôsob by využíval podporu pre pluginy[1] vstavanú do gcc. Tým pádom by odpadla potreba lexikálnej analýzy a parsovania zdrojového kódu, naviac by sa otvorila možnosť interpretovať jeden z jednoduchších medzijazykov – GIMPLE[2] alebo SSA[3] formu.

Problémom však je, že ku kompilácii sa nepoužíva štandardné gcc, ale špeciálny kompilátor pre triedu procesorov ARM, konkrétne arm-none-eubi-gcc. Respektíve, problém nastáva až po zistení, že gcc-plugin.h v arm-none-eubi-gcc vyžaduje systémový hlavičkový súbor system.h, ktorý ku kompilátoru nebol pribalený (pochopiteľné, vzhľadom k faktu, že jedine gcc-plugin.h tento súbor vyžadoval). K zamietnutiu tejto alternatívy teda viedli nasledujúce dôvody:

- neschopnosť vyriešiť problém s použitím gcc-plugin.h (nepomohol ani pokus s fixincludes[4])
- neistota miery kompatibility arm-none-eubi-gcc a gcc, nakoľko pre vývoj pluginov pre kompilátor arm-none-eubi-gcc som nebol schopný nájsť žiadnu dokumentáciu

[1] https://gcc.gnu.org/wiki/plugins
[2] https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html
[3] https://gcc.gnu.org/onlinedocs/gccint/SSA.html
[4] https://github.com/arm-embedded/gcc-arm-none-eabi.debian/tree/master/src/fixincludes

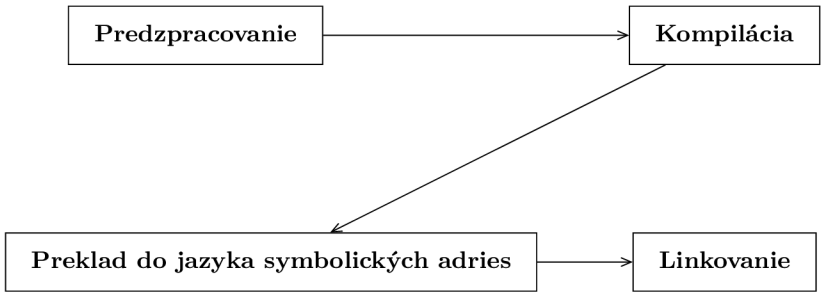
# Kapitola 6

# Implementácia

V kapitole implementácia sú popísané knižnice, na ktorých nástroj regcheck stavia a samotná implementácia nástroja regcheck. V sekcií 6.1 je popísaná predpríprava súborov pred samotným parsovaním. Ďalej v sekcii 6.2 sú priblížené knižnice, pomocou ktorých regcheck premení predzpracované zdrojové kódy na abstraktný syntaktický strom (popísaný v 3.3). V sekcií 6.3 je popísaná samotná implementácia nástroja regcheck. Sekcia 6.4 popisuje rozšírenia implementované nad rámec zadania. Na záver, v sekcií 6.5 je priblížená podmnožina jazyka C, ktorú je nástroj regcheck schopný spracovať.

## 6.1 Predzpracovanie zdrojových súborov

Predzpracovanie (angl. preprocessing) je prvá etapa v kompilácii. Graf znázorňujúci etapy kompilácie je znázornený na obrázku 6.1.




Obr. 6.1: Diagram znázorňujúci jednotlivé etapy kompilácie programu

Pri svojom prechode transformuje text. Medzi jeho transformácie patrí:

- Riadky ktoré končia spätným lomítkom sú spojené
- Komentáre sú nahradené medzerou
- Nahradenie `#include` direktív korešpondujúcimi súbormi
- Rozvinie makrá

Pre získanie výstupu preprocessoru je možné použiť utilitu `cpp`, ktorá je súčasťou `gcc` (Gnu Compiler Collection), prípadne použitie zaužívaného prepínača `-E` (možno nájsť ako v `gcc` tak aj v `clang`). Výstupom je zdrojový súbor/súbory po zpracovaní, na štandartnom

výstupe `stdout`. Pozor, výstupy preprocessorov nemusia byť syntakticky rovnaké naprieč rôznymi preprocessormi.

```
#ifdef USE_1
  #define CONSTANT 1
#else
  #define CONSTANT 2
#endif

5 + CONSTANT
```

Obr. 6.2: Zdrojový súbor C obsahujúci makro pred preprocessingom

```
# 1 "/tmp/preprocessor_demo.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 346 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "/tmp/preprocessor_demo.c" 2

5 + 1
```

Obr. 6.3: Zdrojový súbor C po preprocessingu (použitý príkaz `clang -E -DUSE_1`), pre kompaktnosť boli odstránené prebytočné prázdne riadky

Príklad 6.2 a teda aj v 6.3 nie je valídny C program, jeho kompilácia by zlyhala. Demonštruje to ale možnosť použitia preprocesoru aj mimo sféry jazyka C, nie je to však odporúčané.

## 6.2 Parsovanie

Táto sekcia sa venuje postupu, ktorý bol použitý k tvorbe abstraktného syntaktického stromu (popísaného v 3.3) zo zdrojového kódu. Sekcia začína základnými nástrojmi `lex` a `yacc`, na ktorých myšlienke stavia nástroj `ply`. Na záver sú popísané knižnice `pycparsing` a `pycparsingext` ktorých zameraním je parsovanie jazyka C.

### 6.2.1 Lex

V podsekcii predstavím nástroj Lex. Lex (prípadne jeho open-source implementácia `flex`) je zaužívanou súčasťou skladby nástrojov na poli lexikálnej analýzy. Jeho funkciou je rozdelenie vstupu na tokeny podľa predom definovaných pravidiel. Token je ďalej nedeliteľná jednotka, ktorá popisuje časť vstupu – v kontexte lexikálnej analýzy zdrojového kódu to môže byť názov premennej, číselný literál, kľúčové slovo jazyka a podobne. Pravidlá identifikujúce jednotlivé tokeny v reťazci sú popísané pomocou regulérnych výrazov. Informácie pre túto podsekciu sú prevzaté z [16].

```
%{
int znaky = 0;
%}
%%
. znaky++;
%%
int main(){
    yylex();
    printf("Pocet znakov vo vstupe: %d\n", znaky);
    return 0;
}
```

Obr. 6.4: Jednoduchý príklad práce s Lex, počítadlo znakov v zdrojovom súbore – v tomto prípade je každý znak jeden druh tokenu a akcia je inkrementácia počítadla. Súbor je preložiteľný do C pomocou volania utility lex, pri kompilácii je nutné prilinkovať lexovú knižnicu pomocou `-ll` prípadne `-lfl` pre `flex`

Tokeny `%%` sú predely medzi časťami zdrojového programu lex. Obsahuje 3 časti:

- Deklarácia, môže slúžiť na definíciu konštantánt pre tokeny.
- Prekladové pravidlá vo forme regulérny výraz, naviazaná akcia.
- Pomocné procedúry, môžu definovať ďalšie akcie potrebné pri analýze.

### 6.2.2 Yacc

Nástroj yacc je používaný pre syntaktickú analýzu – pomocou gramatiky vytvára z tokenov takzvané vety – skupiny tokenov podliehajúce definovaným gramatickým pravidlám. Zároveň môže byť jasné prečo sa tieto 2 nástroje používajú v tandeme. Lex vytvára tokeny pre yacc ktorý na ne naviaže akcie, napríklad generovanie medzikódu. Podobne ako lex aj yacc, je rozdelený do 3 častí - deklarácia, prekladové pravidlá a pomocné procedúry oddelené pomocou `%%`. Prekladové pravidlá sú definované vo forme podobnej BNF (Backusova–Naurova forma). Keďže pre zmysluplnú demonštráciu použitia yacc by bolo zároveň nutné definovať aj pravidlá pre lex, funkčná kompletná ukážka yacc nie je súčasťou podkapitoly. Informácie pre túto podsekciu sú prevzané z [16].

```
%token CISLO
%%
VYRAZ : CISLO '+' CISLO
    | CISLO '-' CISLO
```

Obr. 6.5: Príklad časti definície pre yacc.

### 6.2.3 Ply

Ply je skratka pre python lex yacc, je to python implementácia týchto dvoch nástrojov v jednom balíku. Väčšina funkcionality knižnice je implementovaná v dvoch moduloch

`lex.py` a `yacc.py`[1]. Narozdiel od predlôh, zdrojové súbory tvoria validný zdrojový kód v jazyku python a teda krok prekladajúci definície do hostiteľského jazyka (v predchádzajúcich podkapitolách jazyk C) odpadá. Ply tieto definície hľadá v špeciálne pomenovaných premenných a funkciách, spoliehajúc sa na dostupné možnosti introspekcie v jazyku python. Naviac Ply ponúka optimalizovaný mód, kedy medzivýsledok prechodu definícii uloží do súboru `lextab.py` a `yactab.py`, pričom ich zmene (napr. zmena regulárneho výrazu pre token) je nutné tieto súbory zmazať, aby sa vytvorili z aktualizovaných dát. Informácie nachádzajúce sa v tejto podskecií sú prevzaté z [20]. Príklad časti definície je na obrázku 6.6.

```
tokens = (CISLO, PLUS, MINUS)
t_PLUS = r'\)' # tokeny zacinaju t_<nazov tokenu>
t_MINUS = r'-'
t_CISLO = r'[0-9]+'

...

def p_binop(t): # gramticke pravidla zacinaju p_
'''vyraz : CISLO PLUS CISLO
         | CISLO MINUS CISLO'''
if t[2] == '+' : t[0] = t[1] + t[3] # zmeny sa deju na mieste
elif t[2] == '-': t[0] = t[1] - t[3]
```

Obr. 6.6: Príklad časti definície pre ply.

### 6.2.4 Pycparser

Knižnica používajúca Ply obsahujúca definície pre spracovanie jazyka C. Očakávaný vstup je zdrojový kód po preprocessingu, výstup je hierarchia objektov predstavujúca abstraktný syntaktický strom. Pycparser naviac obsahuje minimalistické signatúry funkcii zo štandardnej knižnice, nakoľko v zdrojových kódoch sa často na tieto funkcie odkazuje. Hlavičkové súbory sú v zložke `fake_libc_include` a na jej hlavičkové súbory sa dá odkázať štandardným prepínačom preprocesora `-I`.

Bázová trieda pre objekty popisujúce jednotlivé neterminály je trieda `Node`. Z nej sú odvodené ďalšie triedy ako `BinaryOp`, `UnaryOp`, `Constant` a ďalšie.

### 6.2.5 Pycparserext

Rozšírenie knižnice pycparser o schopnosť parsovať niektoré rozšírenia štandardu ako kľúčové slovo `asm`, prípadne kľúčové slova špecifické pre niektoré kompilátory ako `__inline` prípadne `__inline__`. Knižnica definuje dva parsery – parser pre OpenCL a parser pre GNU.

[1] Od verzie 4.0, na ktorej sa v dobe tvorby práce stále pracuje, sú to iba tieto 2 moduly

## 6.3 Regcheck

V tejto sekcií je popísaná samotná implementácia nástroja regcheck. Implementácia je popísaná po moduloch, každý modul ma vlastnú podsekciu.

### 6.3.1 Modul interpreter

Interpreter je implmenetovaný ako post-order prechod abstraktným syntaktickým stromom (post-order prechod je popísaný v 3.4.1).

Hlavná funkcia modulu je `interpret`, ktorá akceptuje instanciu triedy `Node` z knižnice pycparsing popísanej v podkapitole 6.2.4. Ta je obalená dekorátorom `singledispatch` z modulu `functools`. Tým sa funkcia zmení na takzvanú všeobecnú (anglicky generic), pričom jej konkrétne implementácie sa líšia na základe typu argumentov – v prípade `singledispatch` iba na základe typu prvého argumentu.

Konkrétne implementácie sú v podtržítkových funkciách (ich názov pozostáva z jedného podtržítka, tento idióm sa používa na označenie faktu že názov nie je podstatný). Každá funkcia implementuje logiku pre jeden typ uzlu stromu. Vo funkcii sa podľa potreby znova volá funkcia `interpret` na spracovanie operandov.

### 6.3.2 Modul utils

Z tohto modulu stojí za vyzdvihnutie dekorátor `symbol_to_name`, ktorý zjednodušuje hľadanie objektov v abstraktnom syntaktickom strome. Dekorovaná funkcia musí spĺňať nasledovnú podmienku: prvý argument je vyhradený pre funkčnosť dekorátora a musí obsahovať typehint[2] ktorý odpovedá typu hľadaného symbolu. Odekorovaná funkcia je potom volaná s názvom symbolu ako hodnotou prvého argumentu. Podľa kombinácie názvu a symbolu je potom nájdený odpovedajúci objekt v strome a je poslaný ako hodnota prvého argumentu obalenej funkcie. V prípade že objekt s daným názvom sa v strome nevyskytuje je vyvolaná výnimka `NodeNotFoundError`, v prípade že sa vyskytuje, ale je iného typu ako bolo deklarované v typehinte, je vyvolaná výnimka `TypeError`.

Ďalší implementovaný dekorátor je `with_coords`, ktorý ukladá súradnice (meno súboru, riadok, stĺpec) momentálne spracovaného objektu do globálnej premennej `coords` v module `state`.

### 6.3.3 Modul device

Modul zaoberajúci sa spracovaním špeciálneho súboru popisujúceho mikrokontrolér (taktiež rozobraný v podsekcií 4.1.1. Súbor je súčasťou SDK a obsahuje popis použitého procesoru a popis periférii nachádzajúcich sa na mikrokontroléri spolu s popisom registrov, ktoré ich používajú. Z tohto popisu je pre regcheck potrebné zistiť hodnoty registrov po inicializácii – tie sa použijú na inicializáciu triedy Memory popísanej v podsekcii 6.3.4

Modul definuje triedy, ktoré odrážajú stromovú štruktúru XML formátu, k tomu používa návrhový vzor kompozíciu. Spolu obsahuje 5 tried:

- `Device` zastrešuje celý XML súbor, obsahuje zoznam instancií triedy `Peripheral`, ktoré popisujú dostupné periférne zariadenia

[2] možnosť, ako v jazyku python deklarovať očakávaný typ, deklarovaný typ však slúži iba pre účely dokumentácie – jeho dodržianie nie je vynútené

- `Peripheral` trieda zodpovedá za informácie o perifénom zariadení – jej názov a pozíciu v pamäti (posun od začiatku adresného priestoru a dĺžka alokovaného priestoru), ako aj zoznam instancií Register popisujúcu registre ovládajúcu danú perférne zariadenie
- `Register` podobná ako trieda `Peripheral`, drží dáta o registry a zoznam instancií `Bitfield`
- `Bitfield` drží názvy a pozície bitfieldov nachádzajúcich sa v registry. Korešponduje k elementu `field` v XML súbore.
- `Cluster` je špeciálny prípad, kedy register obsahuje podregistre.

### 6.3.4 Modul memory

Modul poskytujúci triedy pre emuláciu pamäte mikrokontrolléru. Obsahuje 2 triedy – `Memory` a `Symtable`. Trieda `Memory` obsahuje mapovanie adries na ich hodnoty – interne používa natívny typ `dict`. Taktiež zaznamenáva zápis a čítanie jednotlivých adries. Trieda `Symtable` implementuje tabuľku symbolov – prekladá názov symbolu na instanciu triedy `Variable`.

### 6.3.5 Modul state

Obsahuje globálne premenné potrebné k správnemu chodu nástroja:

- `ast` – referencia na koreň abstraktného syntaktického stromu
- `memory` – referencia na instanciu triedy `Memory`, ktorá je použitá pri interpretácii
- `symtable` – referencia na instanciu triedy `Symtable`, taktiež použitá pri interpretácii
- `coord` – pozícia práve spracovávaného tokenu
- `device` – instancia triedy `Device` popísaného v podsekcii 6.3.3
- `all_paths` – rozhoduje či interpreter má prejsť obidve vetvenia podmienky v abstraktnom syntaktickom strome, funkcionalita je popísaná v podsekcií 6.4.2

### 6.3.6 Modul structs

Modul obsahuje triedy `Struct` a `Union`, ktoré obsahujú funkcionalitu pre zistenie atribútov či už celej štruktúry alebo jednotlivých položiek. Medzi tieto atribúty patrí zistenie pozície alebo veľkosti položky, veľkosť celej štruktúry a podobne.

### 6.3.7 Modul values

Obsahuje triedu `Variable` ktorá je použitá v tabuľke symbolov a slúži k popisu hodnoty uloženej v triede `Memory`.

### 6.3.8 Modul formatter

Modul zodpovedný za konverziu vnútorného stavu intrepretru do parsovateľného formátu. Momentálne podporuje iba formát CSV[3].

[3]Comma separated values (prekl. hodnoty oddelené čiarkou)

## 6.4 Implementované rozšírenia

Nástroj regcheck je rozšírený o dve funkcionality, ktoré sú nápomocné pre praktické využitie firmou NXP. Motivácia vzniku funkcionality ako jej implementácia sú popísané v nasledujúcich podsekciách.

### 6.4.1 Rozšírenie č. 1

Niektoré obslužné rutiny periférneho zariadenia nemusia byť súčasťou zdrojového kódu obdržaného na vstupe. Ich implementácia sa totiž nachádza na predom danej adrese v adresnom priestore mikrokontroléru. V prípade že je žiadúce použiť tento typ rutiny, je nutné danú adresu pretypovať na ukazateľ ktorého typ odpovedá signatúre rutiny. Pre lepšiu predstavu je zdrojový kód volania tohto typu rutiny ukázaný na obrázku 6.7.

```
(*((void (*)(uint32_t freq))(ROM_ADDRESS)))(value);
```

Obr. 6.7: Príklad volania rutiny ktorej implementácia nie je súčasťou analyzovaného zdrojového kódu

Regcheck rozpozná výraz uvedený na obrázku 6.7 a volanie danej adresy nahradí za symbol funkcie s menom odpovedajúcim `ROM_ADDRESS` v hexadecimálnom tvare (formát čísla odpovedá formátovaciemu reťazcu `%x`) a pridaním znaku podtržítka na začiatok názvu funkcie. (tzn. pre adresu `0x123` je volaná funkcia s názvom `_0x123`). Funkcia s týmto názvom musí byť obsiahnutá niekde v abstraktnom syntaktickom strome – môže byť súčasťou samostatného súboru, ktorý bude súčasťou analyzovaného projektu alebo priamo súčasťou zdrojového kódu. Toto rozšírenie je použité v príklade s mikrokontrolérom LPC845, funkcia je implementovaná v súbore `custom.c`.

### 6.4.2 Rozšírenie č. 2

Regcheck interpretuje obslužné rutiny – tým pádom v prípade podmienky je vždy vykonaná iba jedna vetva, tá ktorá vyplýva z počiatočného stavu mikrokontroléru dodaného v konfigurácii regchecku. Vznikla však nová požiadavka, zistiť ktoré všetky registre sú danou rutinou obsluhované.

Hodnoty registrov v tomto prípade nebudú mať zmysel – napríklad ak sú vykonané obe vetvenia podmienky a obe vetvenia nadstavujú jeden register jeho výsledná hodnota nemusí odpovedať stavu kedy by bola vykonaná iba jedna vetva. Toto obmedzenie by bolo možné obísť použitím symbolických hodnôt, ale to už bolo mimo rámca bakalárskej práce a konečné hodnoty z pohľadu firmy NXP neboli zaujímavé.

Rošírenie je zapnuté pomocou prepínača `--all-paths`. Tento prepínač vynúti vykonanie oboch vetiev podmienky po sebe.

## 6.5 Podporovaná podmnožina jazyka C

Regcheck podporuje všetky výrazy ktoré sa vyskytovali v testovaných projektoch. Prehľad podporovaných konštrukcií ponúka tabuľka 6.1.

| Operácia | Poznámky |
|---|---|
| Binárne operácie | podporuje multiplikatívne, aditívne,<br>relačné, posuvné, porovnávacie, bitové a logické |
| Unárne operácie | podpora referencie, dereferencie, bitovej a artimetickej negácie,<br>inkeremntácie, dekrementácie a záporu |
| Zložené výrazy | interpretuje všetky podvýrazy |
| Čítanie konštánt | podporuje konštanty zapísané decimálne a hexadecimálne<br>podporuje U a UL modifikátory |
| Deklarácia premennej | podporuje klasickú inicializáciu a inicializáciu poľom |
| Pretypovanie | podporované,<br>pretypovanie na void môže spôsobiť neštandartné správanie |
| Priradenie hodnoty | podporované |
| Volanie funkcie | podporované |
| Podmienky | podporované,<br>pomocou prepínačna `--all-paths` je možné interpretovať<br>pravdivú aj nepravdivú vetvu |
| While cyklus | podporovaný iba prázdny cyklus,<br>pri použití `--all-paths` je telo vykonané práve jeden krát |
| Prístup k prvku poľa | podporovaný |
| Štruktúry | podporované |
| Uniony | podporované |
| Switch | podporovaný |
| Vložený assembler | ignorovaný, súčasťou funkcionality regcheck nie je podpora assembleru |
| Ternárny operátor | podporovaný |
| Polia | podporovaný prístup k poľu, podprovaná nie je deklarácia poľa |
| Makrá | podporované v rámci použitého preprocessora |

Tabuľka 6.1: Funkcionalita podporovaná nástrojom regcheck

# Kapitola 7

# Grafické rozhranie

Súčasťou implementácie je aj grafické rozhranie – Regcheck-gui. Rozhranie je založené na multiplatformovej knižnici wxWidgets a predstavuje jednoduchú nadstavbu nad nástrojom regcheck.

## 7.1 wxWidgets

WxWidgets je multiplatformová C++ knižnica poskytujúca API k vytvoreniu grafického rozhrania, ktoré vypadá ako prirodzená súčasť daného operačného systému. Toho je dosiahnuté linkovaním oproti rôznym platformám, ktoré implementujú wxWidgets API. Východzie varianty platforiem pre jednotlivé operačné systémy sú popísané v tabuľke 7.1, dostupné sú však aj iné varianty ako napríklad wxQt a wxX11. WxWidgets je okrem C++ použiteľný aj s inými programovacími jazykmi, konkrétne Perl, Python a C#. Informácie o wxWidgets boli prevzané z [25].

Projekt s Python implementáciou má názov wxPython[1]. K vytvoreniu previazaní s wxWidgets používa nástroj sip[2].

| Operačný systém | Platforma | Prerekvizity |
|---|---|---|
| Windows | wxMSW | od verzie windows XP |
| MacOS | wxOSX/Cocoa | od verzie macOS 10.7 |
| Linux | wxGTK | odporúčaná verzia GTK+ >= 2.6 |

Tabuľka 7.1: Východzie platformy podľa operačného systému

## 7.2 wxFormBuilder

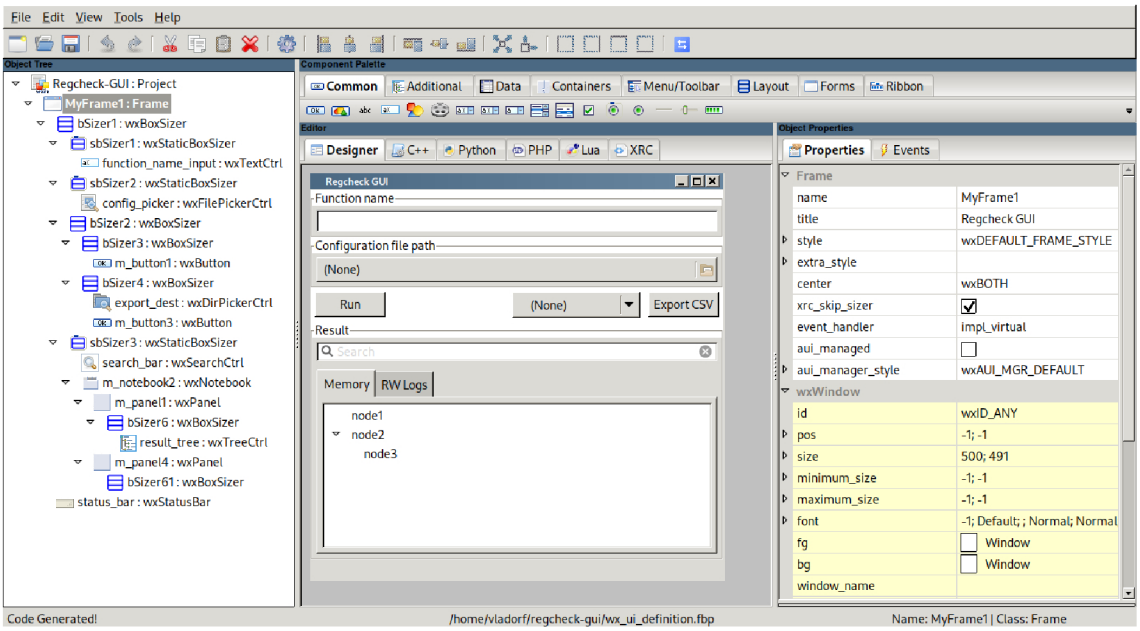
WxFormBuilder je dizajnér grafického rozhrania postaveného nad wxWdigets. Podporuje generovanie kódu do niekoľkých jazykov, konkrétne C++, Python, PHP, Lua a XRC[3]. Vo verzii 3.9.0, ktorú som použil k vývoju grafického rozhrania, však neobsahuje niektoré widgety, konkrétne zamrzel chýbajúci `ListCtrl` (widget vhodný pre zobrazenie dát ktoré sú štrukturované riadkovo). Taktiež nie je možné perzistentne uložiť obsluhu udalostí do

[1] https://github.com/wxWidgets/Phoenix

[2] https://www.riverbankcomputing.com/software/sip/intro

[3] XML formát špecifický pre wxWidgets, vhodný pre separáciu definície widgetov od logiky kódu

generovaného súboru. Rozhranie wxFormBuilder je ukázané na obrázku 7.1. Súbor projektu regcheck-gui je pribalený pod názvom `wx_ui_definition.fpb`.




Obr. 7.1: WxFormBuilder počas vývoja regcheck-gui

## 7.3 Implementácia

Implementácia je rozdelená do dvoch modulov:

- `model` – zodpovedný za sprostredkovanie dát
- `view` – zodpovedný za prezentačnú logiku

Nasledujúce podsekcie budú venované bližšiemu popisu týchto 2 modulov.

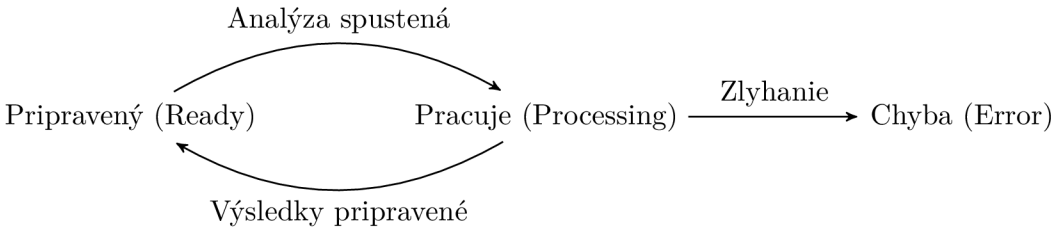
### 7.3.1 Model

Modul `model` obdrží vstupné argumenty od modulu `view` – tie spracuje do formy akceptovateľnej pre rozhranie nástroja regcheck a zavolá ho. Výsledok uloží do textového bufferu `io.StringIO` pre ďalšie spracovanie. To sa deje v metódach `_populate_peripherals` a `_populate_rw_logs`. Pripravené dáta sú sprístupnené metódami `peripherals` a `rw_logs`.

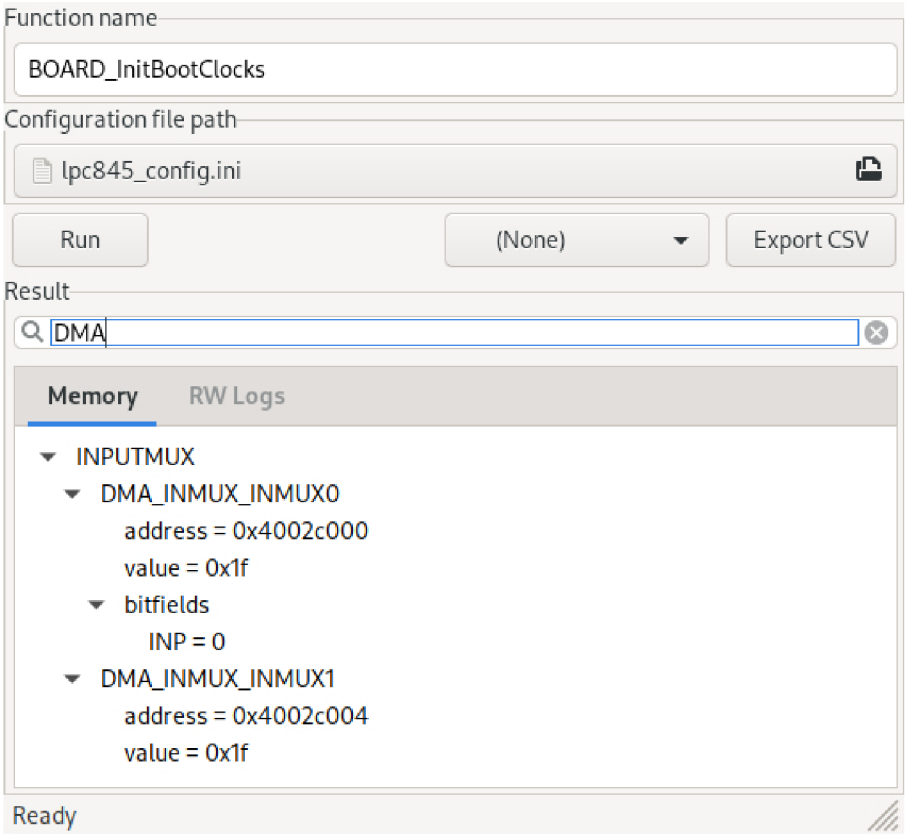
Model podporuje aj filtrovanie podľa názvu registrov, pričom hodnota filtra je predaná ako voliteľný argument pri preberaní dát pomocou týchto metód.

### 7.3.2 View

Modul obsahuje jedinú triedu `Frame`, ktorá rozširuje triedu `MyFrame1` z modulu `generated_view`. Modul `generated_view` je generovaný pomocou nástroja WxFormBuilder popísaného v sekcii 7.2. View taktiež aktualizuje stavovú lištu na odpovedajúci stav. Stavy sú znázornené na obrázku 7.2. Výsledná forma regcheck-gui je ukázaná na obrázku 7.3.




Obr. 7.2: Graf možných stavov v regcheck-gui




Obr. 7.3: Demoštrácia regcheck-gui

# Kapitola 8

# Testovanie

Regcheck je nástroj, ktorý analyzuje zdrojový kód v jazyku C, vyhodnotí ho a vráti informácie o zápisoch do registrov definovaných v špeciálnom súbore, popísanom v podskecii 4.1.1. Účelom tohto nástroja je automatizovaná kontrola stavu po ukončení danej rutiny – konkrétne, že registre, za ktoré je rutina zodpovedná, majú očakávané hodnoty a tým pádom aj prevencia možných chybových stavov plynúcich z porušenia tohto kontraktu. V nasledujúcich sekciách budem rozoberať metodiku testovania nástroja regcheck.

## 8.1 Testovanie na zapožičaných mikrokontroléroch

K testovaniu boli zapožičané dva mikrokontroléry – LPC845[1] a MK22FN512xxx12[2]. Vďaka ich fyzickej prítomnosti bolo možné k vývoju a testovaniu použiť informácie z ladiacej funkcionality ktorá je súčasťou MCUXpresso IDE popísaného v podsekcii 4.5. Jedna z funkcionalít je karta s aktuálnym stavom periférii korešpondujúcim k miestu, v ktorom ladiaci proces momentálne je. Ten ide využiť na získanie stavu registrov pred spustením testovanej rutiny (počiatočný stav) a po vykonaní testovanej rutiny (koncový stav). Za predpokladu, že regcheck obdrží počiatočný stav ako vstupný argument, je možné testovať jeho koncový stav oproti koncovému stavu získaného z ladiaceho procesu.

S postupom navrhnutým v predchádzajúcom odseku je však nasledujúci problém – v súčasnej verzii MCUXpresso (11.1.0) som nenašiel možnosť exportovania všetkých registrov. Je možné exportovať úseky pamäte zadané užívateľom s tým, že táto funkcionalita je použitá ako možnosť exportovania jednotlivých periférii (keďže informácia o pozícii jednotlivých periférii je predom známa, viz device súbor 6.3.3). Proces je teda poloautomatický, z manuálnej inšpekcie zdrojového kódu (prípadne zo znalostí užívateľa) je nutné získať množinu periférii, ktoré daná rutina obsluhuje, exportovať počiatočný a koncový stav týchto periférii a otestovať funkcionalitu nástroja regcheck použitím týchto dát.

Postup popísaný v predchádzajúcich dvoch odsekoch je implementovaný v testoch pre nástroj regcheck a bol použitý pri vývoji a testovaní tohto nástroja.

[1] https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/lpc800-cortex-m0-plus-/lpc845-breakout-board-for-lpc84x-family-mcus:LPC845-BRK

[2] https://www.nxp.com/design/development-boards/freedom-development-boards/mcu-boards/nxp-freedom-development-platform-for-kinetis-k22-mcus:FRDM-K22F

## 8.2 Testovanie výkonnosti

Jazyk Python je pomalší ako jeho kompilované alternatívy C/C++, je však vhodnejší na prototypovanie a rýchly vývoj (rapid application development, ako popísané v [3]).

Procesy testujúce výkonnosť sú spustené pod real-time plánovaciou triedou FIFO s prioritou 99 (najvyššia), aby sa čo najviac zamedzilo zmenám kontextu počas exekúcie programu. Príkaz použitý k dosiahnutiu tohto stavu bol `chrt --fifo 99 <príkaz>` (v závislosti na konfigurácii systému môže byť nutné príkaz spúšťať pod účtom root). K meraniu času bola využitá utilita `hyperfine`[3] vo verzii 1.9.0. Meranie bolo vykonané 100 krát s tým, že priemerná doba behu a jej odchýlka je zachytená v tabuľkách 8.1 a 8.2. Testy boli vykonané na dvoch procesoroch - Intel(R) Core(TM) i5-2520M @ 2.50GHz a Intel(R) Core(TM) i5-8265U @ 1.60Ghz, konkrétny procesor je v tabuľke označený jeho modelovým číslom. Použitý python interpreter bol verzie 3.8.2.

| MCU | Rutina | i5-2520M | i5-8265U |
|---|---|---|---|
| LPC845 | BOARD_InitBootPins | $3.744s \pm 0.044s$ | $2.349s \pm 0.035s$ |
| LPC845 | BOARD_InitBootClocks | $3.717s \pm 0.056s$ | $2.371s \pm 0.026s$ |
| MK22FN512xxx12 | BOARD_InitBootPins | $4.868s \pm 0.048s$ | $2.678s \pm 0.025s$ |
| MK22FN512xxx12 | BOARD_InitBootClocks | $3.969s \pm 0.044s$ | $2.741s \pm 0.024s$ |

Tabuľka 8.1: Rýchlosti behu nástroja regcheck bez použitia predpripraveného abstraktného syntaktického stromu

| MCU | Rutina | i5-2520M | i5-8265U |
|---|---|---|---|
| LPC845 | BOARD_InitBootPins | $1.980s \pm 0.028s$ | $1.342s \pm 0.019s$ |
| LPC845 | BOARD_InitBootClocks | $1.926s \pm 0.031s$ | $1.312s \pm 0.015s$ |
| MK22FN512xxx12 | BOARD_InitBootPins | $1.950s \pm 0.019s$ | $1.596s \pm 0.014s$ |
| MK22FN512xxx12 | BOARD_InitBootClocks | $2.057s \pm 0.020s$ | $1.657s \pm 0.018s$ |

Tabuľka 8.2: Podobne ako 8.1, v tomto prípade však za použitia predpripraveného stromu

Skript použitý pri tvorbe dát v tabuľkách 8.1 a 8.2 má názov `run_benchmark.sh` a je priložený k projektu regcheck.

Z výsledkov testu je možné usúdiť, že tvorba abstraktného syntaktického stromu tvorí podstatnú časť behu celého programu, konkrétne v priemere 48.3% pre i5-2520M a 38.4% i5-8265U. Predpokladám, že ďalšie zrýchlenie by priniesol prepis nástroja do C++, určite by bolo zaujímavé porovnať doby behu týchto dvoch implementácii.

## 8.3 Testovanie vo firme NXP

Použitie nástroja regcheck ako súčasť automatizovaného testovania vo firme NXP Semiconductors je primárnym cieľom tejto bakalárskej práce. V dobe písania tejto sekcie je regcheck spúšťaný v rámci nástroja Jenkins CI[4] na projektoch pre mikrokontroléry LPC845 a MK22FN512xxx12. Zároveň je výstup regcheck porovnávaný oproti ďalšiemu internému nástroju, ktorý poskytuje podobné informácie ako nástroj regcheck. Po ukončení fázy ove-

[3] https://github.com/sharkdp/hyperfine
[4] https://www.jenkins.io/

rovania výstupu bude regcheck použitý na kontrolu istého modelu podieľajúceho sa na inicializácii mikrokontroléru.

# Kapitola 9

# Záver

Účelom tejto bakalárskej práce bolo vytvoriť nástroj, ktorý má pomôcť k odhaleniu možných problémov v inicializácii periférii. Toho mal dosiahnuť statickou analýzou zdrojového kódu. Výsledný nástroj regcheck túto požiadavku spĺňa, výstup z tohto nástroja je možné využiť k porovnaniu analyzátorom nájdených a očakávaných hodnôt, prípadne ho využiť na kontrolu, či daná zmena neovplyvnila inicializáciu.

Formálne zadanie obsahuje šesť nosných bodov k vypracovaniu. Tieto body zahŕňajú preštudovanie literatúry s tematikou statickej analýzy zdrojového kódu, navrhnutie nástroja vykonávajúceho statickú analýzu kontrolujúcu stav periférnych registrov, daný návrh implementovať, popísať akú podmnožinu jazyka C implementácia podporuje, otestovať nástroj na reálnych mikrokontroléroch a na záver prediskutovať prínos práce a možné rozšírenia nástroja.

Poznatky naštudované čítaním literatúry sú obsiahnuté v kapitolách 2 až 5 tejto bakalárskej práce. Konkrétne diela sú samozrejme odcitované a ich prehľad je dostupný na konci práce. Čerpaním z teoretických základov nadobudnutých splnením prvého bodu zadania som navrhol architektúru nástroja a zavrhol dve možné alternatívy postupu. Nástroj som implementoval v jazyku Python za použitia dostupných knižníc a poznatkov získaných jednak štúdiom literatúry, jednak štúdiom šiestich semestrov bakalárskeho programu. Výsledkom je nástroj regcheck, konfigurovateľná aplikácia schopná určiť stav pamäte mikrokontroléru po behu inicializačnej rutiny a vypísať operácie vedúce k tomuto stavu. Regcheck bol taktiež otestovaný na dvoch mikrokontroléroch zapožičaných firmou NXP. V rámci rozšírenia zadania som ešte vytvoril grafické rozhranie, regcheck-gui, slúžiace k zväčšeniu užívateľského pohodlia práce s nástrojom regcheck. Týmto odstavcom som zhrnul dosiahnuté výsledky, zatiaľ čo ďalej v tejto kapitole je možné nájsť návrhy na rozšírenie nástroja regcheck o ďalšiu funkcionalitu.

Vďaka tejto práci som si uvedomil šírku disciplíny akou je statická analýza. Dokážem si predstaviť že v praxi odchytia kvantá zjavných chýb, ktoré sú jasné až keď je na ne poukázané. Je potrebné si ale uvedomiť že tieto nástroje nemajú nahradiť testovanie a ako také slúžia len ako prvé sito k dispozícii či už pre vývojárov alebo pre testerov.

Nástroj regcheck však nie je hotové dielo, nepodporuje všetky možné konštrukcie ktoré sú povolené v jazyku C. Jasný cieľ je podporovať ich aspoň toľko aby sa nástroj mohol použiť aj v širšej sfére ako mikrokontroléry vyvíjané firmou NXP. Taktiež k zrýchleniu nástroja by pomohol prepis do kompilovaného jazyka ako C++ prípadne niektorej z jeho modernejších alternatív.

V súčasnom stave je však nástroj pripravený na použitie a je ďalšou možnosťou v zálohe k zlepšeniu povedomia o vlastnostiach inicializačných rutín.

# Literatúra


[1] Arusoaie, A., Ciobaca, S., Craciun, V., Gavrilut, D. a Lucanu, D. A Comparison of Open-Source Static Analysis Tools for Vulnerability Detection in C/C++ Code. In:. September 2017, s. 161–168.

[2] Benso, A., Chiusano, S. a Prinetto, P. A software development kit for dependable applications in embedded systems. In: *Proceedings International Test Conference 2000 (IEEE Cat. No.00CH37159)*. Oct 2000, s. 170–178. ISSN 1089-3539.

[3] Beynon Davies, P., Carne, C., Mackay, H. a Tudhope, D. Rapid application development (RAD): An empirical review. *European Journal of Information Systems*. September 1999, roč. 8.

[4] Chess, B. a West, J. *Secure programming with static analysis*. 1. vyd. Addison-Wesley, 2007. ISBN 0-321-42477-8.

[5] *Domovská stránka nástroja Clang* [online]. [cit. 2020-01-06]. Dostupné z: https://clang.llvm.org/.

[6] *Domovská stránka nástroja Clang Scan-Build* [online]. [cit. 2020-01-11]. Dostupné z: https://clang-analyzer.llvm.org/scan-build.html.

[7] *Domovská stránka nástroja CodeChecker* [online]. [cit. 2020-01-11]. Dostupné z: https://github.com/Ericsson/codechecker.

[8] Cooper, K. D. a Torzcon, L. *Engineering a compiler*. 2. vyd. Elsevier, 2012. ISBN 978-0-12-088478-0.

[9] *Domovská stránka nástroja Cppcheck* [online]. [cit. 2020-01-06]. Dostupné z: http://cppcheck.sourceforge.net/.

[10] *Cppcheck git repozitár* [online]. 2019 [cit. 2019-12-19]. Dostupné z: https://github.com/danmar/cppcheck/tree/master/rules.

[11] *Domovská stránka nástroja Flawfinder* [online]. [cit. 2020-01-06]. Dostupné z: https://dwheeler.com/flawfinder/.

[12] *Domovská stránka nástroja Flint++* [online]. [cit. 2020-01-06]. Dostupné z: https://github.com/JossWhittle/FlintPlusPlus.

[13] *Domovská stránka nástroja Frama-C* [online]. [cit. 2020-01-06]. Dostupné z: https://frama-c.com/.

[14] Gudivada, A. a Rao, D. L. Chapter 2 - Languages and Grammar. In: Gudivada, V. N. a Rao, C., ed. *Computational Analysis and Understanding of Natural Languages: Principles, Methods and Applications*. Elsevier, 2018, s. 15 – 29. Handbook of Statistics, sv. 38. Dostupné z: http://www.sciencedirect.com/science/article/pii/S016971611830018X. ISSN 0169-7161.

[15] *Domovská stránka nástroja Infer* [online]. [cit. 2020-01-06]. Dostupné z: https://fbinfer.com/.

[16] John R. Levine, T. M. a Brown, D. *Lex yacc*. 1. vyd. O'Reilly, 1990. ISBN 1-56592-000-7.

[17] *Popis MCUXpresso SDK* [online]. [cit. 2020-01-06]. Dostupné z: https://www.nxp.com/docs/en/fact-sheet/MCUXPRESSOSDKFS.pdf.

[18] *Popis MCUXpresso ConfigTools* [online]. [cit. 2020-01-06]. Dostupné z: https://www.nxp.com/docs/en/user-guide/MCUXIDECTUG.pdf.

[19] *Domovská stránka nástroja OCLint* [online]. [cit. 2020-01-06]. Dostupné z: http://oclint.org/.

[20] *Domovská stránka PLY* [online]. [cit. 2020-04-18]. Dostupné z: https://www.dabeaz.com/ply/ply.html.

[21] Shiraishi, S., Mohan, V. a Marimuthu, H. Test suites for benchmarks of static analysis tools. In: *2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. Nov 2015, s. 12–15. ISSN null.

[22] Singh, A. K. a Natarajan, R. An Outline of Separation Logic. *CoRR*. 2017, abs/1703.10994. Dostupné z: http://arxiv.org/abs/1703.10994.

[23] *Domovská stránka nástroja Sparse* [online]. [cit. 2020-01-06]. Dostupné z: https://sparse.wiki.kernel.org/index.php/Main_Page.

[24] *Domovská stránka nástroja Uno* [online]. [cit. 2020-01-06]. Dostupné z: https://spinroot.com/uno/.

[25] *Domovská stránka wxWidgets* [online]. [cit. 2020-04-18]. Dostupné z: https://www.wxwidgets.org/about/.