VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

INSTRUMENTACE PROGRAMŮ PRO MĚŘENÍ POKRYTÍ PŘI TESTOVÁNÍ SW

PROGRAM INSTRUMENTATION ENABLING COVERAGE MEASUREMENT IN SW TESTING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. PETR KAPOUN**
AUTHOR

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

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2021/2022

Zadání diplomové práce

Student: **Kapoun Petr, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Softwarové inženýrství
Název: **Instrumentace programů pro měření pokrytí při testování SW**
Program Instrumentation Enabling Coverage Measurement in SW Testing
Kategorie: Analýza a testování softwaru
Zadání:

1. Nastudujte metody testování softwaru na základě modelů. Nastudujte kritéria pokrytí kódu při testování softwaru.
2. Navrhněte nástroj pro instrumentaci programů při překladu s injekcí kódu pro měření vybraných kritérií pokrytí.
3. Implementujte nástroj pro instrumentaci programů v jazycích C a C++. Pro instrumentaci využijte infrastrukturu překladače clang nebo gcc. Implementujte kód pro počítání pokrytí pro kritéria zahrnující kritéria řádků kódu, rozhodovací logiky a datových toků.
4. Vytvořte demonstrační testovací sadu. Ověřte základní funkcionalitu automatickými testy.

Literatura:

- ISO/IEC/IEEE 29119-4:2015(E) Software and system engineering -- Software testing -- Test techniques.

Při obhajobě semestrální části projektu je požadováno:

- První dva body zadání

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

Vedoucí práce: **Smrčka Aleš, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 18. května 2022
Datum schválení: 3. listopadu 2021

Abstrakt

Tato práce se zabývá návrhem a tvorbou instrumentačního nástroje pro měření pokrytí při softwarovém testování. Nástroj při překladu získá reprezentaci vybraných částí programu v podobě grafu toku řízení a instrumentuje dané části programu vložením zpětných volání funkcí. Pomocí dat generovaných při volání vložených funkcí instrumentovaného programu nástroj vyhodnotí měření kritérií pokrytí. Mezi podporovaná kritéria pokrytí patří pokrytí řádků kódu a vybraná kritéria pokrytí toku řízení a toku dat.

Abstract

This work deals with the design and creation of an instrumentation tool for measuring coverage in software testing. During compilation, the tool obtains a representation of selected parts of the program in the form of a control flow graph and instruments the given parts of the program by inserting function callbacks. Using the data generated when calling the function callbacks of the instrumented program, the tool evaluates the measurement of the coverage criteria. Supported coverage criteria include line coverage and selected control flow and data flow coverage criteria.

Klíčová slova

instrumentace, testování, měření pokrytí, formální verifikace, dynamická analýza, graf toku řízení, pokrytí řádků kódu, pokrytí toku řízení, pokrytí toku dat, C++, LLVM, LLVM IR, Clang

Keywords

instrumentation, testing, coverage measurement, formal verification, dynamic analysis, control flow graph, line coverage, control flow coverage, data flow coverage, C++, LLVM, LLVM IR, Clang

Citace

KAPOUN, Petr. *Instrumentace programů pro měření pokrytí při testování SW*. Brno, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

Instrumentace programů pro měření pokrytí při testování SW

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana doktora Aleše Smrčky. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Petr Kapoun
18. května 2022

Poděkování

Rád bych poděkoval panu Ing. Aleši Smrčkovi Ph.D. za vedení mé práce, jeho vřelý přístup a cenné rady.

Obsah

Kapitola 1

Úvod

Vyhodnotit, jak moc je software otestovaný by měl uživateli nástroj vyvinutý v rámci této práce, která se zabývá návrhem a implementací nástroje pro měření pokrytí.

Testování softwaru je dnes neodmyslitelnou částí vývoje softwaru. Jeho hlavním cílem je kvalita produktu, která má vliv na spokojenost uživatele, pověst společnosti i vývoj samotného produktu. Technik a přístupů k testování existuje celá řada. Slovník inženýrských pojmů [8] testování označuje jako *kreativní a náročný úkol*.

U každého softwaru je nutné správně určit potřebnou míru otestování software v rovnováze s cenou testování, která zahrnuje mimo jiné návrh, tvorbu a provádění testů. Do volby míry otestování vstupuje i fakt, že čím více testujeme, tím je nalezení další chyby nákladnější. Velice důkladně musí být otestovaný software, u kterého může mít chyba vážné, drahé nebo nebezpečné následky. Takový software je například v letadlech, autonomně řízených vozidlech nebo bankách. Opakem může být software vytvořený pro zábavné účely, kdy by důkladné testování prodražilo produkt, ale uživatel by cenu příliš důkladného testování nemusel pochopit.

Zjištění míry otestování softwaru je možné pomocí měření kritérií pokrytí. Vývojáři to o sadě testů prozradí, jak moc pokrývá kód, ale i kde jsou málo pokryté části kódu. Existuje velké množství různých kritérií pokrytí založených, kde některá jsou založena na modelech testovaného systému, jako je například graf toku řízení(CFG). Mezi známá kritéria pokrytí patří například *Line Coverage*, *Function Coverage*, *Edge-Pair Coverage*, *Modified Decision Condition Coverage*, *Prime Path Coverage*, *Branch Coverage*, *All Uses*, *All Def-Use Paths* a další.

Vyhodnocením těchto kritérií se zabývá řada nástrojů a mezi nimi i nástroj vytvořený v rámci této práce. Nástroj využívá sadu nástrojů pro překlad *LLVM*. Funguje jako obálka nad překladačem, která pozastaví překlad, instrumentuje kód a dokončí překlad. Monitorovací sondy vložené instrumentací zaznamenávají provádění částí kódu důležitých pro vyhodnocení měření kritérií pokrytí. Samotné vyhodnocení zpracuje zaznamenané události, načte graf toku řízení a vyhodnotí kritéria pomocí pokrytí požadavků daných kritérií v grafu cestami.

Tato práce je členěna do několika kapitol. Po úvodu je první kapitola *Instrumentace a použité technologie* 2, která představuje problematiku, v rámci níž je tato práce řešena. Je zde rozebrána problematika instrumentace, překladu pomocí nástrojů *Clang/LLVM*, jazyka *LLVM IR*, testování, grafu toku řízení a kritérií pokrytí. Následuje *Analýza a specifikace požadavků* v kapitole 3, na kterou navazuje kapitola 4 věnující se návrhu nástroje pro instrumentaci. Další kapitola 5 se zabývá samotnou implementací monitoru pokrytí a přibližuje fungování částí řešení. Zajímavý je zejména způsob organizace výstupů vložených

sond instrumentovaného programu a vyhodnocení pokrytí z těchto výstupů. Předposlední kapitola 6 se věnuje popisu způsobu ověření správnosti řešení a v poslední kapitole 7 je shrnut výsledek práce.

Kapitola 2

Instrumentace a použité technologie

Tato kapitola se zabývá studií, popisem a volbou technologií pro použití v této práci. Nejdříve je popsán úvod k pochopení podstaty testování v podkapitole 2.2, na to navazuje podkapitola 2.3 popisující testování založené na modelech, konkrétně testování založené na modelu grafu toku řízení *CFG*. Následuje část zabývající se sadou nástrojů *LLVM* v podkapitole 2.5, která je zaměřená na popis z hlediska instrumentace pro měření kritérií pokrytí. V další podkapitole 2.1 je popsán význam instrumentace následovaný další podkapitolou 2.6 popisující existující řešení pro instrumentace, která obsahuje i volbu instrumentačního nástroje pro použití v této práci.

2.1 Instrumentace kódu

Termín instrumentace odkazuje na schopnost monitorovat nebo měřit úroveň výkonu produktu a diagnostikovat chyby[13]. Kód instrumentován, tedy upraven tak, aby bylo možné získat potřebné informace. To je provedeno vložením monitorovacích sond. V programování lze instrumentaci využít pro:

- trasování kódu,
- ladění,
- měření výkonnosti,
- výpisy událostí[13].

Instrumentace přidává do provádění programu režii, která program zpomaluje. Například při měření výkonnosti je třeba zajistit, aby zaznamenání doby běhu nebylo započítáno k době běhu nějaké funkce. U trasování kódu nebo u ladění je naopak dobré vymezit část kódu pro instrumentaci, aby byla instrumentace i běh programu rychlejší.

Vložení (injekce) monitorovacích sond může probíhat v různých časech:

- v čase překladu (na zdrojových kódech),
- při spuštění (při zavedení do paměti), nebo
- v běhu (připojení k bežícímu programu, podpora OS)[17],

Vkládané monitorovací sondy mohou být:

- interní (krátké podprogramy uvnitř sledovaného programu) nebo
- externí (sondy způsobují události vně).

Trasování je způsob, jak lze sledovat provádění aplikace, když je spuštěna[13]. V rámci této práce je kód instrumentován právě za účelem trasování. Proces instrumentace probíhá při překladu a vkládá do zdrojového kódu interní sondy.

2.2 Testování software

Testování softwaru je součástí verifikace softwaru a dynamické analýzy. Motivací pro testování je kvalita softwaru. Testovat software lze mnoha způsoby a vždy je třeba hledat vhodný způsob, jak otestovat konkrétní aspekt softwaru.

Testování je jeden ze způsobů verifikace softwaru, která byla definována ve článku *The Growth of Software Testing*[6] následovně:

> *Verifikace je proces vyhodnocování softwarového systému nebo komponenty za účelem zjištění, zda produkty dané vývojové fáze splňují podmínky stanovené na začátku této fáze.*

Samotné testování bylo definováno mnoha způsoby, protože problematika testování je velice rozsáhlá a rozmanitá. Níže jsou uvedeny dvě definice z knihy *Practical Software Testing: A Process-Oriented Approach*[4]:

1. *Testování je obecně popsáno jako skupina postupů prováděných za účelem vyhodnocení některého aspektu části softwaru.*
2. *Testování lze popsat jako proces používaný k odhalení defektů v softwaru a ke zjištění, že software dosáhl určitého stupně kvality s ohledem na vybrané atributy.*

Software je testován kvůli zjištění kvality a vlivu na kvalitu. Ve standartu *IEEE Standard Glossary of Software Engineering Terminology*[8] jsou uvedeny dvě definice kvality:

1. *Kvalita se týká míry, do jaké systém, komponenta systému nebo proces splňuje stanovené požadavky.*
2. *Kvalita se týká míry, do jaké systém, systémová komponenta nebo proces uspokojuje potřeby nebo očekávání zákazníků nebo uživatelů.*

Kvalitu lze měřit pomocí metrik kvality softwaru. Metriky jsou zaměřeny na různé atributy kvality, které jsou dle knihy *Practical Software Testing: A Process-Oriented Approach*[4] následující:

- správnost,
- spolehlivost,
- použitelnost,
- integrita,
- přenositelnost,
- udržovatelnost,
- interoperabilita.

2.3 Graf toku řízení

Aplikace analýzy měření pokrytí je typicky asociována s použitím modelů toku řízení a toku dat k reprezentaci struktury elementů a dat programu:

- programové příkazy,
- rozhodnutí/větve - ovlivňují tok řízení programu,
- podmínky - výrazy, které jsou vyhodnoceny jako *true/false* a neobsahují žádné jiné výrazy vyhodnocující se jako *true/false*,
- kombinace rozhodnutí a podmínek,
- cesty - sekvence uzlů grafu toku řízení[4].

Základem pro abstraktní reprezentaci kódu pomocí tvorby grafů je definice orientovaného grafu $G = (N, N_0, N_f, E)$, kde:

- N je množina uzlů,
- N_0 je množina počátečních uzlů taková, že $N_f \subseteq N$,
- N_f je množina koncových uzlů taková, že $N_f \subseteq N$,
- E je množina hran takových, že $E \subset N \times N$[2].

Cesta v grafu $path \subseteq N^+$ je neprázdná sekvence(řetězec) uzlů $[n_1, n_2, ..., n_M]$ taková, že každá dvojice sousedících uzlů tvoří hranu v daném CFG[17]:

$$(n_i, n_i + 1) \in E, 1 \leq i < M$$

Definice pokrytí grafu: Nechť TR je množina testovacích požadavků pro grafové kritérium C, sada testů T splňuje C na grafu G tehdy a pouze tehdy, pokud pro každý testovací požadavek $tr \in TR$ existuje alespoň jedna cesta $p \in path(T)$, taková, že p splňuje tr[2].

McCabeova cyclomatická složitost

Při zkoumání grafu toku řízení (*CFG*) lze vyjádřit jeho složitost číselnou hodnotou *McCabeovy cyclomatické složitosti*. U funkcí je lepší mít složitost tohoto grafu menší, protože funkce je poté snadněji testovatelná. Graf s menší složitostí obsahuje méně požadavků pro splnění kritérií pokrytí při testování, ať už to jsou hlavní cesty nebo cesty mezi definicemi a použitími proměnných. Znalost této hodnoty při vývoji pomůže odhalit obtížně testovatelné funkce, které by bylo vhodné upravit nebo rozdělit do více funkcí.

McCabeova cyclomatická složitost (*McCabe's Cyclomatic Complexity*) je vztah pro výpočet (vyjádření) složitosti CFG[17]:

$$V(G) = E - N + 2, \text{ kde}$$

- G - control flow graph,
- E - (edges) počet hran v G,
- N - (nodes) počet uzlů v G.

Formule může být aplikována na grafy toku bez odpojených komponent.

2.3.1 Kritéria pokrytí toku řízení

Každý strukturovaný program může být sestaven ze tří druhů prvků: sekvenční (např. přiřazení hodnoty do proměnné), rozhodovací (např. konstrukce *if-else* a *switch*) a iterativní (např. smyčky *while* a *for*). Směr přechodu záleží na výstupu rozhodování v predikátu. V následujícím kódu je jedno rozhodnutí (*decision*) ($a > 5$ && $b > 10$), které obsahuje dvě podmínky (*condition*) $a > 5$ a $b > 10$:

```
1 if (a > 5 && b > 10){
2     std::cout << "Hello world!";
3 }
```

Výpis 2.1: Kód v jazyce *C++* obsahující jedno rozhodnutí se dvěma podmínkami.

Statement adequacy/coverage

Pokrytí je splněno, pokud je každý příkaz (*statement*) alespoň jednou proveden při běhu programu. V grafu *CFG* pokrytí pokrývá všechny uzly.

Decision(branch) coverage

Pokrytí je splněno, pokud každý rozhodovací element v kódu (*if-then*, *case*, *loop*) je s každým možných vstupem proveden alespoň jednou. V *CFG* toto pokrytí pokrývá všechny hrany, a tedy pokrývá i všechny uzly.

Decision-condition coverage

Rozhodnutí (*decision* může obsahovat více podmínek (*condition*). Mějme rozhodnutí obsahující dvě podmínky.

IF A>5 AND B>10, THEN

Každé rozhodnutí musí být nastaveno na všechny možné hodnoty a každá podmínka musí být nastavena na všechny možné hodnoty. Tedy pokrytí *Decision-condition coverage* je splněno při následujících testovacích případech.

a	b	a > 5	b > 10	a > 5 and b > 10
10	20	true	true	true
0	0	false	false	false

Tabulka 2.1: Testovací případy splňující pokrytí *Decision-condition coverage* u podmínky IF A>5 AND B>10.

Multiple condition coverage

Pokrytí *Multiple condition coverage* je splněno, pokud jsou provedeny všechny kombinace podmínek. V tabulce 2.2 je příklad testovacích případů splňující rozhodnutí obsahující dvě podmínky. Počet testů pro C podmínek je 2^C. Nevýhodou pokrytí je právě exponenciální vztah, kvůli kterému je toto pokrytí nepraktické.

a	b	a > 5	b > 10	a > 5 and b > 10
10	20	true	true	true
10	0	true	false	false
0	20	false	true	false
0	0	false	false	false

Tabulka 2.2: Testovací případy splňující pokrytí *Multiple condition coverage* u podmínky IF A>5 AND B>10, THEN.

Modified condition decision coverage

Pokrytí je splněno, pokud je ověřeno, že každá podmínka nezávisle na ostatních ovlivní výstup rozhodnutí. Počet testů při C podmínkách u tohoto pokrytí je $C + 1$ a tedy pro splnění tohoto pokrytí je potřeba menší počet testů než u *MCC*[16].

Testování dle pokrytí *Modified condition decision coverage* (*MCDC*) je vyžadováno u kritických systémů v oblasti letectví (např. standard *DO-178C*[1]) a v oblasti automobilového průmyslu (např. standard *ISO 26262*[2]).

Node coverage

Kritérium pokrytí vyžaduje, aby množina testovacích požadavků (*TR*) obsahovala každý syntakticky dosažitelný uzel[17].

Edge coverage

Kritérium pokrytí hran Edge Coverage (EC) vyžaduje, aby *TR* obsahovaly každou syntakticky dosažitelnou cestu o délce 0 nebo 1[17].

Edge-pair coverage

Kritérium pokrytí párů hran Edge-Pair Coverage (EPC) vyžaduje, aby *TR* obsahovaly každou syntakticky dosažitelnou cestu o délce nejvíce 2[17].

Prime path coverage

Jednoduchá cesta (Simple path) je taková cesta, ve které se žádný uzel neopakuje. Výjimkou je cesta, která začíná a končí stejným uzlem.

Hlavní/primární cesta (Prime path) p je taková jednoduchá cesta, která není podcestou jiné jednoduché cesty, tj. je to nejdelší jednoduchá cesta.

Kriterium pokrytí hlavních cest Prime Path Coverage (PPC) vyžaduje, aby *TR* obsahovaly všechny hlavní cesty[17].

2.3.2 Kritéria pokrytí toku dat

U kritérií toku dat existují dva základní druhy práce s proměnnou, které jsou vázané k příkazu (*statement*):

[1]Standard *DO-178C - Software Considerations in Airborne Systems and Equipment Certification* je vydáván organizací *RTCA, Inc.*: https://www.rtca.org/.

[2]Část standardu *ISO 26262* zabývající bezpečností *ISO 26262-9:2018 Road vehicles — Functional safety — Part 9: Automotive safety integrity level (ASIL)-oriented and safety-oriented analyses*: https://www.iso.org/standard/68391.html.

- Proměnná je definována v příkazu, když je přiřazena nebo změněna její hodnota.
- Proměnná se používá v příkazu, když je v příkazu použita její hodnota. Hodnota proměnné se nezmění[4]

Kritéria, která se zabývají datovými toky, mohou využívat CFG. V jednotlivých uzlech CFG se vytvoří seznam obsahující definice a použití proměnných (*defs* a *uses*). Položky *defs* obsahují proměnné, které byly definované v rámci základního bloku tohoto uzlu. Obdobně položky *uses* obsahují proměnné, které byly použity v rámci tohoto základního bloku.

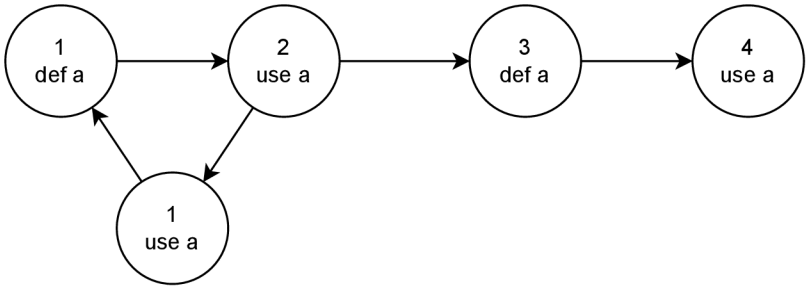
Pro další text budou zavedeny zkratky podle knihy *Practical Software Testing: A Process-Oriented Approach*[4]:

- **def**: definice proměnné,
- **use**: použití proměnné (X) beze změny její hodnoty. Použití proměnné může mít různé formy:
  - **p-use** (*predicate*): predikátové použití ($y = 2 * X$),
  - **c-use** (*computation*): výpočetní použití ($if\ X > 10$).

Krom základního dělení forem definic a použití dat popsaných výše, existuje více forem definic a použití dat. Ty popisuje *Paul C. Jorgensen* ve své knize *Software Testing: A Craftsman's Approach, Fourth Edition*[9]. Krom forem definic a použití popsaných výše, přidává následující formy:

- **o-use** (*output*): použití pro výstup,
- **l-use** (*location*): použití v lokaci (ukazatelé, indexy polí),
- **i-use** (*iteration*): použití ve iteracích (čítače, indexy smyček),
- **i-def** (*input*): definice načtením vstupu,
- **a-def** (*assignment*): definice přiřazením existující hodnoty.

Dvojici definice a použití jedné proměnné v jednom grafu nazýváme *def-use pair*. Cestu od definice proměnné k jejímu použití nazýváme *def-use path*. Jeden pár může mít více cest.



Obrázek 2.1: Příklad definice a použití proměnných pro demonstraci dosažitelnosti použití proměnných od jejich definic.

Při hledání cest a párů definic a použití proměnných *def-use pair/path* je řešena dosažitelnost použití z místa definice proměnné. Na obrázku 2.1 je vidět šest uzlů grafu obsahující definice proměnných. Při prozkoumávání dosažitelnosti použití proměnné *a* od definice

proměnné v uzlu *1* prohledávání pokračuje po následovnících uzlů. Na cestě $[1, 2, 3]$ prohledávání končí směrem do uzlu *4*, protože zde je proměnná přepsána. Na cestě $[1, 2, 3, 5]$ prohledává končí směrem do uzlu *2*, protože uzel *2* je již v cestě obsažen, již je zjištěna jeho dosažitelnost a je tedy detekován cyklus. Výsledkem prohledávání dosažitelnosti od definice proměnné z uzlu *1* budou tři cesty *def-use path* a tři páry *def-use pair*:

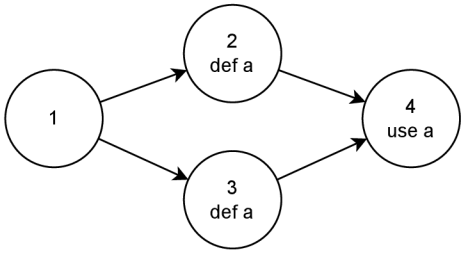
- cesty definic a použití proměnné (*def-use paths*):
  - $[1, 2]$
  - $[1, 2, 3]$
  - $[1, 2, 5]$
- páry definic a použití proměnné (*def-use pairs*):
  - $[1, 2]$
  - $[1, 3]$
  - $[1, 5]$

All defs coverage

Pokrytí jedné definice je splněno, když je definice proměnné pokryta alespoň jedním testovacím případem. Toto pokrytí zkoumá všechny definice proměnných.

All uses coverage

Pokrytí jednoho použití je splněno, když je pokryta alespoň jedna cestou *def-use* obsahující dané použití proměnné testovacím případem.



Obrázek 2.2: Příklad grafu se dvěma páry definice a pokrytí proměnné.

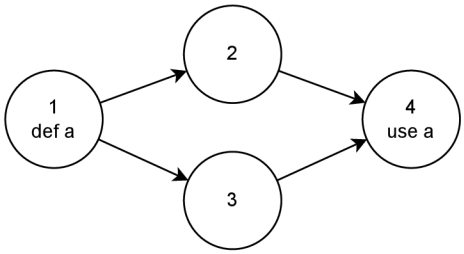
Na obrázku 2.2 jsou obsaženy dva páry *def-use*, kde každý pár má jednu cestu. Kritérium pokrytí *All uses coverage* bude splněno, pokud alespoň jedna cesta bude nalezena v množině testovacích případů.

Pokrytí *All p-uses coverage* zkoumá pouze predikátová použití proměnných. Naopak pokrytí *All c-uses coverage* zkoumá pouze výpočtová použití proměnných.

All def-use pairs coverage

Pokrytí jednoho páru definice a použití je splněno, když je pokryta jedna cestou *def-use* obsahující dané použití proměnné a danou definici proměnné testovacím případem.

Na obrázku 5.7 je definice proměnné *a* v uzlu *1* a použití proměnné *a* v uzlu *4*. K páru *def-use* existují dvě cesty.



Obrázek 2.3: Příklad definice a pokrytí proměnné v uzlech grafu.

cesta 1	1	2	4
cesta 2	1	3	4

Pro splnění pokrytí *All def-use pairs coverage* musí alespoň jeden testovací případ obsahovat alespoň jednu cestu z těchto dvou.

All def-use paths coverage

Pokrytí jednoho páru definice a použití je splněno, když jsou pokryty všechny cesty *def-use* obsahující dané použití proměnné a danou definici proměnné testovacím případem.

U příkladu, který byl použit pro vysvětlení *All def-use pairs coverage*, musí testovací případy pro splnění pokrytí Pro splnění pokrytí *All def-use paths coverage* obsahovat všechny cesty. V tomto případě obě dvě.

2.3.3 Kritérium pokrytí řádků kódu

Toto kritérium pokrytí není nutně vázáno na graf toku pokrytí, ale lze jej použít. Při provádění kódu je pro vyhodnocení tohoto pokrytí nutné zjistit řádky provedených příkazů (*statement*) programu.

Speciálními případy v případě vyhodnocení pokrytí jsou podmínky a prázdné řádky. V případě podmínek stačí, když část podmínky je vyhodnocena, ale pro splnění pokrytí nemusí být vyhodnoceny všechny části podmínky. V případě řádků, které neobsahují žádný příkaz, mezi které patří řádky se samostatnou závorkou, komentářem nebo prázdné řádky, není vhodné tyto řádky do vyhodnocení zahrnout.

Kritéria podobná pokrytí řádků kódu obsahuje například sada *Microsoft Visual Studio*. Zde jsou v tomto ohledu zajímavá zejména tato tři kritéria:

- *Pokrytí nedotčené oblasti* (*Coverage Not Touched Area*),
- *Pokrytí částečně dotčené oblasti* (*Coverage Partially Touched Area*),
- *Pokrytí dotčené oblasti* (*Coverage Touched Area*)[13].

Zajímavá je i vizualizace zmíněných pokrytí, lze si u nich nastavit zvýraznění ve vývojovém prostředí. Například dotčené oblasti mohou být zelené, nedotčené oblastní červené a částečně dotčené oblasti oranžové. Částečně dotčenou oblastí může být například podmínka, jejíž všechny části nebyly v rámci provádění programu vyhodnoceny.

2.4 Existující nástroje pro měření kritérií pokrytí

Existuje několik nástrojů pro měření kritérií pokrytí. V následujících podkapitolách budou představeny čtyři nástroje. Pro první tří nástroje je nutné zakoupit licenci. Čtvrtý nástroj je uvnitř produktu *Visual Studio*[14]. Měření uvnitř sady *Visual Studio* ukazuje jednoduché použití v rámci vývojového prostředí. Další tři nástroje podporují mnoho kritérií pokrytí, ale první dva nástroje *Testwell CTC++*[18] a *Squish Coco*[5] neumí měřit tok dat. Nástroje se liší zejména ve vizualizacích a možnostech použití v rámci vývojového cyklu, kam lze zařadit například použití s nástroji průběžné integrace (*CI*) nebo vyhodnocení kritérií pokrytí z běhů programu na více různých výpočetních uzlech.

2.4.1 Testwell CTC++

Testwell CTC++[18] je analyzátor pokrytí testů pro *C/C++* a vydavatel je *Testwell*.

Podporovaná jsou kritéria: *Line Coverage*, *Statement Coverage*, *Function Coverage*, *Decision Coverage*, *Multicondition Coverage*, *Modified Condition/Decision Coverage (MC/DC)*, *Condition Coverage*.

Mezi klady patří zobrazení grafu toku řízení i vizualizace jednotlivých kritérií pokrytí. Produkt je používán řadou známých firem v oblasti automobilového průmyslu, letectví i přepravy, mezi které patří například *Audi*[3], *BMW*[4], *Bosch*[5], *Honeywell*[6], *Siemens*[7] a další. Nevýhodou je nutnost zakoupení licence.

2.4.2 Squish Coco

Squish Coco[5] je nástroj pro měření pokrytí kódu pro jazyky *Tcl*, *QML*, *C#* a *C/C++* vydávaný firmu *Froglogic*.

Podporovaná jsou kritéria: *Statement Block Coverage*, *Decision Coverage*, *Condition Coverage*, *Multiple Condition Coverage*, *MC/DC*.

Mezi výhody patří podpora mnoha pokrytí a třeba i podpora nástrojů průběžné integrace (*CI*). Nevýhodou je jednodušší vizualizace výstupu a nutnost zakoupení licence, i když cena licence je oproti ostatním zde představeným placeným nástrojům nižší.

2.4.3 BullseyeCoverage

BullseyeCoverage[3] je analyzátor kódu pro jazyky *C/C++* od vydavatele *Bullseye Testing Technology*.

Mezi podporovaná kritéria pokrytí patří: *Statement Coverage*, *Branch Coverage*, *Condition Coverage*, *Condition/Decision Coverage*, *Path Coverage*, *LCSAJ Coverage* a *Data Flow Coverage*.

Mezi klady patří velké množství podporovaných kritérií, kde rozdílem proti předchozím nástrojům, je zvláště podpora kritérií toku dat. Dále lze pouštět program na výpočetních uzlech a pro vyhodnotit kritéria pro všechny běhy. Nevýhodou je vyšší cena licence, která stojí *900$* první rok a poté je každý další rok je prodloužení licence za *200$* (cena udávaná v květnu roku 2022).

[3] Automobilka *Audi*: https://www.audi.com/en.html
[4] Automobilka *BMW*: https://www.bmw.com/en/index.html
[5] Firma *Bosch*: https://www.bosch.com/
[6] Firma *Honeywell*: https://www.honeywell.com/us/en
[7] Firma *Siemens*: https://www.siemens.com/global/en.html

2.4.4 Visual Studio

V rámci produktu *Visual Studio*[14] firmy *Microsoft* je dostupný nástroj pro měření pokrytí. Nástroj je ovládán přes uživatelské rozhraní uvnitř *IDE*. Nástroj umí menší množství pokrytí a lze ho konfigurovat i uvnitř zdrojových kódů tak, jak je ukázáno ve výpisu 2.2.

```
1 [ExcludeFromCodeCoverage]
2 class ExampleClass2 { ... }
```

Výpis 2.2: Příklad značky použitelné pro vyjmutí části kódu z měření pokrytí při použití produktu *Visual Studio*.

Nástroje pro měření pokrytí v produktu *Visual Studio*[14] nelze plně srovnávat s předchozími nástroji. Jde spíše o ukázku vyhodnocení kritérií přímo ve vývojovém prostředí, což je pro vývojáře zjednodušení.

2.5 Infrastruktura pro překlad LLVM

LLVM je modulární sada znovupoužitelných překladačových nástrojů. Transparentnost procesu překladu, jeho modularita a rychlost patří mezi největší výhody sady nástrojů *LLVM*. V rámci této práce je kód instrumentován při překladu prováděného pomocí *LLVM*, kdy do reprezentace mezikódu překladu v jazyce *LLVM IR* jsou vkládána volání funkcí.

Pro *LLVM* je význačná jeho vnitřní reprezentace kódu *LLVM IR*. Právě díky této reprezentaci lze mít všechny nástroje modulární a činnost všech nástrojů je do jisté míry pro uživatele transparentní a konzistentní napříč nástroji. V rámci *LLVM* je poskytnuta sada knihoven pro práci s *LLVM IR* a komponenty jsou na těchto knihovnách postaveny. Překladač může využívat různé modulární komponenty nejvhodnější pro konkrétní překlad. Modularita je jeden z hlavních rozdílů *LLVM* vůči mnoha jiným překladačům.

2.5.1 Architektura

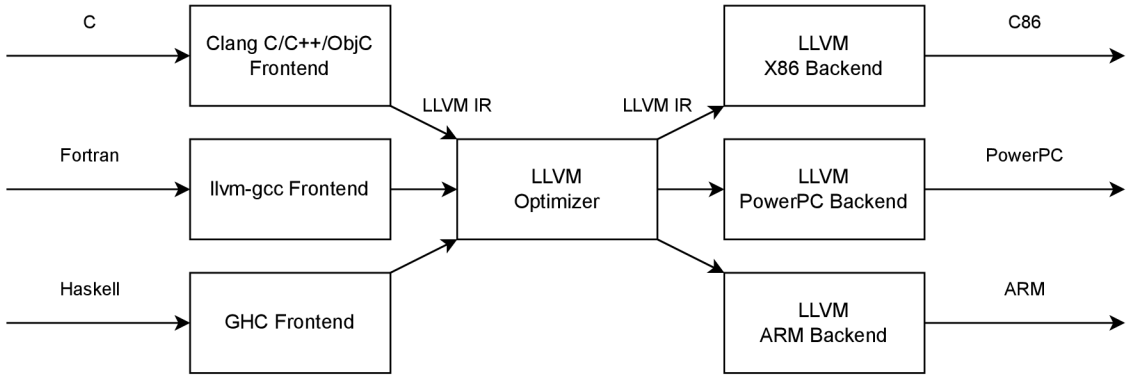
V základu *LLVM* využívá architekturu překladače se třemi fázemi, která je na obrázku 2.4. Úlohy jednotlivých fází jsou následující:

- *front-end*, který provádí převod zdrojového kódu do vnitřní reprezentace kódu,
- *optimalizátor*, který provádí úpravy vnitřní reprezentace kódu,
- *back-end*, který provádí převod vnitřní reprezentace kódu na strojový kód.

Obrázek 2.4: Tří hlavní komponenty překladače se třemi fázemi[10].

Příklad konkrétního propojení vybraných komponent překladače *LLVM* je na obrázku 2.5.

LLVM front-end přeloží zdrojový kód do *LLVM* reprezentace, která je poté zkompletována dohromady pomocí nástroje *LLVM linker*[11]. Různé optimalizační komponenty mohou být spouštěny před nebo po *LLVM linker*u, protože *LLVM linker* je také patří do optimalizační fáze a jeho vstupem i výstupem je *LLVM IR*.



Obrázek 2.5: Implementace překladače *LLVM* jako překladače se třemi fázemi[10].

2.5.2 Využití a rozšířitelnost

Obecným využitím je překlad zdrojových kódů do strojového jazyka cílové platformy. *LLVM* je ale často využíváno kvůli své rozšířitelnosti. Rozšíření nebo tvorba *front-end* modulů, optimalizačních modulů v rámci reprezentace *LLVM IR* nebo *back-end* modulů má různá využití.

Pomocí *LLVM* vzniklo i mnoho nových programovacích jazyků. Při tvorbě překladače pro vlastní jazyk je základ převést reprezentaci vlastního jazyka do reprezentace *LLVM IR*, tedy vytvořit vlastní *front-end* modul. Dále už lze využívat modulární komponenty například pro optimalizace nebo převod reprezentace *LLVM IR* do strojového kódu různých cílových platforem.

Rozšíření a tvorba optimalizačních modulů v prostřední části architektury *LLVM* může být využita například pro analýzy a optimalizace. I tato práce je založena na využití optimalizačního modulu, který upravuje *LLVM IR*.

Využití může *LLVM* najít i při tvorbě nových architektur čipů. V takové úloze je naopak vytvořen *back-end*, který převede *LLVM IR* reprezentaci do strojového jazyka dané architektury. Využití existujících modulů poté znamená, že pro danou architekturu čipu lze psát zdrojové kódy v jazycích, pro které existuje *front-end* modul, tedy například *C++*, *C* nebo *Fortran*.

2.5.3 Důležité instrukce LLVM IR pro měření pokrytí

Pro měření pokrytí jsou potřebné instrukce *LLVM IR* zejména ze 3 kategorií:

- ukončovací instrukce[8],
- operace adresování a přístupu do paměti[9],
- ostatní operace[10].

Ukončovací instrukce jsou na konci základních bloků a určují, který základní blok bude proveden po dokončení aktuálního základního bloku. Tyto instrukce určují tok řízení mezi

[8] Ukončovací instrukce *LLVM IR* https://llvm.org/docs/LangRef.html#terminator-instructions

[9] Operace adresování a přístupu do paměti *LLVM IR* https://llvm.org/docs/LangRef.html#memory-access-and-addressing-operations

[10] Ostatní operace *LLVM IR* https://llvm.org/docs/LangRef.html#other-operations

základními bloky (například instrukce *br*, *switch*) i mezi funkcemi (například instrukce *ret*). Ukázka syntaxe jednotlivých ukončovacích instrukcí je ve výpisu 2.3.

```
; syntax of ret, br and switch instructions
ret <type> <value>    ; Return a value from a non-void function
ret void              ; Return from void function
br i1 <cond>, label <iftrue>, label <iffalse>
br label <dest>       ; Unconditional branch
switch <intty> <value>, label <defaultdest> [ <intty> <val>, label <dest> ... ]
; example:
switch i32 %val, label %otherwise [ i32 0, label %onzero
                                    i32 1, label %onone
                                    i32 2, label %ontwo ]
```

Výpis 2.3: Příklad ukončujících instrukcí jazyka *LLVM IR*, které při použití ukončují základní bloky a funkce.

Operace adresování a přístupu do paměti jsou potřebné zejména pro kontrolu toku dat. Nejdůležitějšími instrukcemi přistupující do paměti pro měření pokrytí jsou instrukce *store* a *load*, jejichž použití je ukázáno ve výpisu 2.4.

```
; store load example:
%ptr = alloca i32                    ; yields i32*:ptr
store i32 3, i32* %ptr               ; yields void
%val = load i32, i32* %ptr           ; yields i32:val = i32 3
```

Výpis 2.4: Příklad instrukcí přistupujících do paměti v jazyce *LLVM IR*.

Mezi ostatními operacemi jsou také operace zajímavé pro větvení kódu a volání podprogramů. Patří mezi ně instrukce *call*, pomocí které jsou vkládána instrumentovaná volání. Dále v této skupině jsou instrukce porovnávací, jako například *icmp* a *fcpm*, které většinou přímo předcházejí instrukcím ukončovacím.

2.5.4 Příklad kódu LLVM IR

Tato práce pracuje přímo s kódem *LLVM IR*, který je instrumentován. V kódu jsou viditelné základní bloky kódu a předávání řízení mezi těmito základními bloky.

Pro příklad je níže napsána funkce *foo()* v jazyce *C++*, která má dva argumenty a s jejich použitím vyhodnotí výraz v podmínce. Pokud výraz vrátí hodnotu *true*, tak je změněna hodnota proměnné, která je poté vrácena

```
int foo(int a, int b){
    if (a == b || b == 0) {
        a = bar(a - b, 0) * 2;
    }
    return a + 1;
}
```

Výpis 2.5: Příklad jednoduché funkce v jazyce *C++*.

Ekvivalentní kód *LLVM IR* reprezentující funkci *foo()* napsanou v jazyce *C++* ve výpisu 2.5 je ve výpisu 2.6.

```
define i32 @_Z3fooii(i32 %0, i32 %1) #0 !dbg !8 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
```

```
  store i32 %0, i32* %3, align 4                    ; line: 1, column: 13, name: "a"
  store i32 %1, i32* %4, align 4                    ; line: 1, column: 20, name: "b"
  %5 = load i32, i32* %3, align 4, !dbg !16         ; line: 2, column: 9
  %6 = load i32, i32* %4, align 4, !dbg !18         ; line: 2, column: 14
  %7 = icmp eq i32 %5, %6, !dbg !19                 ; line: 2, column: 11
  br i1 %7, label %11, label %8, !dbg !20           ; line: 2, column: 16

8:                                                  ; preds = %2
  %9 = load i32, i32* %4, align 4, !dbg !21         ; line: 2, column: 19
  %10 = icmp eq i32 %9, 0, !dbg !22                 ; line: 2, column: 21
  br i1 %10, label %11, label %17, !dbg !23         ; line: 2, column: 9

11:                                                 ; preds = %8, %2
  %12 = load i32, i32* %3, align 4, !dbg !24        ; line: 3, column: 17
  %13 = load i32, i32* %4, align 4, !dbg !26        ; line: 3, column: 21
  %14 = sub nsw i32 %12, %13, !dbg !27              ; line: 3, column: 19
  %15 = call i32 @_Z3barii(i32 %14, i32 0), !dbg !28 ; line: 3, column: 13
  %16 = mul nsw i32 %15, 2, !dbg !29                ; line: 3, column: 27
  store i32 %16, i32* %3, align 4, !dbg !30         ; line: 3, column: 11
  br label %17, !dbg !31                            ; line: 4, column: 5

17:                                                 ; preds = %11, %8
  %18 = load i32, i32* %3, align 4, !dbg !32        ; line: 5, column: 12
  %19 = add nsw i32 %18, 1, !dbg !33                ; line: 5, column: 14
  ret i32 %19, !dbg !34                             ; line: 5, column: 5
}
```

Výpis 2.6: Příklad kódu funkce *foo* v jazyce *C++* ekvivalentní k funkci *foo* napsané v jazyce *C++* ve výpisu 2.5.

Pro příklad další řídící konstrukce je zde ukázka konstrukce *switch*. Tento příklad neobsahuje okolní kód (hlavičku funkce) a je zobrazen ve výpisu 2.7.

```
switch (argc) {
    case 1:
        var = 10;
        break;
    case 2:
        var = 20;
        break;
}
```

Výpis 2.7: Konstrukce přepínače v jazyce *C++*.

Ekvivalentní kód *LLVM IR* reprezentující konstrukci *switch* jazyka *C++* na konci základního bloku je ukázán ve výpisu 2.8.

```
1:
  ; placeholder: instructions of basic block before switch
  %7 = load i32, i32* %4, align 4
  switch i32 %7, label %10 [
    i32 1, label %8
    i32 2, label %9
  ]
```

Výpis 2.8: Konstrukce přepínače v jazyce *LLVM IR*.

2.6 Existující nástroje pro instrumentaci kódu

V této podkapitole jsou popsána vybraná existující řešení pro instrumentaci programů. Podkapitoly postupně představují jednotlivá řešení a v poslední podkapitole 2.6.5 je odůvodněna volba nástroje pro instrumentaci, který je využit v této práci.

2.6.1 XRay instrumentation

Nástroj *XRay instrumentation*[12] je součástí *LLVM* a je určen pro analýzu programů překládaných pomocí *LLVM*. Nástroj *XRay instrumentation* je obsažen v sadě nástrojů *LLVM* pod názvem *llvm-xray*.

Velkou předností tohoto nástroje je právě zahrnutí do sady *LLVM*. Díky tomu je nástroj používán širokou komunitou a je proto stabilní. Dalším důsledkem je, že nástroj není potřeba specificky sestavovat nebo instalovat, ale lze ho jednoduše použít.

Pomocí nástroje lze instrumentovat proměnné a volání funkcí. Také nástroj nabízí analýzu zásobníku volání, která zobrazí počty volání jednotlivých funkcí. Pro vizualizaci instrumentovaných volání umí nástroj generovat *flamegraph*.

2.6.2 SBT-instrumentation

Nástroj *SBT-instrumentation*[19] ze sady nástrojů *Symbiotic toolbox* vznikl jako diplomová práce Ing. Martiny Vitovské na Fakultě informatiky Masarykovy univerzity. Práce získala cena děkana FI MUNI za vynikající závěrečnou práci.

SBT-instrumentation je nástroj pro konfigurovatelnou instrumentaci programů v *LLVM*. Před použitím je nutné nástroj sestavit a poté uživatel vytvoří konfigurační soubor ve formátu *JSON*, kde lze specifikovat, jaké instrukce se do kódu vloží. Na rozdíl od ostatních od ostatních nástrojů lze vložit libovolnou instrukci na přesně specifikované místo. Ostatní nástroje na druhou stranu toto nenabízí a nabízí pouze možnost přidání volání funkce na určené nebo nalezené místo.

2.6.3 Tforc

Nástroj *Tforc*[20] ze sady nástrojů *Testos* vznikl jako diplomová práce Ing. Václava Ševčíka na Fakultě informačních technologií VUT v Brně.

Jedná se o nástroj pro instrumentaci programu během překladu v *LLVM* překladači. Nástroj umožňuje instrumentovat přístupy do paměti a funkce. Instrumentace byla realizována pomocí přidání průchodu v optimalizační fázi překladače *LLVM*. Informace o proměnných jsou spravovány vytvořeným frameworkem, který se připojí k programu během sestavování[20].

Nástroj je ovládán pomocí konfiguračního souboru a souboru s instrumentovanými funkcemi.

2.6.4 Tforc-tool

Nástroj *Tforc-tool*[15] ze sady nástrojů *Testos* vznikl jako bakalářská práce Ing. Kateřiny Muškové na Fakultě informačních technologií VUT v Brně. Nástroj využívá nástroj *Tforc*[20] popsaný v podkapitole 2.6.3.

Nástroje *Tforc-tool* slouží k instrumentaci programů napsaných v jazyce *C++*, a to instrumentaci přístupů do paměti a volání funkcí. Nástroj staví už na existujícím nástroji

Tforc poskytující statickou instrumentaci při překladu, jehož funkcionalitu a použitelnost rozšiřuje. Velkou výhodou oproti stávajícím řešením nabízejícím instrumentaci při překladu je možnost použití nástroje bez změny stávajících překladových skriptů (např. Make)[15].

2.6.5 Volba nástroje

Volba instrumentačního nástroje, který by mohl být využit pro nástroj pro měření pokrytí vychází z následujících požadavků na instrumentační nástroj:

- získání CFG,
- instrumentace základních bloků,
- získání informace o provedených řádcích kódu ideálně v rámci základního bloků, případně v rámci příkazu,
- získání informací o toku dat,
- jednoduchá konfigurace a použití z následujících důvodů:
  - není potřeba velmi přesná specifikace pro nalezení konkrétních instrukcí pro instrumentaci, protože budou instrumentovány všechny základní bloky v instrumentované funkci,
  - velká část instrukcí není pro měření pokrytí relevantní a není nutné ji umět instrumentovat.

Požadavky nesplňuje žádný z popsaných nástrojů. Tedy dvě nabízející se možnosti jsou rozšíření některého existujícího nástroje nebo vývoj nového instrumentačního nástroje.

Pro využití v této práci byl zvolen nástroj *Tforc-tool*[15] z následujících důvodů

- zajistí řízení překladu a všech jeho fází,
- není moc rozsáhlý,
- je lehce rozšířitelný,
- je jednoduše konfigurovatelný a spustitelný.

Samotné řízení překladu je velmi cenné, protože by ho bylo nutné implementovat ve velmi podobné podobě. Dále bude nutné rozšířit nástroj o získání CFG a instrumentaci základních bloků.

Kapitola 3

Analýza a specifikace požadavků

Při vývoji software předchází návrhu analýza a specifikace požadavků, ze kterých návrh a později implementace vychází. Požadavky lze rozdělit do dvou skupin na funkční a nefunkční. Funkční definují, co systém musí umět, a nefunkční popisují, jak systém funguje. Dokumentování požadavků je důležité z následujících důvodů:

- vyhnutí se selhání projektu,
- ujištění, že systém funguje dle očekávání,
- práce v rámci rozpočtu,
- definice rozsahu,
- zvýraznění dalších požadavků[7].

Funkční požadavky

Funkční požadavky jsou specifikací toho, co musí systém dělat. Skládají se jak z vlastností produktu, tak z požadavků uživatele. Obsahují mimo jiné i podrobnosti o tom, co musí být v systému zachyceno a sledováno[7]. Následující požadavky jsou rozděleny do tří skupin.

V první skupině je definována základní a klíčová funkcionalita systému, kvůli které je systém tvořen.

- Vyhodnotit pokrytí řádků kódu
- Vyhodnotit pokrytí toku řízení
  - *Function Coverage*,
  - *Node Coverage*,
  - *Edge Coverage*,
  - *Edge-Pair Coverage*,
  - *Prime Path Coverage*.
  - *(volitelné) Condition Coverage*, *Decision (Branch) Coverage*, *Modified Condition Decision Coverage*, *Multiple Condition Decision Coverage*
- Vyhodnotit pokrytí toku dat

  - *All Defs Coverage*,
  - *All Uses Coverage*,
  - *All Def-Use Pairs Coverage*.
  - *All Def-Use Paths Coverage*.

Druhá skupina požadavků definuje získání dat, která jsou nutná pro vyhodnocení kritérií pokrytí definovaných v požadavcích. Lze zde vidět, že požadavek na získání abstraktního sémantického stromu je volitelný, protože i kritéria, pro jejichž vyhodnocení je nutný, jsou volitelná.

- Získat *CFG* (*Control Flow Graph*) všech funkcí před instrumentací.
- Instrumentovat program tak, aby instrumentovaná volání byla unikátní a tedy byla jednoznačně identifikovatelná ve zdrojovém kódu *LLVM IR*.
- *(volitelné)* Získat *AST* (*Abstract Semantic Tree*) všech funkcí.

Poslední skupina definuje, co by měl systém zachytit nebo sledovat. Zachycená data mohou být použita při běhu a vývoji nebo mohou být výstupem systému pro uživatele.

- Zaznamenat do souborů
  - získaný CFG (Control Flow Graph) všech funkcí,
  - detail vyhodnocení pokrytí obsahující požadavky kritéria pokrytí a důvod nesplnění pokrytí,
  - volání z instrumentovaných programů.

Nefunkční požadavky

Nefunkční požadavky jsou rozděleny do kategorií podle účelu požadavků. Nefunkční požadavky nemají vliv na funkčnost systému, ale mají vliv na to, jak bude fungovat. Nefunkční požadavky se týkají použitelnosti systému. Pokud nejsou splněny nefunkční požadavky, uživatelé mohou být frustrovaní z toho, jak systém funguje[7].

Prvními kategorie požadavků se soustředí použitelnost a spolehlivost. Tento nástroj se soustředí na jednoduchost použití tak, aby se s ním uživatel nemusel dlouze učit pracovat. První požadavek proto definuje existenci příkladu použití nástroje pro měření pokrytí, na kterém uživatel uvidí kroky použití nástroje i reálný výstup.

- Použitelnost
  - Spustitelnost příkladového programu s vygenerováním výstupu pomocí jednoho příkazu bez úprav konfiguračních souborů.
  - Konfigurovatelnost použitého binárního souboru *Clang* (*clang++*) například v případě více nainstalovaných verzí nástroje *Clang* v systému nebo v případě jiného názvu binárního souboru (*clang++-11*).
- Spolehlivost
  - Obnovení prostředí v případě chyby.

Kategorie zabývající se efektivitou a výkonem je v této práci důležitá. Instrumentace a vkládání sond do instrumentovaného programu vytváří režii, kvůli které dochází k znatelnému zpomalení programu. Dále je nutné myslet i na dlouhé běhy programů, kdy doba překladu bude proti délce běhu výrazně menší. S ohledem na to jsou definovány následující požadavky:

- Efektivita a výkon
  - Minimalizovat zpoždění při provádění instrumentovaného programu.
  - Analýzu výstupů z instrumentace provádět až na konci, když už se nepouští instrumentovaný program.
  - Předcházet nedostatkům paměti.

Další kategorie požadavků se zabývají použitými technologiemi, nástroji a jejich verzemi. V případě těchto požadavků se jedná o závislosti systému. Definice těchto požadavků je důležitá pro vymezení prostředí, ve kterém musí systém fungovat.

- Přenositelnost
  - Zachování stability na operačním systému *Ubuntu 20.04*, který byl použit pro vývoj *Tforc-tool*.
  - *(volitelné)* Přidání podpory operačního systému *Ubuntu 21.10*.
  - *(volitelné)* Přidání podpory dalších operačních systémů (například *Debian 11*).
  - Spustitelnost nástroje i testů v *Docker*u.
  - Spustitelnost s *Python*em od verze 3.6 do verze 3.10.
- Rozšířitelnost
  - Možnost budoucího přidání podpory pro novější verze *LLVM*. Nezavedení velké závislosti řešení na konkrétní verzí *LLVM* použité při návrhu a vývoji.
- Implementace
  - Spouštění a provádění analýzy v jazyce *Python*.
  - Kód pro instrumentaci v jazyce *C++*.

V rámci monitoru pokrytí nebude implementována instrumentace od začátku, ale pro zrychlení vývoje budou použity existující nástroje zvolené při studii existujících řešení instrumentace.

- Součinnost
  - Nový nástroj bude pracovat součinně s nástroji *Tforc-tool* a *Tforc*.
    - Parametrizovatelné rozšíření nástroje *Tforc*.
    - Využití a případné rozšíření řízení překladu nástroje *Tforc-tool*.
- Dodání
  - Git repozitář.
- Standardy
  - Referenčními kritérii pokrytí jsou považována kritéria pokrytí popsaná v mezinárodním standardu: *29119-4-2015 - ISO/IEC/IEEE International Standard - Software and systems engineering–Software testing–Part 4 : Test techniques*[1].

Kapitola 4

Návrh nástroje pro měření pokrytí

V této kapitole je popsán návrh nástroje pro měření pokrytí. Návrh vychází z analýzy a specifikace požadavků popsané v kapitole 3.

4.1 Proces použití nástroje

V této podkapitole bude s větší mírou abstrakce představeno použití nástroje *tforcov*. Samotný proces získání výsledků měření pokrytí je zobrazen na diagramu 4.1 a lze ho rozdělit na tři části:

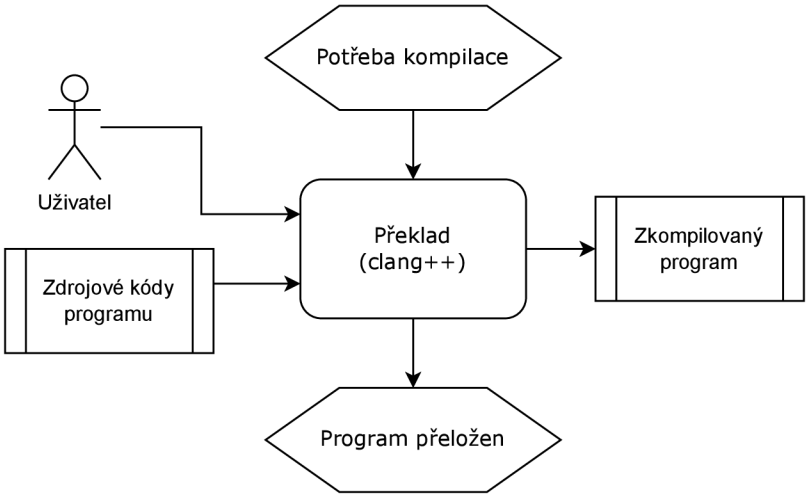
- překlad zahrnující instrumentaci programu,
- testování neboli spouštění instrumentovaného programu,
- vyhodnocení pokrytí z dat vytvořených v předchozích fázích.

Obrázek 4.1: Diagram použití nástroje *tforcov* zobrazující posloupnost procesů pro získání vyhodnocení měření pokrytí (*Process Overview Diagram*).

Tři části procesu na sebe navazují a nelze zaměnit jejich pořadí. Výstupy jednotlivých fází jsou prerekvizitami těch následujících. Překlad vytvoří spustitelný program, který je poté při testování spouštěn. Při překladu je i získán CFG. Spouštěním programu při testování jsou prováděny různé části zdrojového programu, tyto běhy programu jsou zaznamenány. Při vyhodnocení je načten CFG získaný při překladu a pro vyhodnocení pokrytí jsou zpracovány jednotlivé běhy programu. To umožňuje vyhodnotit pokrytí, ať se jedná o pokrytí toku dat, toku řízení nebo řádků kódu.

4.1.1 Proces překladu

Uživatel má mít možnost přeložit program klasickým způsobem i se zapnutou instrumentací. Pro zapnutí instrumentace je nutné zapnout nástroj *tforcov*.



Obrázek 4.2: EPC diagram zobrazující běžný překlad bez zapnuté instrumentace a bez použití nástroje *tforcov* (*Event-Driven Process Chain Diagram*).

Při překladu bez instrumentace uživatel spustí překladač běžným způsobem s běžnými argumenty. Výstupem je neinstrumentovaný spustitelný program. Pro následné porovnání s instrumentovaným překladem je klasický překlad zobrazen na diagramu 4.2.

Instrumentovaný překlad je zobrazen na diagramu 4.3. Spuštěním nástroje *tforcov* je vytvořen *wrapper*, který je spuštěn místo přímého spuštění překladače. Je spuštěn stejným způsobem se stejnými argumenty jako by byl spuštěn překladač přímo. To je důležité pro jednoduchost použití nástroje *tforcov*. Jeho úkolem je připravit prostředí pro běh nástroje *Tforc*, který poté spustí překladač se stejnými argumenty, které dostal *wrapper* nástroje *tforcov*.

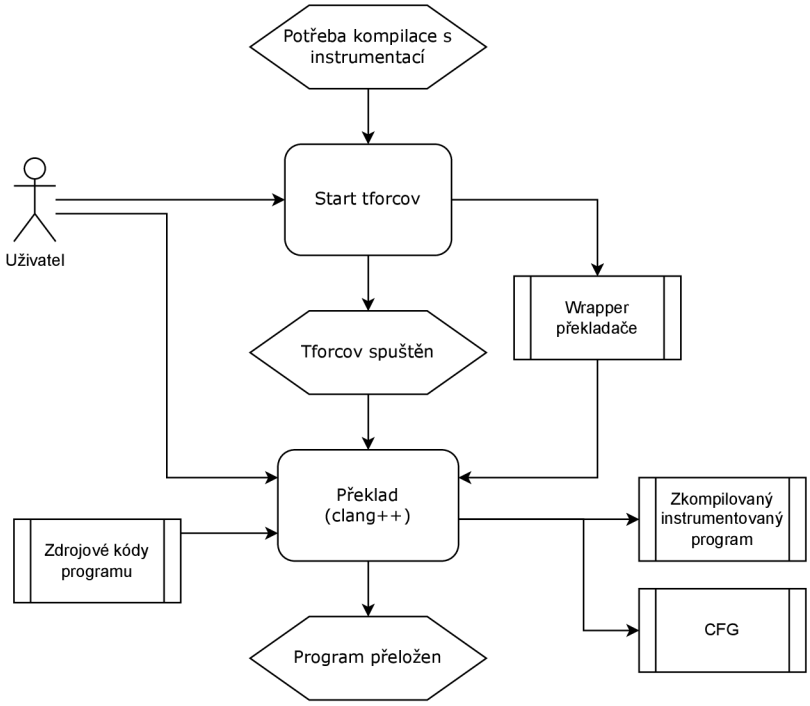
Ve fáze překladu dochází k instrumentaci programu tak, aby bylo možné zachytit a trasovat budoucí běhy programu. Kód programu je při překladu rozšířen tak, aby vybrané části programu měly vedlejší efekt. Tento vedlejší efekt znamená další výstup, ze kterého lze určit, které části programu byly provedeny.

Dalším výstupem ve fázi překladu je získaný CFG. Ten obsahuje i strukturu modulů a funkcí, které nejsou přímo použity pro vyhodnocení, ale udržují členění kódu, které je pro uživatele přehledné a odpovídá struktuře souborů v překladu. V rámci funkcí jsou obsaženy základní bloky, jejichž pomocí jsou později vyhodnocována kritéria pokrytí. V rámci těchto základních bloků jsou získána i další data o zápisech a čtení proměnných nebo umístění na řádcích zdrojového kódu.

4.1.2 Proces testování

Proces testování není pro uživatele z pohledu provedení nijak odlišný. Uživatel spouští testy a program jako vedlejší efekt generuje logovací výstupy. Jedinou odlišností je zpomalení běhu, kvůli přidané režií. Proces testování je zobrazen v diagramu 4.4.

Při testování se pouze používá program instrumentovaný, ale další kód nástroje *tforcov* není použit. Očekávané je zejména testování programu automatickými testy, protože výstup třetí fáze by poté vyhodnotil pokrytí pro danou testovací sadu.



Obrázek 4.3: EPC diagram zobrazující překlad s instrumentací programu pomocí nástroje *tforcov* (*Event-Driven Process Chain Diagram*).

4.1.3 Proces vyhodnocení pokrytí

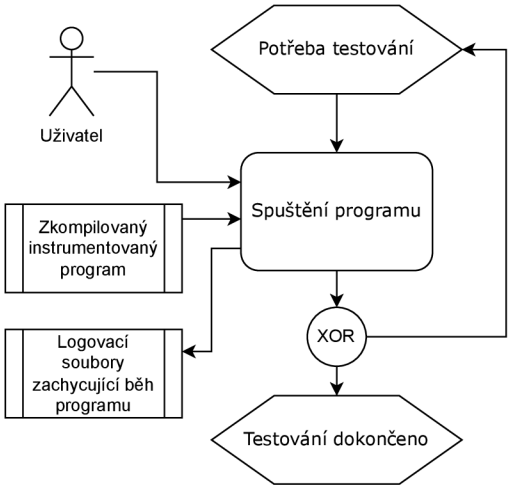
Ve třetí fázi použití nástroje *tforcov* jsou vyhodnocena zvolená pokrytí pro zvolenou část programu. Do této fáze se uživatel přesune, když již nechce dále program spouštět a testovat. Proces vyhodnocení je zobrazen v diagramu 4.5.

Uživatel spustí nástroj *tforcov*, aby toto vyhodnocení provedl. Pokud uživatel testoval program sadou automatických testů, tak tuto sadu může rozšiřovat pomocí identifikovaných nedostatků dané testovací sady. Pro vyhodnocení jsou použity výstupy z překladu a testování.

4.2 Specifikace potřebných výstupů instrumentace

Pro možné vyhodnocení měření pokrytí je základem získat data pomocí instrumentace. Specifikace dat, která jsou nutná pro vyhodnocení měření pokrytí, předchází návrhu rozšíření instrumentačního frameworku. Pokrytí lze z rozdělit do více kategorií.

- pokrytí řádků kódu,
- pokrytí roku řízení,
- pokrytí toku dat.



Obrázek 4.4: EPC diagram zobrazující testování zahrnující spouštění instrumentovaného programu (*Event-Driven Process Chain Diagram*).

4.2.1 Data pro vyhodnocení kritérií toku řízení

Pro vyhodnocení kritérií toku dat je nutné získat reprezentaci zdrojového kódu ve tvaru grafu toku řízení(CFG). Dále je nutné u zkoumaných částí kódu při provedení bloků kódu toto provedení zaznamenat se zachováním pořadí provedení jednotlivých bloků.

4.2.2 Data pro vyhodnocení kritérií pokrytí řádků kódu

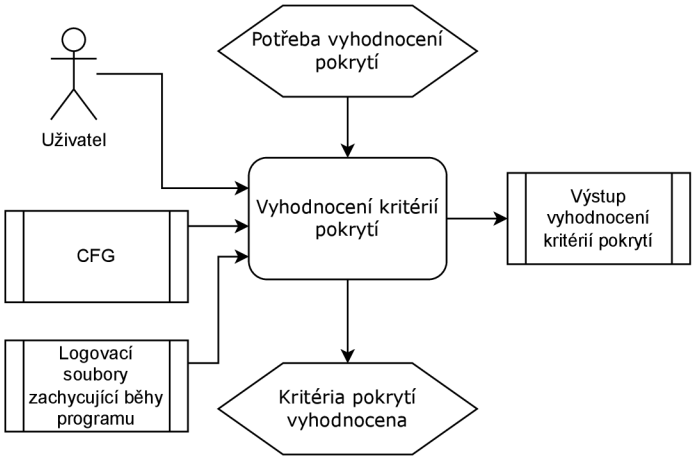
Pro výpočet pokrytí řádků kódu lze využít bloky kódu instrumentované pro měření kritérií toku řízení. V rámci CFG je dostačující zjistit rozsahy jednotlivých bloků na řádcích kódu, tedy například blok kódu začíná na řádku 8 a končí na řádku 12.

4.2.3 Data pro vyhodnocení kritérií toku dat

Pro vyhodnocení toku dat lze opět využít bloky kódu. Pokud daný blok obsahuje zápis nebo čtení proměnné, tak při provedení bloku kódu je zároveň provedeno i toto čtení nebo zápis. Tedy v rámci CFG je nutné získat i seznam zápisů a čtení proměnných se zachováním pořadí.

4.3 Rozšíření nástroje tforc

Nástroj *Tforc* umožňuje více způsobů instrumentace. Pomocí původní funkcionality nástroje by bylo možné vyhodnotit kritéria toku dat a pokrytí funkcí, ale nebylo by možné vyhodnotit pokrytí toku řízení v rámci funkcí ani pokrytí řádků kódu. Několik pokrytí je definováno pomocí CFG a je tedy nutné přidat instrumentaci základních bloků, aby bylo možné tato pokrytí vyhodnotit. Následující podkapitoly obsahují popis důležitých částí původního stavu a návrh rozšíření nástroje.



Obrázek 4.5: EPC diagram zobrazující vyhodnocení kritérií pokrytí pomocí načtení CFG získaného při překladu a logovacích souborů získaných při testování (*Event-Driven Process Chain Diagram*).

4.3.1 Konfigurační soubor

Pro instrumentaci je třeba mít vytvořené konfigurační soubory, pomocí kterých lze instrumentaci nastavit. Výchozí umístění konfiguračního souboru ve zdrojovém adresáři instrumentovaného programu je *tforc_query/main_CB.cpp* a výchozí umístění souboru s funkcemi pro instrumentaci je ve složce *tforc_callback*.

4.3.2 Současný specifikační jazyk pro konfiguraci

Existuje pět různých druhů konfigurace instrumentace.
Specifikace názvu souboru s konfigurací pro instrumentaci.

> configuration: main.query

Specifikace souboru obsahující funkce v jazycích *C* nebo *C++*, které budou používány při instrumentaci.

> callBackFunctions: main_CB.cpp

Použití nepřímého adresování při instrumentaci:

> indirectAddressing: yes

Instrumentace funkce:

> func: afterCountDown after _Z9countDownii [0,1]

Instrumentace globálních a lokálních proměnných:

> load: glob_variable _Z8showInfoPiiPc [ADDRESS, VALUE, LOCATION]
> store: _Z9countDownii:loc_variable showValue [VALUE] CCODE

Tyto volby jsou blíže popsány v práci zabývající se tvorbou nástroje *Tforc*[20] a nebude možné je v rámci nástroje *tforcov* měnit. Nástroj *tforcov* bude tvořit abstrakci nad těmito hodnotami, a proto nejsou v této práci více popsány.

4.3.3 Rozšíření specifikačního jazyka pro konfiguraci

Pro možnost zapnutí měření pokrytí pomocí instrumentace je přidána následující konfigurace, kde *expr* umožňuje zadat název funkce se znakem * pro libovolnou posloupnost znaků.

coverage: m:[expr] **m!:**[expr] **coverage: s:**[expr] **s!:**[expr] **f:**[expr] **f!:**[expr] **!all all**

Jednotlivé volby konfigurace lze uvést vícekrát, lze je vynechat a lze je uvést v libovolném pořadí.

coverage: m!:[expr1] **m!:**[expr2] **all**

Význam jednotlivých položek v konfiguraci má následující význam:

- **coverage**: klíčové slovo pro vložení před ostatní konfigurace pro měření pokrytí,
- **m:[expr]**: modul vyhovující alespoň jednomu výrazu *expr* je zahrnut,
- **m!:[expr]**: modul vyhovující alespoň jednomu výrazu *expr* je vyjmut,
- **s:[expr]**: název zdrojového souboru vyhovující alespoň jednomu výrazu *expr* je zahrnut,
- **s!:[expr]**: název zdrojového souboru vyhovující alespoň jednomu výrazu *expr* je vyjmut,
- **f:[expr]**: funkce vyhovující alespoň jednomu výrazu *expr* je zahrnuta,
- **f!:[expr]**: funkce vyhovující alespoň jednomu výrazu *expr* je vyjmuta,
- **!all**: klíčové slovo pro vyjmutí všech modulů a funkcí,
- **all**: klíčové slovo pro zahrnutí všech modulů a funkcí, toto klíčové slovo je implecitní a není třeba je uvádět.

Volby v konfiguraci mají stanovenou prioritu.

$$\text{m!} > \text{s!} > \text{f!} > \text{m} > \text{s} > \text{f} > \text{!all} > \text{all}$$

Následuje příklad konfigurace.

coverage: m!:*math* **m!:**utils.cpp **f!:**foo **all**

Při této konfiguraci by byl instumentován veškerý kód mimo funkce s názvem *foo*, mimo modul *utils.cpp* a mimo všech modulů, které mají v názvu *math*.

4.4 Rozšíření nástroje Tforc-tool

V této podkapitole je popsáno, jak bude nutné rozšířit a upravit nástroj *Tforc-tool*. Nástroj *Tforc-tool* zajišťuje jednodušší spouštění nástroje *Tforc* a řídí překlad. Tato funkcionalita bude využita téměř v nezměněném stavu. Jde o dvě části:

- přidání podpory pro nové konfigurační možnosti nástroje *Tforc*,

- změna způsobu spouštění testů.

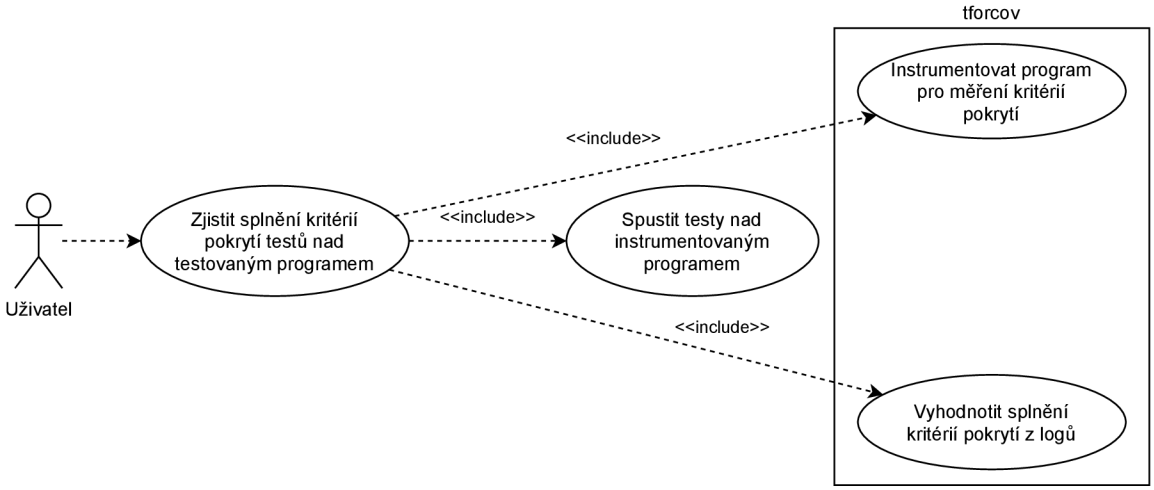
Rozšíření nástroje *Tforc* bude potřebovat podporu i u nástroje *Tforc-tool*. Konkrétně rozšíření možností konfiguračního souboru je třeba promítnout i do nástroje *Tforc-tool* tak, aby konfiguraci nástroj *Tforc-tool* předat nástroji *Tforc*.

Úprava spouštění testů není předmětem této práce, a tedy bude provedena při dostatku času. Nástroj *Tforc-tool* vyžaduje spouštění testů pod jiným uživatelem než je root. Nechtěným důsledkem této podmínky je, že pro automatické testování nelze použít téměř žádné běžné *Docker image* v původním stavu, ale tento *image* je třeba upravit například přidáním uživatele.

Dále jsou testy napsány jako stromová struktura shellových skriptů, což komplikuje i úpravy popsané v předchozím odstavci. Skripty obsahují velké množství duplicitního kódu. Ideálním řešením by bylo každý test nadefinovat jinak než jako zdrojový soubor, soubory pro *Tforc-tool* a shellový skript. Místo skriptu by bylo vhodné mít konfigurační soubor, který by obsahoval informace, jak má být test spuštěn a vyhodnocen. Tím by se odstranil duplicitní způsob spouštění každého testu a bylo by případně možné jednoduše přidat nebo měnit způsob spouštění testů nebo testovací framework nad testy.

4.5 Návrh nástroje tforcov

Tato podkapitola popisuje návrh nástroje *tforcov* na vysoké úrovni neboli systémový pohled na nástroj, který reprezentuje nástroj z pohledu uživatele.



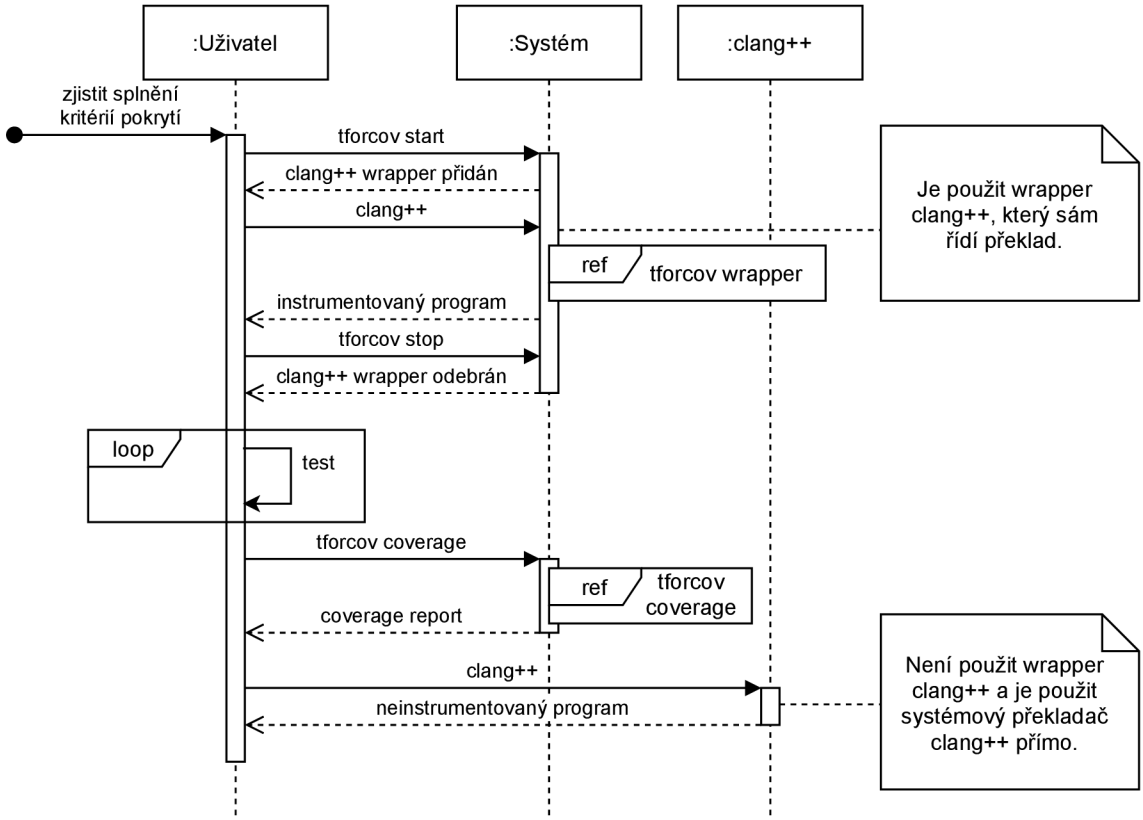
Obrázek 4.6: Diagram použití nástroje *tforcov*.

Uživatelem je vývojář, který má softwarový produkt, u kterého chce změřit, jakých výsledků dosáhne daný softwarový produkt nebo jeho část ve vybraných kritériích pokrytí. Protože je uživatelem vývojář, tak by diagramům v této podkapitole měl porozumět.

Zjištění kritérií pokrytí se skládá ze tří kroků, jak je ukázáno i na obrázku 4.6:

1. instrumentace programu,
2. provádění programu a pouštění testů nad programem a
3. vyhodnocení splnění kritérií pokrytí z logů vytvořených v předchozím bodě.

Způsob použití a návaznosti jednotlivých případů použití je zobrazen v systémovém diagramu na obrázku 4.7.



Obrázek 4.7: Systémový sekvenční diagram nástroje *tforcov*.

V prvním kroku uživatel pomocí skriptu *tforcov* může provést zapnutí nebo vypnutí instrumentace při překladu. Zapnutí provede spuštěním skriptu s parametrem start a vypnutí spuštěním skriptu s parametrem stop. Při zapnuté instrumentaci se při uživatelově zavolání překladače clang++ zavolá místo překladače *tforcov* wrapper, který se postará o konfiguraci a spuštění nástrojů *Tforc-tool* a *Tforc*. Nástroj *Tforc-tool* poté pomocí nástroje *Tforc* instrumentuje zdrojové kódy v jazyce *LLVM IR* a spustí překlad. Uživatel může pomocí konfiguračního souboru zvolit, které části kódu chce instrumentovat. Filtrovat může na úrovni modulů(např. main.cpp) a funkcí(např. foo). Výstupem je instrumentovaný spustitelný program. Podrobněji je fungování wrapperu popsáno v podkapitole 4.6.

V druhém kroku má Uživatel instrumentovaný program a může ho spouštět. Předpokládaným použitím je spuštění testovací sady, kterou uživatel k programu má, ale výpisy budou generovány libovolným pouštěním instrumentovaného programu. Instrumentované části kódu budou vypisovat do souborů, které základní bloky byly provedeny. Tyto výpisy jsou nutné pro výpočet měření pokrytí.

Ve třetím kroku uživatel může spustit výpočet vyhodnocení měření kritérií pokrytí. K tomuto kroku uživatel sám přistoupí, když už nechce žádná další spuštění programu do měření zahrnout. Uživatel může parametrizovat fungování dvěma způsoby. Může zvolit, která kritéria pokrytí mají být vyhodnocena a pro které části kódu. Stejně jako v prvním kroku, tak i zde může uživatel filtrovat části kódu na úrovni modulů a funkcí. Podrob-

nější návrh komponenty nástroje *tforcov* pro vyhodnocení měření kritérií pokrytí je popsán v podkapitole 4.6.

4.5.1 Konfigurace nástroje

Nástroj lze konfigurovat pomocí souboru **.tforcov**. Zde jsou dva způsoby konfigurace. Prvním způsobem je výběr částí kódu pro instrumentaci a druhým způsobem je výběr kritérií pokrytí pro vyhodnocení.

Výběr částí kódu pro instrumentaci je uveden klíčovým slovem **coverage** stejně, jak bylo popsáno v podkapitole 4.3.3 u rozšíření konfiguračního jazyka nástroje *Tforc-tool*.

match: m:[expr] **m!:**[expr] **s:**[expr] **s!:**[expr] **f:**[expr] **f!:**[expr] **!all all**

Výběr kritérií pokrytí, která mají být vyhodnocena je uveden klíčovým slovem **criteria**. Za klíčovým slovem může následovat seznam kritérií pokrytí nebo klíčové slovo **all**.

criteria: [list kritérií | all]

Například při případě volby specifických kritérií pokrytí **MCDC** a **line coverage** by obsah konfiguračního souboru vypadal, jak je uvedeno níže.

criteria: mcdc linecoverage

V případě zapnutí vyhodnocení všech implementovaných kritérií pokrytí by bylo použito samotné klíčové slovo **all**, jak je ukázáno níže. Tato volba je výchozí, a tedy není nutné ji do souboru uvádět.

criteria: all

Pro vložení komentářů do konfiguračního souboru, je zvolen znak #, kdy všechny znaky následující za tímto znakem až po konec řádku jsou považovány za komentář. Komentáře nemají vliv na funkcionalitu.

```
# komentář
criteria: all # další komentář
```

Volba částí kódu **coverage** je použita komponentou nástroje *tforcov* pro instrumentaci blíže představenou v podkapitole 4.6 a komponentou nástroje *tforcov* pro vyhodnocení kritérií pokrytí blíže představenou v podkapitole 4.7.

4.6 Návrh komponenty pro instrumentaci nástroje tforcov

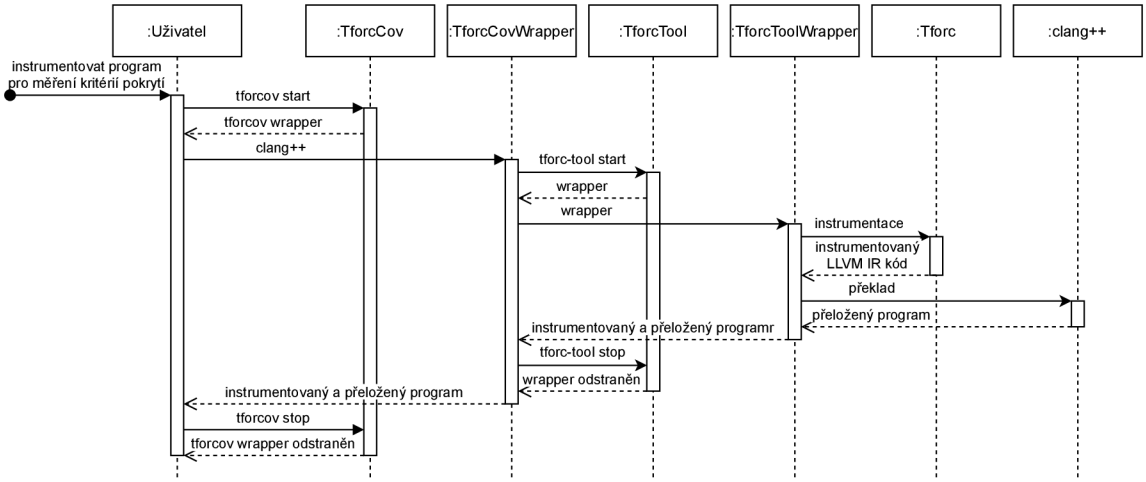
Tato podkapitola popisuje fungování instrumentační komponenty nástroje *tforcov*.

Pro instrumentaci je nutné nejdříve zapnout *tforcov*.

```
tforcov start [-m (path)] [-c name]
```

Zapnutí způsobí, že podle zvolené metody připravení wrapperu *TforcCovWrapper*, bude wrapper připraven pro spuštění. Výchozí metoda pro spuštění je následující:

- **path**, která vrátí cestu s wrapperem pojmenovaným stejně jako překladač, kterou může uživatel přidat na začátek proměnné prostředí *PATH*.



Obrázek 4.8: Sekvenční diagram instrumentačního wrapperu nástroje *tforcov*.

Pro použití nástroje stačí jedna implementovaná metoda, ale je zde případně možné nástroj rozšířit o další metody, které by mohly být například následující:

- **replace** je výchozí metoda, která nahradí binární soubor clang++,
- **set_path**, která nastaví cestu s wrapperem pojmenovaným stejně jako překladač na začátek proměnné prostředí *PATH*,
- **binary**, která přidá spustitelný soubor mezi ostatní systémové programy a uživatel může volat *tforcov-wrapper* místo *clang++*.

Pomocí parametru *-c* lze nastavit název překladače (např. **clang++** nebo například **clang++11**). Výchozí hodnotou je například **clang++**.

Po zavolání wrapperu dojde k načtení konfiguračního souboru **.tforcov** popsaného v podkapitole 4.5.1. Podle něj se připraví konfigurační soubory pro *Tforc* a spustí se *Tforc-tool*, který spustí instrumentaci nástrojem *Tforc* a poté překlad.

Po dokončení překladu lze vypnout *tforcov*.

```
tforcov stop
```

Vývojář má v po překladu instrumentovaný program, který může spouštět a tvořit ním tak výpisy do souboru, ze kterého se později vyhodnotí splnění kritérií pokrytí.

4.7 Návrh komponenty pro vyhodnocení kritérií pokrytí nástroje tforcov

Podkapitola obsahuje podrobnější návrh komponenty nástroje *tforcov* pro vyhodnocení kritérií pokrytí. Nejdříve je ukázán návrh rozhraní a spuštění nástroje v podkapitole 4.7.1. V podkapitole 4.7.2 je diagram tříd. Fungování a opodstatnění návrhu je rozebráno pomocí sekvenčního diagramu v podkapitole 4.7.3 s pomocí obrázku 4.10. Sekvenční diagram se skládá ze čtyř částí, které jsou postupně vysvětleny v podkapitolách s obrázky 4.11, 4.12, 4.13 a 4.14.

- načtení CFG,

- načtení provedených cest,
- vyhodnocení kritérií pokrytí a
- export výsledků.

4.7.1 Spuštění nástroje

Pro spuštění vyhodnocení kritérií pokrytí nad vytvořenými výpisy instrumentovaným program je nutné spustit komponentu nástroje *Tforc* pro vyhodnocení kritérií pokrytí příkazem *tforcov coverage*:

```
tforcov coverage
  [–criteria [(primepath|line|alldefs|edgepair|...)]]
  [–match [(m:[expr]|m!:[expr]|s:[expr]|s!:[expr]|f:[expr]|f!:[expr]|!all|all)]]
  [–export [(xml|html|json|txt|md|...)]]
  [–logs-path path]
  [–cfg path]
```

Všechny parametry jsou volitelné a mají následující význam:

- **criteria** je seznam kritérií pokrytí k vyhodnocení,
- **match** je seznam výrazů pro filtrování modulů a funkcí pro vyhodnocení, který byl blíže popsán v podkapitole 4.3.3,
- **export** je seznam formátů pro export výstupu,
- **logs-path** je adresář, ve kterém budou budou hledány logovací soubory zachycující běhy programu,
- **cfg** je adresář, ve kterém budou budou hledány soubory obsahující výpisy grafů toku řízení daného programu,

V konfiguračním souboru mohou být obsaženy informace se stejným významem, jako mají parametry **criteria** a **match**. V případě použití těchto parametrů platí následující priority:

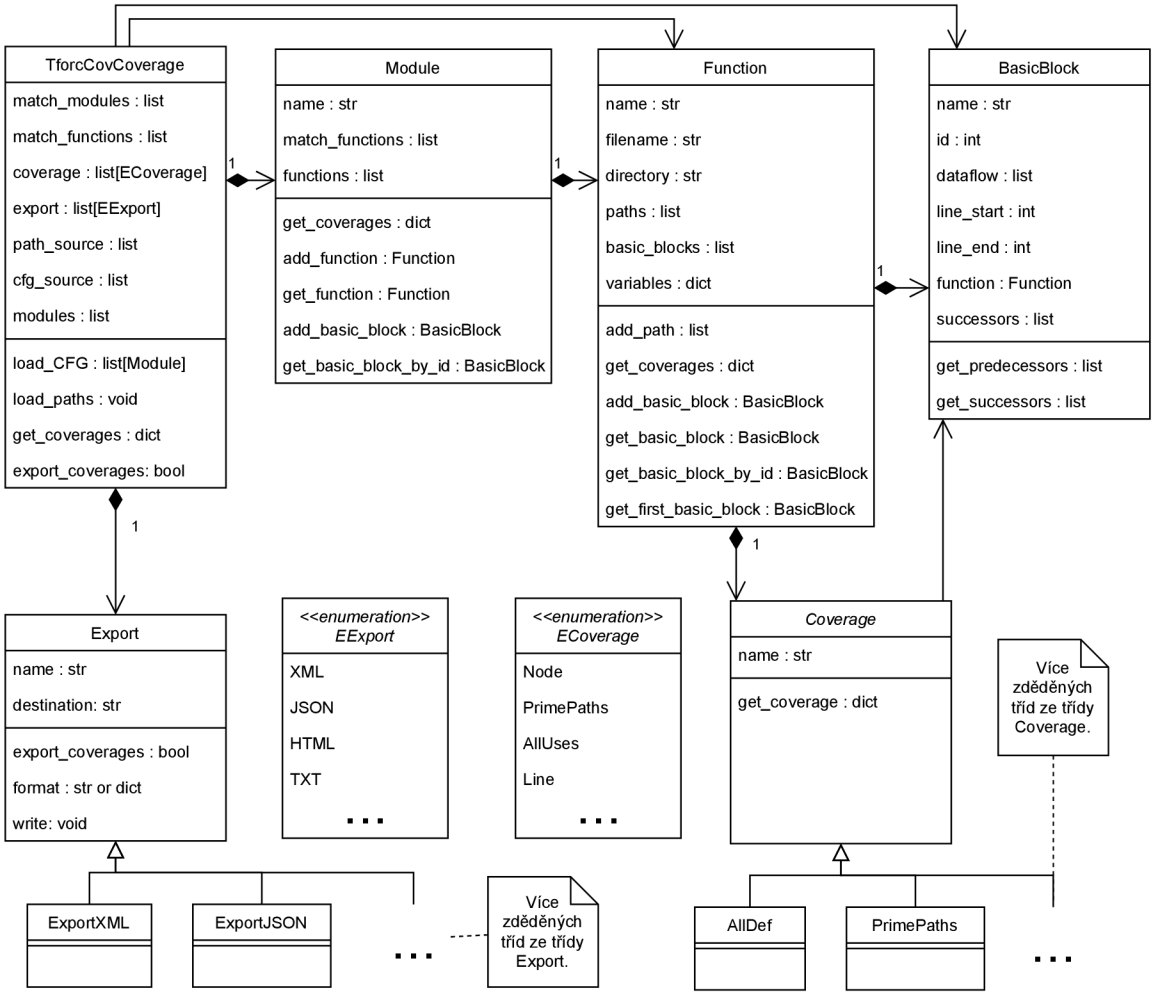
$$\text{criteria} > \text{path}$$
$$\text{match} > \text{path}$$

4.7.2 Diagram tříd

Diagram tříd popisuje vztahy mezi třídami, jejich funkce a atributy. Návrh tříd musí umožnit spouštění nástroje způsobem popsaným v podkapitole 4.7.1.

Hlavní třídou je *TforcCovCoverage*, která řídí chod nástroje. Má na starost propojení a logickou návaznost prováděných kroků v rámci této komponenty nástroje *tforcov*. Fungování čtyř kroků je blíže ukázáno na sekvenční diagramech v podkapitole 4.7.3.

Třídy *Module*, *Function* a *BasicBlock* drží strukturu grafu toku řízení CFG. Struktura těchto objektů kopíruje strukturu prvků v *LLVM*, kde tyto prvky tvoří stromovou strukturu a jednotlivé úrovně jsou modul, funkce, základní bloky, instrukce a operandy. V instrukcích a operandech nejsou pro kritéria pokrytí potřebné všechny informace, a tedy důležité informace jsou abstraktněji drženy v rámci třídy *BasicBlock*.



Obrázek 4.9: Diagram tříd nástroje *tforcov*.

Podtřídy třídy *Coverage* mají za úkol vyhodnotit dané kritérium pokrytí pro v rámci nadřazené funkce.

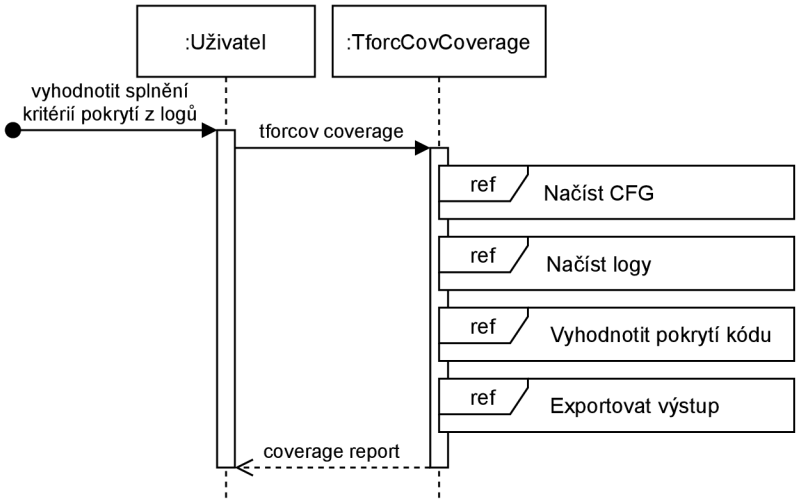
Podtřídy třídy *Export* se starají o formátování výstupu měření pokrytí a zápis do souboru.

Při návrhu bylo v bránu v potaz případné rozšíření o další kritéria pokrytí nebo další možné výstupní formáty. To se konkrétně projevuje dědičností tříd pro jednotlivá kritéria pokrytí ze třídy *Coverage* a dědičností tříd pro export výstupu pro různé formáty ze třídy *Export*. Pro přehlednost byly přidány i výčty možných hodnot pro tyto dvě třídy. Výčty se jmenují *ECoverage* a *EExport* a jsou označeny jako stereotyp *«enumeration»*.

4.7.3 Sekvenční diagram

Sekvenční diagram na obrázku 4.10 zobrazuje funkcionalitu hlavní a řídící třídy *TforcCovCoverage*. Třída postupně provádí následující čtyři kroky, které jsou podrobněji rozebrány v následujících podkapitolách:

- načtení CFG,



Obrázek 4.10: Sekvenční diagram měření pokrytí nástroje *tforcov*.

- načtení provedených cest,
- vyhodnocení kritérií pokrytí a
- export výsledků.

Sekvenční diagram načtení CFG

Sekvenční diagram na obrázku 4.11 zobrazuje postupné plnění datových tříd *Module*, *Function* a *BasicBlock*. Zde je prováděno i filtrování modulů a funkcí podle vstupních parametrů nástroje. Do objektů třídy *BasicBlock* jsou vložena data potřebná pro kritéria pokrytí, tedy rozsah řádků kódu, na kterém se *BasicBlock* nachází následníci, použití proměnných a název.

Sekvenční diagram načtení provedených cest

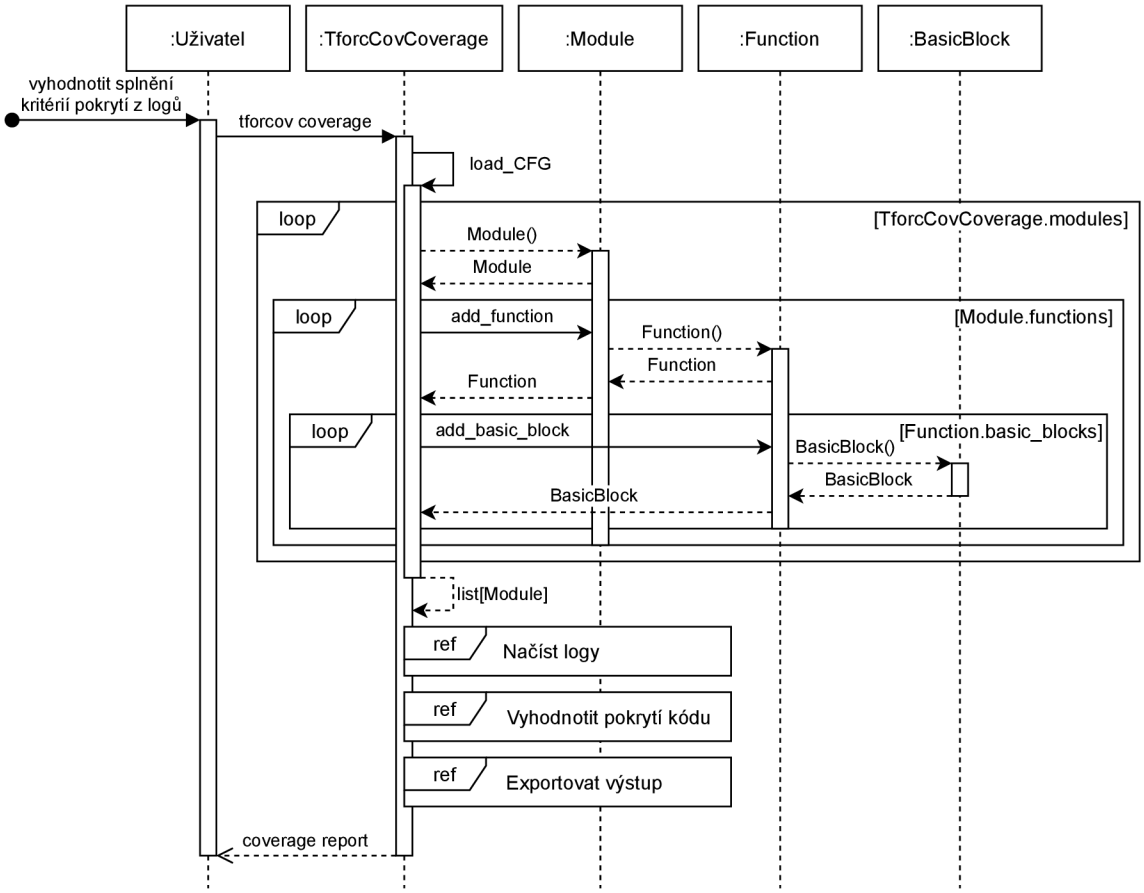
V diagramu tříd na obrázku 4.9 jsou zachyceny vykonané cesty v rámci spouštění instrumentovaného programu. Jednotkou pro kterou se počítají pokrytí je funkce, a tedy tyto cesty lze najít v atributu *paths* třídy *Function*. Sekvenční diagram na obrázku 4.12 zobrazuje načtení cest ze souborů, které řídí třída *TforcCovCoverage*, protože v rámci provádění jedné funkce může být tok řízení předán do jiné třídy nebo jiného modulu. Cesty obsahují pouze provedené základní bloky v rámci dané funkce a základní bloky jiných funkcí a modulů jsou odfiltrovány.

Při načítání vstupních souborů si funkce *load_paths* pamatuje své zanožení. K tomu potřebuje u každé funkce zjistit, který základní blok je první (vstupní) a který je poslední (výstupní). Toto dokáže zpracovat více vnořených volání funkce i rekurzivní funkce.

Zanoření není v diagramu nějak více zvýrazněno z důvodu přehlednosti, ale pro každou funkci se provedou právě tři volání, tak jak je zobrazeno na obrázku 4.12.

Sekvenční diagram vyhodnocení kritérií pokrytí

Pro vyhodnocení pokrytí existuje třída *Coverage*. Z této třídy dědí konkrétní třídy pro jednotlivá kritéria pokrytí. Sekvenční diagram na obrázku 4.13 zobrazuje zjištění výsledků



Obrázek 4.11: Sekvenční diagram měření pokrytí nástroje *tforcov* zobrazující načtení CFG.

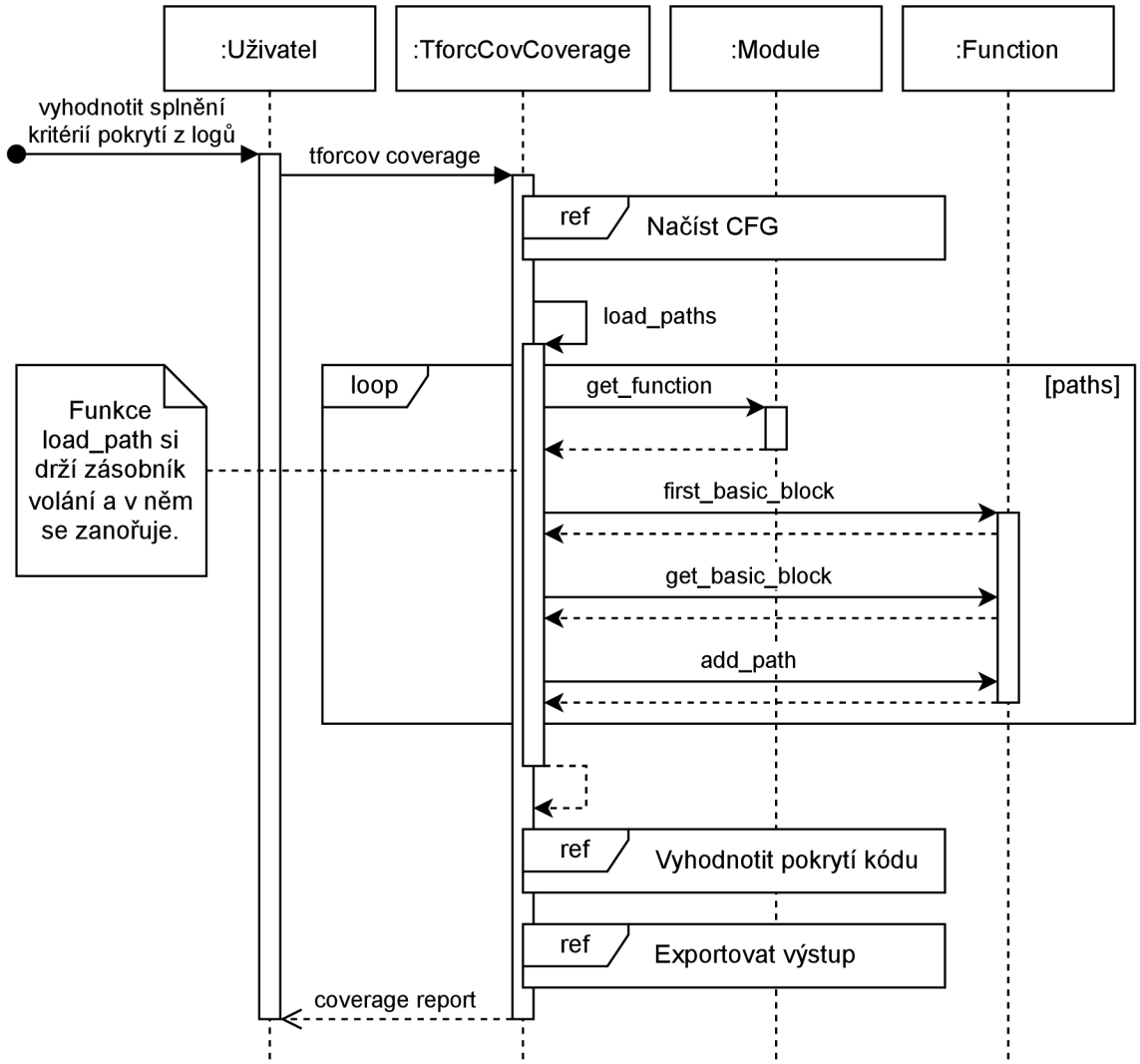
kritérií pokrytí, které se provádí průchodem stromovou strukturou přes *TforcCovCoverage*, *Module*, *Function* a *Coverage*. Jednotlivé podtřídy *Coverage* pro svou funkcionalitu prochází data v třídách *Function* a *BasicBlock* a na základě vlastní funkcionality vyhodnotí pokrytí.

Průběh vyhodnocení splnění kritéria pokrytí se bude skládat se tří kroků:

1. průchod CFG funkce neboli průchod strukturou základních bloků a vyhledání prvků nutných pro splnění daného pokrytí pro danou funkci,
2. průchod cest v atributu *Function.paths* a hledání výskytů prvků definovaných v prvním bodě,
3. ověření splnění pokrytí a návrat výsledků.

Například v případě *Edge-Pair Coverage* budou kroky vyhodnocení kritéria pokrytí vypadat následovně:

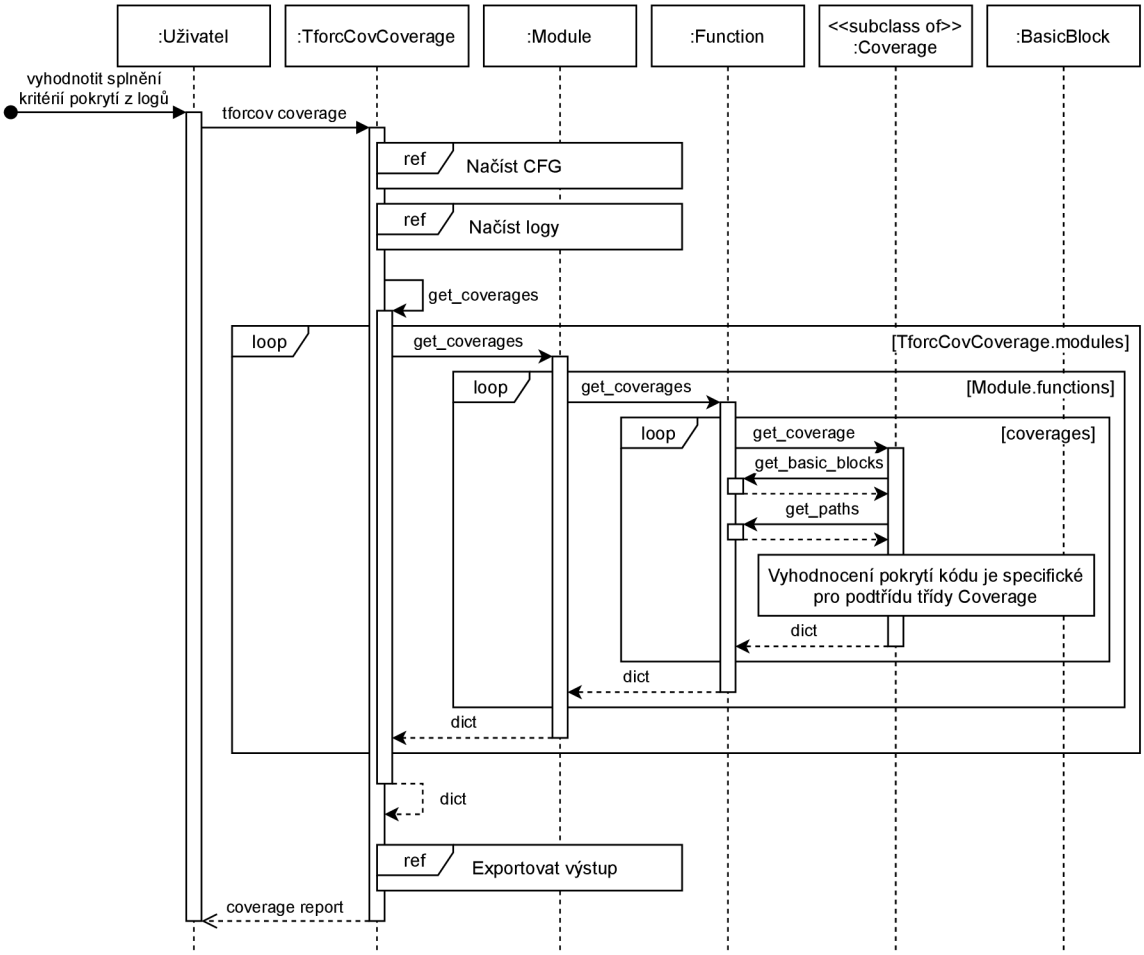
1. vyhledání všech možných kombinací dvojic přechodů mezi bloky,
2. průchod všech cest v atributu *Function.paths* a zaznamenání průchodu ke všem dvojicím nalezeným v prvním bodě, pokud se objeví jako součást cesty,
3. kontrola všech dvojic, zda byly nalezeny v nějaké cestě, a návrat výsledků.



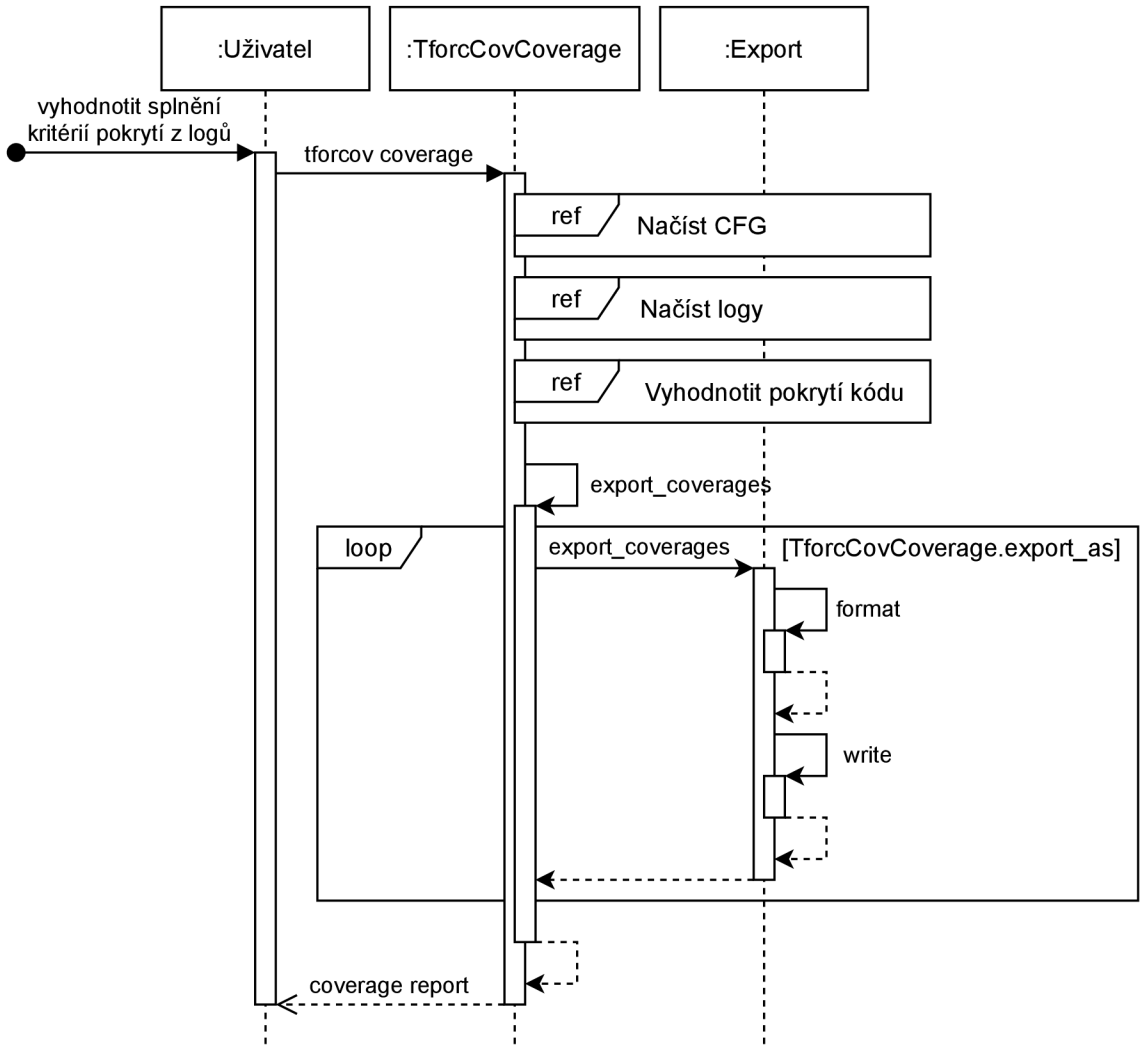
Obrázek 4.12: Sekvenční diagram měření pokrytí nástroje *tforcov* zobrazující načtení provedených cest.

Sekvenční diagram exportování výsledků

Při exportu výstupu třída *TforcovCoverage* předá data získaná při vyhodnocení pokrytí do vybraných podtříd *Export*, které provedou zformátování dat a zápis výstupu do souboru.



Obrázek 4.13: Sekvenční diagram měření pokrytí nástroje *tforcov* zobrazující měření na základě struktury basic blocků a provedených cest.



Obrázek 4.14: Sekvenční diagram měření pokrytí nástroje *tforcov* zobrazující export naměřených výsledků kritérií pokrytí.

Kapitola 5

Implementační detaily monitoru pokrytí

Kapitola obsahuje popis implementace monitoru pokrytí. V jednotlivých podkapitolách je blíže představeno fungování jednotlivých částí implementace s příklady a obrázky představující danou část funkcionality monitoru pokrytí.

První podkapitola 5.1 se zabývá rozšířením nástroje *Tforc*, kde je popsáno získání grafu toku pokrytí a vložení zpětných volání funkcí (instrumentace). Podkapitola 5.2 popisuje zpracování volání funkcí a organizaci dat získaných těmito voláními tak, aby bylo možné trasovat běhy programů. Následující podkapitola 5.3 popisuje načtení dat získaných v předchozích částech, tedy graf toku pokrytí a běhy programů. Na to navazuje podkapitola 5.4, která ukazuje vyhodnocení jednotlivých kritérií pokrytí z načtených dat. Exportování výsledků vyhodnocení kritérií pokrytí popisuje podkapitola 5.5. Vysvětlení fungování monitoru pokrytí při sledování běhu vícevláknových aplikací je v podkapitole 5.6, kde jsou zvýrazněny prvky řešení, které tuto funkcionalitu umožňují. V poslední podkapitole 5.7 jsou popsány závislosti monitoru pokrytí a jeho spuštění. Dále podkapitola ukazuje běhy jednotlivých částí monitoru pokrytí na příkladu změn pracovního prostoru, ve kterém jsou tvořeny soubory s daty z různých fází běhu monitoru pokrytí.

5.1 Rozšíření nástroje Tforc

Nástroj *Tforc* využívá metody *runOnModule* ze třídy *llvm::ModulePass*[1] ve které je zapouzdřena veškerá jeho funkcionalita. Nástroj *Tforc* byl rozšířen o instrumentace základních bloků. Přidání nové funkcionality tvoří většinu implementovaných úprav nástroje *Tforc*, ale úpravy zahrnovaly i opravy drobných chyb.

Instrumentace a získání CFG jsou provedeny v rámci jednoho průchodu strukturami *LLVM*. To umožňuje urychlit instrumentaci, protože se přes dané *LLVM* struktury iteruje pouze jednou, ale umožňuje to i vytvoření unikátních identifikátorů základních bloků, které jsou obsaženy v CFG a poté použity jako parametr o volané instrumentační funkce.

[1] Dokumentace třídy *llvm::ModulePass* https://llvm.org/doxygen/classllvm_1_1ModulePass.html

5.1.1 Získání CFG

Tato podkapitola popisuje získání, strukturu a organizaci jednotlivých CFG. Data reprezentující CFG jsou získána při instrumentaci a jsou vložena do komentářů na začátek souboru obsahující instrumentovaný zdrojový kód v jazyce *LLVM IR*.

Struktura získaných dat

Data jsou ve třech úrovních:

- modul,
- funkce a
- základní bloky.

Modul slouží pouze jako obálka pro množinu funkcí a nemusí mít jméno, proto další možnou identifikací modulu je jméno souboru. Pokud je překládáno více souborů najednou, například

clang++ main.cpp my_lib.cpp -o executable,

tak jsou funkce z obou zdrojových souborů kompilovány v rámci jednoho modulu s názvem *main.cpp*. Příklad získaných dat je ve výpisu 5.1.

```
1 ; module:
2 ; module name: <stdin>
3 ; module identifier: <stdin>
4 ; module file: class_constructor.cpp
5 ; module data layout: e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-
      n8:16:32:64-S128
6 ; module target: x86_64-pc-linux-gnu
```

Výpis 5.1: Příklad dat získaných k modulu.

V rámci modulu je obsaženo několik funkcí. Příklad dat získaných k funkci je ve výpisu 5.2. Každá funkce může obsahovat několik základních bloků. Funkce je identifikována názvem a obsahuje své umístění, které se skládá z:

- jména zdrojového souboru a
- složky umístění souboru.

Mezi funkcemi lze nalézt i systémové funkce, které jsou přidány například při použití vzájemného vyloučení (*std::mutex*), nebo pomocné funkce *LLVM*, mezi které patří například *llvm.dbg.declare*. Tyto funkce nejsou součástí měření pokrytí.

```
1 ; function: main
2 ; function filename: main.cpp
3 ; function directory: /workspace/src
```

Výpis 5.2: Příklad dat získaných k funkci.

Základní bloky obsahují nejvíce dat a tvoří většinu dat. V následujícím příkladu 5.3 je vidět základní blok, který má následující hodnoty:

- *%2* je identifikátor v používaný v *LLVM IR* reprezentaci.

- *id 2* je identifikátor vygenerovaný v rámci průchodu modulem a unikátní v rámci daného modulu. Pomocí tohoto čísla lze určit základní blok i funkci, ve které se základní blok nachází.
- *start 3* znamená, že základní blok v kódu začíná na řádku 3.
- *end 8* znamená, že základní blok v kódu končí na řádku 8.
- *store* a *load* označují zápisy a čtení proměnných. Jsou používané pro vyhodnocení pokrytí kritérií toku dat. Záleží na pořadí hodnot. Pokud je *store X* před hodnotou *load X*, tak se u pokrytí *def-use* bude jednat o jednu cestu, která bude splněna, pokud bude základní blok proveden.
- *variable %4 x* označuje pojmenování. Hodnota *%4* je reprezentace hodnoty(proměnná) v *LLVM IR*. V původním zdrojovém kódu *C++* má tato proměnná název *x*.

```
1  ; bb: %2
2  ; bb id: 2
3  ; bb start: 3
4  ; bb end: 8
5  ; bb store: %3
6  ; bb variable: %3 a
7  ; bb store: %4
8  ; bb variable: %4 x
9  ; bb load: %3
10 ; bb load: %4
11 ; bb store: %7
```

Výpis 5.3: Příklad dat získaných k základnímu bloku.

Vynechání funkcí systémových knihoven

V rámci kódu *LLVM IR* je obsaženo i několik funkcí z standardních knihoven jazyka *C++* nebo pomocných funkcí projektu *LLVM*. Tyto funkce nejsou předmětem instrumentace, a proto jsou z instrumentace i získání CFG vyjmuty. Pro rozeznání funkcí systémových je zkontrolována název souboru a název složky obsahující soubor, kde název složky je prázdný a naopak název souboru obsahuje absolutní cestu k systémové knihovně. Funkce *LLVM* mají název, který není validní pro běžné funkce v *C++*, protože obsahuje tečku.

5.1.2 Instrumentace základních bloků

Při instrumentaci základních bloků je do každého základního bloku instrumentované části kódu *LLVM IR* vloženo jedno zpětné volání funkce. Vložené zpětné volání slouží jako sonda. Zavolání této funkce z daného základního bloku indikuje provedení základního bloku. Podkapitola 5.1.2 popisuje umístění volání této funkce. Zpracování volání funkce je popsáno dále v 5.2.

Umístění volání funkce

Umístění volání funkce do kódu *LLVM IR* je nutné zvolit tak, aby neporušilo pravidla kódu *LLVM IR*. Na příkladu 5.4 je ukázán základní blok s instrukcí *landingpad*[2]. Tato instrukce

[2]Dokumentace instrukce *landingpad* jazyka *LLVM IR* https://llvm.org/docs/LangRef.html#landingpad-instruction

musí být první instrukcí základního bloku. Nelze tedy vložit žádné volání funkce před tuto instrukci. Kód *LLVM IR* obsahující tuto instrukci je vygenerován v tomto případě kvůli implementaci vzájemného vyloučení[3].

```
56:                                               ; preds = %54, %52, %50, ...
  %57 = landingpad { i8*, i32 }
          cleanup, !dbg !2736
  %58 = extractvalue { i8*, i32 } %57, 0, !dbg !2736
  store i8* %58, i8** %4, align 8, !dbg !2736
  %59 = extractvalue { i8*, i32 } %57, 1, !dbg !2736
  store i32 %59, i32* %5, align 4, !dbg !2736
  call void @_ZNSt11unique_lockISt5mutexED2Ev(%"class.std::unique_lock"* %3) #3, !
      dbg !2736
  call void @_Z15enterBasicBlockPci(i8* getelementptr inbounds ([11 x i8], [11 x i8]* @1, i32 0,
       i32 0), i32 16)
  br label %60, !dbg !2736
```

Výpis 5.4: Instrumentovaný základní blok začínající instrukcí *landingpad*, která musí být na první pozici.

Kód v jazyce *C++* 5.5 obsahuje na konci funkci základní bloky, které obsahují jen jedinou instrukci, která předává řízení a ukončuje provádění základního bloku. V takovém případě musí být volání vloženo jako první instrukce základního bloku. Příklad vložení volání instrumentující funkce je ukázán na kódu *LLVM IR* 5.6.

```
int foo(int var) {
    if (var > 1){
        return 1;
    }
    return 0; // return constant without loading any variable
}
```

Výpis 5.5: Funkce vracející konstantu, která je vrácena v krátkém základním bloku.

```
20:                                               ; preds = %19, %18
  call void @_Z15enterBasicBlockPci(i8* getelementptr inbounds ([7 x i8], [7 x i8]* @0, i32 0,
      i32 0), i32 7)
  br label %21

21:                                               ; preds = %20, %14
  call void @_Z15enterBasicBlockPci(i8* getelementptr inbounds ([7 x i8], [7 x i8]* @0, i32 0,
      i32 0), i32 8)
  ret i32 0, !dbg !960
```

Výpis 5.6: Instrumentované základní bloky v *LLVM IR*, které mají původní délku jedné instrukce.

Umístění volání funkce bylo zvoleno s ohledem na obecnost řešení. Z výše uvedených příkladů vyplývá, že vložené volání funkce

- nesmí být vždy na první pozici, protože první instrukce by například mohla být *landingpad*,

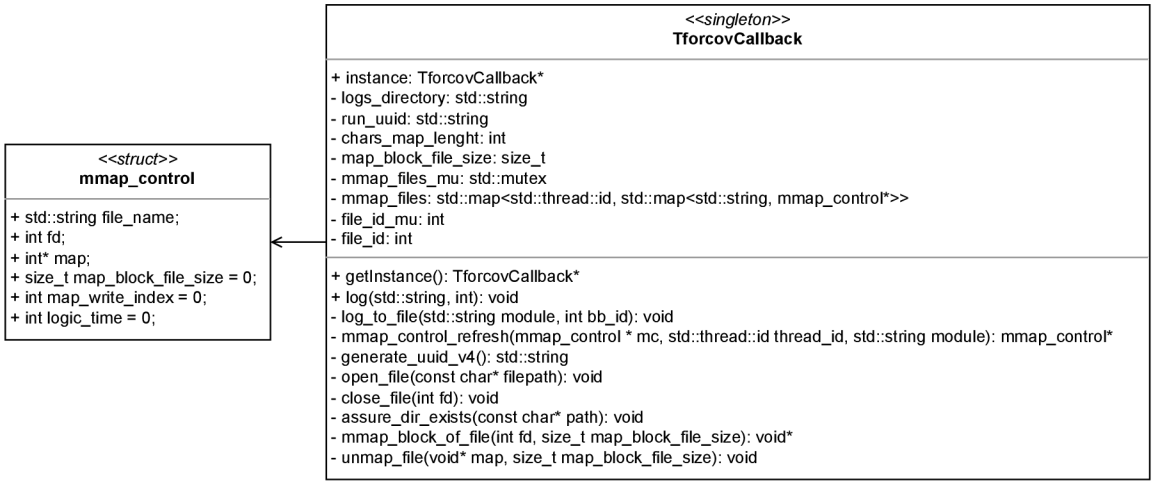
[3]Dokumentace vzájemného vyloučení jazyka *C++* (*std::mutex*) https://isocpp.org/wiki/faq/cpp11-library-concurrency#std-mutex

- nesmí být nikdy na poslední pozici, protože tam mohou být pouze ukončovací instrukce,
- nesmí být na druhé pozici, protože by to mohla být zároveň pozice poslední.

Volání instrumentované funkce je vkládáno na předposlední pozici, protože takové vložení neporuší pravidla jazyka *LLVM IR*.

5.2 Zachycení běhů programů

Tato podkapitola se zabývá použitím rozšířeného nástroje *tforc*. Pro sestavení jednotlivých validních cest pro CFG je při spouštění instrumentovaného programu je nutné sestavit samostatné cesty a odlišit jednotlivá spuštění programu, vlákna, moduly i třeba rekurzi funkcí. Při tomto rozlišení je třeba myslet i na časovou a prostorovou složitost řešení. V následujících podkapitolách bude vysvětlen způsob rozlišení logování provedení základních bloků za účelem sestavení jednotlivých cest CFG. Základní třída *TforcovCallback* pro zapouzdření funkcionality pro logování provedení základních bloků a organizace do jednotlivých běhů programů je na diagramu 5.1.



Obrázek 5.1: Třída *TforcovCallback* zajišťující organizaci a zápis logovacích výstupů.

5.2.1 Identifikace provedení základního bloku

Pro správnou identifikaci provedení základního bloku a možnost budoucího trasování provádění kódu je nutné identifikovat provedení základního bloku pomocí následujících údajů:

- pořadí nebo čas,
- vlákno,
- modul,
- funkce,
- základní blok.

Pořadí nebo čas slouží k sestavení provedených cesty v CFG, které jsou tvořeny posloupností základních bloků, Různá vlákna mohou provádět stejnou funkci a znalost vlákna je nutná k oddělení cest provedených různými vlákny ve stejné funkci. Identifikace modulu je nutná k identifikaci funkce a základního bloku. Funkce je nutná k identifikaci základního bloku. Při vyhodnocení kritérií pokrytí je přiřazena provedená cesta k této funkci. Cesta je tvořena neprázdnou posloupností základních bloků.

5.2.2 Volaná funkce

Funkce, jejíž volání je přímo vloženo do kódu *LLVM IR*, bere 2 argumenty. Prvním je název modulu a druhým unikátní identifikátor základního bloku v rámci modulu.

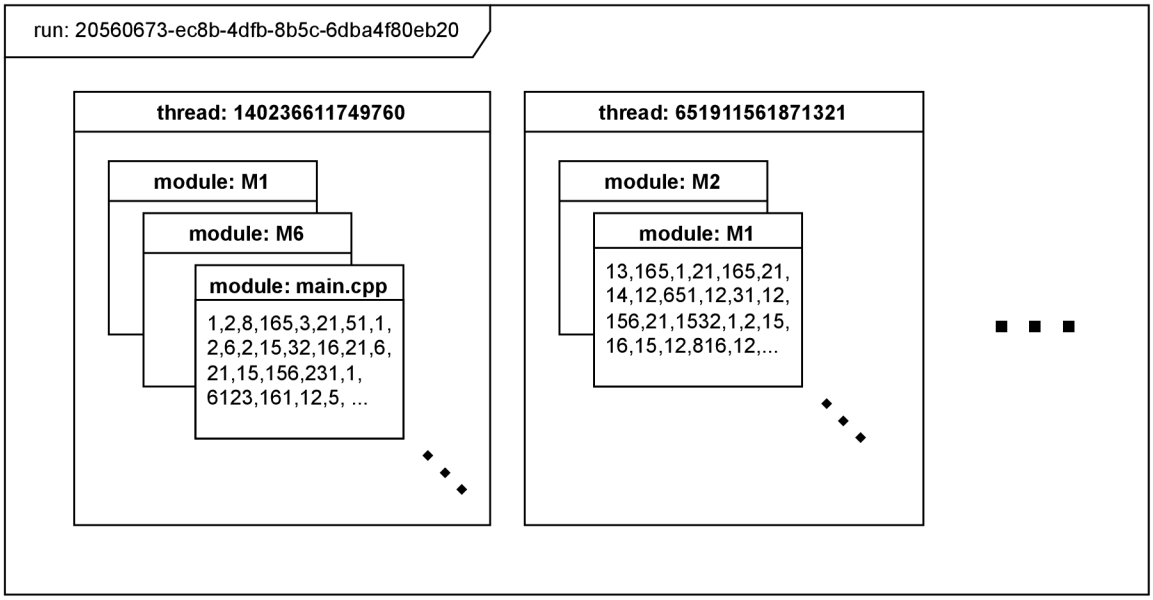
```
1 void enterBasicBlock(char* bb_module, int bb_block);
```

Výpis 5.7: Předpis funkce jejíž volání jsou vkládána do kódu *LLVM IR*

Funkce volá statickou funkci třídy *TforcovCallback::log*, která je vidět na diagramu 5.1. Funkce předá argumenty do této třídy, která se stará o organizaci logovacích výstupů. Volání této funkce splňuje tři z pěti požadavků na identifikaci základního bloku popsanou v 5.2.1 a zbylé dva požadavky jsou splněny uvnitř třídy *TforcovCallback*.

5.2.3 Organizace logovacích souborů

Pro zaznamenání prováděných základních bloků neboli trasování je nutné dodržet identifikaci základního bloku, jak byla popsána v 5.2.1. Funkce a základní blok jsou identifikovány unikátním číselným identifikátorem v rámci modulu, jak je popsáno i v 5.1.1.



Obrázek 5.2: Organizace souborů tvořených třídou *TforcovCallback* z diagramu 5.1.

Provádění základních bloků v různých modulem je zapisováno do různých souborů, protože dva základních bloky by mohly obsahovat základní bloky se stejným názvem, což by porušilo unikátnost základního bloku.

V případě vláken existují pro každé vlákno zvláštní soubory pro výstup. To slouží k oddělení několika různých cest CFG. Například pokud je v rámci běhu dvou vláken zavolána jedna funkce, tak je třeba tyto běhy oddělit.

Kombinace oddělení modulů a vláken tvoří matici souborů pro každé spuštění instrumentovaného programu, jak je vidět na obrázku 5.2. Při jednom spuštění instrumentovaného programu je vytvořen soubor, který obsahuje seznam a identifikaci veškerých logovacích souborů daného běhu programu. Příklad obsahu takového souboru je ve výpisu 5.8.

```
file : 20560673-ec8b-4dfb-8b5c-6dba4f80eb20-1
thread: 140236611749760
module: main.cpp
---
file : 20560673-ec8b-4dfb-8b5c-6dba4f80eb20-2
thread: 651911561871321
module: M1
---
file : 20560673-ec8b-4dfb-8b5c-6dba4f80eb20-3
thread: 140236611749760
module: M1
```

Výpis 5.8: Obsah souboru *20560673-ec8b-4dfb-8b5c-6dba4f80eb20*.

Jednotlivá spuštění programu jsou oddělena ve zvláštních souborech, jak je vidět na příkladu struktury souborů vytvořené dvěma spuštěními instrumentovaného programu 5.9. Z názvů souborů nelze rozeznat modul ani vlákno, pro které jsou v souboru obsaženy výpisy provedení základních bloků.

```
20560673-ec8b-4dfb-8b5c-6dba4f80eb20
20560673-ec8b-4dfb-8b5c-6dba4f80eb20-1
20560673-ec8b-4dfb-8b5c-6dba4f80eb20-2
20560673-ec8b-4dfb-8b5c-6dba4f80eb20-3
e05df113-7997-4f98-b3a3-26ab6b477877
e05df113-7997-4f98-b3a3-26ab6b477877-1
```

Výpis 5.9: Příklad seznamu vytvořených souborů dvěma spuštěními instrumentovaného programu.

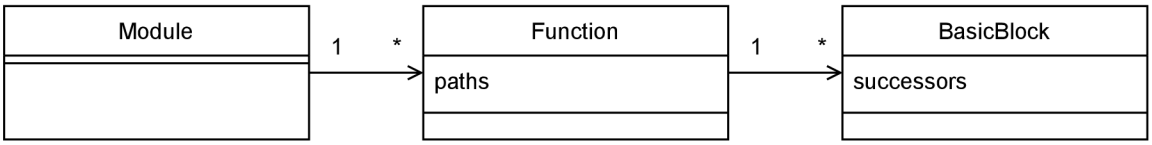
Soubory v jednotlivých spuštění instrumentovaného programu jsou označeny rostoucím číslem. Obsah souborů s logovacími výstupy je definován v souboru bez čísla, kde je uveden modul a vlákno, ve kterých jsou hodnoty získány.

Logovací soubor obsahuje seřazenou posloupnost číselných identifikátorů. Každá číselná odpovídá jednomu identifikátoru základního bloku. Pořadí číselných identifikátorů odpovídá pořadí provedení základních bloků. Volání funkcí jsou v sobě zanořena včetně rekurze, zpracování tohoto souboru je popsáno v 5.3.2.

5.3 Načtení dat pro vyhodnocení pokrytí

Pro vyhodnocení kritérií pokrytí je třeba načíst data, která budou později zpracována. Nejdříve je načtena struktura modulů, funkcí a základních bloků. Všechny základní bloky v rámci funkce představují CFG této funkce. Načtení těchto struktur je popsáno v podkapitole 5.3.1. Po vytvoření struktury objektů, lze k funkcím načíst provedené cesty CFG tak, jak je popsáno v podkapitole 5.3.2. Tato část monitoru pokrytí a částí jí následující jsou implementovány v jazyce *Python*.

Na diagramu 5.3 jsou zobrazeny tři třídy. Struktura objektů těchto tříd je vytvářena ve fázi načítání CFG. Pro načtení cest je potřebná znalost následovníků v objektech *BasicBlock*. Cesty jsou poté uloženy ve třídě *Function*.



Obrázek 5.3: Zredukovaná část diagramu tříd 4.9 pro ilustraci načtení CFG a provedených cest funkcí.

5.3.1 Načtení CFG

Při načtení CFG jsou vyhledány instrumentované zdrojové kódy v jazyce *LLVM IR*. Na začátku těchto souborů jsou obsaženy popisy CFG. Při načítání je vytvářena struktura objektů pro moduly, funkce a základní bloky. Tyto objekty jsou také plněny hodnotami.

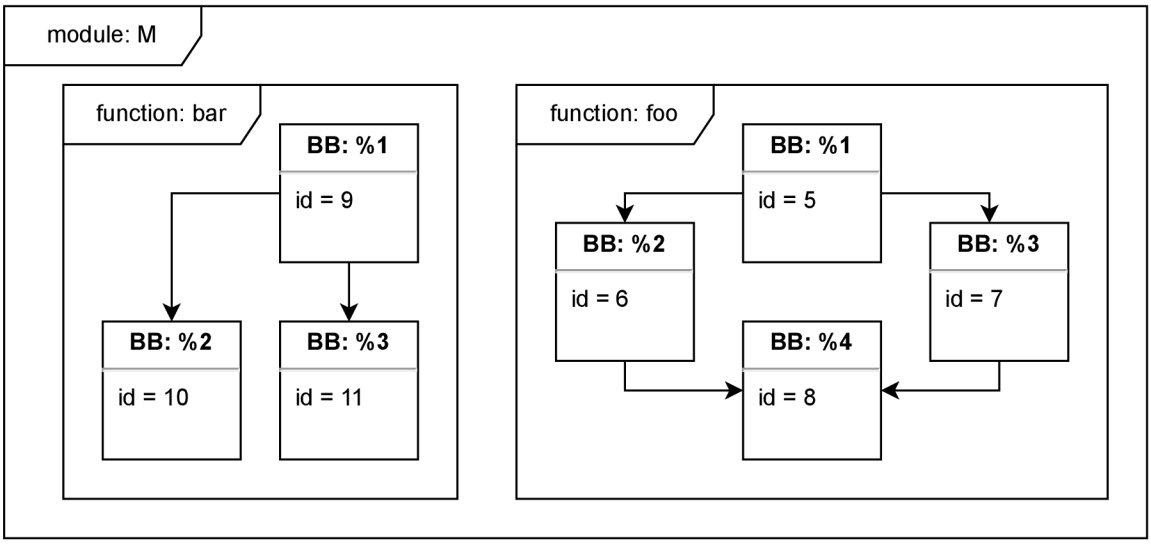
Bez načtení CFG by nebylo možné načíst cesty tak, jak je popsáno v podkapitole 5.3.2. Pro načtení cest je důležité mít u základních bloků seznamy následovníků. Pokud základní blok nemá žádného následovníka, tak v něm vykonávání funkce končí. Pokud základní blok nemá žádného předchůdce, tedy žádný ze základních bloků ve stejné funkci ho nemá jako svého následovníka, tak je to vstupní základní blok funkce. Vstupní základní blok funkce může být pouze jeden. V případě přetěžování funkcí se v kódu *LLVM IR* jedná o více funkcí, protože název funkce obsahuje i typy parametrů.

5.3.2 Načtení provedených cest CFG

Cesty jsou načítány zvlášť pro jednotlivé moduly a vlákna z logovacích souborů popsaných v podkapitole 5.2.3. V každém souboru je posloupnost číselných identifikátorů základních bloků, které jsou unikátní v rámci modulu. Číselné identifikátory jsou sekvenčně načítány. Pro příklad načtení mějme diagram na obrázku 5.4, který si lze představit jako funkce v jazyce *C++* ukázané ve výpisu 5.10.

```
1  int foo(int a) {
2      if (a > 0) { // bb %1 (id = 5)
3          a = bar(a); // bb %2 (id = 6)
4      }
5      else {
6          a = 10 + foo(1 + a); // bb %3 (id = 7)
7      }
8      return a; // bb %4 (id = 8)
9  }
10
11 int bar(int a) {
12     if (a > 0) { // bb %1 (id = 9)
13         return 1; // bb %2 (id = 10)
14     }
15     return 0; // bb %3 (id = 11)
16 }
```

Výpis 5.10: Funkce *foo* a *bar* pro příklad grafů toků řízení 5.4.



Obrázek 5.4: Organizace souborů tvořených třídou *TforcovCallback* z diagramu 5.1.

Při načítání jsou posloupnosti číselných identifikátorů, jejichž cesty ještě nebyly dokončeny, drženy v zásobníku. Každá funkce má vlastní zásobník, ze kterého je odstraněna jedna cesta po nejvrchnější identifikátor vstupního základního bloku funkce v zásobníku, pokud je vkládaný identifikátor ukončovacím základním blokem funkce.

Následující posloupnost čísel představuje vstupní soubor. V případě, že v daném modulu existují funkce s CFG z obrázku 5.4 a funkce *foo* byla zavolána, jak je ukázáno na výpisu 5.11, tak z této posloupnosti identifikátorů budou postupně načteny dvě cesty.

5	6	9	10	8	...

```
1 foo(1);
2 // -- bar(1);
```

Výpis 5.11: Volání funkce *foo*, která zavolá funkci *bar*.

Výstupem takové posloupnosti ve vstupním souboru pro daný CFG jsou dvě cesty. U funkce *foo* bude přidána jedna cesta *[5, 6, 8]* a u funkce *bar* bude přidána jedna cesta *[9, 10]*. Obsahy zásobníků se pro tento vstupní soubor budou postupně měnit, jak je ukázáno v tabulce 5.1.

V případě rekurze funkce je při odebírání cesty ze zásobníku potřebná kontrola, zda načtený základní blok je počáteční nebo koncový v rámci dané funkce. V případě, že na vstupu bude soubor s následující posloupností identifikátorů základních bloků unikátních v rámci modulu, která by byla vytvořena například voláním funkce *foo*, jak je ukázáno na výpisu 5.12.

5	7	5	6	9	10	8	8	...

```
1 foo(0);
2 // -- foo(1);
3 // ---- bar(1);
```

Výpis 5.12: Volání funkce *foo*, která zavolá rekurzivně sama sebe a uvnitř rekurzivního volání zavolá funkci *bar*.

Krok	Funkce	Zásobník (⊏←)			Akce
1	foo	5			vložení
2	foo	5	6		vložení
3	foo	5	6		—
	bar	9			vložení
4	foo	5	6		—
	bar	9	10		cesta dokončena
5	foo	5	6		—
6	foo	5	6	8	cesta dokončena

Tabulka 5.1: Načítání cest a postupně měněný obsah zásobníku při načtení posloupnosti identifikátorů obsahující zanořené volání funkcí.

tak se bude obsah zásobníku funkce *foo* postupně měnit, jak je znázorněno v tabulce 5.2. Výsledkem bude přidání dvou cest *[5, 6, 8]* a *[5, 7, 8]* do funkce *foo* a do funkce *bar* bude přidána jedna cesta *[9, 10]*.

Krok	Funkce	Zásobník (⊏←)					Akce
1	foo	5					vložení
2	foo	5	7				vložení
3	foo	5	7	5			vložení
4	foo	5	7	5	6		vložení
5	foo	5	7	5	6		—
	bar	9					vložení
6	foo	5	7	5	6		—
	bar	9	10				cesta dokončena
7	foo	5	7	5	6	8	cesta dokončena
8	foo	5	7	8			cesta dokončena

Tabulka 5.2: Načítání cest a postupně měněný obsah zásobníku při načtení posloupnosti identifikátorů obsahující rekurzivním volání funkce.

5.4 Vyhodnocení pokrytí

Funkcionalita vyhodnocení pokrytí je umístěna v abstraktní třídě *Coverage* z digramu tříd v návrhu 4.9. Prerekvizitou pro vyhodnocení kritérií pokrytí jsou načtená data CFG a provedených cest popsaná v podkapitole 5.3.

Abstraktní třída *Coverage* byla zavedena z důvodu rozšiřitelnosti a dědí z ní třídy implementující vyhodnocení jednotlivých kritérií pokrytí. Základní metodou pro získání pokrytí je funkce *Coverage.get_coverage()*. Třída má obousměrnou vazbu na třídu *Function*. Vyhodnocení pokrytí probíhá prozkoumáním základních bloků funkce, které tvoří CFG, získáním množiny požadavků daného kritéria pokrytí a zjištěním splnění prvků množiny pomocí porovnání k provedeným cestám.

Vzhledem k implementaci v jazyce *Python*, je výstupem vyhodnocení kritérií pokrytí vnořený slovník (*nested dictionary*), což je víceúrovňová struktura zanořených slovníků (*dictionary*) a seznamů(*list*). Taková struktura dat umožňuje jednoduchou a generickou

práci s výstupem vyhodnocení pokrytí v další části nástroje zabývající se exportem výsledků v podkapitole 5.5.

Veškerá vyhodnocení kritérií použití využívají jednoho jediného volání z každého základního bloku vloženého při instrumentaci. V následujících podkapitolách bude popsáno, jak byla vyhodnocení kritérií pokrytí z volání ze základních bloků implementována. V rámci této práce byla implementována vyhodnocení následujících kritérií pokrytí:

- *Line Coverage*,
- *Function Coverage*,
- *Node Coverage*,
- *Edge Coverage*,
- *Edge-Pair Coverage*,
- *Prime Path Coverage*,
- *All Def Coverage*,
- *All Use Coverage*,
- *All Def-Use Pairs Coverage*,
- *All Def-Use Paths Coverage*.

5.4.1 Line Coverage

Při implementaci pokrytí řádků kódu pomocí provádění základních bloků je nutné znát rozsah řádků, na kterých se základní blok ve funkci nachází. Tyto hodnoty jsou získány pomocí přidaných ladících informací při překladu ze zdrojového kódu v *C++* do kódu v *LLVM IR*.

Ve výpise 5.13 je ukázán kód *LLVM IR*, ke kterému jsou ukázány ladící informace ve výpisu 5.14, ze kterých lze získat informace o lokalizaci základního bloku. Ve výpise 5.15 je ukázka kódu v jazyce *C++* obsahující funkci, která po překladu do jazyka *LLVM IR* bude obsahovat krátké základní bloky, které ve svém těle budou mít pouze jednu instrukci. Ve výpise 5.16 je ukázána část kódu vygenerovaná při získání grafu toku pokrytí, kde pod klíči *bb start* a *bb end* je vypsán rozsah řádků, na kterých se základní blok nachází v souboru, který je napsán u funkce.

```
8:                                                ; preds = %2
  %9 = load i32, i32* %5, align 4, !dbg !20
  %10 = icmp sle i32 %9, 8, !dbg !21
  call void @_Z15enterBasicBlockPci(i8* getelementptr inbounds ([7 x i8], [7 x i8]* @0, i32 0,
      i32 0), i32 2)
  br i1 %10, label %11, label %13, !dbg !22
```

Výpis 5.13: Instrumentovaný základní bloky v *LLVM IR*.

```
!8 = distinct !DISubprogram(name: "foo", linkageName: "_Z3fooii", scope: !1, file: !1, line: 1,
    type: !9, scopeLine: 1, flags: DIFlagPrototyped, spFlags: DISPFlagDefinition, unit: !0,
    retainedNodes: !2)
!20 = !DILocation(line: 2, column: 18, scope: !17)
```

```
!21 = !DILocation(line: 2, column: 20, scope: !17)
!22 = !DILocation(line: 2, column: 9, scope: !8)
```

Výpis 5.14: Ladících informace k základnímu bloku ve výpisu 5.13

```
int foo(int a, int b){          // [%2]
    if (a > 3 && b <= 8){       // [%2, %8]
        return b;               // [%11]
    }                           // []
    return a + b;               // [%13]
}                               // [%17]
```

Výpis 5.15: Funkce vracející konstantu, která je vrácena v krátkém základním bloku.

```
; function: _Z3fooii
; function filename: if.cpp
; function directory: /work/test/system/workspaces/ws5
; bb: %2
; bb id: 1
; bb succ: %8
; bb succ: %13
; bb start: 1
; bb end: 2
; bb store: %4
; bb variable: %4 a
; bb store: %5
; bb variable: %5 b
; bb load: %4
```

Výpis 5.16: Ukázka dat funkce a jednoho základního bloku získaných pro vyhodnocení kritérií pokrytí při instrumentaci *LLVM IR*.

5.4.2 Function Coverage

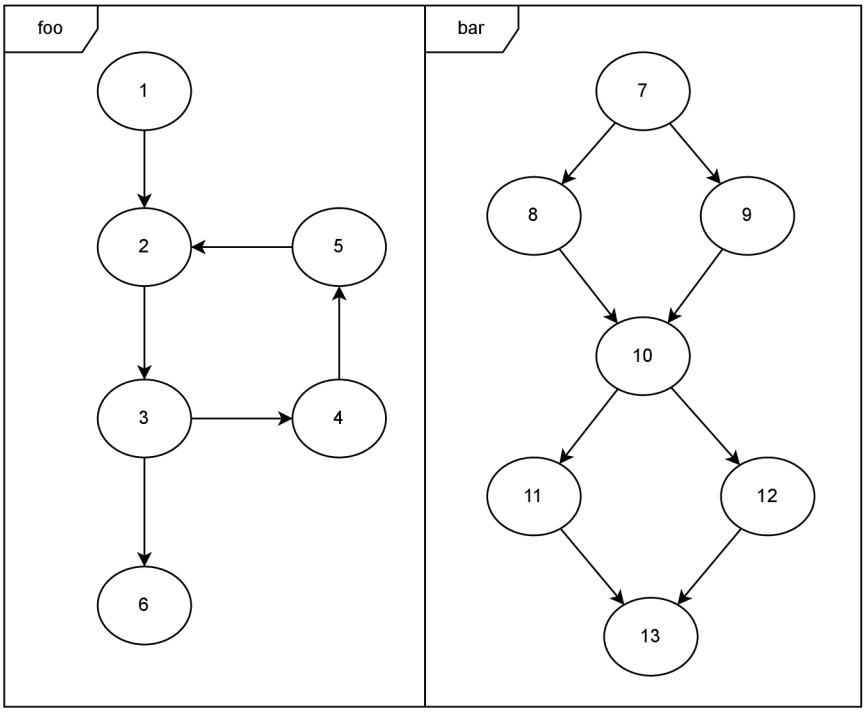
Pokrytí funkce je splněno, pokud byla funkce alespoň jednou zavolána. Provedené cesty jsou neprázdné, proto je u funkce s alespoň jednou provedenou cestou označit toto kritérium pokrytí splněno.

5.4.3 Pokrytí toku řízení

U kritérií pokrytí toku řízení je nejdříve prozkoumán CFG a je sestavena množina požadavků pro dané kritérium pokrytí ve funkci. U implementovaných kritérií toku řízení je vždy hledána podcesta v provedených cestách funkce. U pokrytí uzlů má cesta délku jedna, u pokrytí hran má délku 2 a u pokrytí párů hran má délku tři. V případě pokrytí hlavních cest je délka různá.

Node Coverage

Uzlem je základní blok. Pokrytí uzlu je splněno, pokud je základní blok nalezen alespoň v jedné provedené cestě. Takto je zjištěno pokrytí pro všechny uzly (základní bloky) v rámci zkoumané funkce.



Obrázek 5.5: Příklad CFG funkcí *foo* a *bar*.

Edge Coverage

Vyhodnocení kritéria pokrytí začíná nalezením všech hran. Hrana je dvojice základních bloku taková, že první základní blok má druhý základní blok jako následníka. Pokrytí hrany je splněno, pokud je alespoň v jedné provedené cestě nalezena nepřerušená podcesta sestávající se z těchto dvou základních bloků.

Edge-Pair Coverage

Pro vyhodnocení kritéria pokrytí jsou ve funkci nalezeny všechny dvojice hran, které na sebe navazují. Pár hran je trojice základních bloku, kde první základní blok má jako následovníka druhý blok a druhý základní blok má jako následovníka třetí základní blok. Pokrytí dvojice hran je splněno, pokud je alespoň v jedné cestě nalezena nepřerušená podcesta sestávající se z těchto třech základních bloků.

Prime Path Coverage

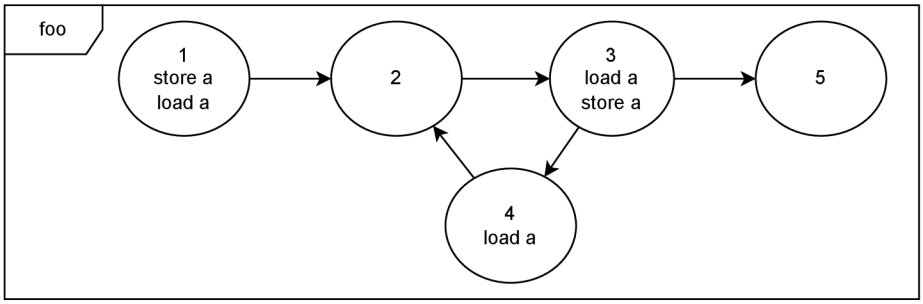
U kritéria pokrytí hlavních cest je hlavní získání hlavních cest z CFG. Nejdříve jsou získány veškeré jednoduché cesty. Množina jednoduchých cest je poté zredukována na množinu hlavních cest.

Hlavní cesta je posloupnost základních bloku. Pokrytí hlavní cesty je splněno, pokud je alespoň v jedné cestě nalezena nepřerušená podcesta sestávající se z těchto základních bloků.

5.4.4 Pokrytí toku dat

Při vyhodnocení kritérií pokrytí toku dat nejsou vyhledávány podcesty na rozdíl od kritérií pokrytí toku řízení. Kritéria *All Def* a *All Use* zkoumají pouze, zda byl základní blok, ve kterém je definice nebo použití proměnné, proveden alespoň v jedné cestě. Kritérium pokrytí *All Def-Use* hledá jeden až dva základní bloky v některé provedené cestě tak, že mezi nimi mohou být provedeny i jiné základní bloky.

Na obrázku 5.6 je CFG funkce foo, pro kterou budou ukázána vyhodnocení kritérií pokrytí níže. K vyhodnocení toku dat se používají pouze volání ze základních bloků vložená při instrumentaci a nebyla tedy při instrumentaci vložena žádná další volání funkcí.



Obrázek 5.6: Příklad CFG funkce *foo*.

All Def Coverage

Pro vyhodnocení kritéria pokrytí jsou nejdříve nalezeny všechny definice proměnné. V CFG na obrázku 5.6 jsou obsaženy dvě. Jsou v základním bloku *1* a *3*. Nechť funkce má jednu provedenou cestu:

1	2	3	4	2	3	5

V cestě je základní blok *1* a tedy pokrytí první definice proměnné je splněno. Stejně je v cestě základní blok *3* a tedy pokrytí druhé definice proměnné je také splněno.

All Use Coverage

Toto pokrytí je podobné jako *All Def coverage*, s rozdílem hledání použití proměnných. V CFG na obrázku 5.6 jsou obsaženy tři použití proměnných. Jsou v základních blocích *1*, *3* a *4*. Nechť funkce má jednu provedenou cestu:

1	2	3	5

Dvě ze tří pokrytí použití proměnných jsou splněny, protože všechny základní bloky *1* a *3* jsou obsaženy v cestě. Základní blok *4* v žádné cestě není a tedy použití proměnné *a* v základním bloku *4* není pokryto.

All Def-Use Pairs Coverage

Při vyhodnocení pokrytí *All Def-Use* je prvním krokem vyhledání cest definic a použití proměnných. To probíhá prohledáváním CFG ze základních bloků, kde jsou definice. Pro základní blok obsahující definici se zjišťuje dosažitelnost ostatních základních bloků. Zkoumání

této dosažitelnosti je implementačně stejný problém jako prohledávání stavového prostoru. Uzly jsou postupně expandovány tak, že jsou cesty k nim přidány do fronty k expandování následně budou expandováni i jejich následníci. Následovník nebude expandován, pokud již před tím byl expandován v rámci cesty, protože to znamená existenci cyklu. Dále expandování cesty končí nalezení další definice stejné proměnné, která hodnotu původní definice proměnné přepíše. Pokud základní blok nemá žádného následovníka, tak do fronty nebude přidána žádná další cesta. V každém z těchto tří případů dokončení cesty jsou v dosažitelných cestách nalezeny všechna použití proměnné, ke kterým jsou pak od definice uloženy cesty *Def-Use Paths*.

V CFG na obrázku 5.6 jsou vidět definice a použitý proměnných použití proměnných. Výstupem hledání dosažitelnosti z míst definic proměnných jsou cesty od definice k použití proměnných (*Def-Use Paths*). Na následujících příkladech bude ukázáno, jak jsou kritéria pokrytí vyhodnocena s prohledáním provedených cest v rámci funkce. Mějme dvě různé provedené cesty:

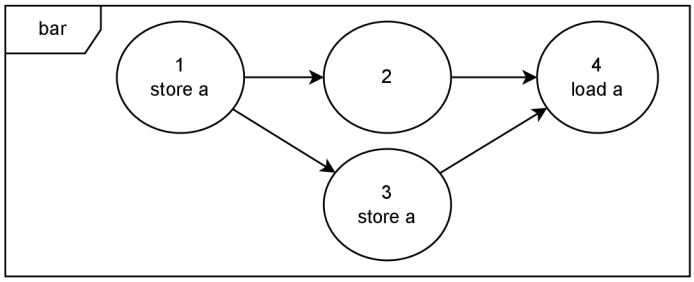
cesta 1	1	2	3	5

cesta 2	1	2	3	4	2	3	5

V následující tabulce jsou vidět čtyři páry definic a použití proměnných. V posledních sloupcích je vyhodnocení, zda pár definice a použití byl danou provedenou cestou pokryt. První cesta pokryla první dva páry a druhé cesta všechny páry.

store	uzel	pozice	load	uzel	pozice	cesta	cesta 1	cesta 2	pokrytí párů
a	1	1	a	1	2	[1]	ano	ano	ano
a	1	1	a	3	1	[1, 2, 3]	ano	ano	ano
a	3	2	a	4	1	[3, 4]	ne	ano	ano
a	3	2	a	3	1	[3, 4, 2, 3]	ne	ano	ano

Na obrázku 5.7 je CFG ve tvaru diamantu. Na příkladu bude ukázáno, že prohledávání dosažitelnosti končí při nalezení další definice stejné proměnné.



Obrázek 5.7: Příklad CFG funkce *bar*.

Pokud existují dvě provedené cesty, tak pokrytí párů definic a použití proměnných bude vyhodnoceno tak, jak je napsáno v tabulce níže.

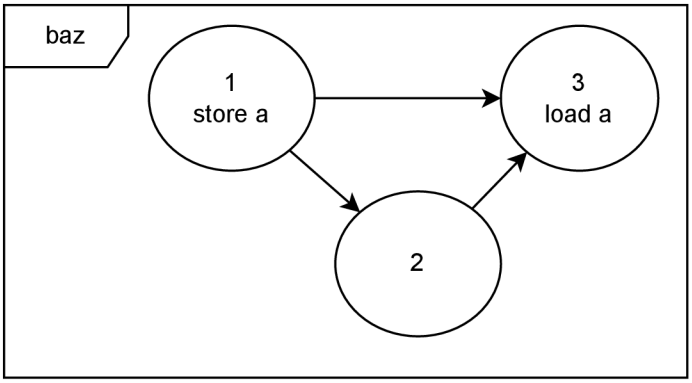
cesta 3	1	3	4
cesta 4	1	2	4

store	uzel	pozice	load	uzel	pozice	cesta	cesta 3	cesta 4	pokrytí párů
a	1	1	a	4	1	[1, 2, 4]	ne	ano	ano
a	3	1	a	4	1	[3, 4]	ano	ne	ano

All Def-Use Paths Coverage

Pokrytí *All Def-Use Paths Coverage* se od pokrytí All Def-Use Pairs Coverage liší v požadavku na pokrytí všech cest. U pokrytí párů stačí, když je pokryta pouze jedna cesta pro unikátní dvojici definice a použití proměnné.

Následující příklad na obrázku 5.8 demonstruje odlišnost pokrytí kritérií cesta a párů (*All Def-Use Paths Coverage* a *All Def-Use Paths Coverage*).



Obrázek 5.8: Příklad CFG funkce *baz*.

cesta 5	1	2	3

store	uzel	pozice	load	uzel	pozice	cesta	cesta 5	pokrytí cest	pokrytí páru
a	1	1	a	3	1	[1, 2, 3]	ano	ano	ano
a	1	1	a	3	1	[1, 3]	ne	ne	

5.5 Export výsledků

Funkcionalita exportu výsledků vyhodnocení pokrytí je umístěna v abstraktní třídě *Export* z digramu tříd v návrhu 4.9. Implementace pomocí abstraktní třídy umožňuje jednoduchou rozšiřitelnost.

Nástroj podporuje více formátů výstupu:

- *XML*,
- *JSON*,
- *YAML*.

Dalším výstup nástroje je vypsán na standardní výstup. Tento výstup neobsahuje veškerá data. Jde o přehled zobrazující výsledky jednotlivých kritérií pokrytí bez výpisu seznamu požadavků na splnění konkrétního pokrytí. Takový výstup uživateli řekne, že například kritérium pokrytí *Prime Path Coverage* je u dané funkce splněno pro 7 hlavních cest z 10 celkových hlavních cest ve funkci, ale nevypíše, jaké hlavní cesty funkce obsahuje ani jaké cesty byly v rámci funkce provedeny.

5.6 Sledování běhů vícevláknových aplikací

Při pouštění vícevláknových aplikací je nutné zabezpečit nebo oddělit zdroje, které vlákna používají, jako je sdílený kus paměti nebo soubor pro zaznamenání běhu programu. Dále je nutné rozlišit běhy programů pro jednotlivá vlákna, aby bylo možné samostatně sestavit provedené cesty CFG pro jednotlivá vlákna.

Při zavolání instrumentované funkce je před zápisem zjištěno, z jakého vlákna byla funkce zavolána. Každé vlákno zapisuje do svých logovacích souborů. Zachycení běhů programů bylo popsáno v podkapitole 5.2 a v podkapitole 5.2.3 je vidět na obrázku 5.2 struktura logovacích souborů tvořených při běhu instrumentovaného programu. To umožňuje rychlejší zápis, protože není nutné používat vzájemné vyloučení při zápisu do souborů namapovaného do paměti, ani není nutné sdílet mezi vlákny data o aktuální zapisovací pozici v souborech. Další výhodou tohoto přístupu je oddělení logovacích dat různých vláken, které je poté jednodušší zpracovat.

5.7 Příklad spuštění monitoru pokrytí

Nástroj byl navržen s ohledem na jednoduchost použití. Následující podkapitoly popisují závislosti, příklad spuštění a popis generovaných souborů při běhu různých částí nástroje *tforcov*.

5.7.1 Závislosti

Před spuštěním je nutné zajistit potřebné závislosti. Podporovaný operační systém je *Ubuntu 20.04*. Závislosti v systému jsou dvou druhů, kde první jsou balíky instalované do systému a druhé jsou moduly pro *Python*. Mezi balíky, které musí být nainstalované v systému, patří:

- *llvm*
- *llvm-dev*
- *clang*
- *make*
- *python3*
- *python3-pip*

Python modul, který musí být nainstalovaný, je jeden a slouží pro podporu výstupu ve formátu *YAML*:

- *PyYAML==5.3.1*

5.7.2 Spuštění

Použití nástroje se skládá ze tří částí, které jsou ukázány ve výpisu 5.17, který pro překlad spouští *Makefile*, kterého ukázka je ve výpisu 5.18. V souboru *Makefile* je pro názornost zadána cesta k *wrapperu* nástroje *tforcov*, ale lze také cestu vložit do proměnné prostředí *PATH*.

- instrumentace,

- spuštění instrumentovaného programu,
- vyhodnocení kritérií pokrytí.

```
TFORCOV=../src/tforcov.py

# instrument program
$TFORCOV instrument start
make
$TFORCOV instrument stop

# run instrumented program
./executable

# evaluate coverage criteria
$TFORCOV coverage
```

Výpis 5.17: Příklad sekvence příkazů pro použití monitoru pokrytí *tforcov*.

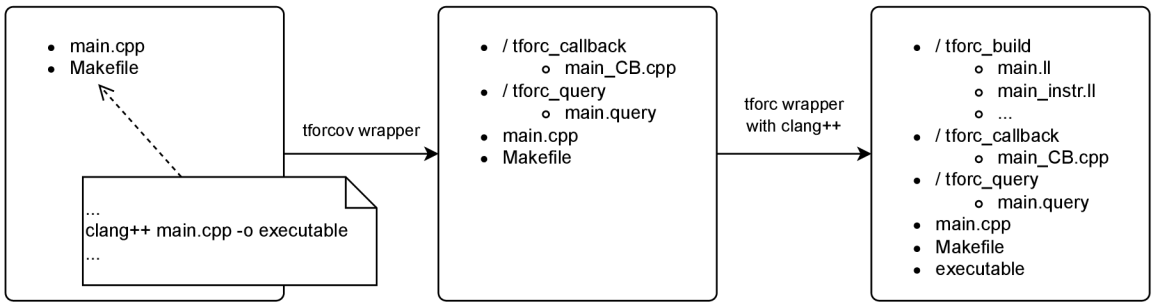
```
CXX := ../src/instrumentation/path_compiler/clang++

all :
	$(CXX) main.cpp -o executable
```

Výpis 5.18: Příklad souboru *Makefile*.

5.7.3 Struktura vytvářených souborů

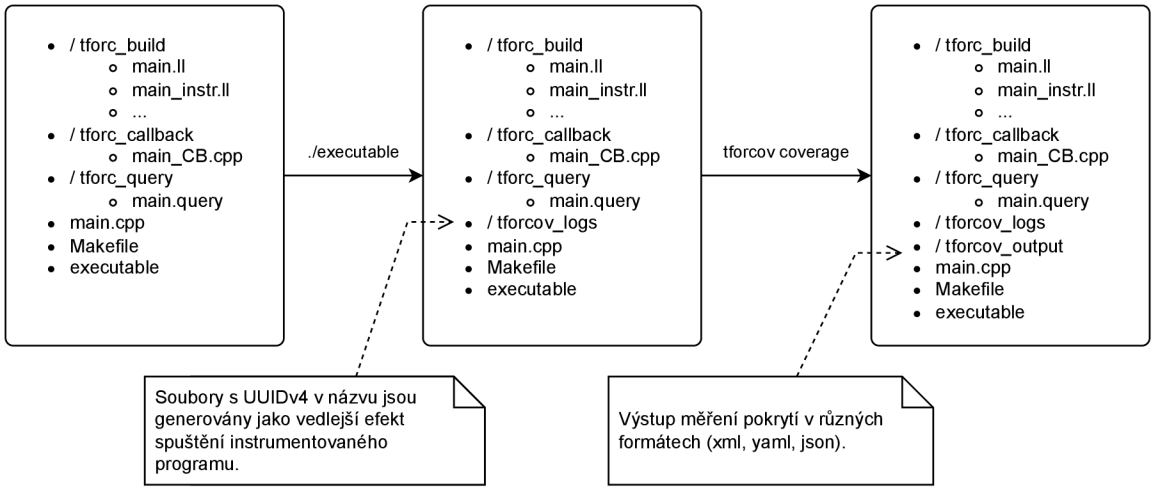
Tato podkapitola popíše postupné vytváření souborů v pracovním prostoru při použití monitoru pokrytí, jak bylo ukázáno v podkapitole 5.7.2

První fází je spuštění instrumentace, pro které je vývoj pracovního prostoru zobrazen na obrázku 5.9. Nejdříve jsou vytvořeny soubory pro konfiguraci nástroje *tforc* ve složkách *tforc_query* a *tforc_callback*. Poté je spuštěn překlad, který vytvoří složku *tforc_build* a spustitelný program. Ve složce *tforc_build* jsou v instrumentovaných zdrojových souborech v jazyce *LLVM IR* vypsány grafy toku pokrytí (*CFG*). Vše co je na obrázku odpovídá zavolání *make* při zapnutém nástroji *tforcov*.



Obrázek 5.9: Soubory tvořené při instrumentaci.

Druhé dva kroky použití monitoru pokrytí jsou ukázány na obrázku 5.10. Druhým krokem je spuštění instrumentovaného programu. Třetím krokem je vyhodnocení pokrytí pro daný instrumentovaný program a provedená spuštění instrumentovaného programu.



Obrázek 5.10: Soubory tvořené při spouštění instrumentovaného programu a při vyhodnocení kritérií pokrytí.

Nejdříve jsou spouštěním instrumentovaného programu tvořeny logovací soubory ve složce *tforcov_logs*, které zachycují jednotlivé běhy programu. Tento přechod odpovídá zavolání *./executable* z výpisu 5.17. Soubory jednotlivých spuštění jsou odděleny, stejně tak jako logovácí soubory jednotlivých vláken a modulů v rámci jednoho spuštění instrumentovaného programu, jak bylo ukázáno v podkapitole 5.2.3.

Posledním krokem je vyhodnocení kritérií pokrytí, které je spuštěno jako *tforcov.py coverage*. Výstupem tohoto skriptu je složka *tforcov_output* obsahující výsledky vyhodnocení kritérií pokrytí nad daným programem a danými běhy programu. Důvodem pro uložení výstupu do složky je generování několika formátů výstupu při jednom spuštění programu, mezi které patří například *JSON*, *XML* nebo *YAML*. Vstupem pro vyhodnocení jsou logy ve složce *tforcov_logs* a grafy toku pokrytí ve složce *tforc_build*.

Kapitola 6

Ověření správnosti implementovaného řešení

Ověření správnosti implementace řešení je provedeno automatickými testy. Testovací framework je popsán v podkapitole 6.1. Při vývoji je využito *Gitlab CI* popsané v podkapitole 6.2. Konečně testy jsou popsány v podkapitole 6.3.

6.1 Použitý testovací framework

Velká část zdrojových kódu této práce je napsána v jazyce *Python*. Vzhledem k této skutečnosti jsou i testy napsány v tomto jazyce. Pro napsání testů je využit testovací framework pytest[1], který je dostupný jako modul pro *Python* na *PyPI*[2].

Testovací framework je rozšiřitelný, existuje pro něj přes 850 externích pluginů. Za účelem zvýšení kvality testů pomocí rozšíření funkcionality testovacího frameworku *pytest* jsou u tohoto nástroje práci použity následující externí pluginy (moduly) pro *pytest*:

- pytest-mock[3] a
- pytest-cov[4].

```
1 pytest==6.1.1
2 pytest-cov==2.11.1
3 pytest-mock==3.5.1
```

Výpis 6.1: Závislosti jazyka *Python* pro automatické testování monitoru pokrytí.

Modul *pytest-mock* umožňuje jednodušší *mockování* částí systému, a tedy izolaci testovaných částí systémů. Modul lze použít v kombinaci se třídami z knihovny pro *mockování* objektů *unittest.mock*[5], která je část standardního modulu *unittest*[6]

Modul *pytest-cov* sbírá při běhu informace a vyhodnotí pokrytí systému provedenými testy v rámci jednoho spuštění modulu *pytest*. Tyto informace byly při vývoji použity

[1] Dokumentace testovacího frameworku *pytest*: https://docs.pytest.org/
[2] Modul *pytest* na *PyPI*: https://pypi.org/project/pytest/
[3] Modul *pytest-mock* na *PyPI*: https://pypi.org/project/pytest-mock/
[4] Modul *pytest-cov* na *PyPI*: https://pypi.org/project/pytest-cov/
[5] Dokumentace knihovny pro mockování objektů *unittest.mock*: https://docs.python.org/3/library/unittest.mock.html
[6] Dokumentace standardního modulu *unittest*: https://docs.python.org/3/library/unittest.html

k odhalení nedostatků testovací sady a jako ukazatel otestovanosti systému jednotkovými testy.

Vzhledem k použití testovacího frameworku, který není součástí standardních modulů jazyka *Python*, je nutné zavést závislosti. Ty jsou v repozitáři obsaženy v souboru *requirements-test.txt*, jehož obsah je zobrazen ve výpisu 6.1.

6.2 Průběžná integrace

V rámci vývoje byl v repozitáři využíván nástroj *Gitlab CI*. Nástroj automaticky spustí úlohy při změně obsahu repozitáře v dané větvi. Konfigurace nástroje probíhá pomocí souboru *.gitlab-ci.yml*. V repozitáři jsou 3 stage:

- stage: *code_quality*
- stage: *test*
- stage: *example*

6.3 Testy

Tato podkapitola popisuje testy vytvořené k ověření správnosti implementovaného řešení. Následující podkapitoly popisují jednotlivé úrovně testování.

6.3.1 Jednotkové testy

Jednotkové testy využívají rozšiřovací moduly *pytest-mock* *pytest-cov* pro testovací framework *pytest*. Nejvíce jednotkových testů je napsáno na část implementovaného řešení zabývající se vyhodnocením kritérií pokrytí. Jednotkové testy lze spustit podle výpisu 6.2.

```
1 python -m pip install --user -r requirements-test.txt -r requirements.txt
2 cd src
3 python -m pytest ../test/unit -v --tb=short --cov . --cov-branch
```

Výpis 6.2: Spuštění jednotkových testů.

6.3.2 E2E testy

Testy *end-to-end* (*E2E*) spouští všechny části monitoru pokrytí velmi podobně, jako kdyby monitor použití používal uživatel. Tyto testy slouží i jako demonstrační sada k použití nástroje *tforcov*. Testy jsou napsány jako jeden test, který je parametrizován následujícími soubory:

- soubor *test.json* definující průběh a vyhodnocení testu,
- zdrojové kódy v jazyce *C++*.

Soubor *test.json* obsahuje informace k provedení a vyhodnocení testu. Tyto informace lze rozdělit do následujících částí:

- překladu programu,
- spuštění instrumentovaného programu, které může být i opakované,

- vyhodnocení kritérií pokrytí.

Překlad programu je definovaný tak, aby bylo možné překlad spustit i bez instrumentace. To je vlastnost implementovaného řešení, protože se používá stejně jako systémový překladač. Zavolání kódu instrumentačního nástroje místo volání systémového překladače řeší až tělo testu.

Když je instrumentovaný program přeložen, tak je v dalším kroku spouštěn. Program může být spouštěn s různými argumenty, které způsobí provádí různých části programu. V této fázi instrumentovaný program tvoří jako vedlejší produkt logovací soubory.

Po dokončení spouštění programu jsou vyhodnocena kritéria pokrytí instrumentovaného programu dle provedených spuštění.

E2E testy lze spustit podle výpisu 6.3.

```
1 apt -y -qq install llvm llvm-dev clang make python3 python3-pip >/dev/null
2 python3 -m pip install --user -r requirements-test.txt -r requirements.txt
3 cd src
4 python3 -m pytest ../test/e2e --tb=native --capture=tee-sys -v
```

Výpis 6.3: Spuštění *E2E* testů.

Kapitola 7

Závěr

Cílem práce bylo vytvořit monitor pokrytí, který provádí instrumentaci programu za účelem vyhodnocení kritérií pokrytí. Záměr se podařilo splnit ve všech bodech.

Nejdříve jsem nastudoval potřebné technologie. Mezi znalosti v oblasti překladu programů patří zejména instrumentace a nástroje pro překlad *LLVM*. Pro druhou část práce jsem studoval informace týkající se testování, kde hlavní byl graf toku řízení (*CFG*) a kritéria pokrytí. Tato studie zahrnovala i průzkum existujících řešení nástrojů pro instrumentaci a vyhodnocení kritérií pokrytí. Po provedené studii jsem analyzoval a specifikoval požadavky. S ohledem na požadavky z analýzy i ze zadání této práce, které zahrnují kritéria pokrytí toku dat, kritéria pokrytí toku řízení, kritérium pokrytí řádků kódu, podporu vícevláknových aplikací, jednoduchost použití nebo minimalizaci režie způsobené instrumentací v době překladu, jsem navrhl monitor pro měření pokrytí skládající se ze dvou částí. První částí je instrumentace při překladu, kde jsem rozšířil nástroj *Tforc-tool* i nástroj *tforc*. Druhou částí je vyhodnocení kritérií pokrytí. Nástroj jsem implementoval v jazycích *Python* a *C++*. Správnost implementovaného řešení byla ověřena automatickými testy. Sada automatických testů obsahuje také demonstrační sadu zdrojových kódů v jazyce *C++*, které jsou instrumentovány při překladu, spouštěny a nakonec jsou pro dané programy vyhodnocena kritéria pokrytí.

Práce mě naučila mnoho nových věcí. Naučil jsem se více o testování, kde zvláště hlubší pochopení kritérií pokrytí mi v budoucnu pomůže lépe testovat software a snad ho i psát tak, aby pro kód byly menší množiny požadavků jednotlivých kritérií pokrytí. Lépe jsem pochopil proces překladu, zvláště s infrastrukturou překladačových nástrojů *Clang/LLVM*, se kterou jsem před tím nepracoval. Velmi zajímavé pro mě bylo i poznání jazyka *LLVM IR*.

V práci by bylo možné pokračovat přidáním dalších kritérií pokrytí, zejména se nabízí kritéria pokrytí pro větvení programu (*Condition Coverage, Decision Coverage, Modified Decision Condition Coverage*). Pro tato kritéria pokrytí by bylo nutné získat pomocí překladače abstraktní sémantický strom (*AST*), což by mělo být proveditelné, protože část nástroje pro vyhodnocení kritérií pokrytí zná umístění zdrojových souborů. Nástroj *tforcov* by bylo možné vylepšit tak, aby tvořil jen jeden soubor mapovaný do paměti pro každé vlákno a modul. Dalším rozšířením by mohl být průzkum nástroje *Tforc-tool*, který nepodporuje veškeré argumenty překladače.

Literatura

[1] 29119-4-2015 - ISO/IEC/IEEE. *International Standard - Software and systems engineering–Software testing–Part 4 : Test techniques*. 1. vyd. IEEE, 2015. ISBN 978-0-7381-9840-8.

[2] Ammann, P. a Offutt, J. *Introduction to Software Testing*. 1. vyd. Cambridge University Press, 2008. ISBN 978-0-521-88038-1.

[3] Bullseye Testing Technology. *BullseyeCoverage*. [cit. 2022-01-23]. Dostupné z: https://www.bullseye.com/index.html.

[4] Burnstein, I. *Practical Software Testing: A Process-Oriented Approach*. 1. vyd. Springer New York, 2003. Springer Professional Computing. ISBN 978-0-387-95131-7. Dostupné z: https://books.google.cz/books?id=0v6HSeqA00oC.

[5] Froglogic. *Squish Coco*. [cit. 2022-01-23]. Dostupné z: https://doc.froglogic.com/squish-coco/latest/index.html.

[6] Gelperin, D. a Hetzel, B. The Growth of Software Testing. *Commun. ACM*. 1. vyd. New York, NY, USA: Association for Computing Machinery. Červen 1988. DOI: 10.1145/62959.62965. ISSN 0001-0782. Dostupné z: https://doi.org/10.1145/62959.62965.

[7] Gorbachenko, P. *What are Functional and Non-Functional Requirements and How to Document These* [online]. 2021 [cit. 2022-05-17]. Dostupné z: https://enkonix.com/blog/functional-requirements-vs-non-functional/.

[8] IEEE. Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*. 1. vyd. 1990. DOI: 10.1109/IEEESTD.1990.101064.

[9] Jorgensen, P. *Software Testing: A Craftsman's Approach, Fourth Edition*. 4. vyd. CRC Press, 2013. ISBN 9781466560697. Dostupné z: https://books.google.cz/books?id=nVLSBQAAQBAJ.

[10] Lattner, C. *LLVM* [online]. The Architecture of Open Source Aplication, leden 2021 [cit. 2021-01-14]. Dostupné z: http://www.aosabook.org/en/llvm.html.

[11] Lattner, C. a Adve, V. *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*. Palo Alto, California: [b.n.], březen 2004.

[12] LLVM Project. *XRay Instrumentation* [online]. 1. vyd. 2016 [cit. 2022-01-23]. Dostupné z: https://llvm.org/docs/XRay.html.

[13] Microsoft. *Tracing and Instrumenting Applications* [online]. 2021 [cit. 2022-01-23]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/framework/debug-trace-profile/tracing-and-instrumenting-applications.

[14] Microsoft. *Use code coverage to determine how much code is being tested.* 2022 [cit. 2022-01-23]. Dostupné z: https://docs.microsoft.com/cs-cz/visualstudio/test/using-code-coverage-to-determine-how-much-code-is-being-tested?view=vs-2022.

[15] Mušková, K. *Instrumentace C/C++ programů při překladu.* Brno, CZ, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/23399/.

[16] Rapita Systems Ltd.. *MC/DC Coverage* [online]. 2021 [cit. 2021-10-17]. Dostupné z: https://www.rapitasystems.com/mcdc-coverage.

[17] Smrčka, A. *Automatizované testování a dynamická analýza.* Fakulta informačních technologií VUT v Brně, 2020. Texty k přednáškám. Dostupné z: https://www.fit.vut.cz/study/course/244772.

[18] Testwell. *Testwell CTC++: Test Coverage Analyzer for C/C++.* 2018 [cit. 2022-01-23]. Dostupné z: http://www.testwell.fi/ctcdesc.html.

[19] VITOVSKÁ, M. *Instrumentation of LLVM IR [online].* 2018 [cit. 2022-01-23]. Diplomová práce. Masarykova univerzita, Fakulta informatiky, Brno. Vedoucí práce Strejček, J. Dostupné z: DostupnézWWW<https://is.muni.cz/th/nhd8u/>.

[20] Ševčík, V. *Rozvoj instrumentace programu při překladu.* Brno, CZ, 2020. Master's thesis. Brno University of Technology, Faculty of Information Technology. Dostupné z: https://www.fit.vut.cz/study/thesis/21397/.

Příloha A

Obsah paměťového média

Paměťové médium přiložené k práci obsahuje finální obsah repozitáře, ve kterém jsou veškeré zdrojové kódy, testy a příklad.

Odkaz na repozitář: https://pajda.fit.vutbr.cz/testos/tforcov.

Obsah paměťového média má následující strukturu:

- `src\` Zdrojové kódy textu práce v Latexu.
- `project\` Finální obsah repozitáře s monitorem pokrytí.
- `readme.txt` Popis struktury paměťového média.

Příloha B

Obsah repozitáře

Repozitář monitoru pokrytí na adrese https://pajda.fit.vutbr.cz/testos/tforcov má následující strukturu (nejsou obsaženy všechny složky a soubory):

- ◇ `example\` Příklad použití monitoru pokrytí.
  - `main.cpp` Hlavní zdrojový soubor.
  - `Makefile` Soubor obsahující příkaz pro přeložení programu.
  - `run.sh` Soubor pro příkladné spuštění překladu, spuštění instrumentovaného programu a spuštění vyhodnocení kritérií pokrytí.
  - `somelib.cpp` Zdrojový soubor importovaný ze souboru `main.cpp`.
- ◇ `src\` Zdrojové kódy monitoru pokrytí.
  - ◇ `coverage_evaluation\` Složka obsahující skripty a soubory pro instrumentaci.
  - ◇ `instrumentation\` Složka obsahující skripty pro vyhodnocení kritérií pokrytí.
  - ◇ `tforc-tool\` Složka obsahující nástroj `tforc-tool`.
  - `tforcov.py` Hlavní skript nástroje `tforcov`.
- ◇ `test\` Složka obsahující testy pro monitor pokrytí.
  - ◇ `e2e\` Složka obsahující *end-to-end* testy pro monitor pokrytí.
  - ◇ `unit\` Složka obsahující jednotkové testy pro monitor pokrytí.
- `.flake8` Konfigurační soubor modulu Flake8[1].
- `.gitignore`
- `.gitlab-ci.yml` Konfigurační soubor pro Gitlab CI[2].
- `Makefile` Makefile obsahující cíle pro lokální spuštění kontejnerů nebo testů.
- `README.md` Dokumentace.
- `requirements.txt` Soubor obsahující závislosti pro Python potřebné pro běh monitoru pokrytí obsahující seznam modulů a verzí.
- `requirements.txt` Soubor obsahující závislosti pro Python potřebné ke spuštění testů obsahující seznam modulů a verzí.

[1] Modul *flake8* na *PyPI*: https://pypi.org/project/flake8/
[2] Dokumentace *Gitlab CI/CD*: https://docs.gitlab.com/ee/ci/