VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

EXTRAKCE GRAFU TOKU ŘÍZENÍ Z FORMÁTU LLVM IR

EXTRACTION OF CONTROL FLOW GRAPH FROM LLVM IR FORMAT

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **VÁCLAV KONDULA**
AUTHOR

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

BRNO 2017

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Kondula Václav**
Obor: Informační technologie
Téma: **Extrakce grafu toku řízení z formátu LLVM IR**
Extraction of Control Flow Graph from LLVM IR Format
Kategorie: Analýza a testování softwaru

Pokyny:

1. Nastudujte architekturu překladače Clang. Nastudujte instrukční sadu LLVM IR.
2. Navrhněte extrakci grafu toku řízení z programů kompilovaných překladačem Clang. Výstupní formát extrahovaných grafů bude odpovídat požadavkům testovací platformy Testos.
3. Implementujte program pro extrakci grafu toku řízení. Velký důraz kladťe na udržovatelnost zdrojových kódů.
4. Ověřte správnou funkcionalitu programu na sadě binárních programů pokrývajících všechny řídicí konstrukce a všechny základní datové typy C a C++.

Literatura:

- *LLVM Language Reference Manual*. Dokument dostupný online: http://llvm.org/docs/LangRef.html
- P. Ammann, J. Offutt. *Introduction to Software Testing*, Cambridge University Press, 2008. ISBN 978-0-511-39330-3.

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání

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

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

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

Vedoucí: **Smrčka Aleš, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

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

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Tato bakalářská práce se zabývá návrhem a implementací extrakce grafů toku řízení ze zdrojových souborů jazyků C a C++ ve formátu LLVM IR. K tomu účelu bylo použito rozhraní LibTooling překladače Clang, pomocí něhož byl tento extraktor implementován jako samostatný nástroj. Výstup programu odpovídá požadavkům platformy Testos, jemuž poskytuje data pro automatické generování testovacích požadavků na základě specifického pokrytí.

Abstract

This Bachelor's thesis deals with design and implementation of control flow graphs extraction from source files in C and C++ language to LLVM IR format. The extractor was implemented as a standalone tool using the LibTooling interface of the Clang compiler. Output of this tool follows the requirements of platform Testos, so it can be used for automated generation of test requirements based on specified coverage criteria.

Klíčová slova

cfg, graf toku řízení, llvm, clang, mezikód, testování, libtooling, jazyk c,
jazyk c++, jednotkové testování

Keywords

cfg, control flow graph, llvm, clang, intermediate representation, testing, libtooling,
c language, c++ language, unit testing

Citace

KONDULA, Václav. *Extrakce grafu toku řízení z formátu
LLVM IR*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Smrčka Aleš.

Extrakce grafu toku řízení z formátu LLVM IR

Prohlášení

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

.....
Václav Kondula
17. května 2017

Poděkování

Tímto bych chtěl poděkovat vedoucímu práce Ing. Aleši Smrčkovi, Ph.D. za odbornou pomoc, cenné rady a věnovaný čas, jenž mi poskytl při vypracovávání této práce.

Obsah

Kapitola 1

Úvod

Testování je nedílnou součástí životního cyklu softwaru. Není výjimkou, že by chyba v programu zavinila nemalé finanční ztráty nebo v horším případě také stála lidské životy. Z těchto důvodů se v dnešní době přikládá stále větší důraz na testování a vznikají nové metody a nástroje na testování. Jednou z metod je testování na základě modelu. K tomu je však nejdříve zapotřebí vytvořit tento model ze zdrojových souborů.

Tato práce se zabývá extrakcí grafu toku řízení z jazyků **C/C++** ve formátu **LLVM IR**. Graf toků řízení programu – známější pod anglickým názvem *control flow graph* – využívá grafové notace k popisu všech možných průchodů programem, tedy reprezentuje abstraktní model systému. Tento formát je velice výhodný pro následnou specifikaci kritéria pokrytí programu a automatické generování testů.

LLVM IR – Low Level Virtual Machine Intermediate Representation – je silně typovaný nízko úrovňový jazyk využívaný překladačem LLVM. Jeho výhodou oproti ostatním jazykům vnitřní reprezentace *(IR)* je určitá míra abstrakce a implementace některých vysokoúrovňových mechanismů, jako jsou například výjimky.

K této extrakci využijeme knihovnu *LibTooling* 3.3, která slouží k psaní samostatných nástrojů založených na front-endovém překladači *Clang*.

Práce je strukturována následovně. Nejprve jsou v kapitole 2 představeny cíle této práce. Následně se v kapitole 3 čtenář seznámí s použitými technologiemi. Návrh a některé implementační detaily jsou popsány v kapitole 4. Verifikací požadavků a testováním se zabývá kapitola 5. V závěrečné kapitole bude provedeno zhodnocení práce 6.

1.1 Motivace

V současné době již existují nástroje na získání grafů toku řízení ze zdrojových souborů obou jazyků *C* a *C++*. Již samotný front-end překladače *Clang* umožňuje vypsat tento graf pomocí přepínače `-analyzer-checker=debug.DumpCFG`. Avšak výstup není v jazyce LLVM IR ani neobsahuje žádné programové lokace.

Obdobně je možné pomocí stejného nástroje přeložit zdrojové soubory do LLVM IR. Opět ale chybí některé ladící informace a tento formát není příliš vhodný k dalšímu zpracování, jak bude upřesněno v následující sekci 1.2.

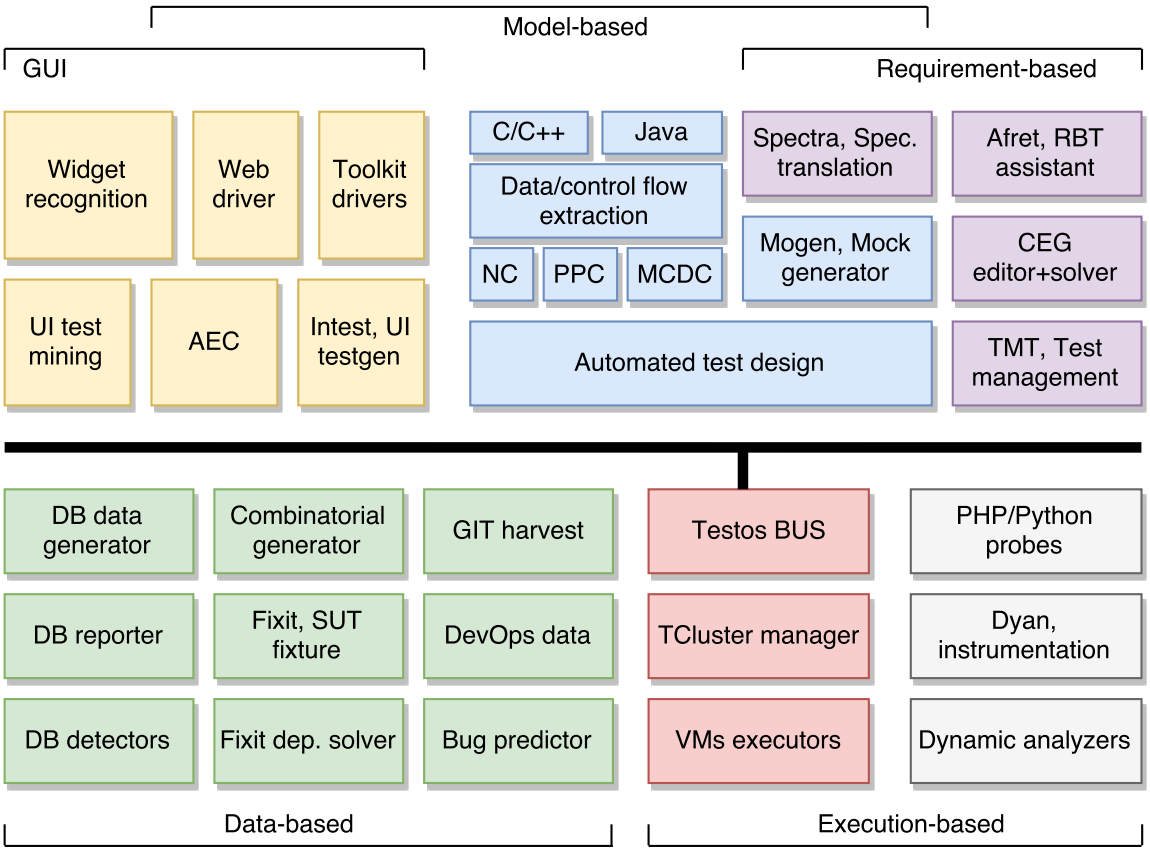
Z těchto důvodů jsem se rozhodl pro vytvoření vlastního nástroje pomocí knihovny *LibTooling* před zpracováním výstupů již existujících nástrojů.

1.2 Testos

Projekt Testos (Test Tool Set) [17] je realizován na Fakultě informačních technologií VUT v Brně a dává si za cíl vytvořit jednotnou sadu testovacích nástrojů, která bude použita pro výuku studentů a také nasazení v praxi. Jednotlivé nástroje vyvíjí studenti fakulty – v rámci svých bakalářských a diplomových prací, nebo jako koníček – a následně budou vydány pod svobodnou licencí.

Nástroje v platformě Testos (viz Obrázek 1.1) kombinují různé úrovně testování a lze je řadit do několika kategorií: testování založené na modelech (Model-based), testování založené na požadavcích (Requirement-based), testování grafického uživatelského rozhraní (GUI), testování založené na datech (Data-based) a dynamická analýza (Execution-based).

V aktuálním vývoji nástrojů pro testování založené na modelech jsou nástroje pro extrakci grafů toku řízení (CFG) ze zdrojových kódů jazyka C/C++ (tato bakalářská práce) a Java [16] a nástroj pro hledání požadavků na testy z CFG, tj. cest v CFG[20].



Obrázek 1.1: Schéma modelu platformy Testos.

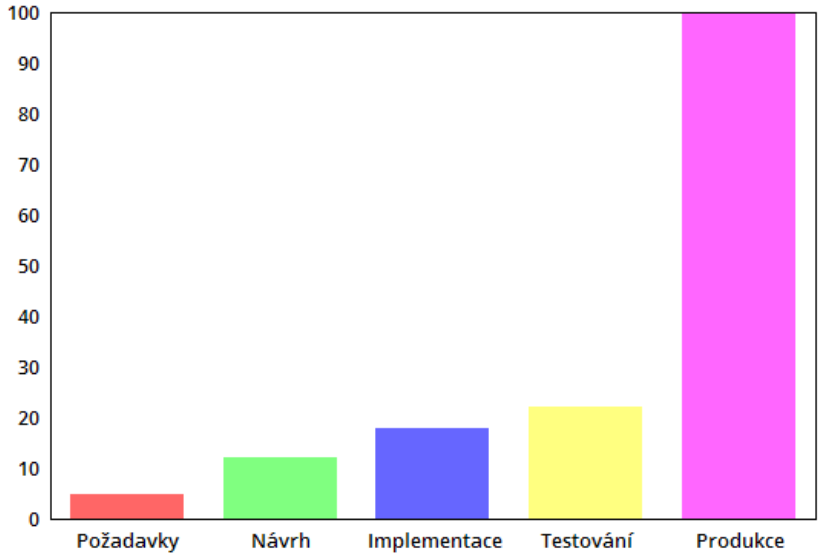
Kapitola 2

Testování softwaru

Testování softwaru je proces, který slouží ke zvyšování kvality softwaru, pomocí identifikování chyb a jiných nedostatků aplikace. Obecně platí, že testování plní tyto důležité funkce:

Verifikace je proces kontroly, že aplikace vyhovuje specifikaci. Specifikace definuje účel systému z hlediska vstupů a výstupů. Na začátku vývoje systému je tedy vytvořen model popisující funkci softwaru a verifikace ověřuje ekvivalenci tohoto modelu k vyvíjenému systému.

Validace je proces ověření, zda aplikace splňuje reálné požadavky uživatele. Pomocí této metody se odhalí chyby v návrhu aplikace způsobené nejčastěji nedokonalým zadáním uživatele, neznalostí programátora o prostředí, kde bude systém nasazen, nebo nesprávně interpretované komunikaci mezi uživatelem a programátorem. Validace se nejčastěji provádí na straně uživatele.



Obrázek 2.1: Závislost ceny opravy chyby ve specifikaci požadavků na fázi životního cyklu softwaru [6] (upraveno).

V procesu vývoje softwaru plní testování neodmyslitelnou roli již od samotného počátku vývoje. Včas odhalena chyba může znamenat ušetření nejen finančních prostředků, ale také spoustu práce. Na základě výzkumu Barry Boehma [6] se ukázalo, že cena opravy chyby má v čase exponenciální složitost. Znázornění můžeme vidět na grafu 2.1. Nejdražší na opravu jsou chyby způsobené v návrhu, pokud jsou odhaleny až v produkční fázi [8]. Jsou-li však odhaleny již na začátku vývoje, je cena jejich opravy minimální.

Z toho důvodu vzniklo mnoho paradigmat, metod a modelů pro vývoj softwaru zahrnujících testování. V dnešní době nejčastěji používaným modelem je *V-model*, který je i součástí standardu ISO 26262 [15]. Znázornění tohoto modelu můžete vidět na obrázku 2.2. Jeho podstata spočívá v tom, že každé návrhové fázi odpovídá určitá fáze testování.

Jednotlivé fáze předpokládají, že entity na nižších vrstvách mají korektní chování. Například při akceptačních testech se sice chyba způsobená špatnou komunikací mezi moduly může projevit, avšak v této fázi je velmi těžké zjistit přesnou příčinu takto vzniklé chyby.

1. **Definice požadavků – Akceptační testování:** Tato fáze zachycuje potřeby uživatelů a definuje samotný účel, k čemu má aplikace sloužit a jak má být využívána. Akceptační testování pak slouží ke zjištění, zda aplikace splňuje tyto požadavky. Chyby v této fázi mají největší finanční dopad, jelikož může dojít ke stavu, že je vyvíjen jiný software, než zákazník požaduje. Je velmi žádoucí, aby se na této fázi testování podílel přímo koncový uživatel softwaru.

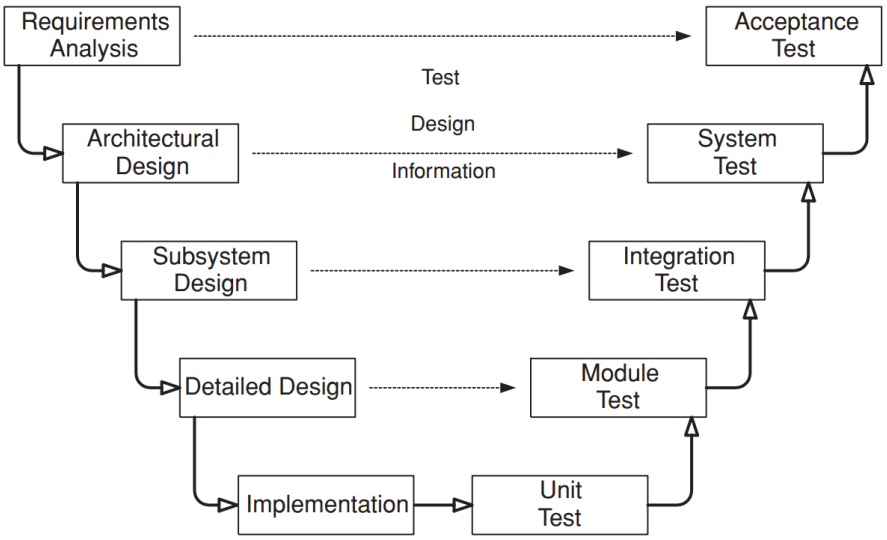
2. **Návrh architektury – Systémové testování:** Při návrhu architektury definujeme jednotlivé části systému a komunikaci mezi nimi. Systémové testování pak slouží k ověření, že takto poskládaný systém z podsystémů splňuje požadavky definované v návrhu.

3. **Návrh subsystémů – Integrační testování:** V této fázi se testují jednotlivé subsystémy jako samostatné celky. Ověřuje se, jestli subsystém plní svojí funkci a také jestli moduly, z nichž se skládá, navzájem korektně komunikují.

4. **Návrh modulů – Modulové testování:** Pojem modul v softwarovém inženýrství představuje kolekci souvisejících pojmenovaných jednotek, které jsou zapouzdřené do společného souboru, třídy nebo balíku. Například v jazyce C představují tyto jednotky funkce. Modulové testování se tedy zabývá vzájemnou komunikací těchto jednotek v rámci modulu. Za tuto fázi testování je ve většině společností odpovědný programátor, jenž moduly implementoval.

5. **Implementace – Jednotkové testování:** Jednotkové testování (angl. Unit testing) označuje testy zaměřující se na konkrétní jednotku, kterou v závislosti na zvoleném jazyce může být procedura, funkce, metoda nebo i celá třída. Jednotkové testy spadají do kategorie testů metody *bílé skříňky*, tedy se znalostí zdrojových kódů. Téměř výhradně tyto testy píší sami vývojáři. V průměru každý vývojář stráví v průměru 50% svého času právě testováním.

Ačkoli některým nezkušeným programátorům může testovaní připadat jako zdržující nebo dokonce zbytečné, tak opak je pravdou. Primární účel testování, jak již bylo zmíněno, je zvyšování kvality softwaru a hledání chyb. Z dlouhodobého hlediska však velká testovací sada také zvyšuje sebevědomí programátora při provádění změn a refaktorování. Programátor nemusí dlouze zkoumat všechny možné dopady na jiné části sytému, jelikož jsou pokryty v testovací sadě. Tím se především u velkých projektů zvýší rychlost a efektivita vývoje, aniž by byla dotčena kvalita softwaru.



Obrázek 2.2: V-Model vývoje software [1]

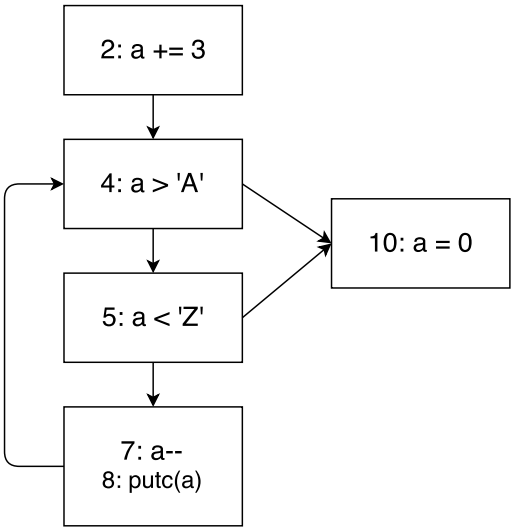
2.1 Testování založené na modelech

Ve své knize *Software Testing Techniques* Boris Beizer napsal (volně přeloženo): „Psaní testů je jednoduché, jediné co musí tester udělat, je najít graf a pokrýt ho." [4] Grafem v tomto výroku má na mysli autor abstraktní model systému, jímž může být například graf toku řízení (angl. *Control Flow Graph*, dále jen *CFG*).

Graf toku řízení je reprezentace programu využívající grafovou notaci k znázornění všech cest, kudy může program při spuštění projít. Uzly v tomto grafu představují základní bloky (angl. Basic Blocks) a orientované hrany pak přechody mezi těmito uzly. Základní blok znázorňuje sekvenci instrukcí, která je vždy vykonána celá v daném pořadí, má jen jeden vstupní bod a jeden výstupní. Aby tato definice byla splněna, musí platit následující pravidla:

1. Skoková instrukce je vždy v sekvenci poslední.
2. Návěští – cíl skokových instrukcí – je vždy v sekvenci první.

Neprázdná sekvence uzlů tvoří *cestu*. Musí platit, že pro každou dvojici sousedních uzlů existuje orientovaná hrana z počátečního do následujícího uzlu. Některé cesty jsou sémanticky neproveditelné. Příklad takové cesty je možné vidět v ukázce 2.2. Ordinální hodnota znaku nemůže být nižší než *65* a zároveň vyšší než *90*. Proveditelné cesty se nazývají *stopy*. Takové stopy, jež začínají v počátečním uzlu grafu a končí v koncovém uzlu, se označují jako *běhy*. Příklad zdrojového kódu v jazyce C reprezentovaného grafem toku řízení je znázorněn na ukázce 2.1.




```
1  void foo(){
2      a += 3;
3      while(
4          (a > 'A') &&
5          (a < 'Z')
6      ) {
7          a--;
8          putc(a);
9      }
10     a = 0;
11 }
```

Ukázka 2.1: Úsek zdrojového kódu v jazyce C převedený na graf toku řízení.

Tento výrok Borise Beizera později doplňují autoři knihy *Introduction to Software Testing* [1]. V rámci testování založeném na modelech grafové znázornění není v některých ohledech dostačující. Proto rozšiřují tuto definici o další tři abstraktní modely systému:

1. **Logické výrazy:** Logické výrazy nabývají dvou hodnot (pravda/nepravda) a v kódu slouží pro vyhodnocování řízení programu. V rámci testování na základě pokrytí pak používáme termíny *predikát* a *klauzule*. Klauzule je logický výraz, který neobsahuje žádnou logickou spojku. Například `c=(a>3)`. Predikát se pak skládá z klauzulí spojenými logickými spojkami. Například `p=(a>3)&&(a<10)`.

   Při vytváření testovací sady, například na splnění pokrytí všech klauzulí (angl. Clause Coverage), je vyžadováno, aby se každá klauzule alespoň jednou vyhodnotila na logickou hodnotu pravda i nepravda. Minimální testovací sada pro predikát `p=(a>3)&&(a<10)` by mohla obsahovat dva testy, kde proměnné `a` jsou přiřazeny hodnoty `0 a 20`.

   Testováním logických výrazů se nejčastěji nachází chyby programátora v použití logických spojek a priority operátorů.

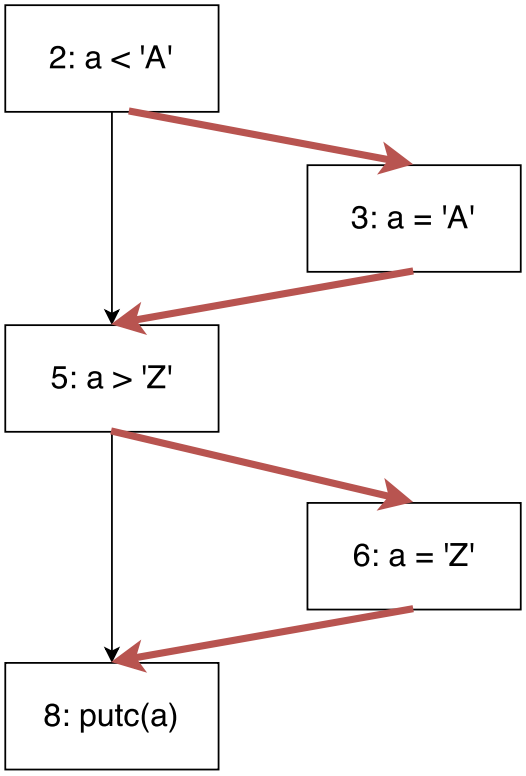
2. **Vstupní domény:** Doména představuje obor hodnot, kterých může proměnná nabýt. Teoreticky, pro některé proměnné, je tento obor hodnot nekonečně velký. Z praktického hlediska v programech například číslo typu *integer* v 32-bitové architektuře může nabýt pouze přibližně 4 milionů hodnot. Bohužel, při testování není možné pokrýt každou z těchto hodnot, proto je nutné rozdělit tuto množinu do kategorií, pro které se program chová stejně. Pro splnění pokrytí vstupních domén je nutné, aby testy pokryly alespoň jednu hodnotu z každé kategorie.

   Rozdělení do kategorií není vždy triviální a vyžaduje znalost systému. Například vstupní doména funkce pro výpočet přirozeného logaritmu `ln(x)` by se skládala z kategorií: $x_1 \in (-\infty; 0)$, $x_2 == 0$, $x_3 \in (0; 1)$, $x_4 == 1$ a $x_5 \in (1; \infty)$.

3. **Popis syntaxe:** Vstupní data na základě syntaktických pravidel můžeme jednoduše dělit na platné a neplatné. Syntaxi platných vstupních dat získáme většinou z formálního nebo semiformálního popisu. K tomu můžeme využít například regulární výrazy

nebo bezkontextovou gramatiku. Jedním z kriterií pokrytí založeném na syntaxi je pokrytí terminálních symbolů (angl. Terminal Symbol Coverage), které vyžaduje, aby testovací sada obsahovala každý terminální symbol gramatiky vstupů.

Z pohledu testování jsou mnohdy zajímavější neplatná vstupní data, která vedou k neočekávanému chování programu, jako jsou výjimky za běhu programu, přístup do neplatných míst paměti nebo vyčerpání zdrojů – například kapacity operační paměti. Mohou ale také způsobit nekonzistenci systému nebo se v nich skrývají bezpečnostní rizika. Nejznámějším útokem za použití neplatných dat je metoda *SQL injection*, při které se přes neošetřený vstup vloží řídící konstrukce pro práci s databází, čímž je možné získat citlivé informace nebo také smazat celou databázi.




```
1  void foo(char a){
2      if(a<'A'){
3          a = 'A';
4      }
5      if(a>'Z'){
6          a = 'Z';
7      }
8      putc(a);
9  }
```

Ukázka 2.2: Příklad sémanticky neproveditelné cesty (znázorněno červeně).

2.2 Kritéria pokrytí grafů

Testování na základě kriterií pokrytí se také někdy nazývá *úplné* nebo *vyčerpávající* testování. Tento přístup je založen na požadavcích na test (angl. *test requirements*), jež určují, který konkrétní element softwaru má test pokrýt. Kritérium pokrytí definuje pravidla pro generování těchto požadavků na test. Metrikou určující míru splnění testovacích požadavků v procentech je *pokrytí* (angl. *coverage*). Princip vytváření požadavků na test na základě kritéria pokrytí je založen na myšlence, že pokud se v systému vyskytuje defekt a máme 100% pokrytí testovacích požadavků, je velká šance, že alespoň jeden test tento defekt odhalí.

V předcházející kapitole bylo zmíněno několik kritérií pokrytí. Jelikož se tato práce zabývá generováním grafu toku řízení, budou nyní uvedeny některá kritéria pokrytí založena na tomto grafu. Obecně můžeme tato kritéria dělit do dvou kategorií.

Kritéria pokrytí řídících toků

Kritéria pokrytí řídících toků jsou založená na *navštívení* určitého uzlu, hrany nebo stopy na základě analýzy struktury grafu.

- **Pokrytí uzlů:** Požadavky na test tohoto pokrytí obsahují všechny dosažitelné uzly. Jedná se o nejzákladnější typ kritéria pokrytí, ze kterého ostatní kritéria vychází.
- **Pokrytí hran:** Cílem je zahrnout každou cestu o délce 0 nebo 1. Délka cesty je určena počtem hran, které obsahuje. Požadavky na test tedy zahrnují všechny uzly a také všechny hrany. Pokrytí hran plně *zahrnuje* pokrytí uzlů. Pokud je dosaženo 100% pokrytí hran, je automaticky dosaženo i 100% pokrytí uzlů.
- **Pokrytí párů hran:** Toto kritérium vyžaduje zahrnutí všech cest do maximální délky 2. Z toho vyplývá, že zahrnuje jak pokrytí uzlů, tak pokrytí hran.
- **Pokrytí jednoduchých cest:** Jednoduchá cesta (angl. *simple path*) je typ cesty, ve které se nepakuje žádný uzel s výjimkou prvního a posledního. Požadavky na test pro splnění tohoto pokrytí musí obsahovat všechny takové cesty.
- **Pokrytí hlavních cest:** Hlavní cesta (angl. *prime path*) musí splňovat stejné náležitosti jako jednoduchá cesta a navíc nesmí být podcestou jiné jednoduché cesty. Jinými slovy hlavní cesta je nejdelší možná jednoduchá cesta. Pokrytí hlavních cest je zahrnuto v pokrytí jednoduchých cest.

Kritéria pokrytí datových toků

Kritéria pokrytí datových toků zkoumají práci s proměnnými. V těchto kritériích pokrytí se využívají řetězce definic a užití (angl. *def-use chains*). Definice představuje zápis hodnoty do proměnné. Užití symbolizuje přečtení hodnoty z proměnné, tedy když je proměnná využita jako operand instrukce nebo volání funkce. Případy užití a definice proměnné se vždy vztahují k základnímu bloku, ve kterém se vyskytuje instrukce, která definici nebo užití obsahuje. Pokud však k jednomu základnímu bloku se vztahuje jak definice, tak užití stejné proměnné, nelze bez znalostí konkretních instrukcí určit, jestli například nepředchází užití dané proměnné její definici [1].

Tato kritéria pokrytí využívají termín *cesta definice a užití* (angl. *definition-use path*). Pro tuto cestu musí platit, že začíná v uzlu, kde je definována proměnná a končí v uzlu, kde se používá. V žádném jiném uzlu na této cestě nesmí být proměnná znovu definována.

- **Pokrytí definic:** Požadavky na test pro toto pokrytí musí obsahovat alespoň jednu cestu definice a užití pro každou definici proměnné. Smyslem těchto testů je dokázat, že každé přiřazení hodnoty do proměnné dává smysl pro nějaké její užití.
- **Pokrytí užití:** Toto pokrytí vyžaduje aby požadavky na test zahrnuly alespoň jednu cestu pro každou dvojici definice a užití proměnné. Tím je možno dokázat, že pro každé přiřazení hodnoty do proměnné dává každé její čtení smysl.
- **Pokrytí všech cest definic a užití:** Jedná se o nejrozsáhlejší pokrytí datových toků, jež požaduje pokrytí všech cest definic a užití. Zahrnuje v sobě pokrytí jak definic, tak užití.

2.3 Ladící informace

Ladění v softwarovém inženýrství je proces hledání a snižování počtu chyb v programu. Při selhání testu je nutné lokalizovat defekt ve zdrojovém kódu, aby mohl být následně odstraněn. K tomu účelu slouží ladící informace. Při vytváření grafů toku řízení programu, je vhodné zahrnout programové lokace základních bloků ve formě názvu souboru a vymezení čísla řádku a sloupce, kde blok začíná a končí. Díky čemuž při manuálním vytváření testů víme, které části kódu máme pokrýt na základě zvoleného kritéria pokrytí. Při automatizovaném vytváření testů využijeme programové lokace jako zpětnou vazbu pro programátora o vzniklé chybě, díky čemu se usnadní proces jejího odstranění.

2.4 Automatické generování testovacích případů

Vytváření testovací sady je složitý a velmi zdlouhavý proces. Kvůli tomu mnoho programátorů testování opomíjí a jsou vydávány programy s množstvím chyb. O to složitější je vytváření testovací sady na základě kritéria pokrytí grafu. Většina projektů, která se chlubí 100% pokrytím kódu, využívá pouze *pokrytí uzlů*, které v přeneseném významu vyžaduje, aby každý řádek programu byl proveden alespoň jednou. Toto chování však nereflektuje funkčnost programu.

V současné době neexistuje žádný ucelený nástroj, který by byl schopen generovat testovací sadu na základě složitějšího a efektivnějšího kritéria pokrytí, jímž je třeba *pokrytí hlavních cest*. O vytvoření takového nástroje se snaží část platformy Testos [17], do které patří i program vytvořený v rámci této bakalářské práce. Automatické generování testů má tyto fáze:

1. V prvním kroku je ze zdrojových souborů extrahován graf toku řízení včetně ladících informací a také definice a případy užití pro jednotlivé proměnné.

2. Následně pomocí knihoven implementované Radkem Vítem [20] získáme požadavky na test na základě zvoleného kritéria pokrytí.

3. Pro každou cestu se serializují instrukce ze základních bloků, přes které cesta prochází. Pomocí nástroje na *řešení problému splnitelnosti omezení* (angl. *constraint satisfaction problem solver*) získáme vektor vstupů pro každou cestu. V této fázi jsme schopni vygenerovat testovací sadu, jež splňuje dané pokrytí. Díky tomu je možné otestovat stabilitu programu, tedy že nedojde k selhání při nějakém průchodu, nikoliv jestli program pracuje správně.

4. V poslední fázi je nutné získat na základě formální specifikace pro každý vstup i očekávaný výstup, který bude při spuštění testu porovnán s reálným výstupem.

V době psaní této práce nejsou ještě nástroje pro řešení problémů splnitelnosti omezení ani zpracování formální specifikace v platformě Testos implementovány. Program na extrakci grafů toku řízení z jazyků C/C++ implementovaný v rámci této bakalářské práce i knihovny na hledání cest pro splnění pokrytí jsou již nyní dostupné na oficiálních stránkách projektu Testos [17].

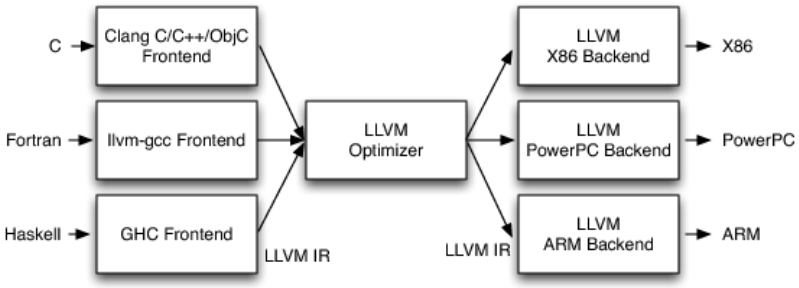
Kapitola 3

Projekt LLVM

Projekt LLVM (původně *Low Level Virtual Machine*) je komplexní sada znovupoužitelných nástrojů a knihoven implementující překladač. LLVM začal jako výzkumný projekt na Univerzitě Illinois v Urbana Champaign v roce 2000. Následně byl vydán pod svobodnou licencí UIUC[1].

LLVM je implementovaný v jazyce *C++* a původně měl sloužit pouze k překladu jazyků *C* a *C++*. Samotný překlad je pak rozdělen do tří fází. V první fází front-endový překladač – v případě jazyků C a C++ se jedná o *Clang* podrobněji popsaný v sekci 3.3 – přeloží zdrojové soubory do LLVM IR. Následně proběhne optimalizace a v poslední fázi je program přeložen do jazyka cílové architektury pomocí *LLVM Core* 3.2.

Tento jazykově nezávislý designe (znázorněný na obrázku 3.1) dal postupem času vzniknout celé řadě front-endových překladačů. LLVM dnes umožňuje překlad z mnoha dalších jazyků jako jsou Ada, Common Lisp, Delphi, Fortran, Haskell, Python, R nebo Ruby. Avšak LLVM zastřešuje i další projekty jako například optimalizované standardní knihovny knihovny jazyků C a C++ s názvem *libc++*, debugger *LLDB* nebo optimalizátor *dragonegg*.



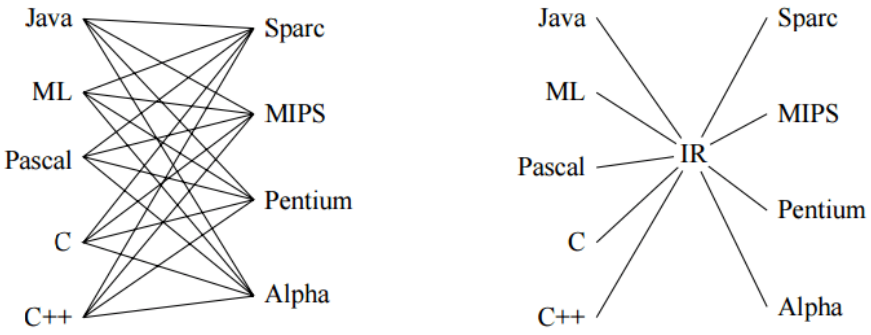
Obrázek 3.1: Znázornění modularity překladu při použití LLVM IR [19].

3.1 Mezikód LLVM

Mezikód – nebo také kód vnitřní reprezentace, angl. *Intermediate Representation (IR)* – slouží k reprezentaci programu při překladu ze zdrojového do cílového kódu. Dobře navržený mezikód je nezávislý jak na zdrojovém, tak na cílovém jazyce. Ku příkladu, jak je znázorněno

[1] University of Illinois/NCSA Open Source License, http://otm.illinois.edu/disclose-protect/illinois-open-source-license

na obrázku 3.2, máme N zdrojových jazyků a M cílových architektur. Bez použití mezikódu by bylo nutné implementovat $N * M$ překladačů. Pokud ale mezikód použijeme, omezíme tento počet pouze na N front-endových a M back-endových překladačů.



Obrázek 3.2: Porovnání překladu při použití vnitřní reprezentace a bez ní [2]

.

Mezikód použitý v projetu LLVM se nazývá LLVM IR. Tento silně typovaný nízko úrovňový jazyk je založen na architektuře s redukovanou instrukční sadou (RISC). Avšak místo pevného počtu registrů používá neomezené množství dočasných proměnných za použití formy *Static single assignment* (SSA) [9]. Tedy každé proměnné je přiřazena hodnota právě jednou a to před tím, než je použita. Tato forma je výhodná pro analýzu nedosažitelného kódu a vytváření řetězců definice a užití. Jak je vidět na ukázce 3.1, tento kód je na první pohled velmi neefektivní, avšak tento problém při překladu vyřeší optimalizátor. Díky silnému typování ukazatelů je také možné analyzovat přístup k proměnným a tedy dokázat, že proměnné nejsou adresovány mimo rozsah platnosti.

LLVM IR lze zapsat ve třech izomorfních formách:

1. v textové podobě jako člověkem čitelný kód
2. kód uložený v paměti při překladu
3. v binární podobě na disku, kterou využívají JIT překladače

```
1 %1 = %x
2 %2 = 21
3 %3 = %1 + %2
4 %y = %3
```

Ukázka 3.1: Zápis pseudokódu využívající SSA formu

Textová podoba

K získání LLVM IR v textové podobě můžeme použít přepínač `-emit-llvm` při překladu C/C++ zdrojového programu pomocí nástroje Clang. V ukázce 3.2 vidíme textovou LLVM IR jednoduchého programu *Ahoj světe*. Lokální identifikátory začínají vždy znakem `%` a obsahují jméno, které bylo definováno ve zdrojovém souboru. Pokud je v daném místě identifikátor překryt jiným, se stejným jménem, pak se na konec přidává číslo indexované od jedničky. Pokud se jedná o dočasnou proměnnou, místo jména se využívá číslo opět

indexované od jedničky. Názvy globálních identifikátorů pak začínají znakem `@`. Definice funkcí se skládají z klíčového slova *define* následovaného návratovým typem, názvem a výčtem parametrů, včetně jejich typů. Volitelně je také možné použít další atributy jako jsou konvence volání nebo viditelnost funkce.

```
1  @.str = private unnamed_addr constant [13 x i8] c"hello!\0A\00"
2
3  declare i32 @puts(i8* nocapture) nounwind
4
5  define i32 @main() {    ; i32()*
6          %1 = getelementptr [13 x i8]* @.str, i64 0, i64 0
7          call i32 @puts(i8* %1)
8          ret i32 0
9  }
10
11 !1 = metadata !{i32 42}
12 !foo = !{!1, !1}
```

Ukázka 3.2: Program *Ahoj světe* zapsaný v LLVM IR.

Podoba uložená v paměti

Program v LLVM je při překladu rozdělen do modulů, které reprezentuje třída *Module*. Každý modul pak obsahuje svou tabulku symbolů, závislé moduly a seznam globálních proměnných a funkcí. Funkce jsou následně implementovány v instancích třídy *Function*, které představují graf toku řízení.

Uzly v tomto grafu jsou základní bloky (třída *BasicBlock*) a představují speciální typ instrukce návěští. Veškeré skokové instrukce odkazují vždy na toto návěští, respektive začátek základního bloku.

Základní bloky se skládají z posloupnosti instrukcí (třída *Instruction*). Každá instrukce má svůj typ a obsahuje vektor operandů.

3.2 LLVM Core

LLVM Core je soubor knihoven pro implementaci back-endové části překladače. V první fázi proběhne analýza kódu, který byl dodán front-endem, tedy ve formátu *LLVM IR*. Během této fáze se vytváří graf toku řízení a také graf volání. Následně proběhne generování cílového kódu, tedy převod LLVM IR do instrukční sady cílové architektury procesoru. LLVM Core podporuje většinu dnes používaných architektur, jako jsou X86, X86-64, PowerPC, NVPTX, ARM a mnoho dalších.

3.3 Clang

Projekt *Clang* je znám hlavně díky svému front-endovému překladači jazyků C, C++, Objective-C, Objective-C++, OpenMP, OpenCL a CUDA, který využívá právě LLVM back-end. Nicméně tento projekt obsahuje také množství nástrojů na statickou analýzu kódu. Cílem projektu je plně nahradit stávající nejpoužívanější překladač GCC (GNU Compiler

Collection). Nespornou výhodou pro koncové uživatele je kompatibilita s GCC, tedy sdílí s ním téměř totožné rozhraní.

Důvodů pro vytvoření nového front-endového překladače, místo použití nebo modifikace GCC bylo několik. GCC je implementován jako monolitický překladač, tudíž je velmi obtížné jej integrovat do jiných nástrojů. Na druhou stranu Clang již od počátku se snaží o rozdělení jednotlivých částí do oddělených knihoven, které mohou být znovu použity pro jiné účely, jako jsou statické analyzátory, nástroje na refaktorizaci, interaktivní vývojové prostředí nebo generaci kódu. Clang byl také navržen tak, aby poskytoval jasnou a stručnou diagnostiku při překladu formou varování a chyb. Jelikož si během fáze překladu udržuje více dat z předchozích kroků, než GCC, tak v mnoha případech dodá přesnější informace o vadě ve zdrojovém kódu.

Clang je vydáván pod stejnou svobodnou licencí jako LLVM, tedy UIUC. Tato licence spadá do rodiny BSD licencí, které jsou použitelné i pro komerční účely. Jelikož je GCC vydáváno pod Všeobecnou veřejnou licencí GNU (angl. *GNU General Public License*), může být použito pouze v projektu, který je vydáván pod stejnou licencí.

Clang AST

Front-endová část překladače využívá pro svou vnitřní reprezentaci zdrojového kódu abstraktní syntaktický strom (angl. *Abstract Syntax Tree*, dále jen *AST*). Vnitřní uzly v tomto stromu reprezentují operátory, zatímco listy představují operandy. AST je produktem syntaktické analýzy a tedy již neobsahuje některé detaily vyskytující se v původních zdrojových kódech. Tím tvoří mezistupeň překladu mezi jazykem zdrojového kódu a LLVM IR.

Clang pomocí rozhraní popsaných v následující sekci umožňuje velké množství operací nad již sestaveným stromem. Díky tomu lze daleko jednodušeji implementovat nástroje analyzující nebo upravující tento strom. Takto vytvořené nástroje naleznou uplatnění především při testování, jelikož lze modifikovat běh programu bez zásahu do zdrojových kódů.

Clang umožňuje vypsat AST v člověku čitelné formě, která syntaxí připomíná jazyk C++, pomocí přepínače `-ast-dump`. Příklad takto získaného AST z primitivního programu je uveden v ukázce 3.3.

```
$ cat  tests/c/test025.c
int foo(int a){
    if (a > 0) return foo(a - 1);
    else return 0;
}

$ clang -Xclang -ast-dump -fsyntax-only tests/c/test025.c
`-FunctionDecl 0x2c45540 <tests/c/test025.c:1:1, line:4:1> line:1:5 referenced foo 'int (int)'
  |-ParmVarDecl 0x2c45478 <col:9, col:13> col:13 used a 'int'
  `-CompoundStmt 0x2c45860 <col:15, line:4:1>
    `-IfStmt 0x2c45830 <line:2:5, line:3:17>
      |-<<<NULL>>>
      |-BinaryOperator 0x2c45690 <line:2:9, col:13> 'int' '>'
      | |-ImplicitCastExpr 0x2c45678 <col:9> 'int' <LValueToRValue>
      | | `-DeclRefExpr 0x2c45630 <col:9> 'int' lvalue ParmVar 0x2c45478 'a' 'int'
      | `-IntegerLiteral 0x2c45658 <col:13> 'int' 0
      |-ReturnStmt 0x2c457e0 <col:16, col:32>
      | `-CallExpr 0x2c457b0 <col:23, col:32> 'int'
      |   |-ImplicitCastExpr 0x2c45798 <col:23> 'int (*)(int)' <FunctionToPointerDecay>
      |   | `-DeclRefExpr 0x2c456b8 <col:23> 'int (int)' Function 0x2c45540 'foo' 'int (int)'
      |   `-BinaryOperator 0x2c45740 <col:27, col:31> 'int' '-'
      |     |-ImplicitCastExpr 0x2c45728 <col:27> 'int' <LValueToRValue>
      |     | `-DeclRefExpr 0x2c456e0 <col:27> 'int' lvalue ParmVar 0x2c45478 'a' 'int'
      |     `-IntegerLiteral 0x2c45708 <col:31> 'int' 1
      `-ReturnStmt 0x2c45818 <line:3:10, col:17>
        `-IntegerLiteral 0x2c457f8 <col:17> 'int' 0
```

Ukázka 3.3: Abstraktní syntaktický strom primitivního programu.

Rozhraní Clangu

V současné době Clang nabízí tři různá rozhraní pro psaní nástrojů.

LibClang nabízí nejjednodušší rozhraní pro psaní syntaktického analyzátoru nad knihovnou Clang. Poskytuje velkou míru abstrakce pro práci s abstraktním syntaktickým stromem a nevyžaduje po programátorovi podrobnou znalost vnitřní implementace knihoven Clangu. Toto rozhraní – psané v jazyce C – je stabilní, tedy nemění se s novými verzemi. Taktéž umožňuje psaní nástrojů v jazyce Python pomocí *Python bindings*. Bohužel velká míra abstrakce sebou přináší omezenou kontrolu nad AST.

Clang zásuvný modul (angl. *Clang plugin*) je nástroj pro vytváření dynamicky alokovaných knihoven, pomocí nichž je možné přidat akci nad AST během překladu.

LibTooling je rozhraní, které lze využít ke psaní nástrojů s využitím knihovny Clang, které jsou však nezávislé na samotném překladači. Projekt Clang zastřešuje také několik nástrojů, které byly takto vytvořeny. Například syntaktický analyzátor *clang-check* nebo nástroj na automatické formátování *clang-format*. Využití rozhraní LibTooling je vhodné, pokud chceme umožnit spuštění nástroje pouze na omezené podmnožině zdrojových souborů. Taktéž nabízí plnou kontrolu nad AST. Nevýhodou pak jsou větší nároky na znalosti programátora ohledně vnitřní implementace Clangu a také nutnost použít jazyk *C++*.

Vstupním bodem pro LibTooling nástroje bývá většinou rozhraní *FrontendAction*, které umožňuje provedení konkrétní akce. Příkladem může být rozhraní *ASTFrontendAction*, které zpřístupňuje AST, nebo *EmitCodeGenOnlyAction*, jež dovoluje číst a modifikovat mezikód v LLVM IR.

Kapitola 4

Návrh a implementační detaily

V této kapitole bude popsána tvorba nástroje, který je předmětem této bakalářské práce. Nejprve v sekci 4.1 si nastíníme architekturu programu. Následně v sekci 4.3 popíšeme, jak byl použit nástroj Clang 3.3. V druhé části kapitoly si pak popíšeme samotnou extrakci grafu toku řízení, řešení některých problémů při implementaci a formátování výstupu programu.

4.1 Architektura programu

Program je psán v jazyku C++ v normě C++11 a využívá knihovnu LLVM a Clang ve verzi *3.8*. Vzhledem k měnícímu se API nelze zaručit, že program bude kompatibilní s jinými verzemi.

Samotný program se pak dělí do tří částí. Hlavní část programu definovaná v souboru `main.cpp` se stará o řízení programu, zpracování parametrů a napojení na knihovnu Clang. O průchod jednotlivými moduly a extrakci dat pomocí knihovny LLVM se stará třída `ModuleMeta`, definovaná ve zdrojovém souboru `cfg_gen.cpp`. Převod z vnitřní reprezentace dat do výstupního formátu je zastřešen tovární metodou `FormatFactory` ze zdrojového souboru `formater.cpp`, která je založená na návrhovém vzoru *Továrna*. Konkrétní výstupní formát lze specifikovat pomocí příkazové řádky. Avšak pro účely projektu Testos a této bakalářské práce byl zatím implementován jediný výstupní formát JSON, specifikovaný v sekci 4.7.

4.2 Využití knihovny Clang

K napojení knihovny Clang jsem využil rozhraní LibTooling. Nejprve jsou pomocí třídy `OptionsParser` propagované argumenty příkazové řádky do knihovny Clang. Díky tomu tento nástroj přijímá stejné argumenty pro překlad jako front-endový překladač Clang. K těmto parametrům je navíc explicitně přidán přepínač `-g`, který zajistí přítomnost ladících informací jako jsou třeba programové lokace.

K přístupu k programu reprezentovaném v mezikódu LLVM IR jsem použil vlastní rozhraní `EmitMetadataAction`, které je odvozené od `EmitLLVMAction`. Funkcionalita rozhraní LibToolingu je podrobněji popsána v sekci 3.3. Tímto způsobem jsem k překladu přidal callback metodu, která je volána pro každý modul přeložený do mezikódu LLVM IR.

4.3 Extrakce grafů toku řízení

Z důvodu pozdějšího rozšíření projektu nejsou při průchodu modulem jednotlivé elementy rovnou vypisovány na výstup, ale vytváří se strom vnitřní reprezentace. Kořenovým elementem je třída `ModuleMeta`, která stejně jako všechny ostatní elementy tohoto stromu implementuje virtuální třídu `MetadataWrapper`. Jelikož se počítá s dalším rozšířením, ve vnitřní reprezentaci jsou uložena i data, která nejsou nutná pro výstup ve formátu JSON.

Modul v LLVM v případě jazyků C a C++ většinou reprezentuje právě jeden zdrojový soubor. Přístup k této reprezentaci v LLVM je možný v metodě zpětného volání `EndSourceFileAction` třídy `EmitLLVMAction`, pomocí veřejné metody `takeModule`. Takto získaný modul obsahuje vektor globálních proměnných, které jsou zpracovávány totožným způsobem jako instrukce popsané níže, a vektor funkcí.

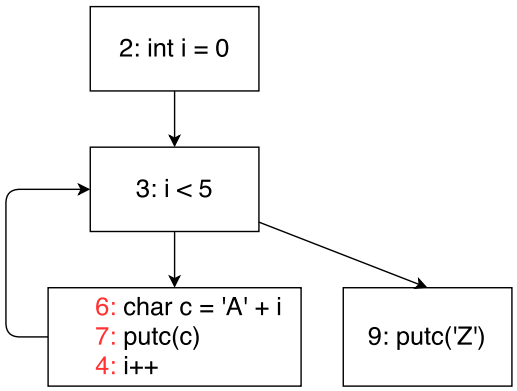
Funkce jsou v LLVM reprezentovány třídou `llvm::Function`, které ve vnitřní reprezentaci nástroje odpovídá třída `CFGMeta`. LLVM pro každou funkci vytváří samostatný graf toku řízení. `CFGMeta` tedy obsahuje ladící informace, jako jsou název funkce, název zdrojového souboru a množinu základních bloků, ze kterých se skládá.

Nad množinou základních bloků každého CFG je nutné iterovat dvakrát. Při prvním průchodu se vytvoří pro každý základní blok instance třídy `BasicBlockMeta` a přidá se záznam do mapy, která jako klíče obsahuje ukazatele na základní bloky LLVM a jako hodnoty ukazatele na instance třídy `BasicBlockMeta`. V druhém průchodu pak se podle záznamů v této mapě korektně nastaví ukazatele na předchůdce a následníky tříd `BasicBlockMeta`, na základě znalosti ukazatelů na předchůdce a následníky ze základních bloků LLVM.

Kromě těchto informací nás u funkce reprezentované pomocí CFG také zajímá množina počátečních a koncových základních bloků. Tyto informace nejsou přímo uložené v `llvm::Function`, nicméně je triviální tyto informace získat prostou iterací nad základními bloky, jež obsahuje. Vybíráme ty základní bloky, které nemají předchůdce (tedy jsou počáteční) a nebo nemají následníka (tedy jsou ukončující).

Základní bloky jsou reprezentovány třídou `BasicBlockMeta`. Základní bloky obsahují uspořádaný seznam instrukcí, přičemž platí, že první instrukce musí být typu návěští a poslední instrukce musí být skoková. Zároveň platí, že skokové instrukce ani návěští nesmí být na jiných než těchto specifikovaných místech, jinak by byla porušena definice základního bloku 2.

Při určování programové lokace základních bloků jsem narazil na problém, že pořadí instrukcí v základním bloku nemusí odpovídat pořadí instrukcí ve zdrojových souborech. Toto lze vidět na příkladu 4.1 jednoduchého for-cyklu. Není tedy možné vzít pouze lokaci první a poslední instrukce, ale je nutné stanovit minimum a maximum ze všech obsažených instrukcí.




```
1  void foo(){
2    for(int i = 0;
3        i < 5;
4        i ++
5    ){
6        char c = 'A' + i;
7        putc(c);
8    }
9    putc('Z');
10 }
```

Ukázka 4.1: For-cyklus v jazyce C s rozdělením do základních bloků. Červenou barvou je zvýrazněná odchylka pořadí instrukcí ve zdrojovém souboru od pořadí v základním bloku.

Instance třídy `BasicBlockMeta` také obsahují informace o definicích a užití proměnných a volaných funkcích za účelem sestavení řetězců definicí a užití popsaných v sekci 2.2. Clang během překladu tyto řetězce sestavuje a je možné k nim jednoduše přistoupit. Avšak je komplikované zpětně provázat sestavené základní bloky s těmito řetězci. Z toho důvodu byl zvolen přístup získání případu definic a užití analýzou instrukcí v základním bloku. Zkoumá se jejich typ a operandy. Pokud je do pojmenované proměnné přiřazena hodnota pomocí operátoru `=` nebo slouží jako cíl operace pro práci s pamětí, jedná se o *definici*. Pokud proměnná slouží jako operand v instrukci, jedná se o *užití*.

Obdobných způsobem jsou také zjišťovány funkce, které jsou volány v rámci základního bloku. Při volání funkce pomocí instrukce `call` je název funkce předáván jako první operand.

Instrukce jsou reprezentovány třídou `InstructionMeta`. Každá instrukce má svůj typ, seznam operandů a programovou lokaci. Samotná instrukce je pak uložena jako textový řetězec reprezentující člověkem čitelnou formu LLVM IR.

4.4 Zpracovávání knihoven

Při testování tohoto nástroje na zdrojových souborech reálných aplikací jsem narazil na problém, že při spuštění nad knihovnami výstup neobsahuje žádné instrukce. Clang v tomto případě vytvoří pouze prázdný modul, který neobsahuje žádné grafy toku řízení. Při zpracovávání knihovních funkcí a tříd Clang sice vytvoří abstraktní syntaktický strom, avšak pokud nejsou volány, pak negeneruje mezikód v LLVM IR. Toto chování se dá změnit přímou modifikací zdrojových souborů Clangu. Nicméně tento přístup je nevhodný, protože je nutné přeložit Clang ve svém prostředí a s přechodem na novou verzi budou tyto změny ztraceny.

Alternativním přístupem je vytvoření jednoduchého programu, který využívá zkoumané knihovny. Pokud chceme tímto nástrojem vygenerovat CFG například pro knihovnu *nlohmann/json*, která je mimo jiné využita i v tomto projektu, je nutné vytvořit následující minimální program, jak je uvedeno v ukázce 4.2.

```
1 #include "json.hpp" // inspected library
2
3 using json = nlohmann::json;
4
5 int main(int argc, char *argv[])(
6     json j;
7     j["pi"] = 3.141;
8     return 0;
9 )
```

Ukázka 4.2: Jednoduchý program využívající knihovnu *nlohmann/json* a umožnující generování grafu toku řízení.

4.5 Překlad nástroje

Projekt LLVM využívá multiplatformní systém pro automatizaci překladu s názvem *CMake*. Z důvodů minimalizace závislostí tohoto nástroje jsem se rozhodl tento systém nepoužít a vytvořit nezávislý *Makefile*, který obsahuje postup pro překlad nástroje. Ke generování správných přepínačů a cest ke knihovnám nutným pro překlad se využívá pomocný program s názvem `llvm-config`, který je součástí balíku `llvm-dev`. Tento balík – a další, jež jsou vyčteny v příloze A – je i tak nutný k překladu nástroje, tedy nevytváří se nová závislost.

Makefile, který naleznete na přiloženém DVD B společně se zdrojovými soubory, je do velké míry inspirován projektem Eli Benderského [5], protože oficiální prameny LLVM nenabízí žádný návod bez použití programu CMake.

4.6 Zahrnutí základních knihoven

Téměř každý program napsaný v jazycích C/C++ využívá základní knihovny. Pokud však budeme chtít tímto nástrojem vygenerovat graf toku řízení ze zdrojových souborů, které tyto základní knihovny využívá, může nastat problém, že tyto knihovny nebudou nalezeny. Tento problém je způsoben tím, že překladač Clang, na kterém je postaven tento nástroj, hledá základní knihovny na relativní cestě `../lib/clang/3.8.0` vůči sobě samému [18]. Správnou cestu ke knihovnám lze na základě případu užití každého uživatele nastavit třemi různými způsoby. V následujících příkladech předpokládáme standardní Unixový systém se standardními knihovnami v adresáři `/usr/include`.

Parametr příkazové řádky - Při spuštění nástroje lze přidat cestu ke standardním knihovnám v systému přepínačem `-I /usr/include`. Podrobnou ukázku můžete nalézt v instalačním manuálu v příloze A.

Symbolický odkaz - Aby uživatel nemusel při každém spuštění přidávat tento přepínač, je možné vytvořit symbolický odkaz z adresy `../lib` ke standardním knihovnám. Toho docílíme v terminálu příkazem `ln -s /usr/include ../lib/clang/3.8.0/include`, pokud se nacházíme ve stejné složce, jako přeložený nástroj.

Přesun nástroje - Taktéž můžeme přesunout nástroj do složky `/usr/bin`. Tedy do složky, kde se nachází většina standardních programů v Unixovém systému. Tímto docílíme toho, že knihovny budou na očekávaném místě, vůči tomuto nástroji.

```
1  #include <iostream>
2  #include "json.hpp"
3
4  using json = nlohmann::json;
5
6  int main(int argc, char **argv){
7    json a = {
8      {"item", "Vaclav"},
9      {"list", {1,2,3}},
10     {"boolean", true},
11     {"some_number", 42},
12     {"object": {
13         {"key1", "value1"},
14         {"key2", "value2"}
15     }}
16   };
17   a["item"] = true;
18   json b = {{"key3", "value3"}};
19   a["object"]["map"] = b;
20   std::cout << a;
21   return 0;
22 }
```

Ukázka 4.3: Ukázka práce s knihovnou nlohmann/json v jazyce C++.

4.7 Výstupní formát

Jak již bylo nastíněno v sekci o architektuře tohoto nástroje 4.1, výstup programu zastřešuje tovární metoda `FormatFactory`, která vytváří instance třídy implementující rozhraní `Formater`.

V současné době jedinou třídou implementující toto rozhraní je třída `JsonGen`. Účelem této třídy je převést vnitřní reprezentaci vytvořenou pomocí `ModuleMeta` do formátu JSON.

K tomu byla využita knihovna *JSON for Modern C++* od Niels Lohmanna [13]. Důvodem k použití právě této knihovny bylo především její efektivní využívání paměti a vysoká rychlost. Tato knihovna je také velmi dobře testována a splňuje 100% pokrytí řádků kódu [12]. Knihovna je šířená jako jediný hlavičkový soubor, který nemá žádné závislosti na další projekty, její integrace je tedy velmi jednoduchá. Pro práci s JSON objekty definuje intuitivní syntaxi, která je velmi podobná jazyku Python, jak je vidět na ukázce 4.3. *JSON for Modern C++* je šířen pod MIT licencí, tedy je možné ji použít i ke komerčním účelům.

Třída `JsonGen` tedy prochází rekurzivně stromovou strukturu uloženou v `ModuleMeta` a pro každý typ uzlu (modul, funkce, základní blok, instrukce a programová lokace) definuje metodu, která jej převede do formátu JSON. Následně tento JSON převede na textový řetězec, aby byla zajištěna kompatibilita s ostatními třídami implementující `Formater` roz-

hraní. Výpis na zvolený datový tok se provádí v souboru `main.cpp`. Datový tok lze zvolit pomocí přepínače příkazové řádky. Pokud tak uživatel neučiní, implicitně je použit standardní výstup.

JSON schéma

Návrh schématu pro výstup extraktorů grafů toku řízení ze zdrojových kódů a zároveň vstupu pro program vyhledávání cest na základě pokrytí jsme společně s dalšími členy skupiny Testos 1.2 sestavili na samotném začátku práce na tomto projektu. Díky tomu jsme mohli každý nezávisle pracovat na svoji části, protože bylo jasně definované rozhraní. Postupem času tento návrh prošel drobnými změnami a vznikl požadavek jej standardizovat. Rozhodl jsem se tedy využít projektu *JSON Schema* [11], které pod záštitou organizace IETF [10] definuje standard, jak vypadají schémata pro validaci JSON dokumentů. V současné době existuje řada validátorů založených na tomto standardu. Jedním z nejpoužívanějších je validátor *JSON spec* [3] napsaný v jazyce Python a šířený pod svobodnou licencí.

Sestavil jsem tedy schéma, díky němuž se velmi usnadnila syntaktická validace výstupního JSON dokumentu. Z důvodu délky tohoto schématu nebylo vhodné jej zahrnout přímo v této technické zprávě. Jeho plné znění můžete nalézt na přiloženém DVD B v souboru `schema.json` v kořenovém adresáři.

V ukázce 4.4 je znázorněna malá část schématu validující JSON, který popisuje LLVM instrukci. Následující ukázka 4.5 pak představuje ukázkový JSON dokument validní vůči tomuto schématu. Informace o programové lokaci jsou nahrazeny textovým řetězcem `"PLOC"` za účelem zkrácení ukázky. Jak je vidět, schéma pro validaci JSON dokumentu je taktéž psáno v JSON formátu. Hodnoty klíčů `"title"` a `"description"` mají pouze informační a dokumentační charakter. Hodnota klíče `"type"` udává, jakého typu musí být entita. V případě LLVM instrukce se jedná o objekt.

Další možné typy jsou řetězec, číslo, vektor, logická hodnota a null. V případě objektu se definují možné klíče v `"properties"`. Pro každou hodnotu takového klíče lze rekurzivně specifikovat nové schéma se stejnými parametry. Pomocí vlastnosti `"required"` je možné nastavit pole povinných klíčů, které objekt musí mít. JSON schéma také umožňuje omezenou sadu podmíněných vlastností, jako je třeba `"oneOf"`. Tato vlastnost je splněna, pokud platí právě jedno z uvedených pravidel.

```
1 {
2     "ploc": {
3         "col_begin": 15,
4         "line_min": 2,
5         "source_file": "test001.c"
6      },
7     "raw_llvm": "%3 = add nsw i32 %2, 1, !dbg !17"
8 }
```

Ukázka 4.5: Příklad JSON dokumentu znázorňující LLVM instrukci.

```
{
  "title": "Instruction",
  "description": "Information about specific instruction.",
  "type":"object",
  "oneOf": [
    {
      "required":["raw_llvm"],
    },
    {
      "required":["opcode, operands"],
    },
  ],
  "properties":{
    "raw_llvm":{
        "type": "string"
    },
    "ploc":{
        "description": "Program location",
        "type": "PLOC"
    },
    "meta": {
        "description": "Custom metadata",
        "type": "object"
    },
    "opcode" : {
        "type": "object"
    },
    "operands" : {
        "type": "array",
        "items": { "type": "string" }
    }
  }
}
```

Ukázka 4.4: Zjednodušená část JSON schématu k validaci LLVM instrukcí.

Kapitola 5

Ověření funkcionality

Správná funkcionalita programu byla ověřena na sadě zdrojových souborů v jazycích C a C++. Tyto soubory můžete nalézt na přiloženém paměťovém médiu v adresáři `tests`. V závislosti na programovacím jazyce, ve kterém byly testované programy napsány, obsahují jejich zdrojové soubory příponu `.c` nebo `.cpp`. Očekávaný výstup ve formátu JSON pak s tímto souborem sdílí *vlastní jméno*, ale obsahuje příponu `.json`.

Testování extrakce grafů toku řízení

První část testovací sady pokrývá všechny základní datové typy obou jazyků. V této části byl kontrolován korektní překlad instrukcí do mezikódu LLVM.

Druhá část testovací sady se zaměřuje na všechny řídící konstrukce, které poskytují tyto jazyky. Testy se zaměřují na strukturu vygenerovaných grafu toku řízení, tedy rozdělení kódu do základních bloků a přechody mezi nimi.

Jak již bylo zmíněno v kapitole o testování 2, vytváření testovacích požadavků na základě kritérií pokrytí řídících toků je využíváno především pro větší projekty. Přiložená testovací sada tedy ověřuje funkčnost programu, ale nereflektuje typický případ užití. Za tímto účelem je v příloze A obsažen návod na extrakci grafů toku řízení pomocí tohoto nástroje z vlastních zdrojových souborů.

Testování formátu výstupu

K validaci výstupního JSON dokumentu 4.7 byl použit nástroj *JSON Schema Lint*, který je dostupný online [14]. Díky tomu byla nejen ověřena správnost JSON schématu, ale také to, že výstup implementovaného nástroje je validní vůči tomuto schématu.

Těmito testy bylo dokázáno, že nástroj odpovídá požadavkům testovací platformy Testos, tedy i účelu, za nímž byl nástroj vytvořen.

Kapitola 6

Závěr

Cílem této práce bylo nastudovat architekturu překladače Clang a navrhnout způsob extrakce grafů toků řízení pomocí tohoto překladače pro účely platformy Testos. Následně tento návrh implementovat a ověřit funkcionalitu sadou testů pokrývající všechny základní řídící konstrukce a primitivní datové typy jazyků C a C++.

Vytyčené cíle se podařilo splnit a navržený nástroj byl implementován v jazyce C++ pomocí rozhraní LibTooling překladače Clang. Díky tomu nástroj umožňuje extrahovat grafy toků řízení ze všech zdrojových souborů jazyků C a C++, které je schopen překladač Clang přeložit. Výstupní formát odpovídá formálním požadavkům platformy Testos, avšak samotná integrace do této platformy nebyla v rámci této bakalářské práce uskutečněna.

Díky této práci jsem se dozvěděl mnohé o architektuře překladačů a také si rozšířil obzory v oblasti testování. Rád bych pokračoval v rozvíjení toho projektu, především tedy v integraci tohoto nástroje do platformy Testos. V rámci budoucího vývoje se také naskýtá příležitost rozšířit extrakci grafu toků řízení o další jazyky, jež umožňuje Clang překládat, kterými jsou například *Objective C/C++*.

Literatura

[1] Ammann, P.: *Introduction to Software Testing*. Cambridge University Press, 2008, ISBN 978-0-512-88038-1.

[2] Appel, A.: *Modern Compiler Implementation in ML*. Cambridge University Press, 1998, ISBN 0-521-60764-7.

[3] Barbosa, X.: JSON Spec. 2017, [Online; navštíveno 08.05.2017].
URL http://py.errorist.io/json-spec

[4] Beizer, B.: *Software Testing Techniques*. Van Nostrand Reinhold Co., 1990, ISBN 0-442-20672-0.

[5] Bendersky, E.: LLVM Clang samples. 2017, [Online; navštíveno 08.05.2017].
URL https://github.com/eliben/llvm-clang-samples

[6] Boehm, B.: *Software Engineering Economics*. Prentice-Hall, 1981, ISBN 0-13-822122-7.

[7] Docker, Inc.: Get Started with Docker. 2017, [Online; navštíveno 08.05.2017].
URL https://www.docker.com/

[8] Glass, R.: *Facts and Fallacies of Software Engineering*. Addison Wesley, 2002, ISBN 0-321-11742-5.

[9] GNU Project: GNU Compiler Collection: Static Single Assignment. 2017, [Online; navštíveno 15.05.2017].
URL https://gcc.gnu.org/onlinedocs/gccint/SSA.html

[10] Internet Society: IETF. 2017, [Online; navštíveno 08.05.2017].
URL https://www.ietf.org/

[11] json-schema organisation: JSON Schema. 2017, [Online; navštíveno 08.05.2017].
URL http://json-schema.org/

[12] Lemur Heavy Industries: Coveralls - nlohmann/json. 2017, [Online; navštíveno 08.05.2017].
URL https://coveralls.io/github/nlohmann/jsonn

[13] Lohmann, N.: JSON for Modern C++. 2017, [Online; navštíveno 08.05.2017].
URL https://github.com/nlohmann/json

[14] Maynard, N.: JSON Schema Lint. 2017, [Online; navštíveno 08.05.2017].
URL https://jsonschemalint.com/

[15] Reactive System, Inc.: Achieving ISO 26262 Compliance with Reactis. 2017, [Online; navštíveno 10.05.2017].
URL http://www.reactive-systems.com/iso-26262-asil.html

[16] Sečkařová, P.: Java2CFG. FIT VUT v Brně, 2017, [Online; navštíveno 14.05.2017].
URL https://pajda.fit.vutbr.cz/testos/java2cfg

[17] Skupina Testos: Domovská stránka projektu Testos. FIT VUT v Brně, 2017, [Online; navštíveno 14.05.2017].
URL http://testos.org

[18] The Clang Team: Clang - Frequently Asked Questions. 2017, [Online; navštíveno 08.05.2017].
URL https://clang.llvm.org/docs/FAQ.html

[19] The Clang Team: Static analysis tools: using Clang in CppDepend. 2017, [Online; navštíveno 08.05.2017].
URL
http://blog.llvm.org/2013/04/static-analysis-tools-using-clang-in.html

[20] Vít, R.: Control Flow Graph Query Engine. FIT VUT v Brně, 2017, [Online; navštíveno 14.05.2017].
URL https://pajda.fit.vutbr.cz/testos/cfgqe

Přílohy

Příloha A

Instalační manuál

Nástroj lze spustit dvěma různými způsoby. První způsob popisuje spuštění přímo v hostitelském systému. K tomu jsou nutné některé balíky, které jsou však dostupné v téměř všech linuxových distribucích pomocí zabudovaných správců balíčků (apt, yum, dnf, brew a další) Druhou možností je spustit docker kontejner z obrazu na přiloženém DVD, ve kterém jsou všechny závislosti již vyřešeny a nástroj je přeložen a připraven ke spuštění.

Překlad v hostitelském systému

V prvním kroku je nutné nejprve získat zdrojové soubory. Ty je možné nakopírovat z přiloženého DVD nebo stáhnout z veřejného git repositáře.

```
git clone git@pajda.fit.vutbr.cz:testos/cpp2cfg.git
```

Následně pomocí správce balíků nainstalujeme potřebné závislosti, kterými jsou:

- g++
- clang-3.8
- llvm
- llvm-3.8
- llvm-3.8-dev
- libclang-3.8-dev
- libz-dev

V závislosti na používané distribuci unixového systému můžeme použít jeden z těchto příkazů:

```
# FEDORA >=18, RHEL >=7, CentOS >=7
dnf install g++ clang-3.8 llvm llvm-3.8 llvm-3.8-dev libclang-3.8-dev libz-dev
# FEDORA <18, RHEL <7, CentOS <7
yum install g++ clang-3.8 llvm llvm-3.8 llvm-3.8-dev libclang-3.8-dev libz-dev
# ubuntu, debian, mint, bash for windows
apt install g++ clang-3.8 llvm llvm-3.8 llvm-3.8-dev libclang-3.8-dev libz-dev
# OSX
brew install g++ clang-3.8 llvm llvm-3.8 llvm-3.8-dev libclang-3.8-dev libz-dev
```

Pokud využíváte linuxové distribuce společnosti Red Hat (RHEL, Fedora, CentOS) je nutné zajistit, aby ve souboru `/etc/ld.so.conf` byl řádek obsahující `/usr/local/lib`. Tedy aby se dynamicky linkované knihovny vyhledávaly i v adresáři `/usr/local/lib`. Pokud provedete nějakou změnu v tomto souboru, je nutné spustit příkaz `sudo ldconfig`, aby se změny aplikovaly.

Když máme vyřešené všechny závislosti, samotný překlad provedeme příkazem `make` v adresáři, kde se nachází dodaný soubor `Makefile`.

Volitelné - Pro snadnější spuštění programu je vhodné nastavit cestu ke standardním knihovnám. Toho docílíme příkazem:

```
ln -s /usr/include ../lib/clang/3.8.0/include
```

Spuštění docker kontejneru

Jediným požadavkem ke spuštění tohoto nástroje pomocí dockeru je mít nainstalovaný program *docker* v hostitelském systému. Postup instalace v závislosti na operačním systému můžete nalézt zde [7].

Následně pak stačí importovat obraz `cfg2cpp` z přiloženého DVD a následně spustit. Toho docílíme v terminálu těmito příkazy:

```
docker include cfg2cpp.tar
docker run --rm -ti cfg2cpp bash
```

Spuštění nástroje

Nezávisle na tom, jestli spouštíte nástroj přímo v systému, nebo docker kontejneru, jsou následující kroky shodné.

Parametry spuštění programu mají následující schéma. Nejprve definujeme přepínače pro samotný nástroj. Pomocí `-format JSON` můžeme vybrat formát výstupu a nebo využitím `-output cfg.json` přesměrovat výstup do souboru místo standardního výstupu.

Následuje seznam cest zkoumaných zdrojových souborů, ze kterých chceme extrahovat grafy toku řízení. Parametry pro překlad jsou shodné s překladačem Clang a jsou oddělené dvěma pomlčkami. Můžeme například zvolit normu jazyka pomocí `-std=C++11` nebo přidat cestu ke standardním knihovnám za použití `-I /usr/include`.

```
./build/cfg-gen [tool-options] <source0> [... <sourceN>] -- [clang-options]

##příklady
# extrakce CFG z jazyka C, přesměrování výstupu do /tmp/out.json
#a nastavena cesta ke knihovnám
./build/cfg-gen -output=/tmp/out.json tests/c/test001.c \
-- std=c99 -I /usr/include

# extrakce CFG z jazyka C++
./build/cfg-gen tests/cpp/test001.cpp -- std=c++11

# explicitně specifikovaný výstupní formát
./build/cfg-gen -formater=JSON tests/cpp/test001.cpp -- std=c++11
```

```
# příkaz k vygenerování CFG ze zdrojových
# souborů tohoto nástroje
CFGGENOPTS=$(llvm-config --cxxflags |
  sed 's/ \-W[-a-zA-Z]*//g' |
  sed 's/ -pedantic//' |
  sed 's/ -fno-exceptions//'
)

LDOPTIONS=$(llvm-config --ldflags --libs --system-libs)

./build/cfg-gen  src/main.cpp src/cfg_gen.cpp src/formater.cpp \
src/external_lib/json.hpp -- -O0 -g  $CFGGENOPTS \
-Wl,--start-group \
-lclangAST \
-lclangASTMatchers \
-lclangAnalysis \
-lclangBasic \
-lclangCodeGen \
-lclangDriver \
-lclangEdit \
-lclangFrontend \
-lclangFrontendTool \
-lclangLex \
-lclangParse \
-lclangSema \
-lclangEdit \
-lclangRewrite \
-lclangRewriteFrontend \
-lclangStaticAnalyzerFrontend \
-lclangStaticAnalyzerCheckers \
-lclangStaticAnalyzerCore \
-lclangSerialization \
-lclangToolingCore \
-lclangTooling \
-lclangFormat \
-Wl,--end-group \
$LDOPTIONS -o build/cfg-gen  \
-I /usr/lib/llvm-3.8/lib/clang/3.8.0/include
```

Příloha B

Obsah DVD

```
/
├── xkondu00/ ... tato technická zpráva v LaTeX
├── src/ ... zdrojové soubory implementovaného programu v jazyce C++ k
│           extrakci grafu toku řízení
│   ├── external_lib/ ... zdrojové soubory externích knihoven, které nejsou
│                         mnou implementované
│   ├── cfg_gen.cpp
│   ├── cfg_gen.hpp
│   ├── formater.cpp
│   ├── formater.hpp
│   ├── main.cpp
│   └── main.hpp
├── tests/ ... testovací sada
│   ├── c/
│   │   ├── test{001-028}.c
│   │   └── test{001-028}.json
│   └── cpp/
│       ├── test{001-033}.cpp
│       └── test{001-033}.json
├── build/
├── lib/
├── cfg2cpp.tar ... docker image s přeloženým nástrojem a všemi nainstalovanými
│                   závislostmi
├── README.md ... průvodní soubor se základními informacemi o projektu
├── Makefile
├── xkondu00.pdf ... tato technická zpráva ve formátu pdf
└── schema.json ... schéma pro validaci a verifikaci výstupu programu
```