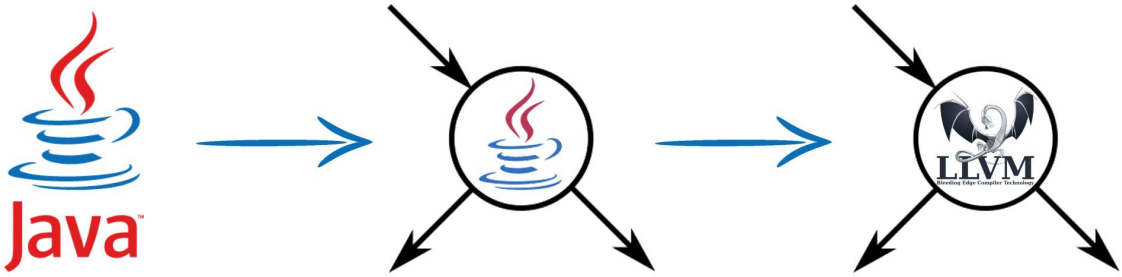
BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2017
http://excel.fit.vutbr.cz

Extrakce grafu toku řízení z bajtkódu Java

Petra Sečkařová*



Abstrakt

Většina analýz vyhodnocujících kvalitu programů je odvozena od grafů toku řízení (Control Flow Graph – CFG). Mezi tyto analýzy patři testování založené na modelech (model based testing), kde jsou grafy používány pro návrh testovacích případů. Ty vznikají z cest vyhledaných v CFG, které jsou později jednotlivě analyzovány jako souvislé sekvence instrukcí. Aby bylo možné další analýzu provádět co nejobecněji, je nutné, aby zkoumané instrukce patřily do některé z obecných instrukčních sad. Tato práce se zabývá extrakcí grafů toku řízení z bajkódu jazyka Java za současného překladu instrukcí do LLVM IR. Přesněji je v první fázi provedena identifikace základních bloků programu a z nich složeny odpovídající CFG. Ve druhé fázi jsou pak jednotlivé instrukce bajtkódu jazyka Java překládány do LLVM IR. Výsledný program dokáže získat grafy toku řízení z libovolného programu v Javě zadaného v jakékoli z jeho nejběžnějších forem (.jar archiv, .java nebo .class soubory). Korektnost extrakce byla testována na kódech obsahujících všechny obvyklé konstrukce jazyka Java, na nichž program funguje spolehlivě bez dalších omezení. Pro další analýzu nad instrukcemi přeloženými do LLVM IR je vhodné nastudovat si hranice a specifika provedeného překladu.

Klíčová slova: Java — Analýza — CFG — JBC

Přiložené materiály: Repozitář projektu — Dokumentace tříd

*xsecka02@stud.fit.vutbr.cz, *Fakulta informačních technologií, Vysoké učení technické v Brně*

1. Úvod

V rámci vznikající platformy Testos, která si za jeden z hlavních cílů klade automatizaci testování softwaru, vznikla potřeba získat z libovolné formy programu v jazyce Java grafy toku řízení s instrukcemi v sadě LLVM IR. Ty mají být později zpracovávány společně s ostatními grafy extrahovanými z ostatních masově používaných programovacích jazyků.

Získáváním CFG z kódů v jazyce Java se zabývá hned několik nástrojů, např. ConFlEx [1], Zhao [2], BytecodeToCfg [3] nebo Dr Garbage Tools [4]. Bohužel většina z těchto projektů je jednoúčelově zaměřená, dnes již neudržovaná, případně jen složitě rozšiřitelná. Tato práce si klade za cíl vytvořit nástroj, který je jednoduchý z hledisek (i) pochopitelnosti pro nové vývojáře, (ii) rozšiřitelnosti pro podporu složitých datových struktur, (iii) udržovatelnosti pro kontinuální vývoj a (iv) integrovatelnosti do platformy Testos.

Navržený překladač pracuje ve dvou fázích. V první je provedena extrakce CFG z Java bajtkódu, k čemuž je využit nástroj `javap`. V další fázi pak probíhá

překlad posloupností instrukcí bajtkódu do odpovídajících sekvencí v LLVM IR. Výsledný graf je po libovolné fázi možné uložit ve formátu JSON používajícím objekty definované platformou Testos[1].

V následující kapitole bude vysvětlen princip extrakce grafů toku řízení. Třetí kapitola je pak věnována základním myšlenkám použitým při překladu jednotlivých sekvencí instrukcí. Kapitola 4 demonstruje použitelnost aktuálního nástroje a posledníkapitola shrnuje dosažené výsledky a popisuje vize do budoucna.

2. Extrakce CFG

Celý proces začíná u bajtkódu, což je binární reprezentace programu napsaného v jazyce Java. Každý zdrojový soubor v tomto jazyce reprezentuje jednu třídu objektů, proto jsou soubory s bajtkódem ukládany s příponou .class. Pokud se jedná o celý projekt, je vytvořen .jar archiv, který obsahuje v adresářové struktuře všechny .class soubory daného projektu. Pokud je programu na vstup zadán přímo zdrojový (.java) soubor, je před fází extrakce potřebný .class soubor získán jeho kompilací.

2.1 Bajtkód jazyka Java a JVM

JVM (Java Virtual Machine), který obstarává provádění binárních programů, je v základu zásobníkový počítač. Kromě zásobníku s operandy využívá také tabulky lokálních proměnných a indexovaný Constant Pool, který obsahuje všechna konstantní data jako jsou textové řetězce, číselné konstanty nebo jména použitých tříd, metod a atributů.

Jednotlivé instrukce tedy typicky pracují nad zásobníkem, připadně kombinací zásobník-proměnná nebo zásobník-pool. Pro předávání parametrů mezi funkcemi jsou použity zásobníkové rámce.

2.2 Využití nástroje `javap`

Binární forma programu je pro člověka typicky nečitelná. Nástroj `javap` dokáže bajtkód z binární formy přeložit zpět do čitelné textové reprezentace, jak je vidět na obrázku 1.

Rozsah tištěných informací je ovlivněn přepínači, z nichž jsou v této práci využity právě tři zmíněné (popis všech možností se nachází v oficiální dokumentaci[2] tohoto nástroje od firmy Oracle):

-c zobrazí instrukce jednotlivých metod (bez něj jsou vytisknuty pouze deklarace metod a třídních atributů).

```
javap -c -l -s Example.class
Classfile /path/to/classfile/Example.class
 ...
Compiled from "Example.java"
public class Example
{
  public Example():
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
        0: aload_0 //!> reference na lokální
                     proměnnou ve slotu 0
        1: invokespecial #1 //!> odkaz do Constant Poolu
             // Method java/lang/Object."<init>":()V
        4: return
      LineNumberTable:
        line 1: 0 //!> číslo první instrukce
      LocalVariableTable:   na daném řádku
        Start  Length  Slot  Name   Signature
            0       5     0  this   LExample;

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
     ...
```

Obrázek 1. Příklad výstupu programu `javap` s přepínači `-c -l -s`.

-l zahrne tabulky lokálních proměnných a mapování instrukcí na řádky původního kódu.

-s připojí ke každé deklaraci atributu signaturu popisující jeho typ.

2.3 Průběh extrakce CFG

Výstup programu `javap` je v tomto projektu načítán po řádcích, z nichž jsou všechny pro náš účel využitelné informace přepisovány do vnitřních struktur.

V rámci identifikace základních bloků v bajtkódu jsou v sekvencích načtených instrukcí vyhledávány skokové instrukce a cíle těchto skoků. Instrukce skoková – např. `goto` nebo `ifcond` – značí konec aktuálního bloku. Instrukce identifikovaná jako cíl některé z předchozích je pak označena jako první instrukce bloku nového.

Při rozlišování základních bloků jsou ukládány také informace o jejich vzájemném uspořádání v grafu, jednotlivé instrukce jsou rozděleny na kód s operandy a doplněny o programové lokace. Z každé metody zpracovávané třídy je vytvořen jeden CFG, jejichž výsledná kolekce může být už v této fázi tištěna ve formátu platformy Testos.

3. Překlad do LLVM IR

Pro další zpracování grafů toku řízení je důležitá nejen korektnost jejich extrakce z původního programu, ale také jednotlivé sekvence instrukcí obsažené v identifikovaných základních blocích. Při analýzách cest v CFG jsou sekvence inststrukcí po sobě jdoucích bloků spojovány za sebe do jediné sekvence v pořadí daném nalezenou cestou. Následně probíhá rozbor operací prováděných programem v této sekvenci.

V rámci platformy Testos chceme pomocí grafů toku řízení analyzovat programy psané v různých pro-

[1] https://pajda.fit.vutbr.cz/testos/cfgqe/blob/master/doc/cfglang.md

[2] http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javap.html

gramovacích jazycích jednotným způsobem. Proto je nutné, aby extrahované grafy obsahovaly instrukce některé z obecných instrukčních sad. Jako taková byla vybrána pro platformu Testos sada LLVM IR, protože má silnou nástrojovou podporu a díky rostoucí popularitě překladače *clang* velkou uživatelskou bázi.

Dalším krokem je tedy překlad instrukcí bajtkódu jazyka Java do LLVM IR tak, aby sekvence instrukcí jednotlivých základních bloků měly před překladem i po něm stejnou operační sémantiku. Na výstupu jsou očekávány ty samé CFG s tím rozdílem, že místo instrukcí bajtkódu budou obsahovat instrukce LLVM IR. Tento úkol už není tak přímočarý jako extrakce CFG, především kvůli rozdílným principům fungování obou instrukčních sad.

Bajtkód je konstruovaný pro práci nad zásobníkem operandů, se kterým však LLVM IR nepočítá, protože veškeré operandy ukládá jako pomocné proměnné. Součástí kódu v LLVM IR je také deklarace používaných metod a konstant, společně s alokací prostoru pro lokální proměnné jednotlivých metod, kterou v bajtkódu nenajdeme.

3.1 Průběh překladu instrukcí

Samotný překlad v rámci jednoho CFG proto začíná právě alokací prostoru pro lokální proměnné. Typickou ukázku tohoto úseku kódu v LLVM IR představuje obrázek 2 .

```
define i32 @foo(i32 %a, i32 %b) {
  %1 = alloca i32, align 4 //!> alokace prostoru
  %2 = alloca i32, align 4    pro argumenty
  store i32 %a, i32* %1, align 4 //!> přiřazení
  store i32 %b, i32* %2, align 4  předaných hodnot
  %3 = load i32* %1, align 4
  ...
```

Obrázek 2. Typická počáteční sekvence instrukcí metody v LLVM IR.

Po deklaraci lokálních proměnných jsou do LLVM IR překládány sekvence instrukcí bajtkódu každého ze základních bloků. Aby bylo možné správně vytvářet potřebné pomocné proměnné, je v překladači použit zjednodušený model operandového zásobníku. Instrukce lze pak podobně, jak je zmíněno v [5], rozdělit do několika skupin podle způsobu překladu.

3.2 Skupiny instrukcí dle způsobu překladu

Instrukce modifikující obsah zásobníku s operandy nejsou do LLVM IR nijak překládány. Patří mezi ně například `dup` (duplikace hodnoty na vrcholu zásobníku), `swap` (prohození dvou hodnot na vrcholu zásobníku), `pop` nebo načítání konstant na vrchol zásobníku. Ty se v LLVM neukládají do pomocných proměnných, ale jsou používány přímo, a tak jejich vložení na zásobník nemá ekvivalentní překlad.

Instrukce přeložitelné 1 ku 1 patří k nejjednodušším problémům tohoto úkolu. Jedná se typicky o aritmetické operace, základní načítání a ukládání lokálních proměnných, nebo volání statických metod. Pro tyto instrukce existují přímé ekvivalenty v obou sadách a jediným úkolem překladače je tedy správně převést operandy ze zásobníku do pomocných proměnných. Příkladem tohoto případu je převod instrukce `imul` z bajtkódu, která se zde uvádí bez operandů, na instrukci `mul` v LLVM IR. Podobně jako v JVM dosadí překladač jako operandy pro násobení dvě hodnoty z vrcholu zásobníku. V těch jsou načtené buď konstanty, nebo jména pomocných proměnných, kam byla v některé z dřívějších instrukcí nahrána hodnota z některé lokální proměnné daného grafu.

Instrukce ekvivalentní sekvencím operací, jakou je například práce s poli nebo vyhodnocování podmínek nad složitějšími datovými typy, jsou o něco komplikovanější právě kvůli zvolení správné sekvence instrukcí v LLVM IR tak, aby byla zachována sémantika. Konkrétně do této skupiny patří instrukce `baload`, která má za úkol načíst na zásobník jeden byte z pole, jehož reference je uložena na vrcholu zásobníku, z prvku na indexu, který je uveden pod referencí. Při překladu vzniká z této instrukce nejprve `getelementptr`, který získá adresu požadovaného prvku, následován instrukcí `load`, která z dané adresy přečte hodnotu a uloží ji do pomocné proměnné pro další využití.

Specifické instrukce, které provádějí operace nad objekty JVM, mezi které řadíme např. `arraylength`, nebo `instanceof`, jsou v mnohých případech přeložitelné buď obtížně nebo vůbec, např. protože JVM udržuje mnoho metadat způsobem, které není možne do LLVM předat. V kódu jazyka Java však nejsou tyto instrukce ničím neobvyklým, a proto i ty, pro které vhodný ekvivalent neexistuje, musí být alespoň symbolicky přeloženy. Jako řešení tohoto problému bylo zvoleno nahrazení daných operací voláním imaginárních funkcí, které si zachovávají jména původních instrukcí. Při analýze se tento úsek kódu bude tvářit jako volání knihovní funkce, u které může případný čtenář předpokládat chování podle významu dané instrukce v JVM.

3.3 Známé problémy

Kromě překladu jednotlivých instrukcí se při pokusu o získání ekvivalentního přepisu bajtkódu do LLVM IR setkáváme i s několika dalšími zajímavými problémy. Ty pramení např. z faktu, že JVM při práci se zásobníkem operandů přímo vyhodnocuje výsledky některých operací, což ovšem není úkolem překladače.

Problematické situace vytvařejí třeba konstrukce

ukládající v rámci základního bloku na zásobník s operandy hodnoty, které nemusí být přečteny. Nejjednodušším příkladem takovéto konstrukce je ternární operátor mající za úkol výběr konstantní hodnoty, která bude následně uložena do lokální proměnné. V bajtkódu se objeví sekvence podobná této:

1. *Pokud podmínka platí pokračuj bodem 4.*
2. *Načti na vrchol zásobníku konstantu C_1.*
3. *Přeskoč do bodu 5.*
4. *Načti na vrchol zásobníku konstantu C_2.*
5. *Ulož hodnotu z vrcholu zásobníku do lokální proměnné.*

Výsledkem automatického překladu instrukcí bude v tomto případě sekvence, ve které se body 2 a 4 nebudou vyskytovat, protože, jak bylo popsáno dříve, načítání konstant na vrchol zásobníku se do LLVM nepřekládá do samostatných instrukcí. V přepisu se tedy objeví dva skoky, mezi nimiž se nebudou vyskytovat žádné další instrukce, následované přiřazením konstanty C_2 do lokální proměnné, zatímco konstanta C_1 zůstane ležet na vrcholu zásobníku.

Tento problém ještě na definitivní řešení čeká, zatím je tedy nutno poznamenat, že analýza kódu obsahující tento typ konstrukcí nemusí být úplně spolehlivá.

4. Evaluace

Správná funkčnost extrakce CFG byla dokázána na testovací sadě obsahující programy se všemi obyklými konstrukcemi jazyka Java, přejatými z unit testů projektu llvm-project[3]. Vzorové výstupy jsou přepracovány také pro ověření korektnosti překladu do LLVM IR.

4.1 Popis výstupu

Výstupem pogramu, který je produktem této práce, je soubor ve specifickém formátu JSON. Příklady spouštění a další užitečné informace jsou uvedeny na wiki stránkách projektu[4].

Následující příklad obsahuje kód jednoduché třídy pro praktickou demonstraci prováděného překladu, jehož výsledky po obou fázích znázorňuje obrázek 3.

```
public class BpTest {
    public static void main(String args[]) {
        int i = foo(1,3);
    }
    public static int foo(int a, int b) {
        if (a > 5) return a;
        else return b;
    }
}
```

Na uvedeném výstupu lze sledovat např. rozdílné zpracování načtení konstanty – instrukce `iconst_5` v 3a, která se projeví až přímo v instrukci porovnání uprostřed základního bloku s id 0 v 3b – a načtení hodnoty z lokální proměnné – instrukce `iload_X`, které jsou přeloženy jako vytvoření pomocných proměnných `%6` a `%8`.

Ve výsledném grafu existují dvě primární cesty (cesty, které nejsou podcestami jiných cest). Další analýza by se tedy sestávala z vytvoření dvou sekvencí instrukcí, z nichž jedna by řetězila instrukce bloků 0 a 1 a druhá instrukce bloků 0 a 2. Byly by formulovány podmínky, za nichž má program postupovat tou či onou cestou (v tomto případě podle hodnoty parametru a), a na základě těchto podmínek by byly generovány požadavky na testovací případy (minimálně jeden kdy bude $a > 5$ a druhý s $a <= 5$).

4.2 Problematické instrukce

Ne všechny metody je možné překládat přímočaře, a málokterá je tak jednoduchá jako uvedený příklad. V kapitole 3 už byly nastíněny nejzásadnější problémy této snahy. Krom zmíněného je však nutné vynakládat nemalé úsilí také na řešení drobnějších úskalí, jakými je například adresace atributů jednotlivých objektů, práce s poli, nebo vyhledání implementace virtuálních metod.

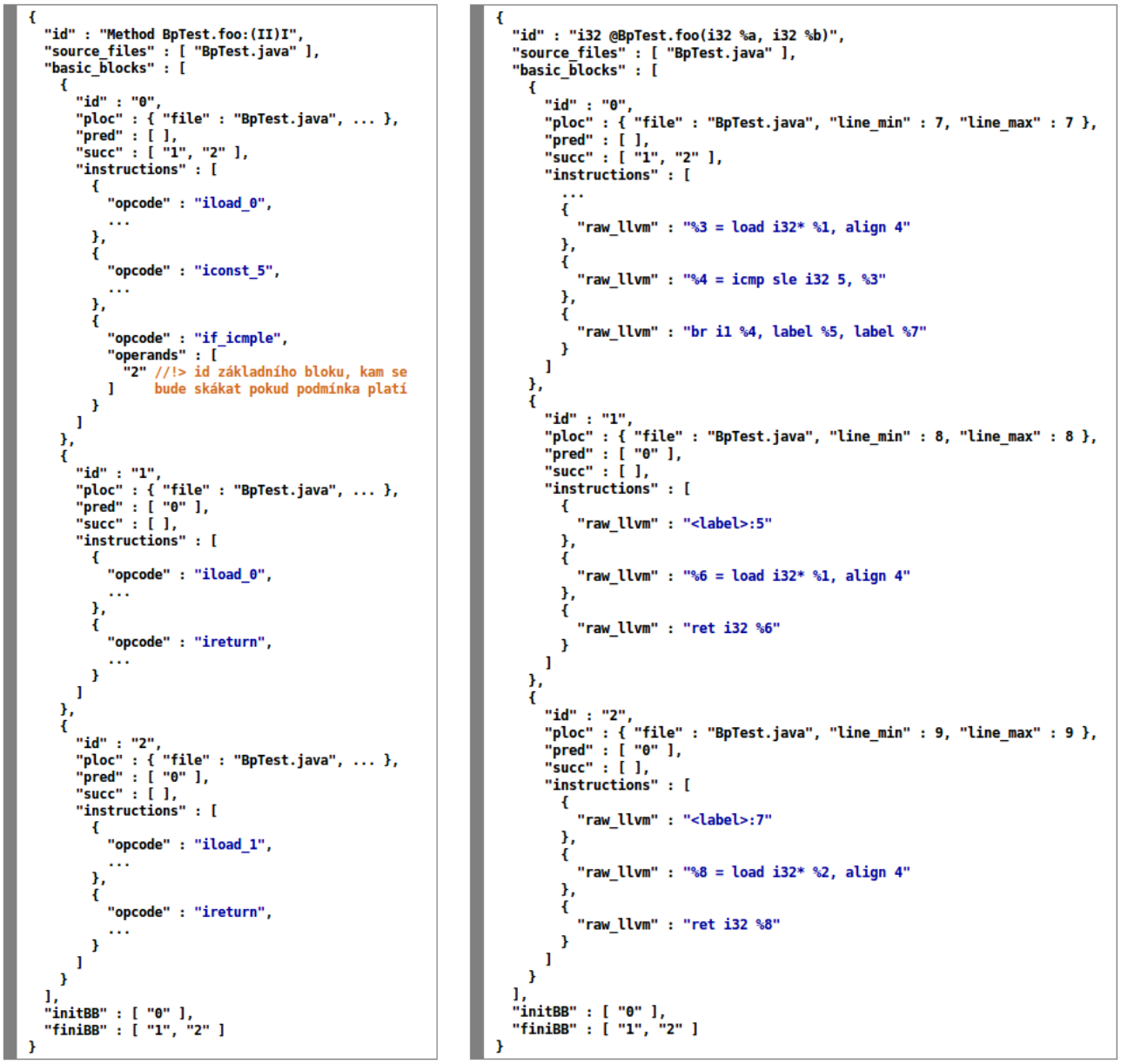
Typickým problémem, který se vyskytuje například u zmiňované práce s atributy objektů, je nedostupnost některých informací, které JVM uchovává interně, zatímco LLVM veškerá metadata očekává v operandech sémanticky ekvivalentních instrukcí. Konkrétně je v LLVM IR pro přístup k některému ze členů pole, struktury nebo objektu používána instrukce `getelementptr`, která potřebuje znát pořadí uložení jednotlivých členů v paměti. JVM oproti tomu identifikuje jednotlivá pole podle jmen a signatur, a proto není možné z bajtkódu zjistit celkový počet členů datové struktury ani jejich uspořádání v paměti.

Toto je v současnosti řešeno tak, že si překladač při zpracování bajtkódu vytváří záznamy o použitých polích jednotlivých tříd objektů. Protože z hlediska analýzy kódu na uložení v paměti příliš nezáleží, přistupuje k atributům, jakoby byly ukládány v pořadí, jak jsou v programu používány.

Jistě stojí za zmínku také výjimky, které jsou běžnou součástí většiny programů v jazyce Java. Podobně jako další specifické operace, které se v LLVM IR

[3] https://llvm.org/svn/llvm-project/java/trunk/test/Programs/SingleSource/UnitTests/

[4] https://pajda.fit.vutbr.cz/testos/java2cfg/wikis/home



(a) CFG s instrukcemi bajtkódu

(b) CFG s instrukcemi LLVM IR

Obrázek 3. Výsledky zpracování metody `foo`.

nevyskytují je tento problém řešen jako volání imaginární funkce.

V případech kdy překladač sahá k této možnosti, jde často o možnost poslední, avšak o nic méněcennější než ostatní způsoby překladu. Pokud je instrukce simulována jako knihovní volání, bude při analýze akceptována jako úsek kódu, za jehož správnost autor analyzovaného kódu nezodpovídá, což vyhovuje i pro získávání výsledků těchto specifických instrukcí. Je ovšem potřeba s tímto řešením při důkladnější analýze výsledných CFG počítat.

5. Závěr

Tato práce osvětluje způsob spolehlivé automatizované extrakce grafů toku řízení z programů v jazyce Java. Ten je obohacený o překlad sekvencí instrukcí bajtkódu do obecné instrukční sady LLVM IR, aby bylo umožněno uniformní zpracování výsledků společně s podobně získanými CFG z kódů v jiných programovacích jazycích.

Extrahované grafy lze využít pro různé metody testování založeného na modelech, například lze generovat testovací případy podle různých kritérií pokrytí kódu. Analýzou toku řízení programů v Javě se zabývá více nástrojů, ale jejich výsledky nejsou narozdíl od této práce obecně dobře rozšiřitelné nebo dále jednoduše použitelné pro další analýzu.

V budoucnu má být tato práce součástí software pro tvorbu a správu automatizovaných testů vyvíjeného platformou Testos. Už nyní existuje první verze navazu-

jícího projektu, který bude ve výsledných CFG vyhledávat cesty a ke kterému budou jistě brzy přibývat další podobné analyzátory.

Poděkování

Za ochotnou spolupráci a odborný dohled děkuji vedoucímu mé bakalářské práce, Ing. Aleši Smrčkovi, Ph.D.

Literatura

[1] Pedro de Carvalho Gomes. Sound Modular Extraction of Control Flow Graphs from Java Bytecode. Master's thesis, KTH Royal Institute of Technology, November 2012.

[2] Jianjun Zhao. Analyzing control flow in java bytecode. In *in Proc. 16th Conference of Japan Society for Software Science and Technology*, pages 313–316, 1999.

[3] Anthony Pena, Nicolas Brondin, and Jérémy Bardon. Github repo of BytecodeToCfg project, December 2014. https://github.com/masters-info-nantes/bytecode-to-cfg.

[4] Sergej Alekseev and Sebastian Reschke. Dr. Garbage Tools, August 2015. http://drgarbagetools.sourceforge.net/.

[5] Nicolas Geoffray. The VMKit project: Java and .Net on top of LLVM. pdf, August 2008. http://llvm.org/devmtg/2008-08/Geoffray_VMKitProject.pdf.