VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

NÁSTROJ PRO GENEROVÁNÍ TESTOVACÍCH CEST DLE ZADANÉHO KRITÉRIA

A TOOL FOR GENERATING TEST PATHS BASED ON A GIVEN CRITERION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. JAN BÍL**
AUTHOR

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

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání diplomové práce

155691

Ústav:	Ústav inteligentních systémů (UITS)
Student:	**Bíl Jan, Bc.**
Program:	Informační technologie a umělá inteligence
Specializace:	Verifikace a testování software
Název:	**Nástroj pro generování testovacích cest dle zadaného kritéria**
Kategorie:	Analýza a testování softwaru
Akademický rok:	2023/24

Zadání:

1. Nastudujte testování založené na modelech. Nastudujte formát GCC Gimple. Seznamte se s řešiči SMT. Nastudujte kritéria pokrytí běžně vyžadovaná při testování kritických aplikací.
2. Navrhněte nástroj pro generování testovacích cest za účelem splnění daného kritéria pokrytí. V generování předpokládejte existující nástroj pro extrakci dat ze zdrojových kódů a existující řešič formulí SMT.
3. Implementujte nástroj, který na základě zdrojového kódu a daného kritéria pokrytí generuje testovací cesty. Implementujte aplikační rozhraní za účelem integrace nástroje s jinými testovacími nástroji.
4. Ověřte funkcionalitu nástroje na automatizované testovací sadě. Demonstruje funkčnost nástroje na jednoduchém tutoriálu.

Literatura:

- Charvát Lukáš, Smrčka Aleš a Vojnar Tomáš. An Abstraction of Multi-Port Memories with Arbitrary Addressable Units. In: *Computer Aided Systems Theory - EUROCAST 2013*. Lecture Notes in Computer Science, roč. 8111. Berlin Heidelberg: Springer Verlag, 2013, s. 460-468. ISBN 978-3-642-53855-1.
- Kraut Daniel. Generování modelů pro testy ze zdrojových kódů. 2019. Diplomová práce. FIT VUT v Brně.
- Sušovský Tomáš. Generování testovacích vstupů podle stopy programu. 2019. Diplomová práce. FIT VUT v Brně.

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:	1.11.2023
Termín pro odevzdání:	17.5.2024
Datum schválení:	6.11.2023

Abstrakt

Automatické generování testovacích vstupů dle zadaného kritéria pokrytí má potenciál ušetřit velkou část nákladů při vývoji kritických aplikací. V rámci této práce byl navržen a naimplementován nástroj, který postupně generuje cesty grafem toku řízení odpovídající funkcím takové aplikace, které splňují zadané kritérium pokrytí. Tyto cesty převádí na odpovídající SMT (Satisfiability Modulo Theories) formuli pro kterou se ověří její sémantická splnitelnost SMT řešičem Z3. Pro sémanticky splnitelné cesty je zároveň vygenerováno ohodnocení vstupů, pro které formule platí. Tato ohodnocení dohromady tvoří sadu testovacích vstupů splňující zadané kritérium pokrytí. Tyto testovací vstupy jsou hodnoty parametrů testované funkce a stavu globálních proměnných.

Abstract

The automatic generation of test inputs according to a specified coverage criterion has the potential to significantly reduce costs in the development of critical applications. This work focuses on designing and implementing a tool that systematically generates paths in the control flow graph generated from function of such application, that meet the specified coverage criterion targets. These paths are then transformed into the corresponding SMT (Satisfiability Modulo Theories) format, and their feasibility is verified using the Z3 SMT solver. For paths that are feasible, a set of input valuations for which the formula holds is simultaneously generated. This collection of valuations formes a set of test inputs that effectively meet the desired coverage criterion. Final test inputs are tested function parameters and state of global parameters.

Klíčová slova

automatické generování, testování, kritéria pokrytí, CFG, SMT řešič, kritické systémy

Keywords

automatic generation, testing, coverage criteria, CFG, SMT Solver, critical systems

Citace

BÍL, Jan. *Nástroj pro generování testovacích cest dle zadaného kritéria.* Brno, 2024. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

Nástroj pro generování testovacích cest dle zadaného kritéria

Prohlášení

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

.....
Jan Bíl
16. května 2024

Poděkování

Rád bych poděkoval vedoucímu mé diplomové práce panu Ing. Aleši Smrčkovi, Ph.D za cenné rady a připomínky, za věnovaný čas a odborné vedení.

Obsah

Kapitola 1

Úvod

Kritické systémy z pohledu bezpečnosti jsou takové systémy, které vyžadují vysokou spolehlivost a jejich selhání může mít významné negativní důsledky. Příkladem takových systémů můžou být letecké či pozemní dopravní prostředky, kde selhání může znamenat velké škody či dokonce smrt pasažérů. [10] V dnešní době se software stále více stává integrovanou součástí kritických systémů. Vývoj software, který je součástí takových systémů je kvůli požadavku na absenci kritických chyb velmi nákladná záležitost. Pro znázornění například software avionického systému ve stíhacím letounu F-22 Raptor plní 80 % funkcionality a stál 30 % všech nákladů na vývoj. Obvykle je velká část nákladů vynaložené na vývoj tohoto software věnována validaci a verifikaci, pro co možná nejlepší vyloučení chyb systému. Často je na tuto část vynaloženo i přes 50 % z celkových nákladů na vývoj. [8]
Vývoj software pro takové systémy podléhá standardům, jako například ISO 26262 pro automobilové odvětví nebo DO-178C pro letecké odvětví. Tyto dokumenty popisují nároky na bezpečnost celého systému stejně tak jako samotného software a obsahují také popis a požadavky na tvorbu a testování tohoto software. Kvůli omezení možnosti chyb je software pro kritické systémy navrhován v jednoduchých funkcích a částech. Tvorba jednotkových testů těchto funkcí splňující kritéria daných požadavků je oblast, která ve vývoji zabírá velké množství lidského času a prostředků. V posledních letech je proto možnost automatizace tvorby testů předmětem výzkumu.
Na automatickou tvorbu testů cílí různé metody a nástroje, které je využívají. Obecně tyto metody můžeme rozdělit na 2 odlišné přístupy, statický a dynamický. [15, 16] Dynamická analýza generuje testy analyzováním chování softwaru během jeho opakováném spouštění. Tento přístup má několik výhod: lehce se vypořádá se složitostí kódu, generuje sémanticky splnitelné cesty a zajišťuje vysoké pokrytí kódu testy. Nicméně počet vygenerovaných testovacích vstupů může být vysoký a generování může trvat velmi dlouho. Oproti tomu statická analýza analyzuje kód bez jeho spuštění. Tento přístup typicky generuje menší testovací sady. Obsahuje metody zejména z oblasti formální verifikace, zmíníme například symbolickou exekuci programu či model checking. [3, 4]
Tato práce má za cíl vytvořit open-source nástroj, který dle uživatelem zadaného kritéria pokrytí pomůže automaticky najít všechny nutné testovací vstupy. Toho je docíleno pomocí vytvořené metody řadící se do statického přístupu generování testovacích vstupů. Konkrétně se zaměřuje na generování testovacích vstupů z grafu toku řízení (CFG - Control Flow Graph) vytvořeného z analyzované funkce. CFG je model programu/funkce, který je možné vygenerovat překladači programovacích jazyků, jako je například GCC (GNU Compiler Collection) pro jazyky C, C++ a další. V práci je vyvinuta metoda generování syntakticky splnitelných cest takovým CFG podle určitého ohodnocení těchto cest. Tím může být

například počet projitých uzlů či počet výrazů v dané cestě, čímž metoda zajišťuje hledání řešení od nejjednodušších průchodů ke složitějším. Kvůli tomu se cesty negenerují přímo z grafu, ale z regulárního výrazu, na který je převeden přes konečný automat.
Z cest vygenerovaných touto metodou jsou potom vytvořeny formule pro SMT řešič, který ověří, zda je cesta sémanticky splnitelná, tedy existuje ohodnocení vstupních parametrů takové, které odpovídá průchodu této cesty. Pokud je formule splnitelná, SMT řešič, zároveň dodá ohodnocení vstupních parametrů, které odpovídají testovacímu vstupu. Takové vstupy jsou generovány pro cesty, které zaručují splnění nějakého požadavku zadaného kritéria pokrytí, dokud toto kritérium není ze 100% splněno nebo není dosaženo limitu počtu vygenerovaných cest.
Struktura práce je následující: V kapitole 2 je popsán teoretický základ nutný k pochopení práce, včetně formálních definic.,V kapitole 3 je popsán návrh nástroje a používané algoritmy. Kapitola 4 se věnuje implementaci navrženého nástroje a konečně v kapitole 5 je funkcionalita nástroje demonstrována na praktickém příkladě.

Kapitola 2

Současné techniky pro strukturální testování

V této kapitole je popsán teoretický základ nutný k pochopení práce. Je popsáno testování založené na modelech, konkrétně zaměřeno na generování testovací sady z grafu toku řízení, zahrnující tvorbu tohoto grafu ze zdrojového kódu pomocí GCC, popis řešičů SMT a nakonec jsou popsány kritéria pokrytí vyžadována při testování. Jedna sekce je také věnována převodu konečného automatu na regulární výraz.

2.1 Testování založené na modelech.

Testování založené na modelech je moderní technika testování software, která využívá abstraktího modelu testovaného software. Takový model je zjednodušeným popisem celkového testovaného systému - System Under Test (dále jen SUT) a reprezentuje vybrané chování SUT. Cílem je efektivní vytvoření testovací sady (množiny testů), která otestuje korektnost systému v rámci modelovaném chování. [12]

2.1.1 Graf toku řízení

Existuje několik používaných modelů systému, které se používají k testování. Jde o modely založené na jazycích UML a SysML, konečné automaty, časované automaty, grafy závislostí nebo rozhodovací tabulky a další. V této práci se však budeme zabývat modelem ve formě grafu toku řízení - Control Flow Graph (CFG).
CFG je abstraktní model SUT, který slouží jako grafová reprezentace řízení běhu SUT. Skládá se z uzlů a hran. Uzly reprezentují jednotlivé příkazy nebo posloupnosti příkazů (basic blocks) a hrany reprezentují přenesení kontroly mezi uzly. *Základní blok* (Basic block) je maximální posloupnost příkazů, které mají právě jeden vstupní a jeden výstupní bod a pro které tedy platí, že pokud je první z nich vykonán, jsou vykonány všechny, čili neobsahují žádné větvení, respektive pokud obsahují větvení, tak na konci bloku, kde předává řízení dále.
Formálně je CFG definován jako čtveřice $G = (N, E, N_0, N_f)$, kde

- N je konečná množina uzlů (Nodes),
- $E \subseteq N \times N$ je množina hran (Edges),
- $N_0 \subseteq N, N_0 \neq \varnothing$ je neprázdná množina počátečních uzlů,

- $N_f \subseteq N$ je množina koncových uzlů (final nodes).

V dnešní době je již generování CFG z kódu součástí statické analýzy překladačů.

2.2 Tvorba CFG z kódu

V této sekci je popsáno, jakým způsobem je ze zdrojového kódu generovaný graf toku řízení pomocí překladače GCC [14]. GCC (GNU Compiler Collection) je sada překladačů vyvinutá v rámci projektu GNU. Překladače původně sloužily pouze pro jazyk C, dnes jsou však již součástí vytvořené překladače pro jazyky Rust, Java, D a další.

Celkový proces překladu můžeme rozdělit do 4 částí: předzpracování, kompilace, převod do strojového kódu (assemble) a linkování. Při předzpracování se kód upraví pro další použití. V rámci této fáze probíhá: odstranění komentářů, vložení zahrnutých souborů (*#include* ...), distribuce názvu makra jejich hodnotou a kompilace podmínek, které zapojují nebo vynechávají kód na základě definice makra (#ifdef, #endif atd.). Další fází je kompilace, ve které se převede předzpracovaný kód do assembleru, který je v třetí fázi převeden do binárního strojového kódu. Poslední fází je linkování, ve které se spojí soubory z předchozího kroku do jediného spustitelného souboru. Pro tuto práci je důležitá fáze kompilace, ve které probíhá generování grafu toku řízení.

Fáze kompilace začíná analýzou předzpracovaného kódu a pokračuje tvorbou abstraktního syntaktického stromu. Na tom se provede sémantická analýza programu. Poté se generuje přechodná reprezentace GIMPLE, na které se provedou různé optimalizační techniky a vygeneruje se z ní kód pro další fázi.

2.2.1 GCC Gimple

Jak již bylo zmíněno, GIMPLE je přechodná reprezentace programu, která je článkem mezi člověkem čitelným kódem a assemblerem. Jde o reprezentaci programu, která je nezávislá na zdrojovém jayzce a cílové architektuře. Gimple vychází z GENERIC reprezentace, která reprezentuje celé funkce jako strom. Instrukce z GENERIC reprezentace jsou přepsány na n-tice o maximálně 3 operandech (s výjimkami jakou je například volání funkcí). Toho je dosaženo pomocí odstranění vysoko-úrovňových konstrukcí, jako jsou například for nebo while cykly, které jsou nahrazeny skoky (goto instrukcemi) a také zavedením dočasných proměnných, které dovolují zjednodušit komplexní výrazy.
Tyto principy jsou demonstrovány na příkladu 2.1, kde vidíme funkci s jednoduchým while cyklem, který je v Gimple reprezentaci nahrazen goto příkazy. Taktéž vidíme zjednodušení komplexního výrazu pomocí dočasných proměnných _1, _2 a _3.

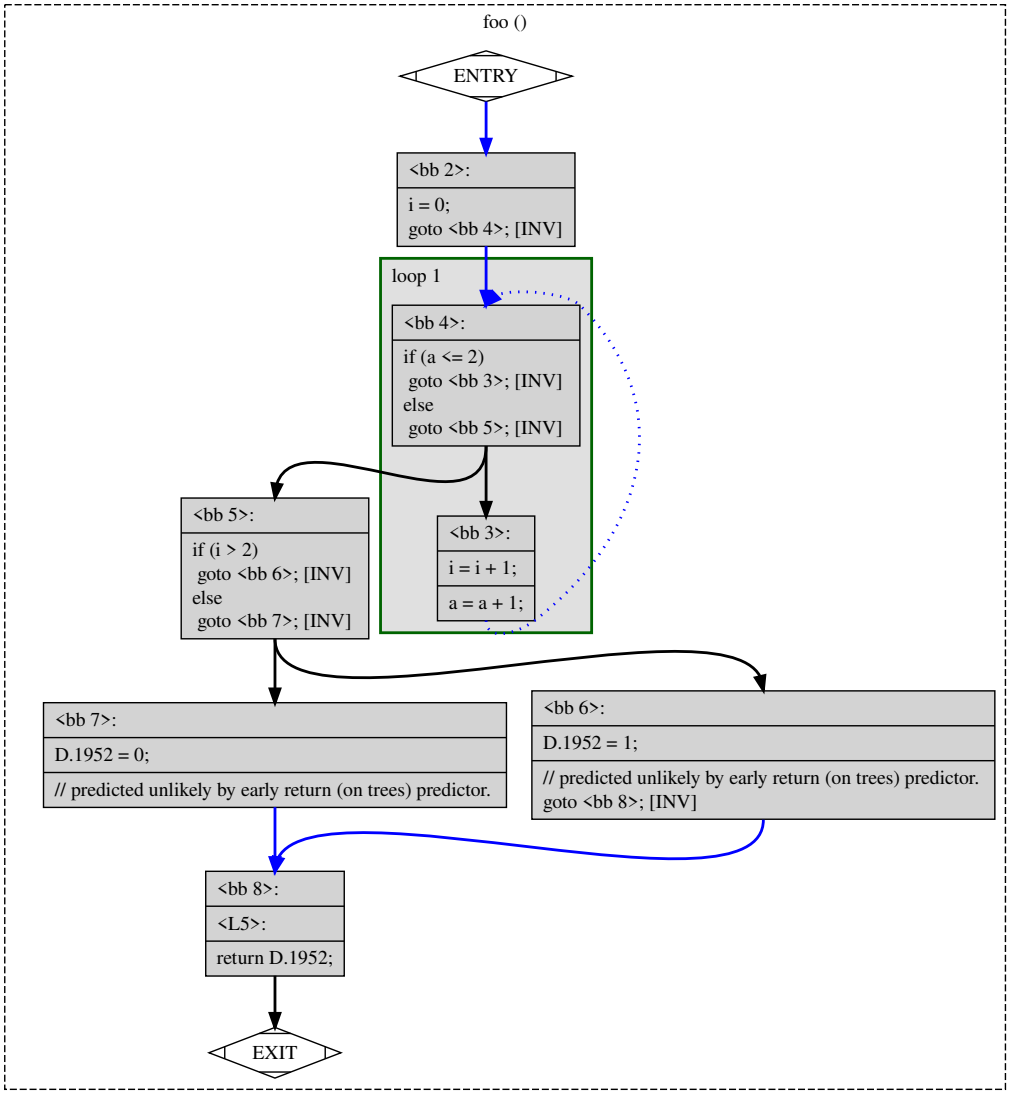
Za účelem optimalizací je poté z Gimple reprezentace vytvořen graf toku řízení detekováním základních bloků, kdy je rozpoznám vstupní a výstupní bod. Základní bloky jsou následně propojeny hranami grafu. Následně jsou provedeny optimalizace na základě vygenerovaného grafu, jako jsou například optimalizace smyček, odstranění mrtvého kódu nebo eliminace (nahrazení) stejných podvýrazů (common sub-sequence elimination).
Na obrázku 2.2 můžeme vidět funkci ukázanou dříve převedenou do podoby grafu toku řízení, kde každý blok v grafu znázorňuje základní blok. Můžeme také vidět, že byla detekována while smyčka.

```
int foo(int a){
    int i = 0;
    while(a<3){
        i = a*2+8/a + 2;
        a++;
    }
    if(i>2){
        return 1;
    } else {
        return 0;
    }
}
```

```
int foo (int a)
{
  int D.1952;
  int i;

  i = 0;
  goto <D.1947>;
  <D.1948>:
  _1 = a * 2;
  _2 = 8 / a;
  _3 = _1 + _2;
  i = _3 + 2;
  a = a + 1;
  <D.1947>:
  if (a <= 2) goto <D.1948>;
  else goto <D.1946>;
  <D.1946>:
  if (i > 2) goto <D.1950>;
  else goto <D.1951>;
  <D.1950>:
  D.1952 = 1;
  return D.1952;
  <D.1951>:
  D.1952 = 0;
  return D.1952;
}
```

Obrázek 2.1: Jednoduchá funkce v jazyce C (vlevo) a její reprezentace v GCC Gimple (vpravo).



Obrázek 2.2: Funkce z 2.1 převedená na Graf toku řízení pomocí překladače GCC

2.3 Převod konečného automatu na regulární výraz

V práci je použitý převod konečného automatu na regulární výraz jako analýza grafu toku řízení, který je převeden na konečný automat, pro možnost generování cest podle určitého ohodnocení. V této sekci je definovaný teoretický základ nutný k pochopení tohoto převodu. Jsou tedy formálně definovány konečné automaty, regulární výrazy a další nutné prerekvizity.

2.3.1 Konečné automaty

Konečný automat je nástroj pro reprezentaci jazyka, na který umíme lehce převést i graf toku řízení, kde množina všech testovacích cest odpovídá danému jazyku.
Formálně je konečný automat M 5-tice:

$M = (Q, \Sigma, \delta, q_0, F)$, kde:

- Q je konečná množina stavů,
- Σ je konečná vstupní abeceda,
- δ je funkce přechodů (přechodová funkce) tvaru $\delta : Q \times \Sigma \to 2^Q$,
- $q_0 \in Q$ je počáteční stav,
- $F \subseteq Q$ je množina koncových stavů.

V definici můžeme vidět podobnost s CFG, kde konečný automat obsahuje množinu stavů (uzlů), přechody mezi nimi a počáteční a koncové stavy obdobně jako CFG.

2.3.2 Regulární výrazy

Definujeme si také regulární výrazy a to za pomoci regulárních množin: Nechť Σ je konečná abeceda. Regulární množinu nad Σ definujeme rekurzívně takto:

- $\varnothing$ (tj. prázdná množina) je regulární množina nad Σ,
- ϵ je regulární množina nad Σ,
- a je regulární množina nad Σ pro všechny $a \in \Sigma$,
- jsou-li P a Q regulární množiny nad Σ, pak také
  - (a) $P \cup Q$,
  - (b) $P.Q$,
  - (c) P^*,

  jsou regulární množiny nad Σ.
- Žádné jiné množiny, než ty, které lze získat pomocí výše uvedených pravidel, nejsou regulárními množinami.

Regulární výrazy nad Σ a regulární množiny, které označují, jsou rekurzívně definovány takto:

- $\varnothing$ je regulární výraz označující regulární množinu $\varnothing$,
- ϵ je regulární výraz označující regulární množinu ϵ,
- a je regulární výraz označující regulární množinu a pro všechny $a \in \Sigma$,
- jsou-li p, q regulární výrazy označující regulární množiny P a Q, pak
  - $(p + q)$ je regulární výraz označující regulární množinu $P \cup Q$,
  - (pq) je regulární výraz označující regulární množinu $P.Q$,
  - (p^*) je regulární výraz označující regulární množinu P^*.
- Žádné jiné regulární výrazy nad Σ neexistují.

Jelikož regulární výrazy a konečné automaty mají ekvivalentní vyjadřovací sílu, lze převést konečný automat na regulární výraz reprezentující ekvivalentní jazyk. Existuje několik způsobů, jak toho lze docílit. Nejznámější metody jsou: metoda eliminace stavů, metoda tranzitivního uzávěru a brzozowského algebraická metoda. V této práci je použita poslední zmíněná metoda odpovídající převodu za pomoci soustavy rovnic.

Soustava rovnic nad regulárními výrazy je ve standardním tvaru vzhledem k neznámým $\Delta = X_1, X_2, \ldots, X_n$, má-li soustava tvar

$$\bigwedge_{i \in \{1,\ldots,n\}} X_i = \alpha_{i0} + \alpha_{i1}X_1 + \alpha_{i2}X_2 + \cdots + \alpha_{in}X_n$$

, kde kde α_{ij} jsou reg. výrazy nad nějakou abecedou $\Sigma, \Sigma \cap \Delta = \varnothing$.
Na takovou soustavu rovnic můžeme převést také konečné automaty nad neznámou X_i vyjadřující slova, která jsou přijímány automatem, pokud začíná ve stavu q_i:

$$\bigwedge_{i \in \{1,\ldots,n\}} X_i = \bigcup_{q_i \xrightarrow{a} q_j} aX_j + \begin{cases} \{\varepsilon\} & , \ q_i \in F \\ \emptyset & , \ \text{else} \end{cases}$$

,kde $q_i \xrightarrow{a} q_j$ znamená, že existuje cesta z q_i do q_j s a a počet stavů n. Pokud čteme $\cup$ jako $+$, dostaneme stejný formát jako mají rovnice nad regulárními výrazy. Výsledek převodu získáme vyřešením rovnice pro Q_i, kde i je počáteční stav automatu. Vyřešit rovnici lze za použití distributivity operátorů $+$ a $.$ a asociativity operátoru $.$ a použití Ardenova pravidla, které říká, že pokud máme jazyky $L, U, V \subseteq \Sigma^*$ a platí $\epsilon \notin U$, potom

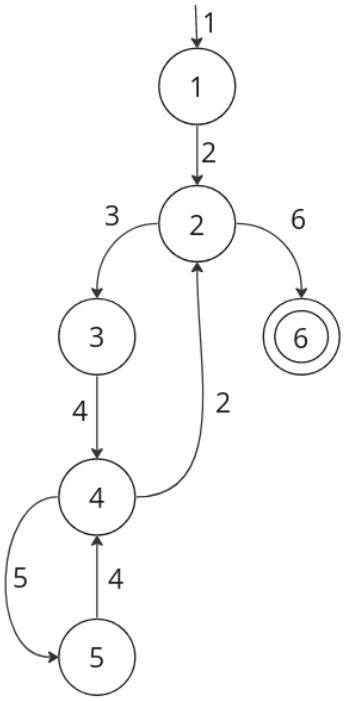
$$L = UL + V \quad \Longleftrightarrow \quad L = U^*V$$

2.3.3 Příklad

Převod konečného automatu na regulární výraz si ukážeme na příkladu. Použijeme CFG ukázaného na Obrázku 2.4. Ten převedeme na konečný automat označením hran jménem cílového uzlu viz Obrázek 2.3. Z tohoto automatu vytvoříme jednotlivé rovnice:
$X_1 = 2X_2$
$X_2 = 3X_3 + 6X_6$
$X_3 = 4X_4$
$X_4 = 2X_2 + 5X_5$
$X_5 = 4X_4$
$X_6 = \epsilon$
Dosazením X_5 do X_4 vznikne: $X_4 = 2X_2 + 54X_4$.
Aplikujeme Ardenovo pravidlo: $X_4 = (54)^*2X_2$
Dosadíme do X_3: $X_3 = 4(54)^*2X_2$
Do X_2 dosadíme X_3 a X_6 a dostaneme: $X_2 = 34(54)^*2X_2 + 6$
Aplikujeme Ardenovo pravidlo a dosadíme do X_1: $X_1 = 2(34(54)^*2)^*6$
Přidáme počáteční stav a máme konečný výraz: $12(34(54)^*2)^*6$. Jazyk který generuje tento regulární výraz odpovídá všem možnostem průchodu grafem a tvoří tedy v podstatě syntakticky správné testovací cesty.

2.3.4 Algoritmus

V této části se podíváme na převod konečného automatu z pohledu implementovatelného algoritmu. I když řešení soustavy rovnic z konečného automatu se může zdát příliš symbo-



Obrázek 2.3: Konečný automat vytvořený z CFG na obrázku 2.4

lická, algoritmus je vhodný pro implementaci.
Mějme automat ve formě rovnic ve tvaru:

$$X_i = B_i + A_{i,1}X_1 + \ldots + A_{i,n}X_n$$

, kde $A_{i,j}$ představuje přechod z q_i do q_j a B_i představuje množinu obsahující ϵ pro $q_i \in F$, jinak je rovno $\emptyset$.
Indukcí nad i v kroku n dostaneme rovnici:

$$X_n = B_n + A_{n,1}X_1 + \ldots + A_{n,n}X_n$$

Použitím Ardenova pravidla upravíme rovnici:

$$X_n = A^*_{n,n}(B_n + A_{n,1}X_1 + \ldots + A_{n,n-1}X_{n-1}) \Leftrightarrow$$
$$X_n = A^*_{n,n}B_n + A^*_{n,n}A_{n,1}X_1 + \ldots + A^*_{n,n}A_{n,n-1}X_{n-1}$$

Nastavením $B'_n = A^*_{n,n}B_n$ a $A'_{n,i} = A^*_{n,n}A_{n,i}$ dostaneme:

$$X_n = B'_n + A'_{n,1}X_1 + \ldots + A'_{n,n-1}X_{n-1}$$

Nyní již můžeme odstranit veškeré X_n ze soustavy převodem pro $i, j < n$:

$$B'_i = B_i + A_{i,n}B'_n$$
$$A'_{i,j} = A_{i,j} + A_{i,n}A'_{n,j}$$

Pokud vyřešíme X_n pro $n = 1$ (tedy pro počáteční stav), dostaneme rovnici:

$$X_1 = B'_1$$

bez $A'_{1,i}$, což je výsledný regulární výraz.
Samotný algoritmus vychází z myšlenky těchto úprav. Označme počáteční stav q_i a počet stavů jako m. Začněme inicializací polí A a B:

```
for i = 1 to m:
  if final(i):
    B[i] := ϵ
  else:
    B[i] := ∅
```

```
for i = 1 to m:
  for j = 1 to m:
    for a in Σ:
      if trans(i, a, j):
        A[i,j] := a
      else:
        A[i,j] := ∅
```

Vlastní algoritmus potom odpovídá jednotlivým zmíněným úpravám:

```
for n = m decreasing to 1:
  B[n] := star(A[n,n]) . B[n]
  for j = 1 to n:
    A[n,j] := star(A[n,n]) . A[n,j];
  for i = 1 to n:
    B[i] += A[i,n] . B[n]
    for j = 1 to n:
      A[i,j] += A[i,n] . A[n,j]
```

, kde $star(a)$ vyjadřuje regulární výraz a^*. Výsledný regulární výraz je potom roven B[1]. Popsaná Brzozowského algebraická metoda byla zvolena zejména kvůli tomu, že oproti ostatním metodám generuje kompaktní regulární výrazy. [2]

2.4 Kritéria pokrytí vyžadovaná při testování kritických aplikací.

Při tvorbě testovací sady, tedy množiny testovacích případů, je důležitým faktorem, jakou část programu sada vlastně otestuje a tedy jaký má sada potenciál k objevení případné chyby v programu. Při tvorbě testovací sady používáme požadavky na testy - test requirments (TR), které vyjadřují cíl pro tvorbu testovacích případů. Tyto požadavky jsou definovány kritériem pokrytí. Formálně je kritérium pokrytí předpis pro systematické generování požadavků na test. Pokrytí potom vyjadřuje míru, s jakou testovací sada zkoumá SUT (System under test) vztaženo k danému kritériu pokrytí. Při tvorbě testovací sady, v našem případě množině testovacích vstupů, je za cíl splnit 100 % pokrytí předem daného kritéria. Nejčastějším kritériem je pokrytí všech řádků (code coverage), které znamená, že spuštění testů testovací sady projde programem tak, aby každý řádek zdrojového kódu byl proveden alespoň jednou. Toto kritérium však často není dostatečné pro otestování všech chování systému. Pro splnění našich účelů se zaměříme na kritéria pokrytí grafů, jelikož pracujeme s grafem toku řízení. Před výčtem a popisem jednotlivých kritérií je potřeba formálně popsat některé pojmy pomocí kterých jsou kritéria pokrytí grafů běžně popisovány nebo s kritérii pokrytí souvisí: [1]

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

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

Délka cesty je určena počtem hran cesty.

Testovací cesta je cesta v CFG grafu, která začíná v počátečním uzlu a končí v některém z koncových uzlů, tedy je to sekvence uzlů $[n_1, n_2 \dots n_M]$ taková že:

$$n_1 \in N_0, n_M \in N_f$$

Podcesta cesty p je souvislý podřetězec z p.

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

Definujeme predikát **path(t)** pro testovací případ t pro cestu spuštěnou testem t.
Podobně definujeme predikát **path(T)** jako množinu cest, které jsou spouštěny z testovacími případy z testovací sady (množiny testů) T.

Uzel n je **syntakticky dosažitelný** z uzlu n_i, pokud existuje cesta z n_i do n.

Kritérium C_1 **zahrnuje** kritérium C_2 ($C_1 > C_2$) právě tehdy, když každá testovací sada, která splňuje kritérium C_1 splňuje také kritérium C_2. Pomocí takového porovnání můžeme určit určitou hierarchii kritérií. Pokud je pokrytí silnějšího kritéria 100 %, bude stejně tak splněno pokrytí i každého slabšího kritéria.

Kritéria pokrytí můžeme rozdělit na **strukturální** - definované pouze na grafu, tedy zahrnují pouze uzly a hrany CFG, kritéria pokrytí **logických výrazů**, které zkoumá podmínky při větvení grafu a kritéria **toku dat**, které vyžadují informaci o proměnných.

2.4.1 Strukturální kritéria pokrytí

Nejjednodušším a také nejslabším strukturálním kritériem pokrytí je pokrytí uzlů grafu - **Node Coverage (NC)**, které vyžaduje projití každého syntakticky dosažitelného uzlu, formálněji:
Testovací sada T splňuje NC pokud a pouze pokud pro každý syntakticky dosažitelný uzel z $n \in N$ existuje cesta $p \in path(T)$ obsahující uzel n.
Jednodušeji můžeme kritérium popsat pomocí požadavků na testy TR (Test Requirments): NC vyžaduje, aby TR obsahovaly každý syntakticky dosažitelný uzel.

Další jednoduché kritérium je kritérium pokrytí hran - **Edge Coverage (EC)**, které definujeme podobně:
EC vyžaduje, aby TR obsahovaly každou dosažitelnou cestu o délce do 1 včetně.
I díky zahrnutí cesty o délce 0 je pokrytí hran o něco silnější než pokrytí uzlů grafu.

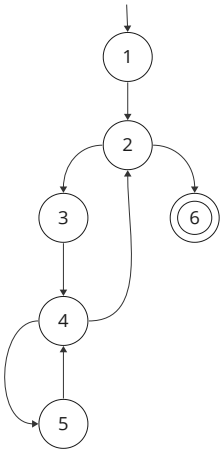
Další kritéria vyžadují zapojení více hran. Jedním z nich je **Edge pair Coverage (EPC)**, které vyžaduje, aby TR obsahovalo každou dosažitelnou cestu o délce 2 nebo menší. Zmínka o menších délkách cest je zde pro zahrnutí grafů s jednou nebo žádnou hranou.

Nejsilnějším kritériem by bylo kritérium pokrytí všech cest - **Complete Path Coverage**

(CPC), které vyžaduje aby TR obsahovalo všechny cesty v grafu. Vzhledem k možnosti cyklů a tedy potencionálnímu nekonečnému počtu cest v grafu je toto kritérium v praxi nepoužitelné.

Při snaze o *vyřešení problému cyklů* vznikla kritéria, které prochází cykly právě jednou, 0 krát, jednou a vícekrát a další, ale v praxi se uchytilo elegantní řešení s konečným počtu cest, které vyžaduje projití cyklu stejně jako jeho vynechání:
Kritérium pokrytí primárních cest **Prime Path Coverage (PPC)**: TR obsahuje všechny primární/hlavní cesty v grafu, kde:
Primární cesta je taková jednoduchá cesta, která není podcestou žádné jiné jednoduché cesty, jde tedy o nejdelší jednoduchou cestu.

Demonstrujeme splnění jednotlivých kritérií na příkladu ukázaném na obrázku 2.4.



Obrázek 2.4: Příklad CFG pro demonstrování jednotlivých kritérií pokrytí.

V tabulce 2.1 můžeme vidět, jaké podcesty vyžadují kritéria pokrytí uzlů, hran a dvo-

Kritérium pokrytí	EC	NC	EPC
Nutné ke splnění	[1,2], [2,3], [2,6],[3,4],[4,5], [4,2],[5,4]	1, 2, 3, 4, 5, 6	[1,2,3], [1,2,6], [2,3, 4],[3,4,2] ,[3,4,5],[4,5,4],[4,2,6], [4,2,3], [5,4,2],[5,4,5]
Cesty splňující kritérium	[1, 2, 3, 4, 5, 4, 2, 6]	[1, 2, 3, 4, 5, 4, 2, 6]	[1, 2, 3, 4, 5, 4, 5, 4, 2, 6], [1, 2, 6], [1, 2, 3, 4, 2, 3, 4, 2, 6]

Tabulka 2.1: Tabulka ukazující kritéria pro splnění jednotlivých kritérií a příklady jejich splnění pro příklad 2.4.

jic hran. Zatímco první 2 kritéria, tedy **NC** a **EC** lze jednoduše splnit jedinou testovací cestou, pro silnější kritérium **EPC** již musíme testovací sadu rozšířit na více cest. Barvy jednotlivých podcest kritéria vyjadřují, jakou cestou byly splněny.

V další tabulce 2.2 je ukázána tvorba jednoduchých cest. V každém sloupci jsou syntakticky dosažitelné cesty o jednotlivých délkách a v řádcích cesty začínající určitým uzlem. Tabulka vznikala rozšiřováním cest z jednotlivých uzlů, tak aby nebyly porušeny podmínky definující jednoduchou cestu. Vykřičníkem jsou pak označeny takové cesty, které končí koncovým uzlem nebo mají stejný počáteční uzel jako koncový. Zbytek cest nelze rozšířit, protože dosažitelné uzly už jsou přítomné v dané cestě a nejsou prvním uzlem. Cesty které jsou zabarveny červeně jsou zároveň primární cesty, protože nejsou podcestou jiné jednoduché cesty. Můžeme si také všimnout, že kritérium pokrytí primárních cest (**PPC**) je silnější, než **EPC**, protože cesty, které kompletně splňují EPC ukázané v Tabulce 2.1 nesplňují PPC a to kvůli podcestě [5,4,2,3], která v dané testovací sadě není obsažena.

Délka 1	Délka 2	Délka 3	Délky 4 a 5
[1]	[1,2]	[1,2,3] [1,2,6]!	[1,2,3,4] ->[1,2,3,4,5]
[2]	[2,3] [2,6]!	[2,3,4]	[2,3,4,5] [2,3,4,2]!
[3]	[3,4]	[3,4,2] [3,4,5]	[3,4,2,6]! [3,4,2,3]!
[4]	[4,2] [4,5]	[4,2,3], [4,2,6]! [4,5,4]!	[4,2,3,4]!
[5]	[5,4]	[5,4,5]! [5,4,2]	[5,4,2,3] [5,4,2,6]!
[6]!			

Tabulka 2.2: Tabulka ukazující jednoduché cesty pro jednotlivé délky cest a primární cesty (červeně) pro příklad 2.4.

2.4.2 Kritéria pokrytí logických výrazů

Dalším pohledem na pokrytí je pohled vyhodnocování logických výrazů, které jsou použity ve větvení v grafu toku řízení. Často jsou v těchto výrazech chyby, které nemusí být odhaleny prostým projitím všech větví. Typicky tedy zkoumáme logický výraz, kterému říkáme **predikát**. Predikát je výraz, který se vyhodnocuje na booleovskou hodnotu (nejvyšší struktura, neobsahuje další predikáty). Může obsahovat proměnné, relační operátory, vo-

lání funkcí nebo logické operace. Logický výraz, který neobsahuje žádnou logickou spojku nazýváme **klauzule**. V odborné literatuře se pro tyto termíny rovněž používají termíny decision (predikát) a condition (klauzule) Dále si ještě nadefinujeme:
P je množina všech predikátů testovaného systému.
$C_p = \{c|c \in p\}$ je množina všech klauzulí predikátu p.
$C = \bigcup_{p \in P} C_p$ je množina všech klauzulí SUT.

Intuitivním kritériem pokrytí logických výrazů je kritérium pokrytí všech predikátů (**Predicate coverage - PC**). Kritérium vyžaduje, aby se každý predikát $p \in P$ vyhodnotil jak na *true*, tak na *false*. Jelikož každý predikát představuje jedno větvení CFG, znamená to vlastně, že se vezme každá hrana v SUT a toto pokrytí je srovnatelné s pokrytím hran (EC).

Podobně je definováno i kritérium pokrytí všech klauzulí (**Clause Coverage - CC**), které vyžaduje, aby se každá klauzule $c \in C$ vyhodnotila jak na *true*, tak na *false*. Cílem tohoto kritéria je zjistit, zda není chyba v klauzulích. Porovnání s ostatními kritérii není.

Dalším, kvůli počtu možných přiřazení nepraktickým kritériem, je kritérium pokrytí všech kombinací (**Combinatorial Coverage - CoC**). Toto kritérium vyžaduje aby pro každý predikát $p \in P$ TR obsahovaly požadavek pro klauzule C_p, aby byly vyhodnoceny všechny kombinace jejich pravdivostních hodnot.

V praxi používané kritérium, které je dokonce součástí standardu DO-187C je kritérium **Modified Condition/Decision Coverage - MCDC**. To zahrnuje všechny pravdivostní hodnoty všech predikátů i všech klauzulí a zároveň vybírá pouze určité kombinace hodnot klauzulí. Tím zvyšuje šanci odhalení chyby. Pro omezení počtu kombinací jsou zavedeny termíny minoritní a majoritní klauzule:
Majoritní klauzule c_i predikátu p určuje jeho hodnotu, pokud všechny **minoritní** klauzule $c_j \in p, j \neq i$ mají hodnoty takové, že změna pravdivostní hodnoty c_i změní pravdivostní hodnotu p.
Samotné kritérium pokrytí MCDC vyžaduje aby pro každý predikát $p \in P$ a každou majoritní klauzuli $c \in C_p$ TR obsahovaly dva požadavky: c se vyhodnotí na true a c se vyhodnotí na false.

2.4.3 Kritéria pokrytí toku dat

Kritéria pokrytí toku dat jsou určeny tím, jak jsou definovány a použity proměnné v programu. Pokrytí je založeno na myšlence, že pro každý výraz v programu bychom měli brát v potaz každou možnou definici proměnné, která je použitá v daném výrazu. Pro popis jednotlivých kritérií opět zavedeme několik definicí:
Pro uzel v grafu v:

Defs(v) je množina všech proměnných které jsou definovány v uzlu v.

Undef(v) je množina všech proměnných, jejichž hodnota je nedefinovaná po provedení kódu v uzlu v.

c-use(v) jsou všechny proměnné, které jsou použity pro definici jiných proměnných v uzlu v.

p-use(v, v') jsou všechny proměnné, které jsou použity při použití hrany (v, v').

Cesta $v_0, v_1, ..., v_k$ je nazývána **def-clear** (prosta od definic) pro x, pokud x $\notin$ defs(v_i) pro 0 < i < k.

S těmito definicemi již můžeme zmínit některé z kritérií pokrytí toku dat:
Kritérium všech definic (all-defs) vyžaduje, aby všechny definice proměnné x v SUT, sada testovacích cest Π obsahuje def-clear cestu z definice k alespoň jednomu c-use nebo jednomu p-use proměnné x. Intuitivně jde o kritérium, ve kterém musí být použity všechny definice proměnné.

Kritérium všech c-use (all c-uses) vyžaduje, aby aby všechny definice proměnné x v SUT a každé c-use proměnné x dosažitelné z definice, Π obsahuje def-clear cestu z definice k c-use. Tedy v principu jde o využití výpočtu s proměnnou s každou možnou definicí proměnné pro tento výpočet.

Kritérium všech p-use (all p-uses) vyžaduje, aby aby všechny definice proměnné x v SUT a každé p-use proměnné x dosažitelné z definice, Π obsahuje def-clear cestu z definice k p-use. Tedy každé rozvětvení ovlivněné dekaždou definicí je zkoumáno.

Další používaná kritéria kombinují použití p-use a c-use. Kritéria toku dat poskytují lepší měření pokrytí, protože zohledňují závislosti mezi definicí a použitím proměnné. Výpočet přesných závislostí je však problematický a tak jsou využívány aproximace. [1]

2.5 Řešiče SMT

Testovací cesta reprezentuje posloupnost příkazů a podmínek programu. Pro vytvoření testovacího případu z testovací cesty, je potřeba ověřit, zda je cesta *sémanticky dosažitelná*. Sémantická dosažitelnost testovací cesty znamená, že existuje ohodnocení vstupních proměnných, takové že stopa programu, který je spuštěný s takovým ohodnocením vstupů, odpovídá dané cestě. Problém, zda je cesta sémanticky dosažitelná je formulovatelný jako problém splnitelnosti.

2.5.1 SAT

Problém splnitelnosti (označován jako SAT z Satisfiability) je problém, zda pro zadanou logickou formuli existuje takové ohodnocení proměnných formule takové, že je formule vyhodnocena jako *true*. Pokud takové ohodnocení existuje, označujeme formuli jako splnitelnou. Pokud žádné takové ohodnocení neexistuje, je formule označována jako nesplnitelná. Pokud se bavíme pouze o výrokové logice, kde se používají jen booleovské proměnné, označujeme problém jako problém splnitelnosti booleovské formule.

Řešič SAT (SAT Solver) je program, jehož cílem je vyřešit zadaný problém splnitelnosti booleovské formule. Vstupem takového programu je tedy formule obsahující pouze booleovské proměnné a výstupem je, zda je formule splnitelná či nesplnitelná. Jelikož je problém obecně NP-úplný, jsou známé algoritmy, které ho vyřeší s nejhůře exponenciální časovou složitostí. Nicméně v dnešní době existují již velmi efektivní algoritmy s řadou rozšíření, které významně urychlují nalezení řešení pro reálné instance problému. [9]

2.5.2 SMT

Satisfiability modulo theories (SMT) je rozšířením SAT problému, takže zahrnuje i formule které obsahují například celá čísla, reálná čísla, pole atd. Výrazy formule jsou vyhodnocovány v rámci určié formální teorie. Formálně, instance SMT je formule predikátové logiky prvního řádu s rovností, kde predikáty mají další význam definovaný určitou formální teorií. Stejně jako u SAT je cílem problému rozhodnout, zda je formule splnitelná. Motivací pro řešení takového problému je snazší kódování problému a usnadnění optimalizace.

Řešice SMT (SMT Solvers), jsou programy s cílem vyřešit danou instanci SMT problému. Přestože SMT problém je typicky NP těžký a pro mnoho teorií nerozhodnutelný, vývoj SMT solverů v poslední době zaznamenal velký pokrok a využití nachází například ve formální verifikaci programů, analýze programů a automatickém testování. Moderní SMT řešiče spojují SAT řešiče a řešiče teorií. Zjednodušeně fungují na principu, kdy oddělí výrazy na jedinou teorii a zavedou sdílené proměnné. Potom vyhodnotí oddělené výrazy a nakonec si vymění rovnice týkající se sdílených proměnných. Řešiče SMT jsou schopné kromě odpovědi na otázku splnitelnosti poskytnout také ohodnocení vstupních proměnných, které splňuje danou formuli, pokud takové řešení existuje.

Jedním z nejznámějších SMT řešičů je Z3, což je open-source nástroj vyvíjený firmou Microsoft, který používá standardizovaný jazyk SMT-LIB. Z3 umí řešit formule zadané v jedné nebo více teoriích zahrnující aritmetické, vektorové, teorie polí nebo textových řetězců. [13]

2.6 Existující přístupy

V oblasti automatického generování testovacích případů již vzniklo množství nástrojů a metod, které často používají přístupy statické analýzy, jako jsou symbolická exekuce a varianty techniky model-checkingu. V této sekci nabídneme stručný přehled těchto nástrojů a technik. I když všechny tyto nástroje úzce souvisí s problémem této práce, žádný z nich se nezohledňuje splnění zadaného kritéria pokrytí nebo jen základních kritérií.

2.6.1 KLEE

KLEE je nástroj používající symbolickou exekuci určený pro automatické generování testovacích případů pro softwarové programy. Je populární v oblasti testování softwaru a analýzy programů. Symbolická exekuce je technika, která prozkoumává všechny možné cesty programu a to reprezentací vstupů programu pouze symbolicky než přímo konkrétními hodnotami. KLEE používá tento přístup k systematickému zkoumání různých testovacích cest a generování testovacích vstupů, které vedou k významnému pokrytí kódu. Ve svém jádru používá rešič SMT zvaný STP constraint solver pro nalezení mrtvého kódu a nastavení symbolických proměnných tak, aby mohl spustit systém pro danou cestu. [3]

2.6.2 CBMC

Bounded Model Checking (BMC) technika formální verifikace. V principu funguje na základě ověřování, zda systém neobsahuje chybu v exekucích, jejichž délka je omezená délkou k. Pokud chyba není nalezena, zvětšuje k, dokud buď nenajde chybu a nebo k nedosáhne horního limitu, po kterém prohledávání ukončí. BMC může být převeden na problém splnitelnosti booleovské proměnné a tedy se dá řešit SAT Solvery.

CBMC je *Bounded model checker* pro programy v jazyce C/C++. Verifikace probíhá na základě rozbalení smyček až do daného omezení na čemž jsou potom ověřovány dané vlastnosti. [4]

2.6.3 CFT4CUnit

Velmi podobný přístup jako prezentovaná metoda nabízí nástroj CFT4Unit. Nástroj používá převod do grafu toku řízení na kterém potom hledá všechny sémanticky splnitelné cesty splňující kritérium pokrytí za pomoci zpětného prohledávání a SMT řešiče. Cykly řeší nastavením horním limitem počtem průchodů a nabízí algoritmus pro rychlejší zpracování jednoduchých (nezanořených) a zanořených (jeden vnitřní a jeden vnější cyklus) smyček. [5]

2.6.4 WCFT4Cpp

Na nástroj CFT4Unit navazuje nástroj WCFT4Cpp pro C++ funkce, který používá velmi podobný přístup. Hlavním rozdílem je zavedením váhovaného grafu, podle kterého se prohledávají jednotlivé cesty, díky čemuž dosahuje efektivnějšího prohledávání. Oba nástroje používají Z3 Solver. [7]

2.6.5 Diffblue

Diffblue je nástroj, který je schopný vytvářet jednotkové testy pro Java kód a to dokonce plně autonomně. K tomu využívá umělou inteligenci. Konkrétní postupy tohoto nástroje však nejsou zveřejněny.

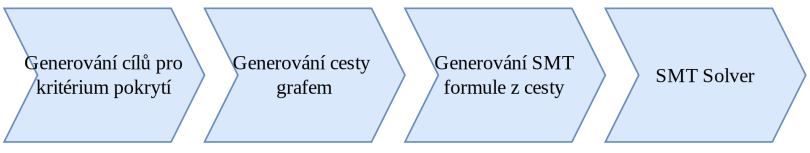
Kapitola 3

Návrh nástroje pro generování testovacích vstupů

V této kapitole je nejprve popsána hlavní myšlenka práce a problém, který má práce za cíl řešit a jsou nastíněny možnosti tohoto řešení. Následně jsou dle zvoleného řešení také definovány formální požadavky na nástroj, které má práce za cíl splnit. Nakonec je popsán návrh nástroje, z jakých logických částí se skládá, jsou popsány veškeré klíčové algoritmy, které jsou finálním nástrojem používány včetně možností pro zlepšení a další práci.

3.1 Popis problému a možnosti řešení

Celková myšlenka finálního produktu, do kterého zapadá tato práce, je vytvoření nástroje pro automatické testování funkcí kritického software. Tedy programátor vytvoří program či funkci, kterou předá na vstup nástroje a ten mu vytvoří testy a otestuje požadovanou funkcionalitu podle vybraného kritéria pokrytí. Nástroj, který vznikl v rámci této práce se zabývá pouze částí celkového nástroje a to částí generování testovacích vstupů z programu převedeného na CFG. Na vstupu je tedy graf toku řízení a kritérium pokrytí, na výstupu hodnoty testovacích vstupů, s jejichž spuštěním otestují zadané kritérium pokrytí. Převod funkcí na CFG pomocí překladače GCC je teoreticky popsán v kapitole 2.2. Nicméně nástroj je navržen univerzálně, jelikož na vstupu není kód, ale graf toku řízení, nástroj tedy může generovat cesty a následně hodnoty vstupů pro kód v jakémkoliv jazyce, pokud existuje převod kódu na CFG.
Na nižší úrovni se nástroj rozpadá na několik částí, konkrétně: generování cílů pro zadané kritérium pokrytí, generování cest vstupním grafem a generování SMT formule z dané cesty a předání SMT Solveru. Poslední částí je pak propojení těchto modulů. Tato posloupnost je ukázána na obrázku 3.1.
Hlavním problémem, který práce řeší je generování sady cest vstupní grafem, tak aby spl-



Obrázek 3.1: Posloupnost jednotlivých částí nástroje

ňovaly zadané kritérium pokrytí a jejich převod na formuli pro SMT Solver. Při návrhu

řešení problému generování cest vyplynuly dvě logické řešení tohoto problému.
Prvním řešením, které bylo vyzkoušeno v první verzi návrhu, je náhodný průchod grafem, ve kterém byly zapamatovány všechny rozhodnutí (podmínky nebo cykly), které se v průchodu použily. Pokud se v rámci jednoho průchodu zopakovalo stejné rozhodnutí, byla zvolena druhá možnost než poprvé. Pokud výsledná cesta nesplnila podcestu kritéria pokrytí nebo nebyla sémanticky správná (feasible), při dalších průchodech se postupně volily jiné možnosti průchodu od posledního zvoleného.
Šlo tedy o jakési upravené prohledávání do hloubky. Při tomto přístupu vyvstal problém s délkou vygenerovaných cest, které mohly být velmi dlouhé již od začátku generování. Jednoduché řešení tedy mohlo být nalezeno v relativně dlouhém čase. Tím vznikl požadavek na postupné generování cest v závislosti na určitém kritériu. Takovým kritériem může být např. počet uzlů v cestě, počet výrazů v cestě nebo počet přiřazení v cestě. Pro splnění tohoto požadavku není možné jednoduše procházet graf na základě jednotlivých rozhodnutí, protože bez analýzy celého grafu není možné postupně generovat cesty s nejlepším ohodnocením, obzvláště pokud graf obsahuje složitější podgrafy, jako jsou například vnořené cykly. Z toho vyplynulo druhé řešení, kterým je přidání analýzy grafu, která předchází generování cest a která detekuje cykly a je schopna zajistit splnění tohoto vzniklého požadavku. V této práci je touto analýzou zvoleno převedení grafu do regulárního výrazu ve formě stromu pro svou elegantnost. Pro tento strom je představen algoritmus pro postupné generování nejlépe ohodnocených cest.

3.2 Formální definice požadavků

V tabulce 3.1 jsou sepsané formální požadavky na výsledný program reprezentující tento nástroj. Požadavky jsou podobné jako v práci [11]. Řešení jednotlivých požadavků je popsáno v této kapitole nebo v kapitole 4 týkající se implementace programu.

3.3 Generování cílů pro zadané kritérium pokrytí

První částí návrhu popsánou v této sekci je návrh generování cílů pro zadané kritérium pokrytí. Cílem v tomto kontextu myslíme podcestu některé testovací cesty, která pokud je v testovací cestě obsažená, splní tím některou část kritéria pokrytí. Pro dané kritérium je potřeba vygenerovat takovou množinu cílů, která splní všechny požadavky kritéria pokrytí. Tento přístup je použitelný pro strukturální kritéria pokrytí.

3.3.1 Strukturální kritéria pokrytí

Pro generování cílů strukturálních kritérií pokrytí si zavedeme množinu cílů (podcest) $TARGETS$, které musí být všechny projity pro splnění daného kritéria. Nejjednodušším kritériem je kritérium pokrytí uzlů grafu (**NC**). Pro to stačí do $TARGETS$ vložit všechny cesty o délce 0, tedy podcesty obsahující pouze jeden uzel.
Podobně pro kritérium pokrytí hran (**EC**) a (**EPC**), do $TARGETS$ vložíme všechny hrany, tedy cesty o délce 1, respektive dvojice hran neboli cesty o délce 2.
Složitější úkol nastává pro kritérium primárních cest (**PPC**). Při generování primárních cest se postupně rozšiřují všechny cesty o délce 0 = jednotlivé uzly. Pokud již cesta nelze rozšířit, tedy narazila na koncový uzel nebo nemá žádné následníky, které by již nebyly obsaženy v cestě (s výjimkou prvního uzlu v cestě) a nebo je první a poslední uzel v cestě identický, je taková cesta přidána do množiny potencionálních primárních cest. Z té jsou

ID	Popis	Kategorie	Závislost
REQ.1	Program generuje množinu testovacích vstupů pro zadaný vstupní graf toku řízení.	funkcionální	
REQ.2	Vstupní CFG je v navrženém tvaru ve formátu JSON	funkcionální	
REQ.3	Uživateli programu je umožněno specifikovat kritérium pokrytí.	funkcionální	
REQ.4	Každý výsledný testovací vstup splňuje některý z požadavků zadaného kritéria pokrytí.	funkcionální	REQ.3
REQ.5	Mezi podporované kritéria pokrytí patří NC, EC, EPC, PPC a MCDC	funkcionální	REQ.3
REQ.6	Každý výsledný testovací vstup odpovídá sémanticky splnitelné cestě.	funkcionální	
REQ.7	Uživateli je umožněno zadat způsob ohodnocení podle kterého jsou tvořeny cesty grafem.	funkcionální	
REQ.8	Uživateli je umožněno zadat maximální počet vytvořených cest grafem.	funkcionální	
REQ.9	Cesty grafem jsou tvořeny od nejlepšího ohodnocení zadané uživatelem	funkcionální	REQ.7
REQ.10	Program vygeneruje omezený počet cest zadaný uživatelem	funkcionální	REQ.8
REQ.11	Splnitelnost cesty je ověřena za pomoci externího SMT Solveru Z3	funkcionální	REQ.6

Tabulka 3.1: Požadavky na navrhovaný nástroj. Sloupce zleva: jednoznačný identifikátor, slovní popis požadavku, typ a závislosti na jiné požadavky.

vyřazeny všechny cesty, které jsou podcestou nějaké cesty v této množině. Tím v množině zbudou pouze primární cesty, které vložíme do množiny $TARGETS$.

Tvorba primárních cest na příkladu je naznačena v tabulce 2.2. Na každém řádku jsou postupně rozšiřovány cesty. Všechny cesty které již nelze rozšířit, označené vykřičníkem a nebo červenou barvou, jsou přidány do množiny potencionálních primárních cest. Z té jsou odstraněny ty které jsou podcestou jiné cesty v množině, např. cesta $[4, 2, 6]$ je podcestou $[5, 4, 2, 6]$. Finální množinu $TARGETS$ tak budou tvořit jen všechny červeně označené cesty.

3.3.2 Kritéria pokrytí logických výrazů

Odlišný přístup řešení musíme zvolit při tvorbě cílů pro kritéria pokrytí logických výrazů. Zaměříme se konkrétně na kritérium pokrytí Modified Condition/Decision Coverage - **MCDC**, které je v praxi nejpoužívanější. Již nebude stačit naplnit naplnit množinu $TARGETS$, protože kritérium vychází z predikátů jednotlivých podmínek a určuje hodnotu klauzulí daného predikátu. Množinu $TARGETS$ tedy upravíme na mapování jednotlivých predikátů (podmínky větvení) na hodnoty jednotlivých klauzulí. Kombinace hodnot klauzulí pro MCDC jsou generovány jednoduchým algoritmem: pro daný predikát jsou nejprve vytvořeny všechny kombinace hodnot jejich klauzulí. Vzhledem k tomu, že jednotlivé klauzule můžou nabývat pouze pravdivostních hodnot (true, false), je počet těchto kombinací

roven 2^n, kde n je počet klauzulí v predikátu. Pro každou takovou kombinaci je ověřeno, zda pro nějakou klauzuli, pokud změníme její hodnotu, tak se změní hodnota celého predikátu (tedy klauzule je majoritní). Pokud taková klauzule pro danou kombinaci hodnot existuje, tato kombinace je přidána do cílové množiny kombinací pro daný predikát, pokud již neobsahuje kombinaci se stejnou hodnotou stejné majoritní klauzule. Vzhledem k návrhu zbytku je potřeba generovat tyto kombinace pro jednotlivé hrany, které jsou označeny buď predikátem pro jednu větev a nebo jeho negací pro druhou větev. Výsledné kombinace predikátu jsou tedy rozděleny do těchto 2 větví a je nutné přidat ověření, že vyhodnocení kombinací hodnot pro daný predikát je *true*. Finální algoritmus je naznačený v Algoritmu 1.
Tento algoritmus negeneruje minimální množinu kombinací hodnot klauzulí. To by šlo napravit například zvolením správného pořadí průchodu nebo postupem popsaným v [6], nicméně to není hlavním cílem této práce, proto se spokojíme s tímto řešením.

Algorithm 1: MCDC pokrytí

Input: Predikát p
Output: Množina kombinací ohodnocení p odpovídající MCDC

Nechť n = počet klauzulí v p;
Nechť $comb$ = všechny kombinace možných hodnot pro n;
Nechť $res := []$;
Nechť $req := []$;
Pro $i = 0$ do n:
    Nastav req[i] = [True, False]
Pro každé $c \in comb$:
    Pro $i = 0$ do n:
        Nechť e_1 := vyhodnoť p s hodnotami v c
        Nastav $c[i] := not(c[i])$
        Nechť e_2 := vyhodnoť p s hodnotami v c
        Nastav $c[i] := not(c[i])$
        Pokud $e1 \neq e2 \wedge e1$:
            Odstraň $c[i]$ z $req[i]$
            Pokud $comb$ není v res: přidej $comb$ do res;
Vrať res;

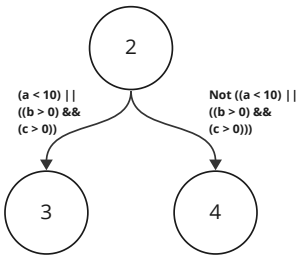
Pomocí takto vygenerovaných kombinací hodnot pro jednotlivé klauzule je potřeba upravit výslednou formuli pro SMT solver tak, že formule bude obsahovat vynucení správné hodnoty všech klauzulí predikátu.
Celý postup generování kombinací hodnot klauzulí tvořící predikát ukážeme na příkladu. Mějme predikát: (a < 10) || ((b > 0) && (c > 0)). Jak můžeme vidět, skládá se ze 3 klauzulí. Pro každou kombinaci hodnot klauzulí je vyhodnocen celý predikát, což je naznačeno pravdivostními hodnotami v tabulce. Hodnoty, které pokud se změní jejich hodnota tak změní hodnotu celého predikátu, tedy majoritní klauzule, jsou označeny zelenou barvou. Pro splnění MCDC pokrytí se každá majoritní klauzule musí vyhodnotit na True a False. Ve sloupci *Targets remaining* je naznačeno, které tyto podmínky ještě zbývají (před průchodem je to (0,1), (0,1), (0,1)). Pak se postupně procházejí řádky tabulky a pokud obsahuje nějakou majoritní klauzuli a zároveň je její hodnota v Targets remaining, tak se celá kombinace přidá do výsledných kombinací. Výsledná sada je znázorněna oranžově. Můžeme si všimnout, že sada není minimální (ta by obsahovala 3+1 kombinací). Kombinace s označením 0 má majoritní první klauzuli s hodnotou False, která je obsažená i v případech

1, 2 a 3, je tedy nadbytečná. Výsledné formule kromě samotného predikátu ještě vynucují právě tyto kombinace hodnot klauzulí:
Pro hranu 2-4:

- 0 - Not((a < 10) || ((b > 0) && (c > 0))) , Not(a<10), Not(b > 0), Not(c>0)
- 1 - Not((a < 10) || ((b > 0) && (c > 0))) , Not(a<10), Not(b > 0), (c>0)
- 2 - Not((a < 10) || ((b > 0) && (c > 0))) , Not(a<10), (b > 0), Not(c>0)
- 3 - Not((a < 10) || ((b > 0) && (c > 0))) , Not(a<10), (b > 0), (c>0)

Pro hranu 2-3:

- 4 - (a < 10) || ((b > 0) && (c > 0)) , (a<10), Not(b > 0), Not(c>0)



Id	a < 10	b > 0	c > 0	a < 10 | | (b > 0 && c > 0)	Edge	Targets remaining
0	False	False	False	False	2->4	(1), (0,1),(0,1),
1	False	False	True	False	2->4	(1), (1),(0,1),
2	False	True	False	False	2->4	(1), (1),(1),
3	False	True	True	True	2->3	(1), (), ()
4	True	False	False	True	2->3	(), (), ()
5	True	False	True	True	2->3	(), (), ()
6	True	True	False	True	2->3	(), (), ()
7	True	True	True	True	2->3	(), (), ()

Obrázek 3.2: Příklad tvorby cílů pro MCDC pokrytí.

3.4 Generování cest vstupním grafem

V této sekci je popsáno, jakým způsobem jsou generovány cesty vstupním grafem s cílem splnění podcest vygenerovaných v předchozím kroku, odpovídající zadanému kritériu pokrytí.

3.4.1 Převod CFG na regulární výraz

Ještě než přejdeme k představení samotného algoritmu generování cest, je ukázáno jakým způsobem je CFG převeden na regulární výraz a taktéž je definována stromová struktura, která je výsledkem tohoto převodu.
Transformace CFG na regulární výraz začíná převedením CFG na konečný automat. To vyžaduje vytvoření přechodové funkce, která je velmi jednoduchá - každá hrana je označena jménem uzlu do kterého daná hrana vede. Když již máme konečný automat, můžeme

použít algoritmus převodu popsaný v části 2.3.4. V rámci převodu jsou rovnou aplikovány i pravidla, které co nejvíce zjednodušují výsledný regulární výraz. Tato pravidla odpovídají Kleeneho algebře nad regulárními výrazy. Výsledkem převodu je stromová struktura, která může obsahovat 4 druhy uzlů:
Listový uzel, který obsahuje hodnotu uzlů v grafu, odpovídající jednomu či více základním blokům (basic blocks) nebo ϵ hodnotu představující prázdnou hodnotu.
Uzel alternace, který obsahuje potomky a a b, odpovídá regulárnímu výrazu $a + b$.
Uzel konkatenace, který obsahuje potomky a a b, odpovídá regulárnímu výrazu ab.
Uzel repetice, který obsahuje potomka a a odpovídá regulárnímu výrazu a^+ (jedno nebo více opakování a - aa^*). Uzly repetice odpovídající a^* jsou převedeny na a^+ pomocí uzlu alternace takto: $\epsilon + a^+$.

3.5 Algoritmus generování cest z regulárního výrazu

V této sekci je popsán algoritmus postupného hledání cest s nejlepším (nejnižším) ohodnocením z regulárního výrazu. Jak již bylo zmíněno, toto ohodnocení může být různé, nejjednodušším je počet basic bloků v cestě.
Protože jde o postupné generování cest, algoritmus si pamatuje jakousi historii již vrácených uzlů. Ta je koncipována jako mapování jednotlivých uzlů na zásobník již navrácených cest. Každý uzel konkatenace si kromě tohoto mapování udržuje ještě 2 fronty, kde jsou uloženy již nalezené cesty, které ještě nebyly navráceny. Formálně jsou zapsané takto:
Mapování uzlů a již nalezených cest z daných uzlů:
$MP : N \to P$; N je množina všech uzlů, $P \subseteq N^*$

Mapování uzlů konkatenace na historii již nalezených kombinací:
$MP_c : N_c \to (Q, S)$; N_c je množina všech uzlů konkatenace; $Q, S \subseteq N^*$

Pro strukturu zásobníku předpokládejme klasické operace:
Přidání položky p na vrchol zásobníku (*push(p)*)
Dotaz na vrchol zásobníku (*top*)
Odebrání položky z vrcholu zásobníku a její vrácení (*pop*)
Dotaz na prázdnost zásobníku (*is_empty*)

Inicializace se provede při prvním vložení do zásobníku. Při generování nových cest je potřeba aktualizovat i příslušné mapování. Funkce, která aktualizuje danou historii přijímá na vstupu uzel, jehož historii aktualizujeme a hodnotu, která je sjednocena s přítomnou množinou. Formálně:
$update\_MP(n, p)$:

$$MP'(m) := \begin{cases} MP(m).push(p), & \text{m = n} \\ MP(m), & \text{otherwise} \end{cases}$$

$MP := MP'$

$update\_MP_c(n, (q, s))$:

$$MP'_c(m) := \begin{cases} (q, s), & \text{m = n} \\ MP_c(m), & \text{otherwise} \end{cases}$$

$MP_c := MP'_c$

Dále si definujeme:

- Min(s) označuje množinu prvků $n \in s$, takových, které mají nejmenší ohodnocení.
- None je speciální prvek, který značí, že žádná nová hodnota nemůže být uzlem vrácena a její ohodnocení je vyšší než jakéhokoliv jiného prvku.

Algoritmus se pro každý typ uzlu ve stromě regulárního výrazu liší, proto je rozdělen na části podle jednotlivých typů. Základem algoritmu je funkce get_next, která vrátí nejlépe ohodnocenou cestu, která ještě nebyla daným uzlem vrácena, tedy není obsažena v mapování nalezených cest z daného uzlu.
Nejjednodušším typem uzlu je listový uzel. Ten při prvním zavolání get_next vrací sám sebe a při každém dalším vrací hodnotu None. Formálně je popsáno v Algoritmu 2 .
Myšlenka pro uzel alternace je taková, že při prvním zavolání si zjistíme nejlépe ohodno-

Algorithm 2: get_next pro listový uzel

Input: Listový uzel n, mapování MP, MP_c
Output: První nejméně ohodnocená cesta p z uzlu n, která není v $MP(n)$, pokud taková cesta existuje, jinak None

1 Pokud $n \in MP(n)$: vrať None, jinak $update\_MP(n, [n])$ a vrať n;

cené cesty z obou potomků (řádek 1) a vrátíme tu lépe ohodnocenou (řádek 5). Při každém dalším zavolání se zjišťuje nová cesta potomka pouze pro toho potomka, jehož cestu jsem posledně vrátil (zajištěno podmínkou na řádcích 2 a 3). Tato cesta se porovná s již některým minulým voláním vrácenou cestou druhého potomka a vrátí se lépe ohodnocená z nich (řádek 5). Algoritmus je napsán v Algoritmu 3.

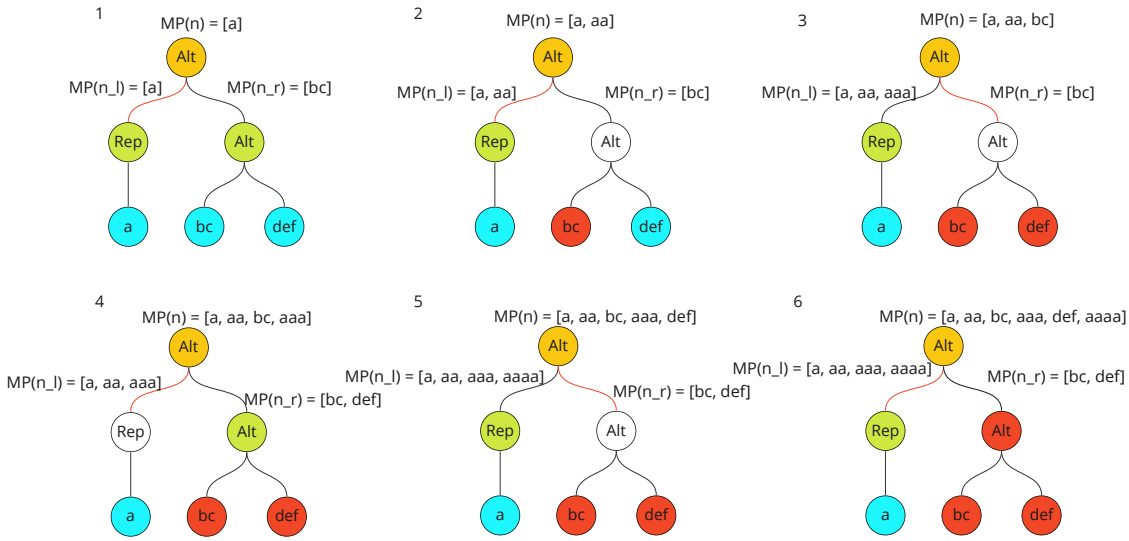
Ukažme si fungování algoritmu na příkladu. Na obrázku 3.3 je vidět jednoduchý strom

Algorithm 3: get_next pro uzel alternace

Input: Uzel alternace n s levým (n_l) a pravým (n_r) potomkem, mapování MP, MP_c
Output: První nejméně ohodnocená cesta p z uzlu n, která není v množině $MP(n)$, pokud taková existuje, jinak None

1 Pokud $MP(n).empty$: Nechť $L_{next} := get\_next(n_l, MP, MP_c)$, $R_{next} := get\_next(n_r, MP, MP_c)$ a pokračuj krokem 5 ;
2 Pokud $MP(n).top = MP(n_l).top$: nechť $L_{next} := get\_next(n_l, MP, MP_c)$, jinak nechť $L_{next} := MP(n_l).top$. Pokud $L_{next} \in MP(n)$: $L_{next} :=$ None;
3 Pokud $MP(n).top = MP(n_r).top$: nechť $R_{next} := get\_next(n_r, MP, MP_c)$, jinak nechť $R_{next} := MP(n_r).top$. Pokud $R_{next} \in MP(n)$: $R_{next} :=$ None;
4 Pokud $R_{next} = L_{next} =$ None: vrať None ;
5 Pokud ohodnocení(R_{next}) < ohodnocení(L_{next}): $update\_MP(n, R_{next})$ a vrať R_{next} , jinak $update\_MP(n, L_{next})$ a vrať L_{next}

regulárního výrazu s kořenovým uzlem alternace. Jeho potomky jsou: uzel repetice s potomkem listového uzlu obsahující řetězec *a* (vlevo) a uzel alternace s potomky listových uzlů obsahující řetězce *bc* a *def*. Jako ohodnocení pro jednoduchost zvolme počet symbolů



Obrázek 3.3: Vizualizace příkladu použití algotitmu pro uzel alternace.

v řetězci. Bez uvážení algoritmů pro jednotlivé uzly by levý potomek měl postupně vracet prvky z nekonečné množiny [*a, aa, aaa, ...*] a pravý potomek vrátí nejdříve *bc*, potom *def*. Na obrázku je ukázán stav uzlu po každém zavolání get_next. Zeleně jsou označny uzly, od kterých se zjišťuje nová cesta
Při prvním zavolání se zavolá get_next pro oba potomky a vrátí se kratší z jejich výsledků - tedy *a* (naznačeno červeným přechodem.
Při druhém zavolání zjišťujeme nový uzel pouze pro levý uzel, protože jeho cesta byla vrácena při prvním zavolání. Porovnává se tedy *aa* a *bc*. Délka je stejná, dle algoritmu je vrácena levá varianta - *aa*.
Při třetím zavolání je znovu zjištěn nová cesta pro levého potomka (protože jeho cesta byla vrácena při minulém volání). Porovná se tedy *aaa* a *bc*. Tentokrát je vrácena pravá varianta - *bc*, protože je kratší.
Čtvrté volání zjišťuje novou cestu od pravého potomka - *def*. Je stejně dlouhá jako posledně nenavrácená levá varianta - *aaa*, tedy levá varianta je vrácena.
Při pátém volání je levým uzlem vrácen řetězec *aaaa*, který je delší než zůstav pravého potomka - *def*
Při dalším volání get_next pro pravý uzel vrátí None a tedy budou vráceny už jen řetězce z levého potomka.

Algoritmus pro uzel konkatenace je oproti algoritmu pro uzel alternace složitější v tom, že pro konkatenaci mohou cesty vygenerované od potomků být použity vícekrát, na rozdíl od alternace, kde je každá cesta vrácena právě jednou. Algoritmus si udržuje 2 fronty pro cesty, které již někdy vygeneroval: jednu výstupní, kde jsou najednou vždy cesty stejného ohodnocení, které mají být již jistě vráceny. Tato fronta je vždy vyprázdněna na začátku algoritmu (řádky 1 a 2). Jakmile jsou všechny cesty v této frontě vráceny, pokračuje se dál.Druhá fronta udržuje cesty, které již vznikly, ale ještě není možné zaručit, že nevzniknou cesty jejichž ohodnocení bude lepší. Po vyprázdnění výstupní fronty se zjistí nové cesty od obou potomků (řádky 3 a 5). Tyto cesty jsou konkatenovány se všemi již navrácenými cestami druhého potomka a všechny takto spojené cesty jsou přidány do druhé fronty (řádky 4 a 6). Tato fronta je potom seřazená podle ohodnocení (řádek 8) a všechny její

prvky s nejlepším ohodnocením jsou přesunuty do výstupní fronty a nakonec je navrácen vrchol této fronty (řádky 9 a 10). Formálně popisuje algoritmus 4

Algorithm 4: get_next pro uzel konkatenace

Input: Uzel konkatenace n s levým (n_l) a pravým (n_r) potomkem, mapování MP, MP_c

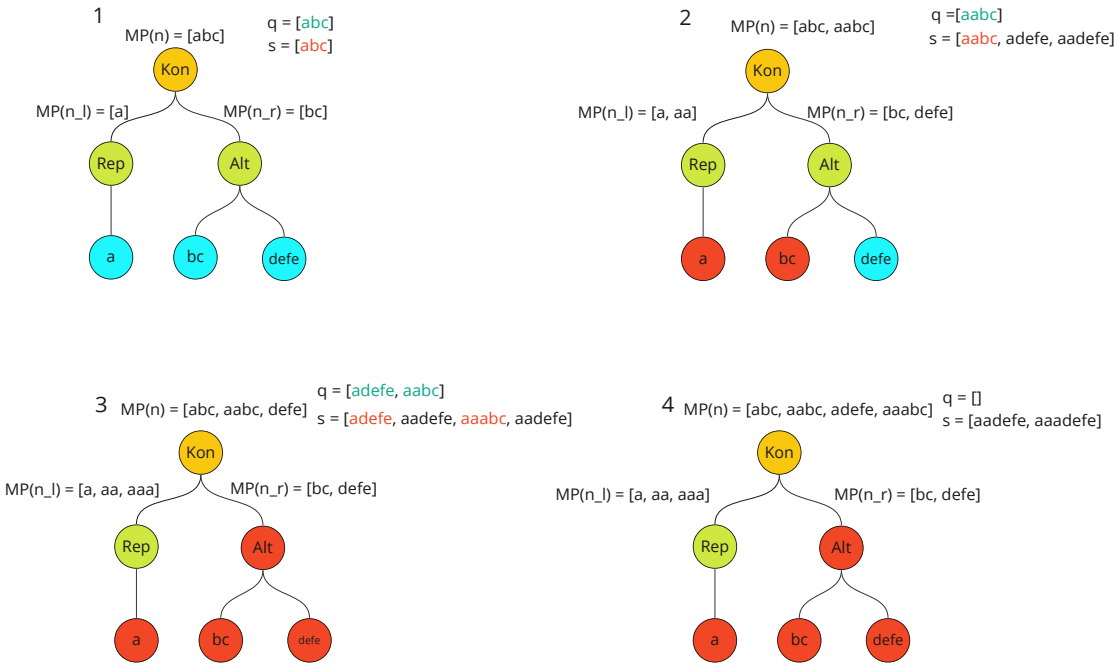
Output: První nejméně ohodnocená cesta p z uzlu n, která není v množině $MP(n)$, pokud taková existuje, jinak None

1 Nechť $q, s := MP_c(n)$;
2 Pokud $q \neq \varnothing$: $top = q.pop$; $update\_MP(n, top)$; $update\_MP_c(n, (q, s))$; vrať top; ;
3 Nechť $L_{next} := get\_next(n_l, MP, MP_c)$ Pokud je $L_{next} = None$, pokračuj bodem 5;
4 Pro každý prvek $r \in MP(n_r)$: nechť $n :=$ concatenate(L_{next}, r); $s.push(n)$;
5 Nechť $R_{next} := get\_next(n_r, MP, MP_c)$ Pokud je $R_{next} = None$, pokračuj bodem 7;
6 Pro každý prvek $l \in MP(n_l)$: nechť $n :=$ concatenate(l, R_{next}); $s.push(n)$;
7 Pokud $s = \varnothing$: vrať None;
8 Seřaď s podle ohodnocení;
9 Pro každý prvek z Min(s) : $top := pop(s)$; $q.push(top)$;
10 $top = q.pop$; $update\_MP(n, top)$, $update\_MP_c(n, (q, s))$, a vrať top;

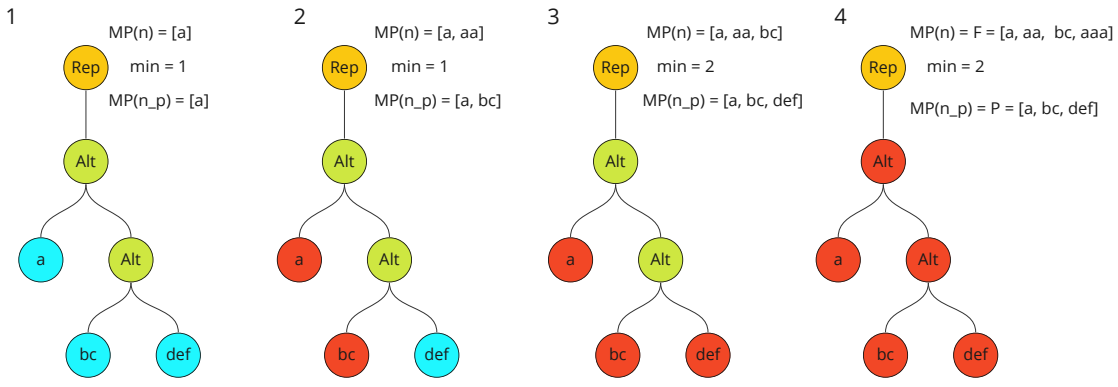
Stejně jako u alternace si ukážeme algoritmus na příkladu, který je znázorněn na obrázku 3.4. Analyzujeme oranžový uzel konkatenace s podobnými potomky jako v předchozím případě. Konkatenace vždy zavolá get_next pro oba uzly a zkonkatenuje je s každým řetězcem z MP druhého potomka a takto spojené řetězce vložíme do fronty s. Z té jsou potom nejkratší řetězce (červeně) přesunuty do q (zeleně) a prvek z q vrácen na výstup. Uzly označené červeně již nemůžou vrátit žádný nový řetězec.
V prvním volání je spojený jediný řetězec *abc*, který je rovnou vrácen.
V druhém kroku se již vytvořilo více řetězců, z nichž některé zůstanou do dalšího kroku.
Ve třetím kroku se do q přesunuly hned 2 řetězce z s, protože mají stejnou délku. Jeden z nich je hned vrácen a při dalším kroku se ihned vrátí druhý z nich bez zjišťování dalších možností.

Myšlenkou generování cesty pro uzel repetice je prozkoumání všech možností kombinace cest, které již vrátil potomek a ještě nebyly navráceny uzlem, pro který je funkce volána (množina F). Počet takových cest je nekonečný, proto se cesty vyhledávají pouze dokud nedosáhnou požadovaného ohodnocení (min). Tyto podmínky jsou ověřeny na řádku 1. Taková cesta je vyhledána pomocí rekurzivní konkatenací cest, které již vrátil potomek. Z takto sestavených cest se zvolí nejlépe ohodnocená (řádky 4 a 5), která je vrácena (řádek 6). Tato rekurzivní funkce je definována v Algoritmu 5.

Pokaždé, když je zavoláno get_next pro uzel repetice, je zjištěná další nejlépe ohodnocená cesta potomka (řádek 1). Potom je zjištěno ohodnocení poslední vrácené cesty (řádek 2) a je zavolána funkce combine, které jsou předány již navrácené cesty potomka, již navrácené cesty aktuálního uzlu a také délka poslední vrácené cesty (řádek 3). Funkce vrátí požadovaný výsledek. Funkce get_next pro uzel repetice je vidět v Algoritmu 6. Příklad pro uzel repetice je na obrázku 3.5. V každém volání get_next je vždy zjištěn další řetězec potomka. V prvním volání potomek vrátil zatím jediný řetězec, min je tedy nastaveno na 1 (chceme řetězec s délkou alespoň 1). Zavolá se funkce combine a ta vrátí řetězec a. Délka



Obrázek 3.4: Vizualizace příkladu použití algoritmu pro uzel konkatenace.



Obrázek 3.5: Vizualizace příkladu použití algoritmu pro uzel repetice.

Algorithm 5: combine

Input: množina cest P, celé číslo min, cesta akt, množina cest F
Output: Nejlépe (minimum) ohodnocená konkatenace možností v P s prefixem akt, jejíž ohodnocení je $\geq min$, která se nenachází v F

1 Pokud ohodnocení(akt) $\geq min$ a zároveň $akt \notin F$: vrať akt;
2 Nechť res := None;
3 Pro každé pos $\in$ P:
4 temp := combine(P, min, Concatenate(akt, pos), F)
5 pokud res = None nebo ohodnocení($temp$) < ohodnocení(res): $res := temp$;
6 Vrať res

Algorithm 6: get_next pro uzel repetice

Input: Uzel repetice n s potomkem n_p, mapování MP, MP_c
Output: První nejméně ohodnocená cesta p z uzlu n, která není v množině $MP(n)$

1 Nechť $p_{next} := get\_next(n_p, MP, MP_c)$;
2 Pokud $MP(n).empty$: nechť $min := 1$, jinak: nechť $min :=$ ohodnocení($MP(n).top$);
3 Nechť $next := combine(MP(n_p), min, \epsilon, MP(n))$;
4 Pokud $next$ = None: vrať $next$;
5 $update\_MP(n, next)$, vrať $next$;

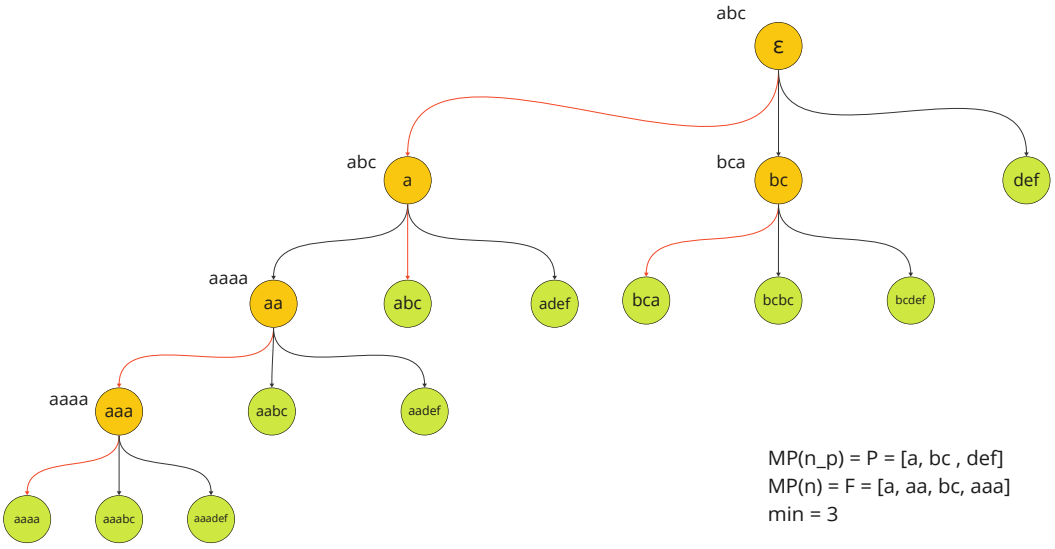
min v druhém volání bude tedy délka a, tedy zase 1. Podobně vše proběhne i pro další dvě volání, min se nastaví na poslední vrácenou hodnotu a combine se postará o nalezení výsledku. Demonstrace rekurzivní funkce combine je naznačena na obrázku 3.6. Uvedený příklad by odpovídal pátému volání z příkladu pro uzel repetice. Parametry jsou dané, hledáme tedy konkatenaci prvků z P, které jsou alespoň 3 znaky dlouhé a nejsou v F. Akt je hodnota v uzlech. Začínáme s prázdným řetězcem. Ten je kratší než 3, proto je rekurzivně volána stejná funkce, tentokrát s akt obsahující konkatenaci akt (ϵ) s každým prvkem z P. Z vrácených řetězců (abc, bca, def, které jsou vedle uzlů) se vybere nejkratší, případně nejlevější pro stejně dlouhé. Která možnost se vybrala je naznačeno červeným přechodem. Oranžové uzly nesplňují podmínky pro vrácení, hledá se tedy dál, dokud se nenajde vhodný řetězec. Výsledná kombinace je nakonec abc.

Pro úplnost prezentovaných algoritmů, je formálně zapsána i funkce *concatenate*, která postupně spojí každý prvek z druhé cesty s první cestou (řádky 3 a 4) - Algoritmus 7.

Algorithm 7: concatenate

Input: Cesty L a R které mají být spojeny
Output: Cesta odpovídající LR

1 Pokud $L = \epsilon$, vrať R;
2 Pokud $R = \epsilon$, vrať L;
3 Pro každý prvek $r \in R$:
4 Přidej r na konec L;
5 Vrať L;



Obrázek 3.6: Vizualizace příkladu pro algoritmus funkce combine.

3.6 Generování SMT formule z dané cesty

Jakmile je vygenerována cesta grafem, pro kterou má být ověřeno, zda je sémanticky dosažitelná, je potřeba vygenerovat SMT formuli, která je předána SMT řešiči pro ověření. Spolu s vygenerovanou cestou známe množinu výrazů v každém uzlu grafu, který odpovídá basic bloku a množinu podmínek, které musí platit pro každou hranu grafu, aby se danou cestou dalo *projít*. V principu je potřeba aby všechny predikáty na hranách cesty, kterou je potřeba ověřit platily najednou. Tedy je potřeba je spojit pomocí logického a. Je však nutné brát v úvahu i veškeré vyhodnocení výrazů v uzlech grafu. Ty mohou upravovat hodnoty proměnných v predikátech. Generování správné formule tedy probíhá postupně - proběhne průchod danou cestou ve kterém se uchovávají hodnoty jednotlivých proměnných. Typicky pro argumenty vstupní funkce tato hodnota nebude konkrétní. V každém uzlu jsou tyto hodnoty aktualizovány dle obsažených výrazů. Predikáty na jednotlivých hranách jsou potom upraveny nahrazením proměnných dle aktuální uchované hodnoty. Takto upravené predikáty jsou postupně přidávány pomocí logického a do výsledné formule.
Pokud je zadáno kritérium pokrytí MCDC, je vybraný predikát ještě upraven, tak aby obsahoval vynucení hodnot odpovídající ověřované variantě. To je provedeno přidáním do predikátu pomocí logického a veškerých klauzulí samostatně - buď neupravených, pokud je potřeba vynutit hodnotu *true* nebo v negaci, pokud je potřeba vynutit hodnotu $false$. V těchto klauzulích musí být také nahrazeny veškeré proměnné aktuální hodnotou.

3.7 Propojení jednotlivých částí nástroje

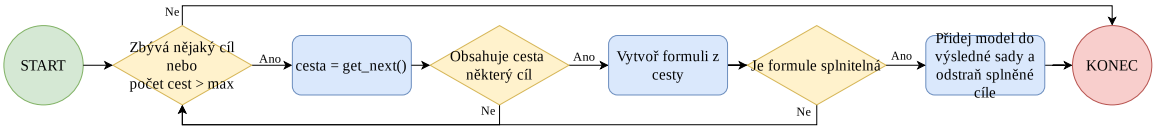
V poslední sekci návrhu je popsáno propojení jednotlivých částí nástroje. Jak již bylo zmíněno, vstupem je graf toku řízení a kritérium pokrytí. Jeden z parametrů je způsob ohodnocení jednotlivých basic blocků v části generování cesty z regulárního výrazu. Dalším nutným parametrem je limit pro počet generovaných testovacích cest. Výstupem nástroje je sada testovacích vstupů do programu. Spuštěním kódu s těmito vstupy jsou projity cesty, které jsou syntakticky i sémanticky splnitelné a dohromady splňují zadané kritérium pokrytí,

pokud jsou požadavky daného kritéria vůbec sémanticky splnitelné a jsou nalezeny v rámci zadaného limitu.
V prvním kroku jsou vygenerovány požadavky na zadané kritérium pokrytí. Pokud se jedná o strukturální kritéria pokrytí, jsou pro podcesty v množině $TARGETS$ vygenerovány formule pro SMT solver, aby se ověřilo, jestli jsou podcesty vůbec splnitelné a zda má cenu se takovým požadavkem dále zabývat. Pokud je požadavek (tedy podcesta) nesplnitelná ($UNSAT$), je vyjmuta z množiny $TARGETS$, aby se nástroj nesnažil najít nastavení vstupních parametrů takových, že splní tento požadavek, když takové nastavení neexistuje.
V dalším kroku je převeden CFG na regulární výraz. Výsledkem tohoto kroku je kořenový uzel regulárního výrazu, který má metodu get_next z algoritmu v sekci 3.5. Tato funkce je postupně volána dokud
a) nejsou splněny všechny požadavky z kritéria pokrytí nebo
b) nebyl dosažen limit počtu vygenerovaných cest.
Pro každou vygenerovanou cestu se zkontroluje, zda daná cesta obsahuje nějakou podcestu v množině $TARGETS$ pro strukturální kritéria pokrytí. Pokud žádnou z podcest neobsahuje, pro tuto cestu nemá smysl ověřovat sémantickou správnost, protože nesplní žádný požadavek ze zadaného kritéria pokrytí. Pokud však alespoň některou podcestu z $TARGETS$ obsahuje, je pro tuto cestu vygenerována formule pro SMT Solveru a ověřena splnitelnost. Pokud formule nemá řešení, celý proces se opakuje s další cestou. Pokud však formule řešení má, je vygenerován model řešení a tento model je přidán do výstupní množiny nástroje. Každá podcesta z množiny $TARGETS$, kterou tato cesta obsahuje je odstraněna z této množiny, protože tento požadavek již je uspokojen.
Pokud je zadáno kritérium pokrytí MCDC, je potřeba změnit proces generování formule z dané cesty a celý postup je tedy lehce odlišný. Stejně jako pro strukturální kritéria jsou postupně generovány cesty, dokud nejsou splněné požadavky nebo není dosaženo limitu. Pro každou cestu je potom vygenerována sada formulí. Tato sada vznikne tak, že pro každou hranu obsaženou v cestě, která je zároveň jako klíč v mapování cílů se vygenerují formule upravené vynucením hodnot klauzulí predikátu dané hrany, které jsou jako hodnoty mapování s klíčem daných hran. Každá formule z této sady je potom ověřena SMT solverem. Pokud existuje řešení, je kombinace hodnot pro daný predikát odstraněna z mapování pro odpovídající hranu. Pokud již nezbývá žádná kombinace, je hrana odstraněna z mapování. Model pro tuto formuli je zase předán na výstup. V tomto případě však mohou vzniknout modely, které jsou duplicitní, proto jsou na výstup vkládány jen unikátní výsledky.

Popsaný postup by se dal nazvat jako prohledávání hrubou silou, jelikož generuje postupně každý syntakticky správný průchod grafem. To znamená, že teoreticky vždy najde řešení, pokud je nastaven dostatečně velký limit počtu generovaných cest a řešení existuje. Návrh je rozšířitelný o určité optimalizace, které urychlí nalezení řešení. V této práci navrhujeme 2 možné přístupy k optimalizaci, které se dají zkombinovat:

- Zamezení generování cest, které obsahují nesplnitelné podcesty.
- Upřednostnit cesty, které mají větší šanci splnit některé kritérium.

Může se stát, že určitá podcesta z vygenerované cesty jen sémanticky nesplnitelná. Tato podcesta se může opakovat i v dalších generovaných cestách. Pokud bychom tedy dokázali detekovat takové podcesty, nemusíme ověřovat cesty obsahující tuto podcestu a nebo ještě lépe takové cesty nemusíme vůbec generovat.
Ne všechny vygenerované cesty splňují některý z požadavků kritéria pokrytí, které chceme



Obrázek 3.7: Zjednodušený diagram aktivit nástroje.

splnit. Proto nemá smysl generovat cesty, které žádný z požadavků nesplní. Příkladem může být program, který na konci obsahuje jednoduché větvení, který rozhoduje o vrácené hodnotě, jako např. funkce na Obrázku 2.1 s CFG na Obrázku 2.2. Pokud v požadavcích zbudou již jen podcesty s uzlem <bb6>, nemá smysl generovat cesty s uzlem <bb7>. Jen takto jednoduchou optimalizací bychom v tomto případě generovali polovinu cest.

Postup je naznačený zjednodušeným diagramem aktivit. Tento diagram je na obrázku 3.7. Diagram ukazuje postup pouze od bodu, kdy je vygenerována struktura regulárního výrazu. Do té chvíle je postup přímočarý.

Kapitola 4

Implementační detaily generátoru

V této kapitole je popsána implementace navrženého nástroje. Nejdříve jsou popsané vybrané technologie, které se v programu používají. Následně je popsán formát vstupu, architektura programu a nakonec také výstupy programu.
Navržený nástroj je implementačně realizován programem nazvaným **Catgenify - Control flow graph Automated Test case GENeration**.

4.1 Technologie a knihovny

Program je implementován v jazyce **Python**. je vysokoúrovňový programovací jazyk, který v roce 1991 navrhl Guido van Rossum. Nabízí dynamickou kontrolu datových typů a podporuje různá programovací paradigmata, včetně objektově orientovaného, imperativního nebo funkcionálního. Python je vyvíjen jako open source projekt, který zdarma nabízí instalační balíky pro většinu běžných platforem (Unix, MS Windows, macOS, Android). Ve většině distribucí systému GNU/Linux je Python součástí základní instalace.
Python byl zvolen kvůli podpoře zmíněných paradigmat a také podpoře různých balíčků, které se v projektu používají:
Argparse je balíček, který umožňuje vytvoření uživatelsky přívětivého rozhraní z příkazové řádky. V programu se definuje, jaké argumenty jsou potřeba a *argparse* tyto argumenty naparsuje z sys.argv. Balíček také automaticky generuje zprávu o způsobu použití a zprávu nápovědy. Pokud nejsou zadány požadované argumenty, modul vygeneruje chybové hlášky.
Graphviz je balíček pro Python, který poskytuje rozhraní k Graphviz, populárnímu open-source softwaru pro vizualizaci grafů. Graphviz uživateli umožňuje vytvářet a zobrazovat diagramy, grafy a sítě programově pomocí Python kódu. Tento balíček umožňuje zobrazit vytvořené reprezentace nástroje, ať už jde o graf toku řízení či strom regulárního výrazu.
JSON je textový formát, který se používá pro výměnu dat. Je snadno čitelný a zapisovatelný pro lidi a zároveň snadno parsovatelný a generovatelný pro počítače. Json je používán jako formát pro vstupní data. Balíček *json* pro Python poskytuje soubor nástrojů pro práci s tímto formátem a nabízí tyto klíčové funkce:

- Kódování a dekódování
- Formátování
- Serializace a deserializace
- Ošetření chyb

- Výkonnost

Pyparsing je balíček v jazyce Python, který poskytuje nástroje pro analýzu a zpracování textových dat. Tento balíček umožňuje uživatelům definovat vlastní gramatiky a provádět analýzu strukturovaných dat v textových souborech, řetězcích nebo jiných zdrojích. PyParsing poskytuje flexibilitu a jednoduchost použití při tvorbě parserů pro různé účely. V nástroji je knihovna používána pro parsování výrazů a podmínek. Tedy pro udržování tabulky symbolů (aktuálních hodnot proměnných) a při tvorbě cílů MCDC kritérií.
Copy modul poskytuje funkce pro vytváření mělkých a hlubokých kopií objektů. Mělká kopie vytváří nový objekt, ale rekurzivně nekopíruje vnořené objekty, zatímco hluboká kopie vytváří nový objekt a rekurzivně kopíruje všechny vnořené objekty uvnitř něj. Tento modul je zvláště užitečný, když potřebujete zdvojit měnitelné objekty, abyste zabránili nechtěným změnám. Zajistí se tím, že změny v původním objektu neovlivní zkopírovaný objekt.
Balíček **Z3** pro Python poskytuje Python API pro Z3 SMT Solver, který je široce používán v různých oblastech, včetně ověřování softwaru, analýzy programů a formálních metod. S balíčkem Z3 pro Python mohou uživatelé využívat možnosti SMT Solveru Z3 přímo v Python kódu. Umožňuje uživatelům definovat podmínky a formule pomocí syntaxe podobné Pythonu a pak použít efektivní Z3 Solvery ke kontrole splnitelnosti a poskytnutí řešení zadaných rovnic jednoduchými funkcemi *check* a *solve*.

4.2 Argumenty spuštění nástroje

Nástroj je možné spustit s několika přepínači. V této sekci je vysvětlen jejich význam:

- **-h,** –**help** Vypíše zprávu nápovědy a ukončí se.
- –**cfg, -c** cesta k JSON souboru vstupního CFG.
- –**coverage, -cc** Zvolené kritérium pokrytí
- –**print, -p** Zobrazit kontrolní výpisy
- –**mode, -m** Způsob ohodnocení uzlů pro generování cest.
- –**limit, -l** Limit počtu generovaných cest

Parametr help je automaticky vygenerovaný pomocí balíčku argparse.
Parametr cfg je jediný povinný parametr. Udává cestu k json souboru, který obsahuje graf toku řízení.
Coverage je zadané kritérium pokrytí. Zadaný parametr musí odpovídat jedné ze zkratek podporovaných kritérií: NC, EC, EPC, PPC nebo MCDC. Pokud není zadáno, je použito PPC.
Parametr print určuje, zda budou vypsány kontrolní výpisy na standardní výstup. 0 (pokud není zadáno) znamená žádný výpis, jinak se vypisují. Parametr mode určuje jakým způsobem jsou ohodnoceny uzly v grafu. To ovlivňuje generování cest. Momentálně jsou podporovány 2 způsoby, první z nich je výchozí:

- 1- cesta je ohodnocena počtem uzlů (basic blocků), tedy každý uzel má ohodnocení 1.
- 2- uzel je ohodnocena počtem výrazů v odpovídajícím basic bloku.

Posledním volitelným argumentem je limit počtu generovaných cest. Výchozí hodnota je nastavena na 100 generovaných cest.

4.3 Formát vstupního CFG

Jak již bylo zmíněno, vstupní formát CFG je zadán v textovém formátu JSON. Vstupní formát obsahuje všechny informace z grafu toku řízení - množinu uzlů, množinu hran ve formě potomků každého uzlu, počáteční a koncové uzly. Dále potom seznam vstupních parametrů a proměnných včetně jejich typu. Každý uzel je také označen unikátním jménem označujícím basic blok, obsahuje seznam výrazů, které se v basic bloku vykonávají a pokud obsahuje více potomků (tedy jde o větvení), obsahuje podmínku za které se jde to které větve. Podmínky a výrazy na uzlech a hranách jsou nutné pro správné vytvoření SMT formule. Vytvoření takového JSON vstupu z CFG vytvořeného např. GCC překladačem pro C/C++ kód nebo jiným překladačem pro jiný jazyk by mělo být jednoduché, není však již součástí nástroje.
Ukázka vstupního formátu můžeme vidět na Obrázku 1. Pod klíčem *parameters* jsou vstupní parametry funkce a proměnné s jejich typem v položce *types*. Typ musí odpovídat jednomu z typů, které podporuje Z3 Solver. V *nodes_list* je seznam uzlů. Položky *inital_node* a *terminal_nodes* jsou vypsány počáteční a koncové uzly. V položce nodes jsou potom veškeré informace k uzlům: jméno *name*, koncové uzly hran *succs*, výrazy *actions*. Pokud succs obsahuje více položek, je přítomná i podmínka *condition*.

4.4 Architektura programu

Architektura programu je navržena tak, aby bylo možné jednoduše nahradit jednotlivé části nástroje. Program je proto rozdělen na jednotlivé moduly. Tyto moduly mají navržené rozhraní, které zaručuje, že spolu budou kompatibilní. Takový přístup přináší výhodu lehké rozšířitelnosti programu. Všechny moduly jsou nadefinovány v *modules.py* jako abstraktní třídy.
Modul na nejvyšší úrovni se jmenuje **Module_App** a implementuje metodu *solve*, která volá metody ostatních modulů a řídí předávání informací mezi nimi. Modul je v programu realizován třídou App, které jsou předány naparsované vstupní parametry. Podle zvoleného kritéria pokrytí (v tuto chvíli se rozlišuje strukturální nebo MCDC) se definují další moduly, které se na sebe správným způsobem napojí.
Další moduly poté odpovídají jednotlivým částem návrhu nástroje tak jak jsou popsané v kapitole 3:

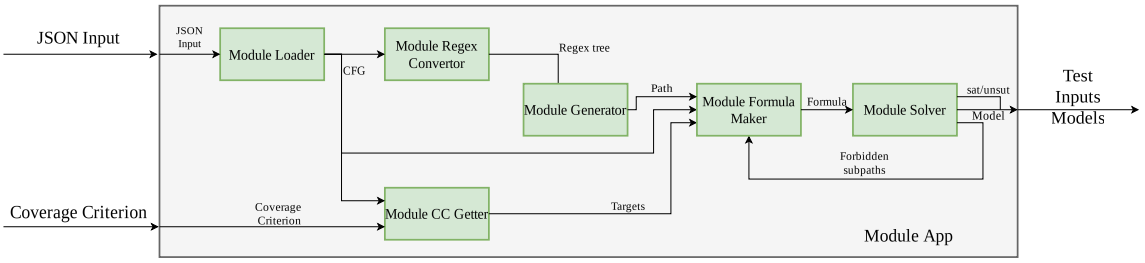
- **Module_Loader** je modul, který se stará o načítání vstupního JSON souboru do instance třídy CFG. Implementuje metodu load_from_json, která na vstupu přijímá cestu k vstupnímu souboru a vrací zmíněnou instanci CFG reprezentující graf toku řízení.

- **Module CC Getter** se stará o vytvoření cílů kritéria pokrytí. Využívá postupů popsaných v sekci 3.3. Rozhraním tohoto modulu je funkce get targets, která na vstupu přijímá načtenou instanci grafu toku řízení a jedno z podporovaných kritérií pokrytí. Výstupem modulu je potom struktura vytvořených cílů pokrytí $TARGETS$. Tato struktura se liší pro různé typy zadaného kritéria: pro strukturální se jedná o pole podcest (pole uzlů), zatímco pro kritéria pokrytí logických výrazů jde o mapování (slovník) hrany na kombinace booleovských hodnot klauzulí v podmínce dané hrany.

- **Module Regex Convertor** má za úkol převod grafu toku řízení na regulární výraz. Popis převodu je v sekci 3.4.1. Vstupem modulu je načtený CFG, ze kterého se vytvoří

```
{   "parameters": [
        {
            "type": "Int",
            "id": "a"
        },
        {
            "type": "Int",
            "id": "b"
        }],
    "nodes_list":["bb2","bb3","bb4","bb5"],
    "initial_node":"bb2",
    "terminal_nodes": ["bb5"],
    "nodes": [
        {
            "name": "bb2",
            "succs": [
                {
                    "name": "bb3",
                    "condition": "(a < b)"
                },
                {
                    "name": "bb4",
                    "condition": "Not((a < b))"
                }
            ]
        },
        {
            "name": "bb3",
            "succs": [
                {
                    "name": "bb5"
                }
            ],
            "actions":[
                "a=a+1"
            ]
        },
        {
            "name": "bb4",
            "succs": [
                {
                    "name": "bb5"
                }
            ]
        },
        {   "name": "bb5"}
    ]
}
```

Listing 1: Ukázka jednoduchého vstupního souboru ve formátu JSON.



Obrázek 4.1: Propojení jednotlivých modulů nástroje a jaká data si předávají.

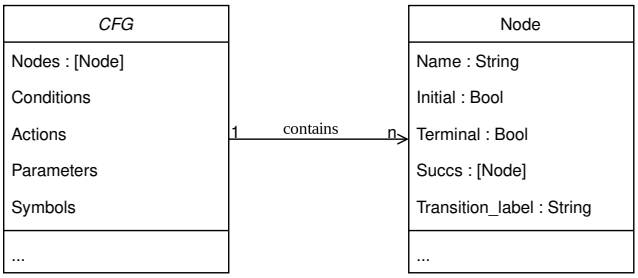
konečný regulární výraz. Výstupem modulu je regulární výraz ve formě stromu. Strom je ve formě kořenového uzlu, který je jedna z tříd konkatenace, repetice, alternace či případně listového uzlu (Literal).

- **Module Generator** převádí strom regulárního výrazu vytvořený v předchozím kroku na třídy, které mají implementované algoritmy generování cest popsané v sekci 3.5. Tento převod také zahrnuje převod uzlů repetice na ty s alespoň jedním opakováním ($a^* \rightarrow (\varepsilon + a^+)$). Vstupem modulu je kromě zmíněného regulárního výrazu také instance CFG a to kvůli možnosti ohodnocení uzlů podle výrazů v basic blocích, které odpovídají jednotlivým uzlům. Výstupem je další stromová struktura, kde každý uzel má implementovanou metodu get_next, která vrací další nejlépe ohodnocenou cestu grafem.

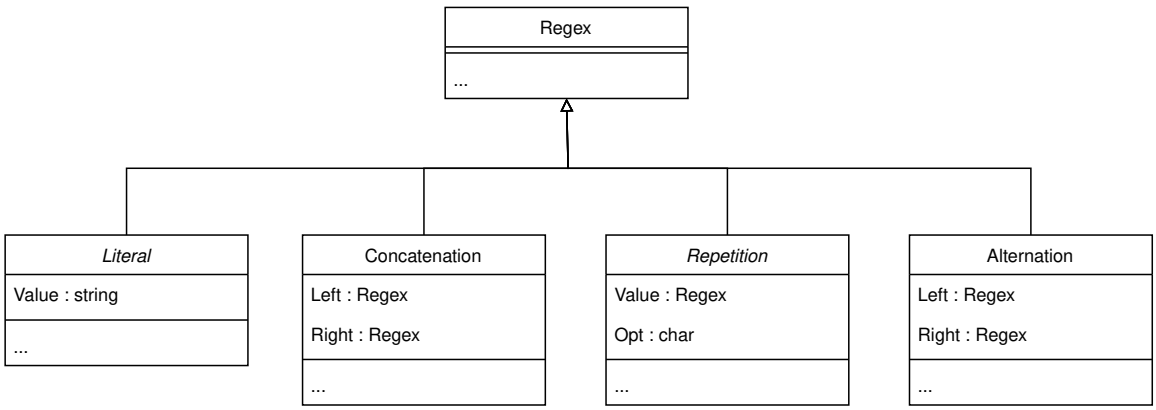
- **Module Formula Maker** je zodpovědný za vytváření formulí pro SMT Solver z dané cesty. Proces generování je stručně popsaný v sekci 3.6. Modul musí mít implementovanou metodu solve_formula, do které vstupuje instance CFG a cesta, která není nic jiného než posloupnost uzlů. Výstupní formule musí být v takovém formátu, aby ji vybraný solver přijímal. To klade omezení na tvar výrazů a podmínek ve vstupním souboru, které nemusí odpovídat formě výrazů zdrojového jazyka. Stejně typ proměnných musí odpovídat podporovaným typů, kterými jsou Int, Float, Bool a Bitvector. V práci jsou naimplementovány 2 třídy tohoto modulu. Jedna je použita pokud je zadané strukturální kritérium pokrytí, druhý pro kritérium MCDC. Druhý zmíněný generuje formule pro každý cíl každé hrany obsaženou v $TARGETS$ a zároveň ve vygenerované cestě. Výstupem je tedy buď jedna či více formulí. V našem případě jsou v textovém formátu.

- Posledním modulem je **Module Solver**. Konkrétní třída tohoto modulu implementuje funkci *solve_formula*, která má za úkol předat formuli, která je vstupem funkce, SMT Solveru, který ověří její splnitelnost. Výstupem je potom výsledek tohoto ověření - *sat* nebo *unsat* spolu s modelem řešení pokud je formule splnitelná. Pokud splnitelná není, funkce vrací také soubor nesplnitelných podcest, které se použijí pro omezení počtu ověřovaných cest - ty které obsahují tyto podcesty již nemá cenu ověřovat. V momentálním stavu je tento soubor vždy prázdný, ale je to jedna z možností rozšíření nástroje.

Na obrázku 4.1 je naznačeno, jaká data si moduly předávají a je tím také naznačen interface vnitřních modulů a také App modulu.



Obrázek 4.2: Diagram tříd CFG a Node.



Obrázek 4.3: Diagram tříd regulárního výrazu.

4.5 Implementační detaily

V této sekci je popsáno rozdělení souborů do jednotlivých souborů a tříd, které využívají jednotlivé moduly. Logika nástroje je rozdělena do souborů podle funkčních částí, tak jak jsou popsány v návrhu programu.
Zásadní část kódu je implementována v souboru *graph.py*, kde se nachází metody a třídy týkající se grafu toku řízení, který používají téměř všechny moduly. Graf toku řízení je, jak již bylo zmíněno reprezentován třídou `CFG`, která obsahuje instance třídy `Node`, reprezentující jednotlivé uzly. Atributy těchto tříd a jejich vztah je ukázán diagramem tříd na obrázku 4.2. Třída `CFG` obsahuje seznam uzlů, které obsahují všechny hrany v atributu *Succs* a příznaky počátečního a koncového uzlu. Tím je určena struktura grafu. *Conditions* je mapování hran na podmínku, pro průchod danou hranou. *Actions* je mapování uzlů na výrazy, které se dějí v uzlu. *Parameters* jsou vstupní parametry funkce, které se inicializují v Z3 SMT Solveru. Atribut *Symbols* je aktuální ohodnocení proměnných konkrétního průchodu.

Třídy pro uchování regulárního výrazu jsou v souboru *regex.py*. Třídy jsou velmi jednoduché a obsahují pouze potomky uzlu daného stromu. Jediný uzel repetice obsahuje navíc znak pro odlišení regulárních výrazů a^* od a^+. Diagram tříd pro tyto třídy je na obrázku 4.3.

Stejnou stromovou strukturu představují třídy v souboru *tree.py*. Tyto třídy implementují funkci get_next, která generuje další nejlépe ohodnocenou cestu regulárního výrazu. Třídy tedy navíc obsahují atributy, které si uchovávají informaci o aktuálním stavu, například historii již vrácených uzlů.

V souboru *parse.py* jsou implementovány funkce, které se zabývají parsováním výrazů a podmínek. Ty využívají balíček pyparsing. Výrazy i podmínky jsou parsovány za účelem nahrazení proměnných aktuálními hodnotami. Pro kritérium pokrytí se podmínky také parsují do stromové struktury logického výrazu.

Stromová struktura regulárního výrazu je tvořena jednoduchými třídami pro logické a, logické nebo a negaci, které obsahují pouze potomka/potomky. Tyto třídy se nachází v souboru *predicate.py* spolu s funkcemi, které tvoří cíle pro MCDC kritéria. Většina pomocných funkcí je rekurzivní průchod stromem logického výrazu.

4.6 Testovací sada

Pro ověření správné funkcionality programu je implementována automatizovaná sada testů. Jsou vytvořeny hned 2 sady testů. Jedna obsahuje jednotkové testy, které ověřují funkcionalitu jednotlivých funkcí. Vytvořeny jsou jednotkové testy pro funkce, které odpovídají algoritmům v návrhu. Tyto testy využívají framework pytest, který umožňuje jejich spuštění pomocí jediného příkazu *pytest*.
Druhá sada testů obsahuje demonstrační testy, které ověřují funkcionalitu programu jako celku. Vstupy testů vychází z reálných funkcí, které byly ručně převedeny na grafy toku řízení. Poté je nad nimi spuštěn vytvořený program pro různá kritéria pokrytí. Výsledky jsou potom uloženy do souborů a jejich hodnoty jsou ověřeny předpokládanými podmínkami platícími pro kritérium NC. Všechny tyto testy jsou rozděleny do složek podle struktur, které kódy obsahují. Složka if testuje podmíky if-then, složka obsahuje testy pro kódy s while cykly a složka comb obsahuje testy s kombinací obou konstrukcí. V každé složce najdeme kromě vstupu a výstupu programu také soubor *expected.txt*, který obsahuje podmínky pro splnění testu. Pro spuštění demonstračních testů postačí spuštění skriptu *run_tests.py*.

Kapitola 5

Demonstrační příklad

V této kapitole je ukázána funkcionalita popsaného nástroje na demonstračním příkladu. Příklad má za cíl ukázat hlavně fungování představených algoritmů na netriviálním příkladu implementovaným programem. Jsou ukázány vstupy a výstupy jednotlivých modulů, tak jak jsou poskládány za sebou.
Příklad byl vytvořen tak, aby obsahoval:

- Zanořený cyklus
- Více vstupních proměnných
- If-else větev takovou, že pro průchod jedné větve je potřeba několikrát projít cykly
- Alespoň jedna podmínka je tvořena více klauzulemi

S těmito podmínkami byl vytvořen příklad 5.1. Z takového kódu je pomocí překladače GCC vytvořen CFG, který je přepsán do vstupního JSONu.

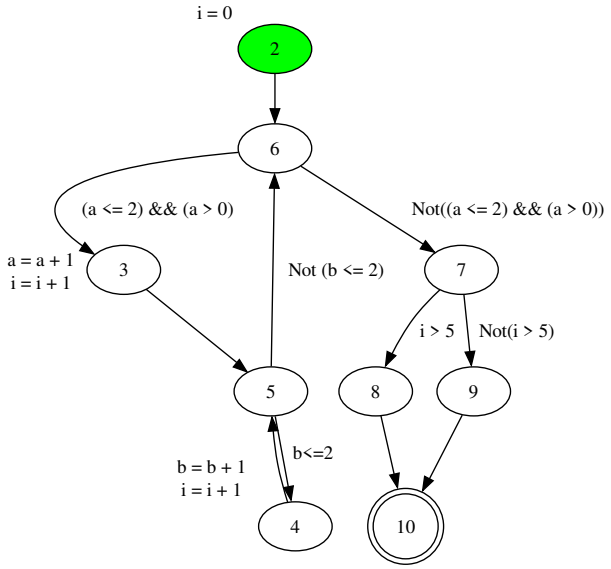
Prvním modulem programu je modul, který se stará o načtení vstupního JSON souboru do vnitřní reprezentace - třídy CFG, která obsahuje potřebné atributy pro graf toku řízení. Správnost načtení můžeme zkontrolovat pomocí funkce, která automaticky vykreslí graf včetně podmínek a příkazů. Tento vygenerovaný graf vidíme na obrázku 5.2. Jakmile je načtený graf, můžeme již vygenerovat příslušné cíle kritérií pokrytí. Které kritérium je použito je zadáno uživatelem, nicméně pro demonstraci ukazujeme všechna kritéria v tabulce 5.1 . Dále bude činnost nástroje demonstrována na kritériích PPC a MCDC.

Kritérium pokrytí	Vygenerované cesty
NC	[2], [3], [4], [5], [6], [7], [8], [9], [10]
EC	[2, 6],[6, 3],[6, 7],[3, 5],[5, 6],[5, 4],[4, 5],[7, 8],[7, 9],[8, 10],[9, 10]
EPC	[2, 6, 3], [2, 6, 7], [6, 3, 5], [6, 7, 8], [6, 7, 9], [3, 5, 6], [3, 5, 4], [5, 6, 3], [5, 6, 7], [5, 4, 5], [4, 5, 6], [4, 5, 4], [7, 8, 10], [7, 9, 10]
PPC	[4, 5, 6, 7, 9, 10], [4, 5, 6, 7, 8, 10], [3, 5, 6, 7, 9, 10], [3, 5, 6, 7, 8, 10], [2, 6, 7, 9, 10], [2, 6, 7, 8, 10], [2, 6, 3, 5, 4], [4, 5, 6, 3], [5, 6, 3, 5], [3, 5, 6, 3], [6, 3, 5, 6], [4, 5, 4], [5, 4, 5]
MCDC	(6, 3): [[True, True]], (6, 7): [[False, True], [True, False]], (5, 6): [[False]], (5, 4): [[True]], (7, 8): [[True]], (7, 9): [[False]]

Tabulka 5.1: Vygenerované cíle pro zadané kritérium pokrytí.

```
int foo(int a, int b){
    int i = 0;
    while((a<3) && (a>0)){
        a++;
        i++;
        while(b<3){
            b++;
            i++;
        }
    }
    if(i>5){
        return 1;
    } else {
        return 0;
    }
}
```

Obrázek 5.1: Kód ze kterého byl vytvořen vstupní CFG pro demonstrační příklad.



Obrázek 5.2: Automaticky vykreslená načtená struktura grafu toku řízení.

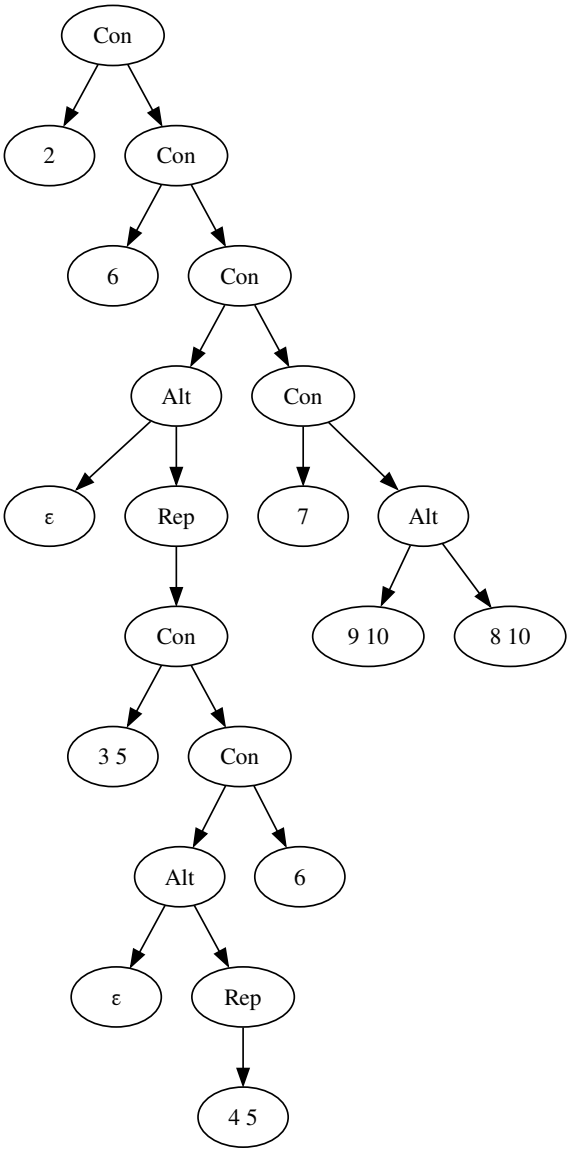
Dalším krokem je převod CFG na regulární výraz. Ten má na starost modul RegexConvertor. Výsledkem převodu je regulární výraz 2 6(3 5(4 5)*6)*7((9 10)+(8 10)). Vidíme 2 repetice, které odpovídají vnořeným cyklům a na konci vidíme alternaci, která odpovídá if-else konstrukci. Nicméně pro další postup je zásadní stromová struktura, ze které je tento výraz sestaven. Tento strom je vykreslen pomocí knihovny graphviz. Tento automaticky vykreslený strom je zobrazen na obrázku 5.3.

5.1 Příklad postupu pro strukturální kritéria pokrytí

Převod na regulární výraz je společná část pro strukturální kritéria i kritérium MCDC. V této sekci ukážeme proces generování testovacích vstupů pro strukturální kritérium pokrytí, konkrétně PPC.

Strom regulárního výrazu je předán dalšímu modulu, který se již stará o postupné generování cest z kořenového uzlu. Ještě před samotným generováním cest se pro strukturální kritéria ověří, zda jsou všechny cíle kritérií pokrytí splnitelné. Jsou z nich tedy vytvořeny formule pro SMT Solver a jsou ověřeny. Pro PPC je nesplnitelná jediná cesta a to cesta [2, 6, 7, 8, 10]. Vygenerovaná formule je ve tvaru $Not(And((a <= 2), (a > 0))), (0 > 5)$. Je jasné, že podmínka $(0 > 5)$ nemůže být splněna, takže po předání formule SMT Solveru, který ji označí jako nesplnitelnou (unsat) je tato cesta z cílů vyřazena.

Jakmile je ověřeno, že jsou všechny cíle splnitelné, začnou se postupně generovat cesty podle zvoleného ohodnocení cest. Pro jednoduchost zvolíme jako ohodnocení počet basic bloků v cestě. Pro všechny takto vygenerované cesty je zkontrolováno, zda splňuje některý z cílů a pokud ano, je vytvořena formule, předána SAT Solveru a pokud je splnitelná, jsou



Obrázek 5.3: Vizualizace stromové struktury regulárního výrazu.

všechny cíle vyřazeny. Celý proces je znázorněn ukázkou kontrolních výpisů, které jsou upraveny pro kompaktnost výpisu:

```
Generated next path: [2 6 7 9 10]
formula: Not(And((a <= 2 ), (a > 0 ))), Not((0 > 5 ))
Formula solved, result is: [a = 3]
Targets satisfied: [2 6 7 9 10 ] Remaining targets:11:
[4 5 6 7 9 10], [4 5 6 7 8 10], [3 5 6 7 9 10], [3 5 6 7 8 10],
[2 6 3 5 4], [4 5 6 3], [5 6 3 5], [3 5 6 3], [6 3 5 6], [4 5 4], [5 4 5]

Generated next path: [2 6 7 8 10] No criterium to satisfy

Generated next path: [2 6 3 5 6 7 9 10]
formula: And((a <= 2 ), (a > 0 )), Not((b <= 2 )), Not(And(((a + 1 )<= 2 ),
((a + 1 )> 0 ))), Not(((0 + 1 )> 5 ))
Formula solved, result is: [a = 2, b = 3]
Targets satisfied: [3 5 6 7 9 10 ] [6 3 5 6 ] Remaining targets: 9:
[4 5 6 7 9 10], [4 5 6 7 8 10], [3 5 6 7 8 10],[2 6 3 5 4],
[4 5 6 3], [5 6 3 5], [3 5 6 3],[4 5 4], [5 4 5]

Generated next path: [2 6 3 5 6 7 8 10]
formula: And((a <= 2 ), (a > 0 )), Not((b <= 2 )),
Not(And(((a + 1 )<= 2 ), ((a + 1 )> 0 ))), ((0 + 1 )> 5 )
Formula unsat

Generated next path: [2 6 3 5 4 5 6 7 9 10]
formula: And((a <= 2 ), (a > 0 )), (b <= 2 ),
Not(((b + 1 )<= 2 )), Not(And(((a + 1 )<= 2 ), ((a + 1 )> 0 ))),
Not((((0 + 1 )+ 1 )> 5 ))
Formula solved, result is: [a = 2, b = 2]
Targets satisfied: [4 5 6 7 9 10 ] [2 6 3 5 4 ] [5 4 5 ] Remaining targets: 6:
[4 5 6 7 8 10], [3 5 6 7 8 10], [4 5 6 3], [5 6 3 5], [3 5 6 3], [4 5 4]

Generated next path: [2 6 3 5 4 5 6 7 8 10]
formula: And((a <= 2 ), (a > 0 )), (b <= 2 ),
Not(((b + 1 )<= 2 )), Not(And(((a + 1 )<= 2 ), ((a + 1 )> 0 ))),
(((0 + 1 )+ 1 )> 5 )
Formula unsat

Generated next path: [2 6 3 5 6 3 5 6 7 9 10]
formula: And((a <= 2 ), (a > 0 )), Not((b <= 2 )), And(((a + 1 )<= 2 ),
((a + 1 )> 0 )), Not((b <= 2 )), Not(And((((a + 1 )+ 1 )<= 2 ),
(((a + 1 )+ 1 )> 0 ))), Not((((0 + 1 )+ 1 )> 5 ))
Formula solved, result is: [b = 3, a = 1]
Targets satisfied: [5 6 3 5 ] [3 5 6 3 ] Remaining targets: 4:
[4 5 6 7 8 10], [3 5 6 7 8 10], [4 5 6 3], [4 5 4]
.
.
.
```

```
Generated next path: [2 6 3 5 4 5 4 5 4 5 4 5 6 3 5 6 7 8 10]
formula:...
Formula solved, result is: [a = 1, b = -1]
Targets satisfied: [3 5 6 7 8 10 ] All criterions satisfied!
```

Ve výpisech vidíme, že délka vygenerovaných cest se postupně zvětšuje. Ze všech nalezených modelů řešení je vytvořena finální sada testovacích vstupů. Ta pro tento příklad vypadá takto:

1. **a = 3**
2. **a = 2, b = 3**
3. **a = 2, b = 2**
4. **b = 3, a = 1**
5. **a = 2, b = 1**
6. **a = 1, b = 2**
7. **a = 2, b = -2**
8. **a = 1, b = -1**

V případě první testovacího vstupu vidíme nastavenou hodnotu pouze jednoho parametru. To značí, že hodnota druhého parametru nemá na průchod programem žádný vliv. Pokud se koukneme na kód, je jasné, že při nastavení $a = 3$ se přeskočí while cyklus a tedy parametr b se vůbec nepoužije.

5.2 Příklad postupu pro kritéria pokrytí logických výrazů

V této sekci je ukázán rozdílný postup při zadání kritéria MCDC. Generování cest zůstává i pro tuto variantu stejné jako u strukturálních kritérií, hlavní rozdíl je v generování formulí. Pro každou cestu není generována pouze jedna formule, ale pro každou hranu, která se nachází v cestě se vygeneruje formule pro každý cíl dané hrany. Ukážeme si upravené kontrolní výpisy pro první vygenerovanou cestu:

```
Generated next path: [2 6 7 9 10]
formula: Not(And((a <= 2 ), (a > 0 ))), Not(a <= 2 ), (a > 0 ), Not((0 > 3 ))
Formula is sat, model is: [a = 3]
Satisfied: ('6', '7'): [False, True]
Remaining:
{('6', '3'): [[True, True]],
('6', '7'): [[True, False]],
('5', '6'): [[False]],
('5', '4'): [[True]],
('7', '8'): [[True]],
('7', '9'): [[False]]}

formula: Not(And((a <= 2 ), (a > 0 ))), (a <= 2 ), Not(a > 0 ), Not((0 > 3 ))
Formula is sat, model is: [a = 0]
Satisfied: ('6', '7'): [True, False]
Remaining:
{('6', '3'): [[True, True]],
('5', '6'): [[False]],
('5', '4'): [[True]],
('7', '8'): [[True]],
```

```
('7', '9'): [[False]]}

formula: Not(And((a <= 2 ), (a > 0 ))), Not((0 > 3 )), Not(0 > 3 )
Formula is sat, model is: [a = 3]
Satisfied: ('7', '9'): [False]
Remaining:
{('6', '3'): [[True, True]],
('5', '6'): [[False]],
('5', '4'): [[True]],
('7', '8'): [[True]]}
```

Můžeme vidět vygenerovanou cestu, která obsahuje 2 hrany, které se nachází v cílech: [6, 7] a [7, 9]. Druhá zmíněná hrana obsahuje jen jednu klauzuli, obsahuje tedy jen jeden cíl, jehož splnění je automatické projití danou hranou. Zajímavější je první hrana, která obsahuje 2 klauzule: $(a \leq 2)$ a $(a > 0)$. V první vygenerované formuli vidíme snahu o splnění cíle (False, True), tedy $(a \leq 2)$ musí být vyhodnocena jako False a $(a > 0)$ musí být vyhodnoceno jako True. To znamená, že musí platit $Not(a <= 2)$ a $(a > 0)$. Přesně tyto klauzule jsou obsaženy v dané formuli. Touto jedinou vygenerovanou cestou jsme našli řešení pro oba cíle pro tuto hranu a v dalších cestách již nebude zkoumána.

Kapitola 6

Závěr

V této diplomové práci jsem navrhnul a implementoval nástroj, který má za cíl generování testovacích vstupů pro funkce kritického software ve formě grafu toku řízení dle zadaného kritéria pokrytí. Tyto testovací vstupy jsou nalezeny SMT Solverem pomocí cesty převedené na SMT formuli, představují tedy syntakticky i sémanticky splnitelné cesty grafem.
V práci je nejprve popsán teoretický základ, poté návrh nástroje, kde jsou představeny algoritmy nezávislé na implementačním jazyce, které vyjadřují myšlenku celé práce. V další kapitole je již popsána samotná implementace cílového nástroje. Ta je demonstrována v poslední kapitole, kde jsou ukázány možnosti nástroje na praktickém příkladě.
Hlavním přínos práce je v nové metodě generování cest grafem toku řízení na základě převodu na regulární výraz, který umožňuje zvolit způsob ohodnocení cest, podle kterého se nové cesty generují. To umožňuje prohledávání od nejjednodušších řešení, například s nejmenším počtem výrazů v cestě. Spojením této metody s ostatními částmi nástroje - generování cílů zvoleného kritéria pokrytí, převodem cesty na SMT formuli a následném ověření sémantické splnitelnosti pomocí SMT formule, jde o univerzální prototyp nástroje, který není vázán na konkrétní programovací jazyk či překladač a je tedy možné použití v mnoha dalších projektech.
Nástroj zároveň nabízí možnosti pro další práci. Zmíním možnost optimalizace hledání vhodné cesty pro zadané kritérium pokrytí, které by zmenšilo množství prohledávaných cest či omezení množství ověřovaných cest SMT Solverem pomocí zavedení zakázaných podcest. Tyto možnosti jsou zmíněny v kapitole návrhu. Dalším nabízejícím rozšířením je množství podporovaných datových typů a podporovaných kritérií pokrytí. Posledním námětem pro další práci je zaintegrování do komplexního nástroje, který automaticky vytvoří a otestuje zadaný kód.

Literatura

[1] Ammann, P. a Offutt, J. *Introduction to Software Testing*. Cambridge University Press, 2008. ISBN 9780521880381. Dostupné z: https://books.google.cz/books?id=BMbaAAAAMAAJ.

[2] Brzozowski, J. A. Derivatives of Regular Expressions. *J. ACM*. New York, NY, USA: Association for Computing Machinery. oct 1964, sv. 11, č. 4, s. 481–494. DOI: 10.1145/321239.321249. ISSN 0004-5411. Dostupné z: https://doi.org/10.1145/321239.321249.

[3] Cadar, C., Dunbar, D. a Engler, D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. USA: USENIX Association, 2008, s. 209–224. OSDI'08.

[4] Clarke, E., Kroening, D. a Lerda, F. A Tool for Checking ANSI-C Programs. In: Jensen, K. a Podelski, A., ed. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*. Springer, 2004, sv. 2988, s. 168–176. Lecture Notes in Computer Science. ISBN 3-540-21299-X.

[5] Duc Anh, N., Ngoc Hung, P. a Nguyen, V. H. A method for automated unit testing of C programs. In:. Září 2016, s. 17–22. DOI: 10.1109/NICS.2016.7725644.

[6] Hayhurst, K. J. *A practical tutorial on modified condition/decision coverage*. DIANE Publishing, 2001.

[7] Huong, T. N., Kha, D. M., Tran, H.-V. a Hung, P. N. Generate Test Data from C/C++ Source Code using Weighted CFG and Boundary Values. In: *2020 12th International Conference on Knowledge and Systems Engineering (KSE)*. 2020, s. 97–102. DOI: 10.1109/KSE50997.2020.9287629.

[8] Kassab, M. Testing Practices of Software in Safety Critical Systems: Industrial Survey. In:. Leden 2018, s. 359–367. DOI: 10.5220/0006797003590367.

[9] Katebi, H., Sakallah, K. A. a Marques Silva, J. P. Empirical Study of the Anatomy of Modern Sat Solvers. In: Sakallah, K. A. a Simon, L., ed. *Theory and Applications of Satisfiability Testing - SAT 2011*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, s. 343–356. ISBN 978-3-642-21581-0.

[10] Knight, J. Safety critical systems: challenges and directions. In: *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. 2002, s. 547–550.

[11] Kraut, D. *Generování modelů pro testy ze zdrojových kódů [online].* 2019. Diplomová práce. Brno: Vysoké učení technické v Brně. Fakulta informačních technologií.

[12] Lochau, M., Peldszus, S., Kowal, M. a Schaefer, I. Model-Based Testing. In: Bernardo, M., Damiani, F., Hähnle, R., Johnsen, E. B. a Schaefer, I., ed. *Formal Methods for Executable Software Models: 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014, Bertinoro, Italy, June 16-20, 2014, Advanced Lectures.* Cham: Springer International Publishing, 2014, s. 310–342. DOI: 10.1007/978-3-319-07317-0_8. ISBN 978-3-319-07317-0. Dostupné z: https://doi.org/10.1007/978-3-319-07317-0_8.

[13] Moura, L. de a Bjørner, N. Z3: An Efficient SMT Solver. In: Ramakrishnan, C. R. a Rehof, J., ed. *Tools and Algorithms for the Construction and Analysis of Systems.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, s. 337–340. ISBN 978-3-540-78800-3.

[14] Seyster, J., Dixit, K., Huang, X., Grosu, R., Havelund, K. et al. Aspect-Oriented Instrumentation with GCC. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I. et al., ed. *Runtime Verification.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, s. 405–420. ISBN 978-3-642-16612-9.

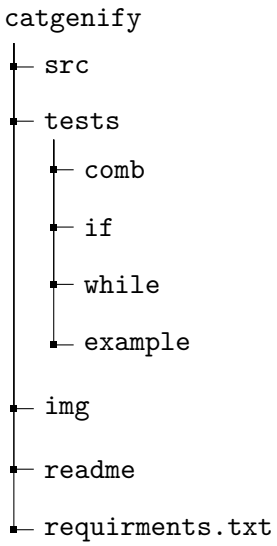
[15] Williams, N., Marre, B., Mouy, P. a Roger, M. PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis. In:. Duben 2005, sv. 3463, s. 281–292. DOI: 10.1007/11408901_21. ISBN 978-3-540-25723-3.

[16] Zhang, S., Saff, D., Bu, Y. a Ernst, M. D. Combined static and dynamic automated test generation. In: *ISSTA 2011, Proceedings of the 2011 International Symposium on Software Testing and Analysis.* Toronto, Canada: [b.n.], červen 2011, s. 353–363.

Příloha A

Obsah odevzdaného paměťového média

Odevzdané paměťové médium obsahuje tyto adresáře a soubory:



- Složka src obsahuje zdrojové soubory
- Složka tests obsahuje demonstrační testy rozdělené podle konstrukcí, které testují.
- Složka img obsahuje vygenerované obrázky z kódu obsažené v práci.
- readme je soubor s krátkým popisem argumentů nástroje
- requirments.txt je soubor s balíčky, které projekt používá

Příloha B

Instalace a spuštění nástroje

Tato příloha obsahuje příkazy k instalaci a spuštění nástroje.
Zdrojové soubory lze stáhnout z GitLab repozitáře https://pajda.fit.vutbr.cz/testos/catgenify

Pro překlad lze využít příkaz `pyinstaller src/catgenify.py`
Tímto příkazem se vytvoří cílový soubor ve složce dist/catgenify pro aktuální operační systém.

Demonstrační příklad je potom spuštěn pro systém Linux z adresáře projektu pomocí příkazu:
`dist/catgenify/catgenify -c tests/comb/test_comb_2/test_comb_2.json`

Příloha C

Uživatelský tutoriál

Tato příloha obsahuje jednoduchý tutoriál provázející uživatele od stažení po spuštění nástroje v systému Linux.

Zdrojové kódy lze stáhnout z přiloženého média nebo z GitLab repozitáře pomocí příkazu:
`git pull https://pajda.fit.vutbr.cz/testos/catgenify.git`

Po úspěšném stažení je nutné nainstalovat veškeré balíčky, které nástroj používá pomocí příkazu:
`pip3 install -r requirements.txt`
Pokud se balíčky nainstalovaly správně, mělo by se na konci výpisu objevit:

```
Installing collected packages: z3-solver, argparse, pyparsing, pluggy,
packaging, iniconfig, graphviz, pytest
Successfully installed argparse-1.4.0 graphviz-0.20.3 iniconfig-2.0.0
packaging-24.0 pluggy-1.5.0 pyparsing-3.1.2 pytest-8.2.0 z3-solver-4.13.0.0
```

Pokud jsou již balíčky nainstalované, objeví se výpis podobný následujícímu:

```
Requirement already satisfied: z3-solver in ...
(from -r requirements.txt (line 1)) (4.12.2.0)
Requirement already satisfied: graphviz in ...
(from -r requirements.txt (line 2)) (0.20.1)
Requirement already satisfied: networkx in ...
(from -r requirements.txt (line 3)) (3.2.1)
```

Poté už by měly být spustitelné testy a celý program. Pro spuštění unit testů přejdeme do složky *src* pomocí `cd src` a spustíme testy příkazem:
`pytest`
Pokud vše proběhne správně, vypíše se následující výpis:

```
==================== test session starts====================
platform linux -- Python 3.10.6, pytest-8.1.1, pluggy-1.4.0
rootdir: ...
plugins: anyio-3.6.2
collected 11 items

test_cc_getter.py .... [ 36%]
test_tree.py ....... [100%]

==================== 11 passed in 0.48s ======================
```

Demonstrační testy se spouští opět ze složky src pomocí příkazu
`python3 run_tests.py`
Při správném postupu se vypíše:

```
Starting test test_comb_1
Test succesfull
.
.
Starting test test_while_1
Starting test test_while_2
Test succesfull
All 9 tests succesfull!
```

Zároveň se ve složkách jednotlivých testů vytvoří výstupní soubory s koncovkou .out pro každé kritérium, jejichž správnost lze ověřit manuálně.

Nakonec ukážeme spuštění jednoduchého příkladu. Vybereme CFG z jednoho demonstračního testu nacházejícího se ve složce `tests/if/test_if_1` a spustíme ho s kontrolními výpisy pro kritérium NC příkazem:

```
python3 src/catgenify.py -c tests/if/test_if_1/test_if_1.json -cc NC -p 1
```

Při správném spuštění se vypíše:

```
bb2((bb4 bb5)+(bb3 bb5))

Generated next path: [bb2 bb4 bb5] formula: Not((a < b ))
Formula solved, result is: [b = 0, a = 0]
Targets satisfied: [bb2] [bb4] [bb5] Remaining targets: 1:
[bb3]

Generated next path: [bb2 bb3 bb5] formula: (a < b )
Formula solved, result is: [b = 1, a = 0]
Targets satisfied: [bb3] All criterions satisfied!

Total number of targets for this criterion: 4
From which feasible: 4
Satisfied with result test cases: 4
Which is 100.0% of all criterions
and 100.0% of all feasible criterions
[[b = 0, a = 0], [b = 1, a = 0]]
```