VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

APLIKACE PRO NAHRÁVÁNÍ A PŘEHRÁVÁNÍ DISKRÉTNÍHO STAVOVÉHO GRAFU

DISCRETE GRAPH STATE APPLICATION DRIVER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **JOSEF MELKUS**
AUTHOR

VEDOUCÍ PRÁCE **doc. RNDr. PAVEL SMRŽ, Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2021/2022

Zadání bakalářské práce

Student: **Melkus Josef**
Program: Informační technologie
Název: **Aplikace pro nahrávání a přehrávání diskrétního stavového grafu**
Discrete Graph State Application Driver
Kategorie: Umělá inteligence

Zadání:

1. Nastudujte problematiku stavových diagramů a diskrétních grafů.
2. Definujte stavový graf a jednotlivé přechodové funkce, relevantní pro úlohy autonomního řízení.
3. Implementujte způsob nahrávání stavů. Nahrávky budou sloužit jako etalon.
4. Implementujte způsob přehrávání nahraných stavů, sloužící pro podrobný debugging a automatické vyhodnocování správnosti programů.
5. Vytvořte stručný plakát prezentující práci, její cíle a výsledky

Literatura:

- dle doporučení vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- funkční prototyp řešení

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

Vedoucí práce: **Smrž Pavel, doc. RNDr., Ph.D.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 11. května 2022
Datum schválení: 1. listopadu 2021

Abstrakt

Cílem této práce je vytvořit balíček nástrojů pro integrační testování C++ systémů na základě stavových diagramů. Všechny komponenty balíčku jsou od sebe oddělené a umožňují snadnou výměnu za jinou. Součástí tohoto balíčku je knihovna definující vytváření a přechody stavového diagramu. Dalším nástrojem je rozhraní pracující nad touto knihovnou, které zaznamenává přechody stavů za běhu systému. V rámci testů se uložené přechody porovnávají s etalonem. Etalon je běh systému, který pro dané vstupy považujeme za správný. Další částí je program na samotné porovnávání a skript sloužící k automatizaci testů. Vytvořený systém byl otestován v rámci partnerské společnosti.

Abstract

The aim of this work is to create a framework for integration testing of C++ systems based on their state diagrams. One part of the framework is a library defining the creation and transitions of a state diagram. Another part is an interface working on this library, that records state transitions in a run of a system. Records are then compared with an etalon. Etalon is the run of the system, that we consider as correct for a given set of inputs. The last part is an application for comparing state transitions and a script for test automation. The created system was tested within a partner company.

Klíčová slova

stavový diagram, stavový graf, integrační testování, C++, debugging

Keywords

state, diagram, graph, integration testing, C++, debugging

Citace

MELKUS, Josef. *Aplikace pro nahrávání a přehrávání diskrétního stavového grafu.* Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. RNDr. Pavel Smrž, Ph.D.

Aplikace pro nahrávání a přehrávání diskrétního stavového grafu

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doc. RNDr. Pavla Smrže, Ph.D. Další informace mi poskytla partnerská firma BringAuto. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Josef Melkus
10. května 2022

Poděkování

Děkuji svému vedoucímu bakalářské práce doc. RNDr. Pavlu Smržovi, Ph.D. za trpělivost, ochotu a odbornou pomoc při vytváření této práce. Dále děkuji partnerské společnosti BringAuto za poskytnutí nástrojů, na kterém mohl být výsledek této práce testován. Také bych rád poděkoval panu prof. RNDr. Alexandru Medunovi, CSc. za doporučení odborné literatury.

Obsah

Kapitola 1

Úvod

V dnešní době vzniká velké množství velmi komplexních systémů, kterými jsou například síťové, či jakékoliv vícevláknové aplikace, nebo stále se rozvíjející autonomní systémy, u kterých nelze predikovat jejich přesné chování. Tyto systémy jsou složité na otestování, které je důležité k ověření správnosti jejich chování, nebo také k odhalení bezpečnostních rizik. Dále v případě nalezení chyby je často velmi složité najít její příčinu.

Systém vytvořený v této bakalářské práci slouží k testování právě těchto systémů, které testuje na základě jejich stavového diagramu. Tento systém je složen z několika nástrojů, pomocí kterých se snaží odhalit nesprávné přechody stavů a slouží jako nástroj pro automatické vyhodnocování správnosti programů, ale také jako nástroj pro podrobný debugging, čehož dosahuje pomocí zaznamenávání přechodů stavů stavového diagramu testovaného systému. Tento balíček nástrojů je univerzální a snadno nasaditelný do výše popsaných systémů, nijak zásadně neovlivňuje výkon a napomáhá jak při objevování chyb, tak při hledání jejich příčin.

Tato práce vzniká ve spolupráci s partnerskou společností BringAuto, která vyvíjí autonomní doručovací roboty. Autonomní systémy jsou velmi složité a náchylné na chyby, řídí však reálnou věc, která může při malé softwarové chybě napáchat velké hmotné škody. Z toho důvodu je důležité tyto systémy podrobně otestovat. Testuje se mnoha různými způsoby, tato práce se zaměřuje konkrétně na testování podle stavového diagramu.

V kapitole 2 je popsána obecná teorie grafů a některé algoritmy pro prohledávání grafů. V této kapitole je popsaný algoritmus pro hledání cyklů, který je poté využíván pro vyhodnocování běhu systému. Následující kapitola 3 se zabývá konečnými automaty a s nimi spojenými stavovými diagramy, které jsou základní jednotkou pro testování vytvořeným systémem. Kapitola číslo 4 popisuje různé strategie a metody testování softwaru. Jsou zde také popsány různé typy testování pro různé fáze vývoje a také testovací nástroje.

Kapitola 5 popisuje návrh výsledného balíčku nástrojů. V této kapitole je popsáno rozdělení na jednotlivé komponenty a jejich funkce. Dále je popsán způsob zaznamenávání informací o přechodech stavů a funkce etalonu při jejich vyhodnocování.

V kapitole 6 je popsána implementace zásadních funkcí pro správné fungování každé z komponent. Mimo to je zde popsán způsob distribuce výsledného balíčku. Postup při vývoji tohoto balíčku nástrojů a podrobnosti o jeho testování v partnerské společnosti je popsán v kapitole 7.

Kapitola 2

Teorie grafů

Teorie grafů je obor diskrétní matematiky, jehož základním nástrojem je graf, definovaný jako množina vrcholů a hran. Grafy jsou využívány například pro matematické důkazy, ale také v různých oborech informatiky, ve společenských nebo i přírodních vědách [26].

Informace v této kapitole, není-li řečeno jinak, jsou převzaty z knihy J. Demela [7].

2.1 Graf

Graf se skládá z `vrcholů` a `hran`. Může být `orientovaný` nebo `neorientovaný`. V orientovaném grafu jsou všechny hrany orientované, to znamená, že u nich rozlišujeme `počáteční vrchol` a `koncový vrchol`. Neorientovaný graf je složen pouze z neorientovaných hran, které chápeme jako symetrické spojení dvou vrcholů. Hrany grafu mohou být ohodnocené, nebo neohodnocené. `Ohodnocení hrany`, nebo také v literatuře používané `ohodnocení přechodu`, může vyjadřovat jakoukoliv veličinu, například čas nebo vzdálenost.

Orientovaný graf

Orientovaný graf je trojice $G = (V, E, \epsilon)$, kde V je neprázdná konečná množina vrcholů, E je konečná množina orientovaných hran a ϵ je zobrazení $\epsilon = E \to V^2$, které nazýváme `vztahem incidence`. Toto zobrazení přiřazuje každé hraně $e \in E$ uspořádanou dvojici vrcholů (x, y), ve které je vrchol x nazýván `počátečním vrcholem hrany` a označován $P_V(e)$ a vrchol y je nazýván `koncovým vrcholem hrany` a je označován $K_V(e)$. Hrana e může být popsána tak, že vede z vrcholu x do vrcholu y, nebo že spojuje vrcholy x a y. Dále o hraně e platí, že je `incidentní` s vrcholy x a y. Vrcholy x, y jsou `incidentní` s hranou e a jsou souhrnně označované jako `krajní vrcholy hrany` e. Jestliže platí $P_V(e) = K_V(e)$, je hrana e nazývána `smyčkou`. Více hran může mít stejné počáteční a koncové vrcholy, tedy pro dvě různé hrany e_1, e_2 platí $P_V(e_1) = P_V(e_2)$ a $K_V(e_1) = K_V(e_2)$. Tento vztah je možné také zapsat jako $\epsilon(e_1) = \epsilon(e_2)$. Tyto hrany jsou označovány jako `rovnoběžné` nebo také `násobné`. Množina hran grafu může být také prázdná. Vrchol, který není `incidentní` s žádnou hranou je nazýván `izolovaným vrcholem`.

Všechny odkazy na graf v této práci jsou vztahovány právě na orientovaný graf.

Prostý graf a multigraf

V prostém grafu je násobnost každé hrany rovna nejvýše jedné. Multigraf je graf, ve kterém násobnosti hran mohou být i větší než jedna.

2.1.1 Sledy a odvozené pojmy

Sled je posloupnost vrcholů a hran $v_0, e_1, v_1, e_2, ..., e_k, v_k$, ve které pro každou hranu e_i platí $P_V(e_i) = v_i - 1$ a $K_V(e_i) = v_i$. Ve sledu na sebe vrcholy a hrany navazují a všechny hrany jsou orientované vpřed ve směru sledu. Vrchol v_0 je označován `počátečním vrcholem sledu` a vrchol v_k je označován `koncovým vrcholem sledu`. Vrchol v_y je z vrcholu v_x `dostupný`, právě když existuje sled, ve kterém je počátečním vrcholem v_x a koncovým vrcholem v_y.

V obecných sledech se mohou vrcholy i hrany opakovat. `Triviální sled` je sled, který obsahuje jediný vrchol a žádnou hranu. Každý sled je jednoznačně určen posloupností hran. V prostém grafu může být sled určen posloupností vrcholů.

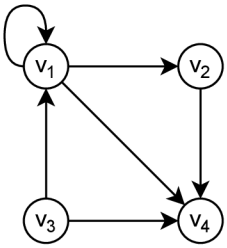
Sled, ve kterém se neopakuje žádný vrchol ani hrana, je nazýván `cestou`. Sled ve kterém se neopakuje žádná hrana se nazývá `tahem`.

Uzavřené sledy

Sled, který má alespoň jednu hranu a počáteční vrchol je stejný jako koncový vrchol, tedy platí $v_0 = v_k$, je označován `uzavřeným sledem`. Uzavřený sled, ve kterém je každý vrchol i hrana maximálně jednou (mimo v_0 a v_k) je označován `uzavřenou cestou`, pro kterou se používá název `cyklus`.

2.1.2 Způsoby reprezentace grafu

Ačkoliv je pro člověka velmi často graf nejvíce srozumitelný, pokud je reprezentován graficky, nemusí tomu tak být v případě velkých grafů. Zásadní problém porozumění graficky značeného grafu však nastává, když je graf zadáván do počítače. V této podsekci jsou popsány různé způsoby reprezentace grafu, od značení grafickou podobou, po způsoby, které jsou lépe strojově zpracovatelné.



Obrázek 2.1: Příklad grafické podoby grafu.

Grafická podoba

Výhodou grafu je možnost ho zobrazovat graficky. To umožňuje člověku lepší porozumění situace modelované grafem. Vrcholy grafu jsou obvykle značeny kroužky a hrany šipkami spojujícími příslušnou dvojici počátečního a koncového vrcholu. Šipka vede z počátečního do koncového vrcholu. Stejný graf lze zakreslit mnoha způsoby, pro přehlednost je preferován ten nákres, kde se hrany grafu nejméně kříží.

Příklad grafické podoby grafu je na obrázku 2.1.

Seznam vrcholů a hran

Množina vrcholů grafu je popsána seznamem prvků a množina hran seznamem dvojic tvořených počátečním a koncovým vrcholem hrany. V případě grafu s pojmenovanými hranami je jméno hrany přidáno k dvojici.

Výhodou tohoto způsobu reprezentace grafu je jeho univerzálnost a relativní rychlost. V praxi je často používán, ač se často liší implementací. Graf z obrázku 2.1 reprezentován seznamem vrcholů a hran:

$$\begin{array}{ll} \text{Vrcholy:} & v_1,\ v_2,\ v_3,\ v_4 \\ \text{Hrany:} & (v_1, v_1),\ (v_1, v_2), \\ & (v_1, v_4),\ (v_2, v_4), \\ & (v_3, v_1),\ (v_3, v_4) \end{array}$$

Seznam sousednosti

Jedná se o úspornější variantu předchozího způsobu. Množina vrcholů je stále popsána seznamem prvků, ale každému vrcholu v náleží seznam jeho sousedních vrcholů. V případě grafu s pojmenovanými hranami seznam obsahuje dvojice se jménem hrany a koncovým vrcholem. Graf z obrázku 2.1 reprezentován seznamem sousednosti:

$$\begin{array}{ll} v_1\text{:} & v_1,\ v_2,\ v_4 \\ v_2\text{:} & v_4 \\ v_3\text{:} & v_1,\ v_4 \\ v_4\text{:} & \emptyset \end{array}$$

Matice sousednosti

Tento způsob je matematicky elegantní a velmi rychlý pro zpracování počítačem. Nevýhodou tohoto způsobu je větší paměťová náročnost, která je určena počtem vrcholů a je stejná jak pro grafy s vysokým počtem hran, tak pro grafy s počtem hran nízkým.

K reprezentaci grafu maticí sousednosti je potřeba očíslovat vrcholy. Číslování může být libovolné, ale neměnné. Očíslované vrcholy slouží jako indexy v matici. Neexistující hrana mezi vrcholy je značená nulou. Existující hrana může být značená:

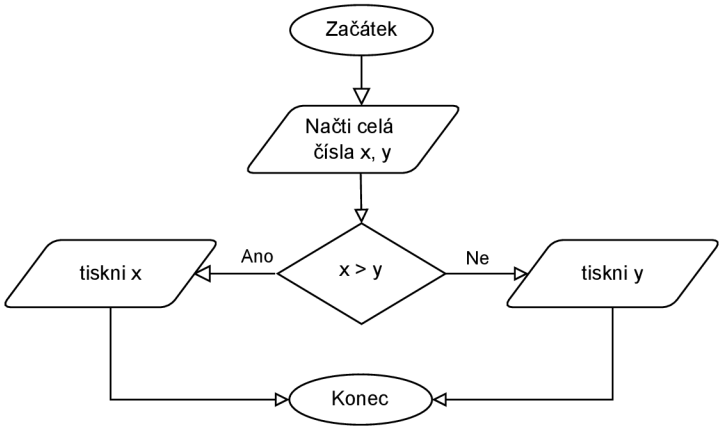
1. ohodnocením hrany v případě grafu s ohodnocenými hranami,
2. jménem hrany v případě grafu s pojmenovanými hranami,
3. počtem hran mezi dvěma vrcholy pro multigrafy
4. jedničkou v ostatních případech.

Matice sousednosti M_G grafu na obrázku 2.1:

$$M_G = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

2.1.3 Využití grafu

Grafem lze popsat a zjednodušit mnoho reálných situací. Příkladem může být silniční síť, kde hrana je silnice, která navíc může být ohodnocená délkou nebo jiným parametrem, a vrchol křižovatka. Tento graf poté může být využit například k hledání nejkratší cesty. V informační technologii se graf používá například ve formě vývojového diagramu, který slouží pro popis programu pomocí sledu instrukcí viz obrázek 2.2. Graf také může sloužit jako určitá abstrakce fungování systému, více o tomto tématu je napsáno v kapitole 3.



Obrázek 2.2: Vývojový diagram popisující aplikaci, která vytiskne větší ze dvou zadaných čísel.

2.2 Algoritmy pro hledání cyklů

V této sekci jsou popsány algoritmy pro hledání cyklů v orientovaných grafech. Seznam cyklů v grafu může být využit například jako součást systémů pro optimalizaci počítačových programů, nebo pro dělání obecné analýzy počítačových programů, jako je například odhad výpočetního času [25]. V této práci bylo potřeba najít cykly pro agregaci stavů v aplikaci pro vyhodnocování přechodů stavů běhu systému 5.2.2.

Definice

V následující sekci se vyskytuje značení:

- V Počet vrcholů v grafu
- E Počet hran v grafu
- C Počet cyklů v grafu
- N Velikost instance úlohy

Dva cykly, které jsou vzájemně cyklickou permutací, jsou považovány za jeden cyklus [21].

2.2.1 Složitost algoritmů

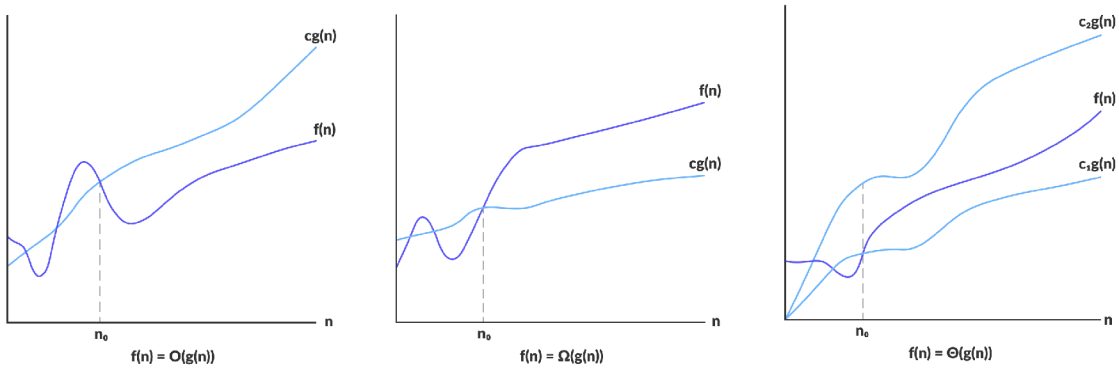
Podsekce je založená na informacích z knihy J. Demela [7] a studijní opory J. Honzíka [9].

Teorie složitosti se zabývá závislostí času řešení úlohy a paměťové náročnosti na velikosti instance úlohy. `Typ úlohy` obecně určuje, jaká data a v jakých strukturách jsou pro úlohu zadána a co má být výsledkem. `Instance úlohy` je konkrétní případ typu úlohy s konkretními daty. Časová složitost se počítá pro algoritmus, který řeší daný typ úlohy.

Pro popis časové složitosti algoritmu je nevhodné používat dobu výpočtu, vzhledem k různým rychlostem procesorů. Používání počtu instrukcí, které je potřeba pro výpočet vykonat, by bylo vhodnější, ale i zde záleží na typu procesoru, protože různé procesory mají odlišné soubory instrukcí. Z tohoto důvodu se používá `asymptotické vyjádření složitosti` algoritmu. Ta vyjadřuje porovnání algoritmu s jistou funkcí pro N blížící se nekonečnu. Porovnání může mít podobu tří různých složitostí:

- O Omikron - vyjadřuje horní hranici chování
- Ω Omega - vyjadřuje dolní hranici chování
- Θ Theta - vyjadřuje třídu chování

Grafické znázornění těchto asymptotických vyjádření je na obrázku 2.3.



Obrázek 2.3: Přehled grafického znázornění asymptotických vyjádření složitosti. Převzato z [2].

Pro praktické aplikace se algoritmy nejčastěji charakterizují složitostí O, která vyjadřuje nejhorší případ. Jeho definice je:

Nechť f, g jsou dvě nezáporné funkce reálné proměnné. Existují-li reálné konstanty k, m takové, že pro všechna $x > m$ platí $g(x) \leq k \cdot f(x)$, pak $g = O(f)$.

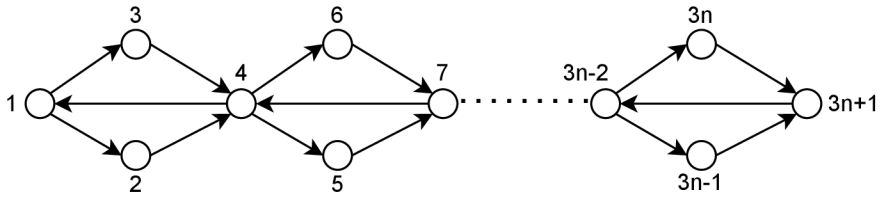
Vyjádření složitosti O se nezískává měřením skutečných výpočtů na počítači, ale teoretickým rozborem činnosti algoritmu. Výhodou tohoto vyjádření je jeho zaměření na velké instance úloh. U malých instancí je zpravidla časová složitost zanedbatelná. Další výhodou je nezávislost na typu počítače, programovacím jazyce, překladači a dalších proměnných faktorech.

2.2.2 Tiernanův algoritmus

Podkapitola je založená na článku J. C. Tiernana [23]. Algoritmus na hledání cyklů v orientovaném grafu podle Tiernana využívá řešení hrubou silou. Při tomto způsobu řešení se systematicky prochází celý prostor možných řešení problému a zjišťuje se, zda nalezené řešení

vyhovuje zadaným podmínkám. V případě algoritmu pro hledání cyklu v grafu to znamená, že se prohledávají všechny vrcholy pro najití všech cest, u kterých se poté vyhodnocuje, zda se jedná o cyklus. Výhodou tohoto způsobu řešení je jednoduchost implementace a nalezení všech existujících řešení problému, nebo důkaz o neexistenci řešení problému. Nevýhodou jsou vysoké výpočetní nároky, které mohou růst exponenciálně k velikosti problému [19].

Tiernan tento algoritmus označuje za nejefektivnější, protože každý cyklus najde právě jednou. Nejhorší případ grafu pro Tiernanův algoritmus je na obrázku 2.4, kde platí exponenciální závislost vůči počtů cyklů a také velikosti grafu [22].

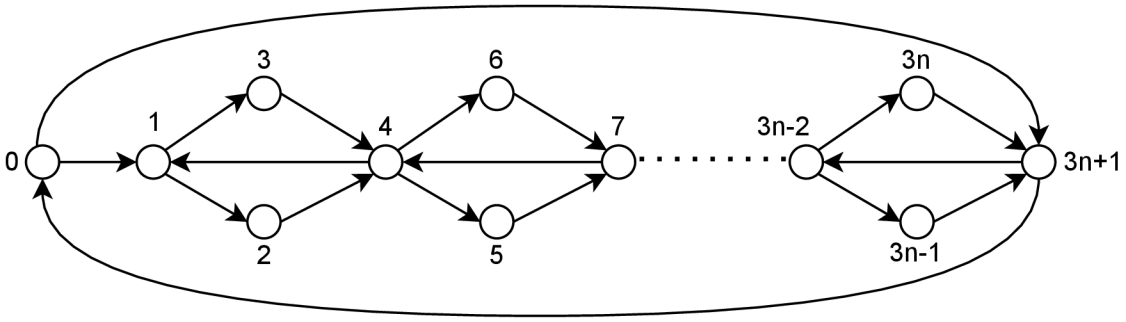


Obrázek 2.4: Nejhorší případ grafu pro Tiernanův algoritmus vyhledávání cyklů.

2.2.3 Weinblattův algoritmus

Weinblatt se snaží vylepšit Tiernanův algoritmus dvěma úpravami. První upravou je snaha zredukovat velikost grafu o vrcholy, které nemůžou patřit do žádného cyklu, ještě před začátkem vyhledávání cyklů. Redukci provádí tak, že odstraňuje vrcholy, které nejsou koncovým vrcholem žádné hrany, a také hrany, pro které tento vrchol je počátečním vrcholem. Tento krok se opakuje až do chvíle, kdy žádný takový vrchol neexistuje. Dále se Weinblatt snaží algoritmus zefektivnit minimalizováním počtu prozkoumávání každé hrany. Toho dosáhl využitím způsobu prohledávání do hloubky popsaným níže a metodou zpětného vyhledávání (backtracking) [25].

Weinblattův algoritmus je ve většině případů méně časově náročný než Tiernanův algoritmus. Paměťově je však náročnější, protože si ukládá prohledané vrcholy. Nejhorší případ grafu pro Weinblattův algoritmus je na obrázku 2.5. Tento algoritmus nalezne v nízkém čase všechny cykly dle vzoru $(3i-2, 3i+1, 3i, 3i-2)$ a $(3i-2, 3i+1, 3i-1, 3i-2)$. V případě prohledání hrany $(0, 3i+1)$ však nastává exponenciální závislost, která je zapříčiněna využitím rekurzivní procedury pro hledání cyklů, která se snaží najít cestu zpět do počátečního vrcholu cyklu kombinováním již nalezených cyklů [22].

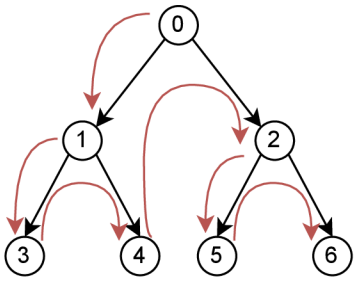


Obrázek 2.5: Nejhorší případ pro Weinblattův algoritmus vyhledávání cyklů

Prohledávání do hloubky

Technické informace jsou čerpány z článků R. Tarjana z roku 1972 [21]. Prohledávání do hloubky je algoritmus pro prohledávání grafů. Tento algoritmus je široce užíván při hledání řešení problémů kombinatoriky a umělé inteligence. Algoritmus vybere libovolný vrchol grafu (kořenový vrchol v případě stromu) a označí ho za startovní. Poté prohledává následující vrcholy. Prohledávaný vrchol je vždy označený. Pokud algoritmus při prohledávání nalezne vrchol, který již nemá žádné incidentní neoznačené vrcholy, vrátí se k prohledávání předchozího vrcholu pomocí metody zpětného vyhledávání.

Na obrázku 2.6 je znázorněn postup při prohledávání do hloubky ve stromu. Postup prohledávání je následující: postup vpřed po vrcholech `0`, `1`, `3`, návrat do vrcholu `1`, postup vpřed do vrcholu `4`, návrat do vrcholů `1`, `0`, postup vpřed do vrcholů `2`, `5`, návrat do vrcholu `2` a postup vpřed do vrcholu `6`.



Obrázek 2.6: Příklad postupu prohledávání vrcholů při algoritmu prohledávání do hloubky.

Algoritmus prohledávání grafu do hloubky lze implementovat iterativně i rekurzivně. Pro záznam prohledaných vrcholů se ve většině případů používá zásobník.

2.2.4 Tarjanův algoritmus

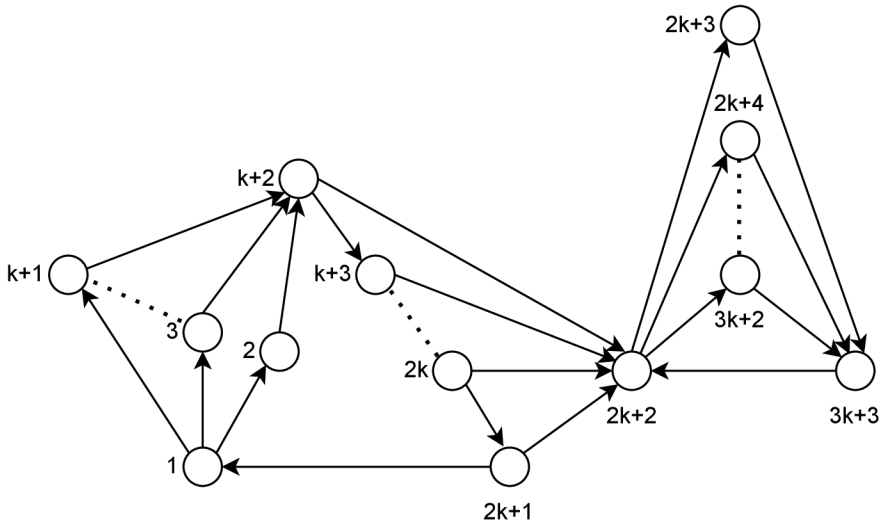
Tarjanův algoritmus je variace na Tiernanův algoritmus, používající metodu zpětného vyhledávání a metodu pohledu dopředu (look-ahead) pro vyhnutí se procházení nevyhovujících vrcholů [22]. Ačkoliv je jeho časová náročnost porovnatelná s Johnsonovým algoritmem 2.2.5 ve většině případů, v nejhorším případě (obrázek 2.7) platí časová náročnost $O(n \cdot e(c+1))$. V tomto grafu Tarjanova look-ahead metoda nedokáže nijak zjednodušit graf. Dále při hledání cyklu z vrcholu `1` až po `k+1` jsou pro každý z vrcholů `k+2` až `2k+1` všechny cykly v podgrafu z vrcholu `2k+2` do vrcholu `3k+3` prohledány celkem `k`-krát. Hledání cyklů obsahující vrchol `1` má ve výsledku časovou náročnost $O(k^3)$ [11].

2.2.5 Johnsonův algoritmus

Tato podsekce je založená na článku D. B. Johnsona [11]. Tento algoritmus byl vybrán pro jeho nejnižší časovou náročnost, která je $O((n+e)(c+1))$, je tedy polynomiální a vhodnější pro velké grafy než předchozí algoritmy.

Vstupem algoritmu je graf, reprezentovaný seznamem sousednosti. Algoritmus předpokládá označení vrcholů čísly od 1 do n. Implementace algoritmu v pseudokódu je v příloze A.1.

Algoritmus postupuje hledáním `cest` z vrcholu s. Vrcholy, které jsou součástí aktuální cesty, jsou ukládány na `zásobníku`. Vrchol je přidán na zásobník voláním rekurzivní procedury `CIRCUIT` a je z něj odstraněn při návratu této procedury. Když je vrchol v přidán



Obrázek 2.7: Nejhorší případ grafu pro Tarjanův algoritmus vyhledávání cyklů.

do cesty, je také označen v poli `blocked` jako zablokovaný. To zabraňuje použití vrcholu v dvakrát v jedné cestě. Vrchol v není odblokován při návratu volání procedury, která jej zablokovala. Odblokování je vždy zpožděno tak, že jakýkoliv z dvou způsobu odblokování v je oddělen buď nalezením cyklu, nebo návratem k proceduře, která volala proceduru `CIRCUIT`.

Správnost algoritmu závisí na tom, že žádný vrchol nezůstane zablokovaný po najití cyklu v grafu. Pro snížení časové náročnosti je však potřeba, aby všechny vrcholy byly zablokované co nejdéle je to možné, v souladu s potřebami pro správnost. Algoritmus z tohoto důvodu používá pro každý vrchol seznam blokovaných vrcholů `B`, který slouží k zapamatování informací získaných prohledávání částí grafu, které nejsou součástí cyklu. Při volání procedury `UNBLOCK(x)` se nejprve zkontroluje, zda je nějaký vrchol `y` v seznamu `B(x)`. V kladném případě je volána procedura `UNBLOCK(y)`. Poté se odblokuje vrchol `x` v poli `blocked`.

Kapitola 3

Konečné automaty

Kapitola je založená na informacích z knih [24], [10] a studijní opory [15]. Pro přesnou formulaci problému, co může počítač řešit a jaká bude efektivita výpočtu, je nutné mít formální model. Tento model musí být univerzální, aby dokázal popsat všechny typy těchto problémů. Mezi nejznámější formální modely patří konečné automaty, zásobníkové automaty, nebo složitější Turingův stroj. Formálními modely se zabývá obor teoretické informatiky, teorie automatů. Tato kapitola popisuje nejjednodušší formální modely, kterými jsou konečné automaty.

Konečné automaty jsou nástrojem pro popis různého hardwaru nebo softwaru. Jsou využívané například v následujících případech:

- Software sloužící pro design a kontrolu chování digitálních obvodů.
- Lexikální analýza kompilátoru, tedy komponenta, která rozděluje vstupní text kódu na logické jednotky, kterými jsou například identifikátory nebo klíčová slova.
- Software na zpracování velkých objemů textu, například pro vyhledávání slov, vět nebo vzorů.
- Software pro verifikaci různých typů systémů s konečným počtem odlišných stavů, kterými mohou být například komunikační protokoly, nebo protokoly zajišťující bezpečnou výměnu informací.

Konečnými automaty lze definovat systémy, které je možné popsat nějakým konečným počtem `stavů`. Stav má za úkol si pamatovat část historie systému. Vzhledem ke konečnému počtu stavů musí být tato část nějak omezená a stav si musí pamatovat jen to, co je opravdu důležité pro další chod systému.

Konečný automat nemá paměť. Jediný způsob, kterým si může něco zapamatovat, je vnitřní stav. Těch však musí být konečný počet a musí být předem definované. Každý počítač je tedy konečným automatem. Ačkoliv mají počítače paměť, není nekonečná, a lze tedy popsat konečným počtem stavů. Tento přístup však není příliš přehledný, tedy ani praktický, a je v tomto případě lepší použít pro popis některý z vyšších formálních modelů, využívající potenciální nekonečnou paměť. Tento model sice nepopisuje přímo realitu, ale pro většinu případů je rozdíl způsobený touto abstrakcí zanedbatelný a návrhu lze jednodušeji porozumět.

Problém konečného automatu je, že výstup je pouze typu `vstup přijat` a `vstup nepřijat`. V případě, kdy je potřeba komplexnějšího výstupu, je nutné použít některou alternativu konečného automatu. Těmito alternativami jsou například Mealyho automat a Moorův automat. Více informací o těchto alternativách lze nalézt například v [24].

3.1 Deterministický konečný automat

Deterministický konečný automat má vždy právě jeden `vnitřní stav`, také nazývaný `aktuální stav`. Tento stav značí právě zpracovávaný stav. Každý stav musí mít definovaný právě jeden `přechod` pro každý `symbol vstupní abecedy`. Prvním vnitřním stavem konečného automatu je `počáteční stav`.

3.1.1 Definice

Deterministický konečný automat je definován pěticí $A = (Q, \Sigma, \delta, q_0, F)$ kde:

1. Q je konečná množina `stavů`
2. Σ je neprázdná konečná množina `vstupních symbolů`, zvaná `vstupní abecedou`
3. $\delta : Q \times \Sigma \to Q$ je `přechodová funkce`, která přijímá stav $q \in Q$ a vstupní symbol $x \in \Sigma$ a vrací stav $p \in Q$.
4. $q_0 \in Q$ je `počáteční stav`
5. $F \subseteq Q$ je množina koncových stavů

Dvojice $(q, x), q \in Q, x \in \Sigma$ je nazývána `konfigurace` automatu. Konfigurace $(q_0, x), x \in \Sigma$ je nazývána `počáteční konfigurace`.

Přechodová funkce přijímá konfiguraci automatu, na základě které provádí `přechod` vnitřního stavu. V deterministickém konečném automatu existuje vždy právě jeden výstup přechodové funkce pro jednu konfiguraci. Tedy je-li automat ve stavu $q \in Q$ a přijímá znak $x \in \Sigma$, výstupem přechodové funkce vždy bude $p \in Q$, pokud platí $\delta(q, x) = p$.

3.1.2 Rozhodování deterministického automatu

Konečný automat zpracovává nějakou sekvenci vstupních symbolů a rozhoduje, zda danou sekvenci `přijme` nebo `zamítne`. `Přijímaný jazyk` automatu je množina všech sekvencí, kterou daný konečný automat přijímá.

Nechť $a_1 a_2 ... a_n$ je sekvence vstupních symbolů. Konečný automat začíná ve svém počátečním stavu q_0. Poté je využita přechodová funkce $\delta(q_0, a_1) = q_1$ pro nalezení stavu, do kterého konečný automat přejde po zpracování prvního symbolu a_1. Po přechodu je zpracováván následující symbol a_2 vyhodnocením funkce $\delta(q_1, a_2)$. Ta vrací následující stav q_2. Takhle se postupuje až do zpracování posledního znaku a_n funkcemi dle vzoru $\delta(q_(i-1), a_i) = q_i$ pro každé i z množiny $M = \{1, 2, ..., n-1, n\}$. Po zpracování posledního znaku se automat nachází ve stavu q_n. Pokud je stav q_n součástí množiny F, je sekvence $a_1 a_2 ... a_n$ `přijata`, naopak když platí $q_n \notin F$, je tato sekvence `zamítnuta`.

3.2 Nedeterministický konečný automat

Nedeterministický automat může být ve více stavech zároveň. Tato schopnost je označována jako zkoušení cest pro vstup.

Ke každému nedeterministickému konečnému automatu existuje nějaký deterministický, který přijímá stejný jazyk. Nedeterministický automat se používá pro zjednodušení návrhu, avšak je náročnější na realizaci. Z jakéhokoliv nedeterministického konečného automatu lze vytvořit deterministický, může se však zvýšit počet stavů. V nejhorším případě je minimální

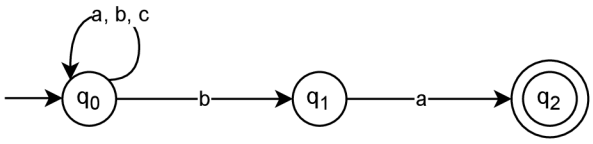
počet stavů deterministického konečného automatu 2^n, kde n je počet stavů ekvivalentního nedeterministického konečného automatu. V praxi po převodu často bývá počet stavů u deterministického konečného automatu podobný, ale zvyšuje se počet přechodů.

Nedeterministický konečný automat je definován stejně jako deterministický konečný automat pětící $A = (Q, \Sigma, \delta, q_0, F)$. Rozdíl je v přechodové funkci δ, o které platí:
$\delta : Q \times \Sigma \to 2^Q$, kde 2^Q označuje množinu všech podmnožin množiny Q.

Podmnožina jakékoliv množiny obsahuje také prázdnou množinu $\emptyset$. Tedy může existovat přechodová funkce, pro kterou platí $\delta(q, x) = \emptyset$, kde $q \in Q$ a $x \in \Sigma$. Ta značí, že pro daný vstupní symbol pro daný stav neexistuje žádný přechod.

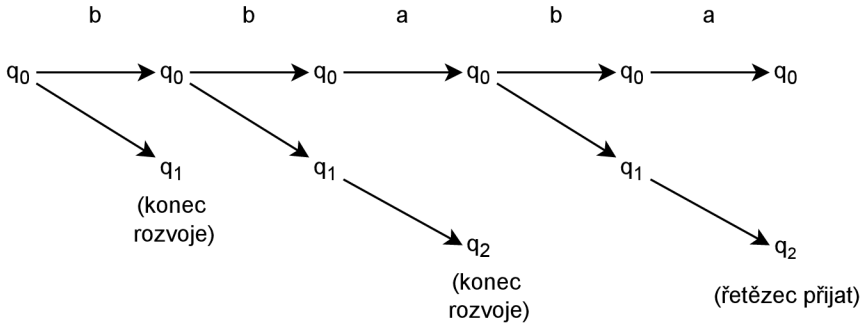
3.2.1 Rozhodování nedeterministického automatu

Nedeterministický konečný automat zkouší všechny možné přechody pro vstupní sekvenci s. Existuje-li sled přechodů, pro který po zpracování sekvence s automat končí v nějakém z koncových stavů, je sekvence s přijata.



Obrázek 3.1: Stavový diagram nedeterministického automatu, přijímající řetězce ukončené řetězcem *ba*.

Příklad: Nechť A je nedeterministický konečný automat, který přijímá řetězce složené ze znaků a, b, c končící sekvencí ba. Automat A je popsán stavovým diagramem na obrázku 3.1. Stavové diagramy jsou popsány v sekci 3.4. Nechť vstupní řetězec tohoto automatu $s = bbaba$. Z počátečního stavu q_0 vedou dva různé přechody pro vstupní symbol b. Automat vyzkouší oba přechody. Je tedy zároveň ve stavu q_0 i q_1 a zkouší opět všechny přechody pro následující znak b. Rozvoj ze stavu q_1 zde končí, protože nemá definovaný přechod pro znak b. Rozvoj stavu q_0 stále pokračuje. Takhle automat pokračuje dále až do zpracování posledního symbolu řetězce s. Pokud je alespoň jeden z vnitřních stavů dosažených po zpracování celého řetězce koncovým stavem automatu, je vstup přijat. Celý postup je graficky popsán v grafu 3.2.



Obrázek 3.2: Rozvoj vnitřních stavů nedeterministického konečného automatu pro řetězec *bbaba*.

3.3 Zápis konečného automatu

Konečný automat lze zapsat několika způsoby. Níže popisovaný automat A je deterministický konečný automat, který má vstupní symboly $\Sigma = \{a, b, c\}$ a přijímá všechna slova obsahující podřetězec $abba$. Definice jazyku L přijímaného automatem A je:

$$\{XabbaY \mid X \text{ a } Y \text{ jsou jakékoliv řetězce znaků } a, b, c\}$$

Formální zápis konečného automatu je $A = (\{q_0, q_1, q_2, q_3, q_4\}, \{a, b, c\}, \delta, q_0, q_4)$. Kde přechodová funkce δ vypadá takto: $\delta(q_0, a) = q_1$, $\delta(q_0, b) = q_0$, $\delta(q_0, c) = q_0$, $\delta(q_1, a) = q_1$, $\delta(q_1, b) = q_0$, a tak dále. Tento zápis automatu však není příliš přehledný a zápis přechodové funkce je navíc velmi zdlouhavý. Proto se pro popis konečných automatů používají stavové diagramy popsané v sekci 3.4 a další způsoby popsané v této sekci.

3.3.1 Tabulka přechodů

Tabulka přechodů je konvenční tabulkové vyjádření funkce δ, která vrací hodnotu odpovídající vstupním argumentům funkce. Řádky tabulky označují stavy a sloupce tabulky označují vstupní symboly. U nedeterministického konečného automatu se dostupné stavy pro danou konfiguraci zapisují jako množiny. Počáteční stav automatu je označen šipkou a koncové stavy hvězdičkou.

Tabulka přechodů odpovídající funkci δ konečného automatu A:

	a	b	c
$\rightarrow q_0$	q_1	q_0	q_0
q_1	q_1	q_2	q_0
q_2	q_1	q_3	q_0
q_3	q_4	q_0	q_0
$*q_4$	q_4	q_4	q_4

Výhodou reprezentace konečného automatu tabulkou přechodů je snadné zpracování automatu počítačem.

3.3.2 Stavový strom

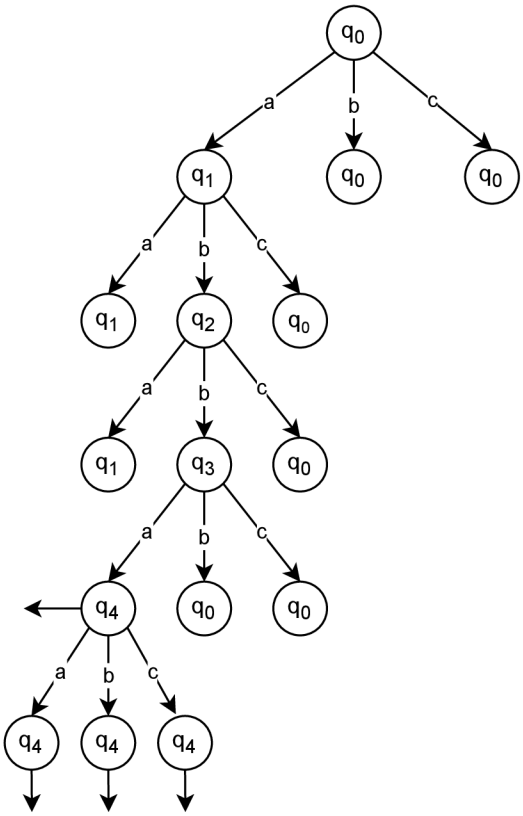
Konečný automat lze zapsat jako hranově a vrcholově ohodnocený strom. Vrcholy jsou ohodnoceny stavy a hrany vstupními symboly. Kořen stromu má hodnotu počátečního stavu. Hrana mezi vrcholy $q, p \in Q$ ohodnocená symbolem $x \in \Sigma$, značí existenci přechodu $\delta(q, x) = p$, tedy pro vstup x existuje přechod z q do p. Aby strom nebyl nekonečný, zastaví se rozvoj při výskytu vrcholu, který se již vyskytl výše ve stromě. Koncové stavy jsou označeny šipkou vedoucí z tohoto stavu ven.

Příklad zápisu automatu A stavovým stromem je na obrázku 3.3.

3.4 Stavový diagram

Stavový diagram je způsob zapsání konečného automatu formou orientovaného grafu. Stavový diagram pro konečný automat $A = (Q, \Sigma, \delta, q_0, F)$ je graf definovaný následovně:

1. Pro každý stav v Q existuje právě jeden vrchol



Obrázek 3.3: Stavový strom pro automat A ze sekce 3.3

2. Existuje-li stav $q \in Q$ a symbol $x \in \Sigma$, musí existovat přechod $\delta(q, x) = p$, který je reprezentován hranou s počátečním vrcholem q a koncovým vrcholem p. Hrana je popsána symbolem x. Existuje-li více vstupních symbolů, které způsobí přechod z q do p, mohou tyto symboly popisovat stejnou hranu.

3. Do počátečního stavu q_0 vede šipka. Tato šipka nevede ze žádného vrcholu.

4. Koncové stavy (stavy v množině F) jsou reprezentovány vrcholy označenými dvojitým kruhem. Stavy mimo F jsou reprezentovány vrcholy označenými jednoduchým kruhem.

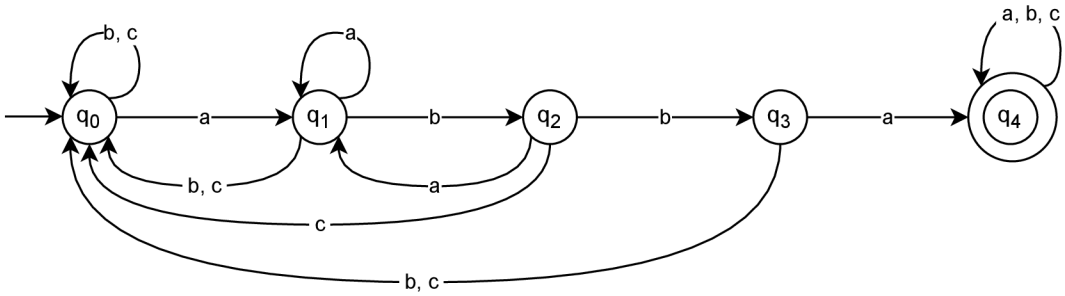
Stavové diagramy jsou široce užívány při návrhu softwarových systému. Jednak tyto diagramy mohou ilustrovat dynamiku a chod celého systému, jednak jsou užitečné pro návrh objektů, tříd, nebo také modulů systému. Stavové diagramy vyjadřují chování objektu založené na událostech, což je velice užitečné při modelování reaktivních systémů [1].

Stavový diagram konečného automatu popsaného v podsekci 3.3 je na obrázku 3.4.

3.5 Existující řešení

Informace v této sekci jsou přebrány z článků [14] a [6].

Implementace kódu podle návrhu stavového diagramu je užitečná technika pro řešení komplexních problémů softwarového inženýrství. Stavové diagramy rozeberou celý navržený



Obrázek 3.4: Stavový diagram automatu A ze sekce 3.3. Tento automat přijímá řetězce obsahující podřetězec `abba`.

systém na jednotlivé moduly, které jsou reprezentovány `stavy`. Každý stav provádí nějaký jednoduše definovatelný úkol. Přechody mezi stavy jsou vyvolávány `událostmi`.

Při implementaci systému podle stavového diagramu je dobré, když je ve zdrojovém kódu zřejmé, kde probíhá jaká událost pro změnu stavu, protože to pomáhá k údržbě a změnám kódu. Dále je potřeba mít způsob pro důkaz, že systém pracuje podle návrhu a všechny přechody byla správné pro daný scénář. K dosažení těchto cílů slouží různé nástroje. Jedním z nich je `StateMachine` popsaný v této sekci.

3.5.1 StateMachine

`StateMachine` je knihovna pro jazyk `C++`, sloužící k definici konečných automatů. Tato knihovna je kompaktní a nevyužívá šablony. Pro ukládání pravidel přechodů stavů používá tabulku přechodů. Každá událost je členská funkce veřejné instance třídy s libovolnými typy argumentů, sloužící pro kontrolu dat nad přechody. Knihovna dále umožňuje implementaci `funkce stavů`, tedy definici toho, co který stav dělá. Tyto funkce mohou v argumentu přijímat data události. Všechny stavy a události mají základní třídu, z které dědí a mohou přepisovat její funkce.

Tato knihovna také definuje pokročilé funkce sloužící pro ochranu podmínek přechodu a rozdělení vstupních a výstupních akcí pro každý stav. Dále také používá softwarové zámky pro vícevláknovou ochranu softwaru.

Při implementaci stavového diagramu pomocí StateMachine jsou z velké části používaná i víceřádková makra, které se snaží implementaci konečného automatu zjednodušit skrytím implementačních detailů.

Cílem této knihovny je být kompaktní, přenosná a univerzální pro řešení široké škály problémů související s konečnými automaty.

Události

Události se dělí na `vnější` a `vnitřní`. Vnější jsou veřejné funkce, které může volat jakékoliv vlákno v libovolném čase. Vnitřní události jsou generované samotným stavovým automatem v době funkce některého stavu. Když je vygenerována nějaká vnější událost, na základě aktuálního stavu je vyhledáváno, zda je potřeba provést nějaký přechod. Pokud ano, je přechod proveden a je vykonána funkce nového stavu. Po provedení této funkce je zkontrolováno, zda byla vyvolána nějaká vnitřní událost. Pokud ano, opět proběhne přechod a je vykonána funkce dalšího stavu. Toto se opakuje až do chvíle, kdy nebyla vyvolaná

žádná vnitřní událost. V této chvíli se vrací volání původní vnější události. Všechny tyto funkce probíhají ve vlákně, ve kterém byla vyvolána vnější události.

Události mohou mít data jako argument, tato data smí být použité pouze při funkci stavu a poté musí být odstraněna.

Přechody

Při vygenerování události je provedeno vyhledávání přechodu stavu. Toto vyhledávání může mít jeden ze tří výstupů:

1. nový stav
2. událost ignorována
3. nemůže nastat

`Nový stav` způsobí přechod stavu a spuštění funkce stavu. Pokud je nový stav stejný, jako aktuální stav, je znovu spuštěna jeho funkce. `Událost ignorována` nemá žádný vliv na přechody, ale pokud měla událost nějaká data, jsou ztracena. `Nemůže nastat` je rezervováno pro situace, kdy událost pro aktuální stav není validní a značí chybu softwaru.

Kapitola 4

Testování softwaru

Informace v této kapitole jsou převážně přebírány z [8] a [12]. Testování softwaru je proces, který probíhá po dobu jeho vývoje a jeho úkolem je co nejdříve odhalit chyby. Testování se používá pro ověření, že software splňuje požadavky určené jeho návrhem. Dále se jím ověřuje, že tento návrh opravdu slouží účelu, pro který byl navrhnut. V neposlední řadě slouží pro odhalování chyb, které pomáhá při tvoření spolehlivého softwaru.

V této kapitole jsou popsány různé metody testování, testovací nástroje a modely popisující v jaké fázi vývoje softwaru by se měla jaká jeho část testovat,

4.1 Modely životního cyklu

Modely životního cyklu popisují sled základních etap při vývoji. Každý projekt je jiný a proto i uspořádání etap jeho vývoje je rozdílné, vždy však lze nalézt řadu společných rysů. Úkolem modelu životního cyklu je definovat jednotlivé etapy a jejich časovou posloupnost. Každá etapa musí vytvořit svůj reálný výstup, podle kterého je oddělena od následující etapy [12].

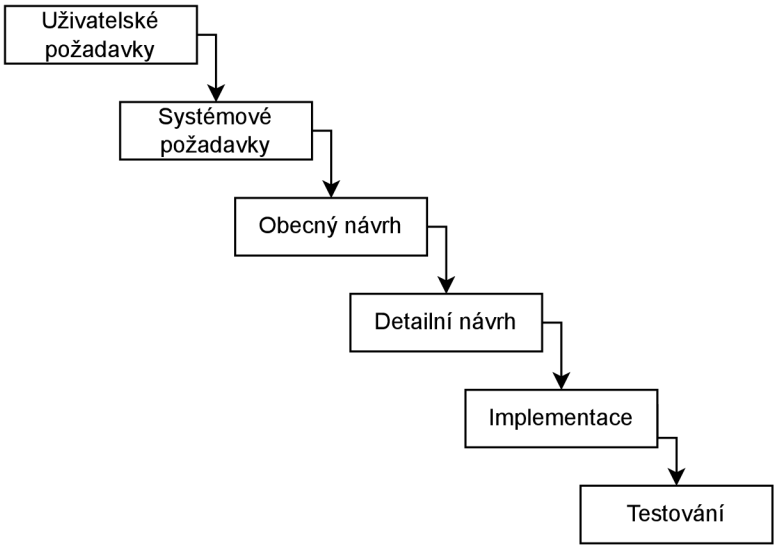
Existuje mnoho různých modelů životního cyklu, některé jsou vhodné v projektech, kde je zadání striktní a nemění se vůbec nebo jen velmi málo. Dva základní modely tohoto typu jsou popsané v této sekci. Jsou zde popsané pro znázornění jednotlivých etap a s nimi souvisejícími fázemi testování. Jiné modely jsou vhodnější pro projekty, u kterých se často mohou měnit uživatelské či systémové požadavky. Příkladem takového modelu je `spirálový model`[1], případně pokud se zadání určuje až při vývoji, jsou vhodné agilní metodiky[2].

4.1.1 Vodopádový model

Vodopádový model je jeden z nejstarších modelů a odvíjejí se od něj pokročilejší modely. Tento model má přirozenou posloupnost a jednotlivé etapy jsou v něm plněny postupně od specifikace uživatelského požadavku až po nasazení do reálného prostředí. Nevýhodou je, že testování v tomto modelu probíhá až ke konci vývoje a je složité dostat zpětnou vazbu do dřívějších fází a chyby se nesou několika vývojovými fázemi, tedy chyba v uživatelském požadavku je přenášená až do etapy implementace, kde po odhalení musí znovu přejít všemi etapami.

[1] http://testovanisoftwaru.cz/manualni-testovani/modely-zivotniho-cyklu-softwaru/spiralovy-model/

[2] https://cw.fel.cvut.cz/old/_media/courses/a4b33si/prednaskove_materialy/si4-agile.pdf



Obrázek 4.1: Vodopádový model životního cyklu softwaru

4.1.2 V-model

V-model je model životního cyklu softwaru, který vychází z vodopádového modelu.

V-model přidává čtyři testovací úrovně do již zmíněného vodopádového modelu. Každá testovací úroveň přímo souvisí s některou úrovní danou vodopádovým modelem. Popis jeho etap je znázorněn na obrázku 4.2.

Jednotkové testování

Při jednotkovém testování se hledají chyby a verifikuje se funkce jednotlivých jednotek, kterými mohou být moduly, funkce, objekty, třídy a tak dále. Jednotkové testování ověřuje funkci nejmenších oddělitelných jednotek izolovaných od zbytku kódu.

Integrační testování

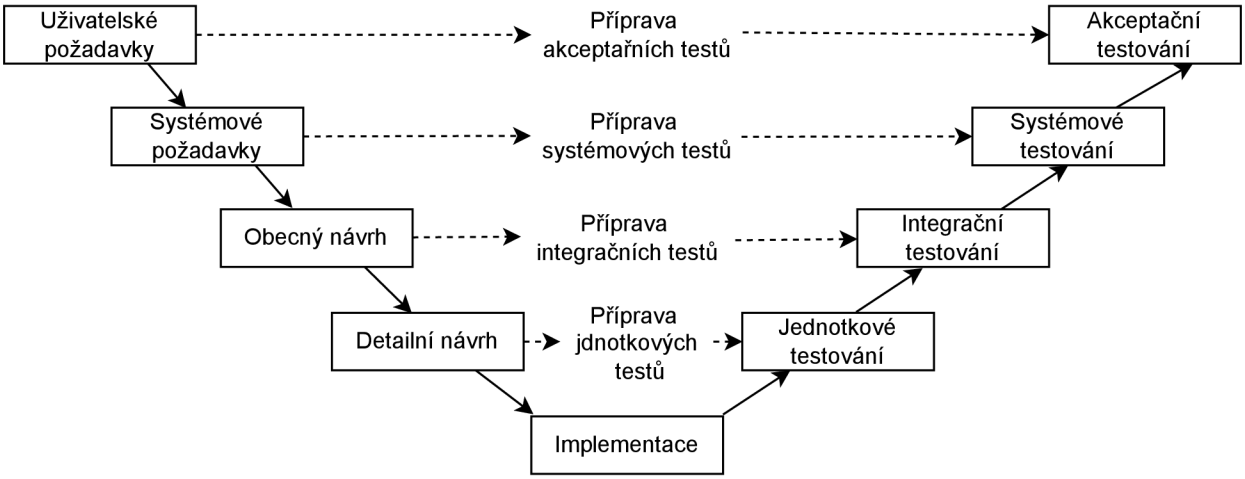
Integrační testování testuje interakce modulů a komponent, které byly vyvíjeny zvlášť, sloučených do jednoho systému. Tento systém se testuje jako celek a ověřuje, zda nenastávají chyby při interakci těchto komponent.

Systémové testování

Při systémovém testování se ověřuje, zda celý systém pracuje podle specifikovaných systémových požadavků.

Akceptační testování

Při akceptačním testování se validuje, zda systém odpovídá uživatelským potřebám a požadavkům a rozhoduje se, zda bude výsledný systém přijat.



Obrázek 4.2: V-model životního cyklu softwaru.

4.2 Strategie testování

Tato sekce je založená na informacích z [12]. Existuje několik strategií postupu při testování. Volba strategie testování často závisí na zvolené strategii implementace.

Testování zdola-nahoru

Při tomto způsobu se nejdříve testují jednotky na nejnižší úrovni abstrakce a až poté se testují části na vyšší úrovni. Tato strategie je vhodná pro testování obecně použitelných modulů.

Testování shora-dolů

Při tomto způsobu se začíná testovat na nejvyšší úrovni abstrakce a postupně se postupuje k jednotlivým modulům. Při této strategii je třeba řešit jistou simulaci funkce modulů na nižší úrovni a vede k omezené znovupoužitelnosti modulů.

Sendvičové testování

Jedná se o kombinaci předchozích strategií. Při této strategii se moduly rozdělí na logické, které jsou testovány metodou shora-dolů, a funkční, které jsou testovány metodou zdola-nahoru.

Jednorázové testování

Tato strategie v otestování modulů a následné integrace do systémů. Tato strategie je vhodná pouze pro malé systémy, protože je obtížné odhalit příčinu chyby.

Testování porovnáváním

Pro tuto strategii je nutné, aby testovaný systém měl více verzí, jejichž výstupy se mohou porovnávat. Tato strategie předpokládá, že výstup systému byl v předchozí verzi správný

a dokáže odhalit pouze rozdíly mezi jednotlivými verzemi, ale nikoliv chybu, která se vyskytuje ve všech verzích, nebo také chybu ve specifikaci systému, protože slouží k ověření, že implementace vyhovuje specifikaci.

4.3 Testovací metody

Existuje mnoho testovacích metod a každá má nějaké výhody a nevýhody. Každá metoda je dobrá pro hledání některého typu chyb, ale slabší při hledání jiného typu. Různé metody jsou také užitečné v různé fázi vývoje softwaru. Testovací metody se dělí do několika kategorií. Dvě hlavní kategorie jsou dynamická a statická. Statické metody se soustředí na inspekci kódu, kontrolu datových toků a podobně. Statické metody kód nespouštějí a jsou používány před dynamickými metodami.

Dynamické metody jsou metody, při kterých se spouští kód a ověřují se jeho výstupy. Pro toto testování je potřeba naplánovat a připravit testovací scénáře a definovat způsob vyhodnocení. Většinou testovací scénáře vznikají, když je implementována část kódu, kterou má scénář testovat, v horším případě po implementaci celého programu. Existuje však také přístup, který se nazývá `programování řízené testy` (anglicky test-driven development), při kterém se nejprve vytvoří testovací scénáře a teprve poté je implementována část kódu, která slouží pro úspěšné projití každého z těchto scénářů [5].

Dynamické metody jsou dále rozdělené na další podkategorie, nejzákladnější dvě jsou popsané v této sekci.

4.3.1 White-box

Metoda white-box (bílá skříňka) je metoda založená na struktuře programu. Tato metoda je označována jako bílá, protože skrze ní lze vidět co se děje uvnitř skříňky, stejně jako u této metody, která vyžaduje znalost implementace daného softwaru. Cílem testu může být například otestovat nějakou smyčku. V tomto případě testovací scénáře budou takové, aby testovaná smyčka proběhla jednou, dvakrát a vícekrát. Pro tyto testy není nutné znát funkcionalitu celého softwaru.

4.3.2 Black-box

Metoda black-box (černá skříňka) je metoda založená na specifikaci softwaru. Černá skříňka vyjadřuje zařízení, u kterého není známo jak pracuje uvnitř, ale známe jeho vstupy a výstupy. V této metodě se tester zaměřuje na to, co software dělá, nikoliv na to, jak to dělá.

4.3.3 Testování přechodů stavů

Testování přechodů stavů je testovací metodou spadající do kategorie black-box. Tato metoda lze využít pro testování systému popsaných konečným automatem 3. Při tomto testování je snaha vytvořit testovací scénáře takové, že pro každý stav konečného automatu existuje testovací scénář, při kterém se tento konečný automat do tohoto stavu dostane. Testovací scénáře se mohou také zaměřovat na pokrytí všech přechodů.

Vytváření testovacích scénářů podle stavového diagramu je přístup odpovídající metodě black-box. Měření míry pokrytí stavů těmito testy je již hraniční s metodou white-box.

Výhodou této metody je, že testovaný stavový diagram může mít míru abstrakce podle potřeby. Tedy některé stavy mohou vykonávat mnoho funkcí, které však nejsou potřeba tolik

otestovat, například protože jsou testované jinou metodou. Jiné stavy mohou reprezentovat jen jedinou funkcionalitu.

4.4 Debugging

Pokud nějaký test odhalí chybu, která má být opravena, musí programátor nejprve najít místo v kódu, ve kterém tato chyba nastala. Tento proces se nazývá `debugging`. Při debuggingu mohou pomoci záznamy běhu aplikace, nebo také pozorování vnitřního stavu za běhu systému. V této sekci jsou popsány příklady nástrojů pro debugging.

4.4.1 Nástroje pro debugging

Pro záznam běhu aplikace jsou často využívané logovací knihovny. Formáty záznamů těchto knihoven jsou více popsány v sekci 5.1.3. Do této kategorie debuggingu spadá také výsledný nástroj této práce.

Další formou debuggingu je sledování toho, co se děje uvnitř programu při jeho běhu. Pro tento účel slouží například nástroj `GNU Debugger` (zkráceně `gdb`). Tento nástroj nabízí čtyři hlavní funkcionality pro odchycení chyb [20]:

- spuštění programu se specifikací čehokoliv, co by mohlo ovlivnit jeho chování,
- zastavení programu při výskytu specifikované podmínky,
- prozkoumání zásobníku volání funkcí a dat, když je program zastaven,
- změnu částí programu pro zjištění, jestli opravení jedné chyby nepovede k další.

4.5 Testovací nástroje

Pro každou testovací metodu existuje nějaký nástroj pro její provedení a automatizaci. Testovací nástroje vznikají pro testování programů napsaných v různých jazycích a mají různé určení. Některé porovnávají vstupy a výstupy funkce, zjišťují pokrytí a výkon kódu, jiné mohou například kontrolovat úniky paměti. Nástroj tvořený v této práci je nástroj pro metodu testování přechodů stavů, který využívá strategii testování porovnáváním.

Níže jsou popsané technologie, které byly využívaný pro testování výsledného balíčku:

GoogleTest

`GoogleTest` je testovací framework pro programy v jazyce `C++`, založený na architektuře `xUnit`. GoogleTest sjednocuje testovací případy do sad pro přehlednost. Každý test je izolován od všech ostatních, testy se tak nijak neovlivňují. GoogleTest sleduje všechny testovací případy sám, není nutné je tedy nikam přidávat, ale jsou vždy spuštěny všechny. Testovací případy využívají `tvrzení` k vyhodnocení chování testovaného kódu. Pokud test spadne, nebo nesplní tvrzení, je označen za neúspěšný [3]. Ve výpisu 4.1 je příklad testovací sady `odmocniny`, ve které je v případě scénáře `kladna` je porovnáván výstup funkce `odmocnina` s konstantou. V scénáři `zaporna` je tvrzeno, že funkce `odmocnina` vyvolá výjimku.

```
1 TEST(odmocniny, kladna) {
2     EXPECT_EQ(odmocnina(4), 2);
3     EXPECT_EQ(odmocnina(9), 3);
4 }
5
6 TEST(odmocniny, zaporna) {
7     EXPECT_THROW(odmocnina(-4));
8 }
```

Výpis 4.1: Příklad testovací sady frameworku GoogleTest.

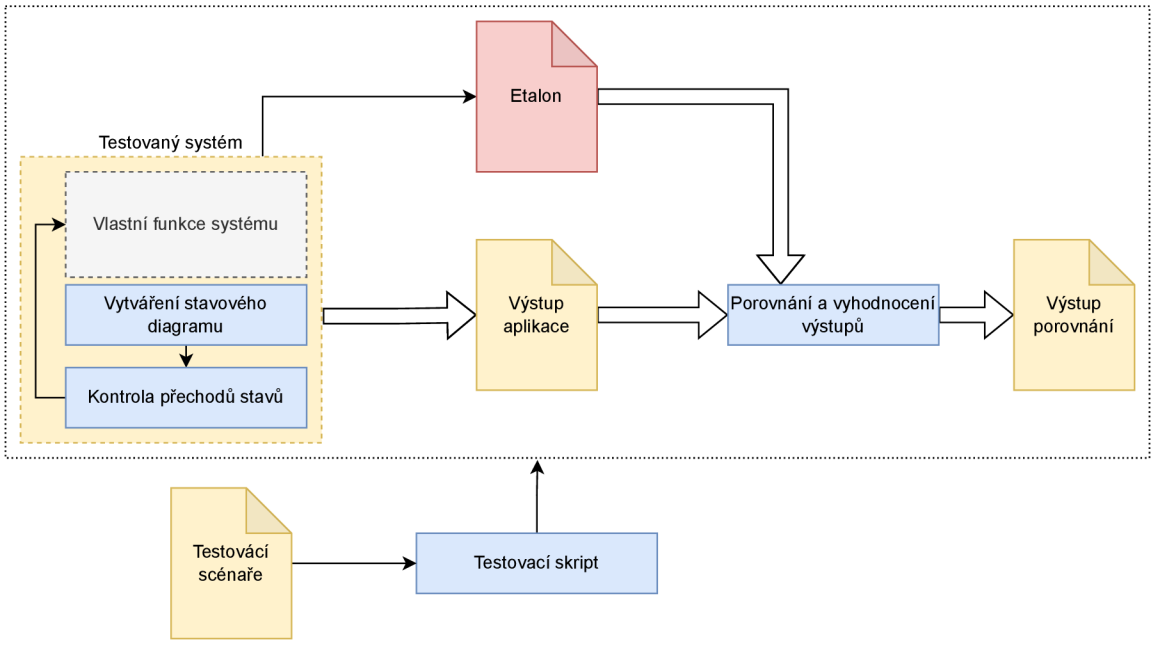
Valgrind

Sada nástojů `Valgrind` poskytuje mnoho nástrojů pro debugging a profilování programů. Tyto nástroje slouží k vylepšení programů ve směru stability, ale také rychlosti. Nejvíce používaným nástrojem je `Memcheck`. Ten objevuje chyby spojené s pamětí, které jsou velice časté v programech v jazyce `C` a `C++` a mohou vést k nepředvídatelnému chování nebo k pádu programu [4].

Kapitola 5

Návrh balíčku nástrojů

Cílem této práce bylo vytvořit balíček nástrojů tvořící modulární systém, který slouží pro integrační testování systémů implementovaných v jazyce `C++` na základě přechodů stavů 4.3.3. Pro dosažení tohoto cíle bylo potřeba vytvořit balíček nástrojů, který bude schopný definovat stavový diagram a přechodové funkce systému, nahrávat přechody stavů za běhu programu a poté vyhodnotit tyto nahrané stavy. Návrh daného balíčku nástrojů je popsán v této kapitole.



Obrázek 5.1: Návrh balíčku nástrojů pro testování systémů podle stavových přechodů.

5.1 Vytváření stavovém diagramu a zaznamenávání přechodů stavů

První požadavek při vytváření tohoto systému byla knihovna pro definici stavového diagramu a zaznamenávání přechodů mezi stavy. Tato knihovna byla rozdělena na dvě části

z důvodu požadavku na modularitu systému. Důvod tohoto požadavku je popsaný níže v podsekci 5.1.4.

Tyto dvě části musí být napsány v jazyce `C++`, protože jsou integrovány přímo do testovaného systému. Také je z tohoto důvodu kladen důraz na jednoduchost jejich použití. Integrace balíčku do již existujícího programu by neměla vyžadovat žádnou změnu jeho struktury. Zároveň by měla pomoci při implementaci nového systému k dodržení logického postupu zadaného stavovým diagramem, který definuje chování tohoto systému. Při integraci těchto komponent do již existujících systémů je zásadní rozdíl oproti knihovně `StateMachine`, popsané v sekci 3.5.1, která hraje silnou roli při řízení programu a pro její správné využití je potřeba, aby byl systém implementován kolem ní.

5.1.1 Definice stavového diagramu

První část návrhu je knihovna, pomocí které bude definován stavový diagram aplikace. Tato knihovna by měla být jednoduchá pro implementaci. V této verzi knihovny není kladen důraz na její výpočetní rychlost, ale je cílem vytvořit koncept testovacího systému. Je však důležité, aby knihovna byla nahraditelná pro pozdější vylepšení systému. Pro tuto práci tedy bylo potřeba mít nějaký prostředek, podle kterého bude možné navrhnout a implementovat další části systému. Tato knihovna pro jednoduchost definuje stavový diagram prostým orientovaným grafem 2.1, který je pouze rozšířen o počáteční stav. Koncové stavy nejsou implementovány, protože nebyly užitečné v žádném ze systému, do kterých byla knihovna integrována. Kontrola dat mezi přechody, jak to dělá například knihovna `StateMachine`, může být příliš komplikovaná a není tedy součástí této knihovny, která má za cíl být jednoduchá pro použití a univerzální.

5.1.2 Kontrola přechodů stavů a jejich zaznamenávání

Další komponentou je rozhraní, které pracuje nad stavovým diagramem definovaným knihovnou popsanou v předchozí sekci. Toto rozhraní má na starost kontrolu mezi přechody stavů a jejich zaznamenávání. Zaznamenání přechodů je implementováno pomocí textových záznamů (logů 5.1.3). Zásadním rozdílem mezi knihovnou `StateMachine` a tímto rozhraním je, že toto rozhraní nemá za úkol řídit program, neslouží k zjišťování momentálního stavu, ani k zajištění konzistence dat u vícevláknových programů.

5.1.3 Zaznamenávání dat

Soubory do kterých se zaznamenávají data běhu systému jsou nazývané `logy`. Pro označení jednotlivých záznamů o činnosti systému a jeho datech se používá jednotné číslo, tedy `log`. Logy obsahují užitečné informace o běhu systému. Tyto informace jsou používány pro analýzu běhu, odhalování chyb a bezpečnostních rizik, měření výkonu a mnoho dalších. Logy jsou často příliš obsáhlé pro manuální analýzu a proto se v poslední době pro analýzu těchto souborů hojně využívají algoritmy `data miningu` a `strojového učení`. Manuální analýza je vhodná například pro odhalení důvodu chyby, když je již známo, kdy k chybě došlo [16].

Formát logů se může měnit v závislosti na použitém logovacím nástroji a typu systému zaznamenávající logy. Například webové servery zaznamenávají typ požadavku (GET, POST), počet přenesených bajtů a tak dále. Jiné systémy mohou obsahovat název programu, který záznam vytvořil a typ záznamu. Většinou však platí nutnost časové značky,

podle které lze zjistit, kdy k jaké události došlo a je zásadní pro analýzu systémů běžících v reálném čase [18].

Logy implementovaného balíčku obsahují:

- časovou značku,
- název programu, do kterého je integrován,
- typ záznamu,
- značku, se kterou z komponent balíčku záznam souvisí,
- data.

Data při začátku běhu systému obsahují tabulku přechodů pro stavový diagram systému. Dále při běhu obsahují informaci o přechodu mezi stavy a název nového stavu.

Pro zaznamenávání dat byla využita knihovna `BringAuto-Logger`[1]. Jedná se o open-source knihovnu partnerské společnosti a byla použita, protože je již použita v systémech této společnosti, do kterých tento nástroj byl integrován v rámci vývoje.

5.1.4 Modularita a nahraditelnost

Ačkoliv knihovna pro vytváření grafu musí implementovat logiku přechodů, v aplikaci je pro přechody stavů použita mezivrstva. Je to z důvodu požadavku na modularitu systému a nahraditelnost jednotlivých komponent. Toto oddělení umožňuje definici stavového diagramu i jiným způsobem než knihovnou, která je implementována v rámci tohoto balíčku, nebo také použití jiného způsobu zaznamenávání přechodů stavů.

Pro nahrazení některé z těchto komponent je tedy potřeba změnit jen mezivrstvu, která je velice jednoduchá a nekontroluje žádnou logiku, tedy při správně provedené náhradě nemůže nastat změna chování systému. Také díky tomuto přístupu například není zapotřebí pochopit knihovnu definující stavový diagram, pokud se vyskytne požadavek na změnu logovací knihovny. Programátor takto pracuje pouze s tím, čemu rozumí, tedy s tím, co plánuje změnit.

Při náhradě některé z komponent se tedy nemění implementace přechodů stavů v systému. Pouze se v rozhraní pro kontrolu přechodů stavů nahradí volání funkcí přímo související s nahrazenou komponentou.

5.2 Porovnání a vyhodnocení přechodů stavů

Předchozí části systému zaznamenaly přechody stavů za běhu programu. Manuální kontrola těchto záznamů může sloužit při podrobném debuggingu, zvláště při implementaci nového systému, nebo při hledání chyby. Pro integrační testování je však potřeba nějaký nástroj pro automatické vyhodnocení správnosti výstupu běhu systému. Správnost se vyhodnocuje porovnáváním se souborem zvaným `etalon`, který je popsaný v podsekci 5.2.1.

Pro filtraci záznamů o přechodech stavů a jejich porovnání je vytvořená aplikace v jazyce `C++`, která přijímá na vstupu etalon a aktuální výstup testovaného běhu programu. Tato aplikace před samotným vyhodnocováním najde cykly ve stavovém diagramu aplikace, vyfiltruje záznamy o přechodech stavů a seskupí je podle cyklu do kterého tyto stavy patří. Tyto seskupené soubory jsou poté mezi sebou porovnány. Výstupem programu jsou záznamy o rozdílných přechodech stavů.

[1] https://github.com/bringauto/ba-logger

5.2.1 Etalon

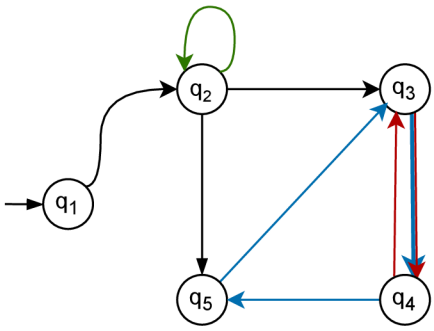
Etalon je měřící prostředek, kterého základní význam je zabezpečení metrologické jednotnosti a správnosti měřidel a měření. Definice etalonu:

Etalon je ztělesněná míra, měřidlo, referenční materiál nebo měřicí systém, určený k definování, realizaci, uchovávání nebo reprodukování jednotky nebo jedné nebo více hodnot veličiny k použití pro referenční účely. Referenčními účely se myslí odevzdávání hodnoty této jednotky méně přesným měřidlům. Etalon vyhovuje stanoveným požadavkům a byl jako takový oficiálně schválen [13].

V této práci se slovo etalon používá ve významu referenčního souboru se zaznamenanými přechody stavů, jehož správnost ověřuje programátor manuální kontrolou kontextu a logiky přechodů stavů společně s nimi spojenými daty. Tento etalon je po ověření správnosti uložen a využíván k validaci následujících běhů programu.

5.2.2 Agregace cyklů

V původním návrhu žádná agregace neprobíhala. To však často vedlo k falešně negativním výsledkům testů například z důvodu zpoždění komunikace mezi klientem a serverem, nebo jiným čekáním.



Obrázek 5.2: Cykly v grafu reprezentující superstavy.

```
1   Prechod do stavu q1
2   Prechod do stavu q2
3   Prechod do stavu q2
4   Prechod do stavu q3
5   Prechod do stavu q4
6   Prechod do stavu q3
7   Prechod do stavu q4
8   Prechod do stavu q5
9   Prechod do stavu q3
10  Prechod do stavu q4
```

Výpis 5.1: Agregace přechodů stavů do superstavů označených barvou podle grafu 5.2

Tento problém řeší agregace záznamů podle cyklů v grafu. Aplikace pro vyhodnocení přijímá stavový graf ze souboru s logy, ve kterém poté najde všechny cykly pomocí Johnsonova algoritmu popsaném v sekci 2.2.5. Záznamy o přechodech stavů jsou poté agregovány

podle těchto cyklů do takzvaných `superstavů`. Tyto superstavy jsou zaznamenány pouze jednou a obsahují všechny přechody stavů, které se opakují v jednom cyklu. Pouhé záznamy superstavů nejsou dostatečně podrobné pro debugging a nemusí být ani dostatečné pro manuální kontrolu záznamů, například pro vytvoření etalonu, protože mohou skrýt nějaká data, ale jsou dostatečné pro automatické vyhodnocení běhu.

Příklad grafů s barevně označenými cykly je na obrázku 5.2. Každý z těchto cyklů může být superstav. Příklad agregace přechodů do superstavů označených podle barev v tohoto grafu je znázorněn na 5.1.

Agregace cyklů také slouží při ukončování běhu systému, kdy systém často čeká na ukončení v nějakém cyklu. Signál pro ukončení může přijít kdykoliv v tomto cyklu a je tedy potřeba poznat, zda v něm systém opravdu ukončil běh, přestože nemuseli proběhnout všechny přechody stavů pro daný cyklus.

5.3 Skript pro automatizaci testů

Poslední částí balíčku nástrojů je skript, který slouží k automatickému spouštění testů. Tento skript je napsán v jazyce `Python`. Vstupem tohoto skriptu je testovací soubor scénářů, popsaný v podsekci 5.3.1. Výstup každého testovacího scénáře je poté předán vyhodnocovací aplikaci. Cílem tohoto skriptu bylo vytvořit nástroj pro automatizaci integračních testů.

Tento skript kromě porovnávání také umí vytvořit etalony. Ty jsou potřeba vytvořit při přidání nových scénářů, nebo při změně systému takové, kdy se záměrně změní jeho chování a tedy i přechody stavů. Vytváření etalonů ručně pro každý scénář by bylo příliš složité a bylo by náchylné na chybu provedenou programátorem, ať již při vytvoření testovacího prostředí, nebo samotném spuštění programu, nebo časování ukončení běhu. Automaticky vytvořené etalony je stále nutné manuálně ověřit pro správnost.

5.3.1 Soubor scénářů

Soubor scénářů je soubor ve formátu `JSON`, který obsahuje příkazy pro přípravu prostředí k testování a jednotlivé testovací scénáře programu, které obsahují mimo argumenty ke spuštění testovaného programu také maximální dobu běhu každého scénáře.

Kapitola 6

Implementace

Finální balíček nástrojů byl pojmenován `StateSmurf`. V této kapitole je popsána implementace jednotlivých komponent balíčku.

Poznámky k implementaci

Celý kód, včetně názvů funkcí a proměnných, je implementován v anglickém jazyce. Tyto názvy nebudu překládat. Privátní proměnné tříd jsou vždy značené podtržítkem na konci názvu, například `privatniPromenna_`. Pomocné funkce a třídy nejsou popisovány.

6.1 Vytváření grafu

Knihovna pro vytváření stavového diagramu se jmenuje `DiagramSmurf`. Tato knihovna obsahuje funkce pro definici hran, vrcholů a počátečních vrcholů stavového diagramu. Tento stavový diagram není přesný popis konečného automatu jak byl popsán v sekci 3, definuje spíše orientovaný graf rozšířený o počáteční vrchol. Je však dostatečný pro splnění požadavků pro tento testovací systém. Dále implementuje funkce pro přechody mezi jednotlivými stavy.

Komponenta `DiagramSmurf` byla implementována ve dvou verzích. První verze byla spíše koncept knihovny. Na začátku práce bylo potřeba upřesnit jak celý systém bude fungovat, proto byl vytvořen prototyp, u kterého byla prioritní jednoduchost implementace.

6.1.1 Vrcholy a hrany

Vrchol reprezentující stav stavového diagramu je implementován třídou `Vertex` tvořenou konstantní privátní proměnnou obsahující jeho název a funkcí pro získání tohoto názvu.

V první verzi byl stavový diagram reprezentován seznamem vrcholů a hran. Pro uložení hran, reprezentujících přechody stavového diagramu, sloužila třída `Edge`, obsahující svůj počáteční a koncový vrchol. Ačkoliv seznam vrcholů a hran je velice jednoduchý a srozumitelný způsob uložení grafu, v pozdější verzi byl nahrazen z důvodu vyšší hardwarové náročnosti. Seznamy vrcholů a hran byly implementovány vektory:

```
std::vector<Vertex> vertexes;
std::vector<Edge> edges;
```

Tato verze byla paměťově náročnější, protože se v paměti mimo vrcholy uchovávali také hrany, které obsahovali pouze reference na dva vrcholy.

V druhé verzi je již stavový diagram reprezentován seznamem sousednosti. V tomto řešení byla odstraněna třída `Edge`, protože povolené přechody jsou popsané konečnými vrcholy v seznamu pro každý počáteční vrchol. Seznam je implementován binárním vyhledávacím stromem `std::map`[1] standardní knihovny `C++`, ve které je každý vrchol klíčem. Ten poté odkazuje na seznam vrcholů implementovaným vektorem. O těchto vrcholech platí, že jsou koncovým vrcholem hran, ve kterých je počátečním vrcholem daný klíč. Seznam sousednosti je uložen jako privátní proměnná `adjacencyList_` třídy `StateDiagram`:

```
std::map<std::shared_ptr<Vertex>,
         std::vector<std::shared_ptr<Vertex>>> adjacencyList_;
```

Reprezentace seznamem sousednosti byla také zavedena pro zjednodušení předávání stavového diagramu mezi aplikacemi, více o důvodu předávání v 6.2.

Vrcholy grafu jsou přidávány funkcí:

```
std::shared_ptr<Vertex> addVertex(const std::string &name);
```

Funkce přijímá řetězec obsahující název stavu. Nejprve vytvoří daný vrchol, který poté přidá jako klíč do mapy `adjacencyList_`. Poté vrací referenci na tento vrchol, která je dále používána pro definici hran a počátečních vrcholů.

Pro přidávání hran mezi vrcholy slouží funkce:

```
void setEdge(const std::shared_ptr<Vertex> &from,
             const std::shared_ptr<Vertex> &to);
```

Tato funkce přijímá referenci na počáteční a koncový vrchol hrany. Funkce nejprve zkontroluje, zda jsou oba vrcholy součástí daného grafu a poté vyhledá v `adjacencyList_` vektor odkazovaný počátečním vrcholem, do kterého poté přidá koncový vrchol.

Počáteční vrchol

Počáteční vrchol stavového diagramu se přidává funkcí:

```
void setStartVertex(const std::shared_ptr<Vertex> &vertex);
```

Funkce přijímá referenci na již existující vrchol ve stavovém diagramu, který označí za počáteční přidáním do seznamu dostupných stavů, tedy vložením do mapy, ze stavu `__START__`. Stav `__START__` je implicitně přidán do seznamu vrcholů a označen za aktuální stav při volání konstruktoru třídy `StateDiagram`. Volání funkce `setStartVertex(destination)` je tedy podobné volání funkce `setEdge(__START__, destination)`, ve které je však použití rezervovaného vrcholu `__START__` zakázáno. Počáteční vrcholy jsou implementovány tímto způsobem z důvodu jejich předávání dalším aplikacím.

6.1.2 Přechody hran

Pro přechody mezi stavy třída `StateDiagram` obsahuje přetíženou funkci:

```
bool changeState(const std::shared_ptr<Vertex> &destinationVertex);
bool changeState(const std::string &vertexName);
```

[1] http://www.cplusplus.com/reference/map/map/

V první variantě funkce `changeState` přijímá referenci na cílový vrchol. Tato varianta je rychlejší, než druhá možnost, protože se v ní porovnávají reference. časová složitost tohoto způsobu ve vektoru je tedy $O(E)$, kde E je počet hran grafu, ale vyžaduje uložení ukazatelů na vrcholy v rámci kontextu. Druhá varianta funkce (implementace v 6.1) přijímá jako parametr název cílového stavu jako řetězec, nevýhodou je vyšší výpočetní náročnost z důvodu porovnávání řetězců. Časová složitost této funkce pro prohledávání vektoru je $O(E \cdot (n + m))$, kde n, m jsou délky řetězců. Tato funkce zjišťuje, zda je možný přechod z aktuálního stavu do cílového. Pokud je přechod možný, proběhne změna aktuálního stavu a funkce vrací `true`. V opačném případě funkce vrátí `false`.

Aktuální stav je privátní proměnnou třídy `StateDiagram` nazvanou `currentState_`, která obsahuje referenci na vrchol, do kterého byl proveden poslední úspěšný přechod. Při volání konstruktoru je do ní vložena reference na vrchol `__START__`.

V první verzi byly všechny hrany uloženy ve vektoru, který se celý prohledával. Pro každou hranu se zjišťovalo, zda se počáteční vrchol rovná současnému stavu a koncový vrchol cílovému stavu.

V druhé verzi jsou vrcholy místo ve vektoru uloženy v binárním vyhledávacím stromě ve kterém platí složitost vyhledávání $O(ln(k))$, kde k je počet prvků binárního stromu. Při hledání hrany je nejprve v této mapě nalezen vektor, pro který je klíčem aktuální stav. Tento vektor je poté prohledáván. Časová složitost tohoto způsobu vyhledávání hran je $O(ln(V) + E')$, (nebo $O(ln(V) + E' \cdot n)$, kde n je délka řetězce, pokud se vyhledává podle řetězce). V je počet vrcholů grafu. E' je podmnožinou množiny všech hran E, pro kterou platí, že všechny její hrany mají počáteční vrchol v aktuálním stavu. Tato podmnožina má v prostém grafu maximální velikost rovnou počtu vrcholů, tedy nejvyšší velikost je $|E'| = V$, zatímco nejvyšší velikost celé množiny E v prostém grafu je $|E| = V^2$.

```
1 bool StateDiagram::changeState(const std::string &vertexName) {
2     for (const auto &vertex: adjacencyList_[currentState_]) {
3         if (vertex->getName() == vertexName) {
4             currentState_ = vertex;
5             return true;
6         }
7     }
8     return false;
9 }
```

Výpis 6.1: Definice funkce pro změnu stavů podle jména.

6.2 Kontrola přechodů stavů

Kontrola přechodů stavů je implementována komponentou s názvem `TransitionSmurf`. Jedná se o komponentu poskytující mimo jiné rozhraní pro kontrolu přechodů nad stavovým diagramem definovaným komponentou `DiagramSmurf`.

6.2.1 Přechody stavů

`TransitionSmurf` pro přechody mezi stavy implementuje funkci:

```
bool goToState(const std::string &stateName);
```

která volá funkce knihovny `DiagramSmurf` pro změnu stavu podle jména stavu předaného parametrem 6.1. Vždy po přechodu mezi stavy TransitionSmurf zaloguje záznam, který je ve formátu popsaném v podsekci 6.2.3 a obsahuje nový stav. V případě neúspěšného přechodu zaznamená chybu. Nevyvolává však výjimku, ani se nijak nesnaží ukončit aplikaci. Jak již bylo popsáno v návrhu, tento systém slouží spíše pro zpětnou kontrolu a nemá za úkol řízení běhu systému. Při neúspěšném přechodu však funkce vrací boolovskou hodnotu `false`, kterou již systém může jakkoliv zpracovat.

Funkce `goToState` dále volá funkci:

```
bool inState(const std::string &stateName)
```

která slouží pro kontrolu, že změna stavu proběhla. Funkce porovnává, zda je aktuální stav `StateDiagram::currentState_` totožný se stavem předaným parametrem `stateName`. Jedná se o kontrolu datové konzistence přechodů ve vícevláknových aplikacích, tedy ověření, že při přechodu stavu nedochází k souběhu (anglicky race condition)[2].

Konzistenci dat lze zajistit použitím synchronizačních zámků, to však není žádoucí používat přímo ve funkcích komponenty `TransitionSmurf`. Jedním z důvodů je, že každá aplikace může používat jinou implementaci zámků a není vhodné používat více různých synchronizačních prostředků v jednom systému. Zajištění ochrany dat ve vícevláknových aplikacích má na starost výhradně programátor vyvíjející danou aplikaci, nikoliv tento nástroj, který slouží pro testování.

6.2.2 Inicializace

Stavový diagram, nad kterým `TransitionSmurf` pracuje je předán do konstruktoru třídy. Ten zkontroluje validitu stavového diagramu, uloží si ho jako privátní proměnnou a zaznamená graf do textového výstupu ve formě seznamu sousednosti 2.1.2. Důvod tohoto výstupu je zaznamenání stavového diagramu nad kterým systém pracuje pro kontrolu a také předání tohoto stavového diagramu aplikaci pro vyhodnocování výstupu 6.3, která jej potřebuje znát pro nalezení jeho cyklů. Poté zaznamená informaci o začátku běhu programu. Tato informace slouží jednak k oddělení seznamu sousednosti od ostatních záznamů, jednak k rozpoznání více běhů programu v jednom souboru. Takto zaznamenaný graf z obrázku 5.2, následovaný záznamem o začátku běhu programu by vypadal následovně:

`[<casova znacka>] [aplikace] [debug] [DiagramSmurf] __START__ :` q_1
`[<casova znacka>] [aplikace] [debug] [DiagramSmurf]` q_1 `:` q_2
`[<casova znacka>] [aplikace] [debug] [DiagramSmurf]` q_2 `:` q_2 q_3 q_5
`[<casova znacka>] [aplikace] [debug] [DiagramSmurf]` q_3 `:` q_4
`[<casova znacka>] [aplikace] [debug] [DiagramSmurf]` q_4 `:` q_3 q_5
`[<casova znacka>] [aplikace] [debug] [DiagramSmurf]` q_5 `:` q_3
`[<casova znacka>] [aplikace] [debug] [TransitionSmurf] Start of Run`

6.2.3 Zaznamenávání stavů

`TransitionSmurf` k zaznamenávání přechodů stavů používá open-source logovací knihovnu `BringAuto-Logger`. Tato knihovna je generickým rozhraním pro jiné logovací knihovny. Při implementaci této komponenty se využíval konkrétně `spdlog`[3]. Knihovna `BringAuto-Logger`

[2] Synchronizační chyba ve vícevláknových procesech, kdy přístupy k paměti nejsou správně synchronizovány a nastává nedeterministické chování [17].

[3] https://github.com/gabime/spdlog

umožňuje logovat na různých úrovních závažnosti. Tyto úrovně jsou již předdefinované a v knihovně chybí funkcionalita nějakou úroveň přidat a také možnost logovat různé úrovně do různých souborů. V optimálním případě by `TransitionSmurf` logoval do své vlastní úrovně a nemusel by poté explicitně značit, že se jedná o záznam právě této komponenty. Proto jsem ve veřejném repozitáři vznesl návrh na přidání těchto funkcionalit. `TransitionSmurf` prozatím využívá úroveň `DEBUG` pro logování změny stavů a `WARNING` v případě neúspěšného přechodu. Pokud funkce `inState` skončí neúspěchem, zaznamená se tato informace na úrovni `ERROR`. Formát logů o přechodech stavů:

```
[<cas>] [<aplikace>] [debug] [TransitionSmurf] Going to state <stav>
```

Špičaté závorky značí proměnnou.

6.3 Vyhodnocování výstupů

Aplikace která porovnává přechody stavů a vyhodnocuje správnost výstupu běhu systému se jmenuje `SmurfEvaluator`. Vstupem této aplikace jsou dva soubory, etalon a testovaný výstup systému, ke kterým se předává cesta v argumentech. `SmurfEvaluator` nejprve najde všechny existující cykly ve stavovém diagramu a poté soubory vyfiltruje, aby obsahovali pouze logy související s přechody stavů. Tyto přechody poté agreguje do cyklů. Poté nastává samotné porovnávání přechodů. Pokud aplikace přechody vyhodnotí jako stejné, ukončí se s návratovým kódem `0`. Agregované přechody lze také uložit pro manuální hledání chyb přepínačem `save-aggregated`.

6.3.1 Hledání cyklů ve stavovém diagramu

Načtení stavového diagramu a nalezení všech cyklů v něm má na starosti třída `CircuitFinder`. Tato třída přijímá v konstruktoru soubor s logy programu. Konstruktor tento soubor poté předává funkci `bool createAdjacencyMatrix(std::istream &srcFile);`, která si nejprve uloží záznamy obsahující seznam sousednosti, které poté dále zpracovává. Nejprve oindexuje každý vrchol a uloží jeho název do pole `stateNames_` na právě přidělený index a do asociativního vyhledávacího stromu uloží dvojici `(název_stavu, index)`. Tento vyhledávají strom je využíván pro správné vytvoření matice sousednosti, kdy je potřeba názvy stavů převést na indexy. Pole `stateNames_` je poté využíváno při samotném nalezení cyklu. Po načtení všech vrcholů je alokována matice sousednosti `adjacencyMatrix_` o velikosti počtu vrcholů obsahující `boolean` hodnoty `false`. Důvod použití matice je konstantní časová složitost přístupu k prvku $O(1)$ a také jednoduchost práce s indexy a minimalizace porovnávání řetězců. Poté se postupně prochází seznam sousednosti. Pro každý vrchol v seznamui je nastavena hodnota `true` na souřadnicích `[index_počáteční_vrchol][index_koncový_vrchol]`. Počáteční vrcholy, tedy vrcholy patřící k vrcholu `__START__`, nejsou přidávány do matice, ale jsou přidávány do vektoru `startingVertexes_`. Když je vytvořená matice sousednosti, konstruktor ještě alokuje paměť pro pole `blocked_` a matici `blockMatrix_`, odpovídající seznamu blokovaných vrcholů `B` v Johnsonově algoritmu 2.2.5.

Pro samotné vyhledání cyklů v grafu slouží funkce:

```
std::vector<std::vector<std::string>> find();
```

Tato funkce je implementací Johnsonova algoritmu popsaného v sekci 2.2.5. Rozdílem je, že nezačíná ve vrcholu s nejnižším indexem, protože jím může být pokaždé jiný vrchol a

vedla by tato implementace k nekonzistentnímu chování, protože by cykly začínaly v různém stavu, což by způsobilo různé chování při agregaci. Proto tato funkce začíná hledat z prvního vrcholu v uloženého ve vektoru `startingVertexes_`. Do tohoto vektoru jsou, po nalezení cyklů začínajících ve vrcholu v, vloženy všechny jeho sousední vrcholy. Funkce navíc po nalezení cyklů s počátečním vrcholem v odstraní všechny hrany, pro které je vrchol v koncovým, aby bylo zabráněno opětovné nalezení stejného cyklu, pouze začínajícího v jiném vrcholu. Funkce vrací seznam všech cyklů, uložených jako vektor vektorů složených z řetězců reprezentujících názvy stavů v cyklu.

Třída dále obsahuje privátní funkce, které odpovídají funkcím Johnsonova algoritmu `UNBLOCK` a `CIRCUIT`:

```
void unblock(const int &vertex);
bool circuit(const int &vertex);
```

Vstupem těchto funkcí je index vrcholu nad kterým se má požadovaná funkce vykonat.

6.3.2 Filtrování souborů

Pro filtrování logů je implementována třída `Filter`. Tato třída obsahuje tři funkce. První dvě funkce jsou velmi podobné:

```
static std::string findDiagramSmurfLog(std::istream &srcFile);
static std::string findNextTransitionLog(std::istream &srcFile);
```

Obě funkce čtou logy a zahazují všechny, které neobsahují klíčové slovo. Když najdou záznam obsahující klíčové slovo, vracejí tento záznam. Funkce `findDiagramSmurfLog` je používána při načítání grafu pro hledání cyklů a jejím klíčovým slovem je `[DiagramSmurf]`. Klíčovým slovem pro funkci `findNextTransitionLog` je `[TransitionSmurf]`. Tato funkce slouží pro hledání záznamů obsahující přechody stavů.

Poslední funkce:

```
static std::vector<std::string>
                createTransitionLogVector(std::istream &srcFile);
```

přijímá soubor obsahující logy a pomocí funkce `findNextTransitionLog` všechny logy přechodů uloží do vektoru, který po nalezení všech logů vrací.

6.3.3 Agregace záznamů o přechodech stavů

Důvod k agregaci stavů a způsobu zpracování koncového cyklu je podrobně popsán v podsekci 5.2.2. Agregaci logů přechodů stavů má na starosti třída `CircuitAggregator`. Konstruktor této třídy nejprve volá funkce třídy `CircuitFinder` pro nalezení cyklů ve stavovém diagramu. Seznam cyklů si uloží do privátní proměnné `circuitList_`. Pro samotnou agregaci přechodů v souboru slouží funkce:

```
std::vector<std::string>
                createAggregatedVector(std::istream &sourceLogFile);
```

Vstupem této funkce je soubor obsahující záznamy o přechodech stavů. Tato funkce nejprve volá funkci `createTransitionLogVector` a vrácený vektor uloží do privátní proměnné `transitionLogVector_`. Index zpracovávaného záznamu v tomto vektoru je uložen v privátní proměnné `transitionIndex_`. Funkce vrací vektor `aggregatedLogVector`, obsahující agregované záznamy. Základní funkce pro hledání cyklů v záznamech je:

```
long getCircuit(long lastCircuit);
```

Tato funkce postupně prochází každý cyklus ze seznamu `circuitList_` a porovnává, zda se stavy cyklu rovnají stavům v právě zpracovávaných záznamech, tedy záznamech od indexu `transitionIndex_` až po `transitionIndex_ + delkaCyklu`. Těchto cyklů může nalézt více, proto po prohledání všech možností zjistí, který z nalezených cyklů má největší délku a vrací jeho index. Není-li nalezen žádný cyklus, vrací funkce záporné číslo odpovídající jedné ze tří možností:

1. `NOT_FOUND` – právě zpracovávaný záznam není počátkem žádného cyklu,
2. `END_FOUND` – kdykoliv při prohledávání byl nalezen začátek/konec běhu,
3. `NO_CIRCUITS` – ve stavovém diagramu nejsou žádné cykly.

Funkce následně vrácenou hodnotu zpracovává následovně:

1. `NOT_FOUND` – záznam uloží do vektoru `aggregatedLogVector`, inkrementuje `transitionIndex_` o 1 a nastaví hodnotu `currentCircuit` na `NOT_FOUND`.
2. `END_FOUND`, aktuální záznam je začátek běhu – k záznamu o začátku běhu přidá poznámku „`- Aggregated`“ a vloží jej do vektoru `aggregatedLogVector`, poté inkrementuje `transitionIndex_` o 1 a nastaví hodnotu `currentCircuit` na `START_FOUND`.
3. `NO_CIRCUITS` – uloží všechny záznamy do vektoru `aggregatedLogVector` a funkce končí.
4. Kladné číslo, index nalezeného cyklu – nejprve zjistí, zda je právě nalezený cyklus stejný jako předchozí cyklus uložený v proměnné `currentCircuit`. Pokud není, zaznamená ho do vektoru `aggregatedLogVector` ve formátu popsaném níže a uloží jeho index do proměnné `currentCircuit`. Nakonec inkrementuje `transitionIndex_` o velikost nalezeného cyklu.
5. `END_FOUND`, aktuální záznam je přechod stavu – funkce zjistí, zda je v proměnné `currentCircuit` uložen nějaký cyklus. Pokud ano, porovná zda jsou poslední záznamy stavů stejné jako stavy v tomto cyklu. Jestliže nejsou stejné, nebo v proměnné `currentCircuit` není uložený žádný cyklus, jsou tyto záznamy vloženy do vektoru `aggregatedLogVector`. Nakonec je inkrementována hodnota `transitionIndex_` o počet těchto záznamů a `currentCircuit` nastaven na `END_FOUND`.

Záznam o cyklu obsahuje časovou značku záznamu prvního přechodu cyklu. Dále obsahuje index nalezeného cyklu, ten však slouží pro orientaci v záznamech při manuální kontrole. Tyto indexy mohou být různé pro různé běhy programu z důvodu implementace stavového diagramu pomocí binárního vyhledávacího stromu. Další informací jsou stavy, které cyklus obsahuje, tyto již slouží k porovnávání. Příklad agregovaných záznamů z výpisu 5.1 je uveden ve výpisu 6.2.

```
[<casova znacka>] [<aplikace>] [debug] [TransitionSmurf] Going to state q_1
[<casova znacka>] In circuit 0: [q_2, ]
[<casova znacka>] In circuit 1: [q_3, q_4]
[<casova znacka>] In circuit 2: [q_3, q_4, q5, ]
```

Výpis 6.2: Příklad agregovaných záznamů aplikací SmurfEvaluator

6.3.4 Porovnávání výstupů

Porovnávání výstupu implementuje třída `LogsComparer`. Její jediná veřejná funkce:

```
bool compareFiles(std::istream &etalonFile,
                  std::istream &comparedFile,
                  std::string saveAggregatedPath);
```

přijímá etalon, výstup běhu systému, který bude porovnáván a cestu pro uložení agregovaných logů. Agregované logy jsou uložené do složky dané argumentem `saveAggregatedPath` v souborech `etalon` a `compared`. Soubor s výstupem běhu systému může obsahovat více různých běhů, tyto běhy jsou oddělené záznamem `Start of Run`, který je sem vložen komponentou `TransitionSmurf`. Etalon smí obsahovat výstup pouze jednoho běhu systému.

Funkce nejprve agreguje oba soubory funkcí `createAggregatedVector` a vrácené vektory poté porovnává. Při nalezení záznamu o začátku běhu je na standardní výstup vytisknuto: `Run number x:`, pro oddělení vyhodnocení jednotlivých běhů. Poté jsou všechny záznamy ve vektorech spolu porovnávány. V záznamech se porovnává každé slovo mimo časovou značku. V záznamech o cyklech se také neporovnává číslo cyklu, protože toto porovnání by vedlo k falešně negativním výsledkům porovnání. Pokud jsou všechny logy běhu stejné, je k danému běhu na standardní výstup vytisknuto `OK`. Všechny záznamy, které nejsou stejné, jsou také vytisknuty k danému běhu. Je li jeden z logů delší, jsou tyto přesahující záznamy také vytisknuty a porovnány s prázdným řetězcem. Porovnávání každého běhu probíhá odděleně. Jestliže mají všechny běhy stejné stejné přechody stavů, funkce vrací `true`, obsahuje-li alespoň jeden z běhů neekvivalentní přechody je vrácena hodnota `false`. Podle této hodnoty celá aplikace vrací úspěch nebo neúspěch.

Třída `LogsComparer` také obsahuje pomocné privátní funkce pro samotné porovnávání záznamů a validaci etalonu.

Příklad výstupu porovnání souboru, který obsahuje dva běhy, kde první běh je stejný jako etalon, ale v druhém je o jeden přechod méně:

```
Run number 1:
    OK
-------------------------------------
Run number 2:
Logs aren't equal:
  Etalon: [2022-01-01 15:31:35.232] [sampleApp] [debug] Going to state A
  Compared: [2022-02-01 10:01:10.456] [sampleApp] [debug] Going to state B
Logs aren't equal:
  Etalon: [2022-01-01 15:31:35.322] [sampleApp] [debug] Going to state B
  Compared:
-------------------------------------
```

6.4 Automatizace testů

Skript v jazyce `Python` pro automatické spouštění testů a vyhodnocování přechodů systému `StateSmurf`, sloužící pro integrační testování se jmenuje `CompareScenarios.py`. Tento skript přijímá následující argumenty:

- `--scenario`: cesta k souboru se scénáři,
- `--executable`: cesta k souboru spouštějící systém,

- `--evaluator`: cesta k spustitelnému souboru `smurf_evaluator`,
- `--create-etalons`: nepovinný argument pro vytvoření etalonů,
- `--output`: nepovinný argument pro změnu složky pro výstup skriptu.

Tento skript nejprve ve složce obsahující soubor se scénáři, případně ve složce dané argumentem `output`, vytvoří složky:

- `etalons` – složka, ve které jsou umístěny etalony, tato složka je změněna pouze, když je skript spouštěn s argumentem `create-etalons`,
- `output` – složka, do které se ukládají výstupy jednotlivých běhů aplikace,
- `aggregated_output` – složka, do které jsou uloženy agregované výstupy z aplikace `SmurfEvaluator`,
- `evaluator_output` – složka pro výstup porovnávání aplikací `SmurfEvaluator`, tyto výstupy slouží pro hledání případných chyb.

Skript nejprve spustí příkazy pro přípravu testovacího prostředí a poté postupně spouští jednotlivé testovací scénáře. Každý testovací scénář obsahuje název, argumenty pro spuštění a timeout, tedy čas v sekundách, po jehož uplynutí je danému běhu poslán signál `SIGINT`, skript poté čeká 15 sekund, aby běžící proces stihl signál zpracovat a provést správné ukončení. Po uplynutí těchto 15 sekund je zaslán signál `SIGKILL`. V případě použití argumentu `create-etalons` je výstup běhu daného scénáře uložen do složky `etalons`. V opačném případě je uložen do složky `output` a je porovnáván se stejnojmenným výstupem ze složky `etalons`. V případě, že tyto soubory nejsou ekvivalentní, je změněna hodnota proměnné `tests_passed` na `False`. Na konci každého běhu proběhne úklid, který může například uvést testovací prostředí do původního stavu.

Po provedení všech testovacích scénářů jsou volány příkazy pro vyčištění, kterým může být například vypnutí testovacího prostředí.

6.4.1 Testovací scénáře

Testovací scénáře jsou uloženy v souboru ve formátu `JSON`. Příklad tohoto souboru je popsán na 6.3. V tomto příkladě se prostředí nastavuje pomocí nástroje `docker-compose`[4], který na začátku zapne určené služby popsáné v souboru `docker-compose.yml`, po každém běhu je restartuje a na konci tyto služby opět vypne. Aplikace popsaná v tomto příkladu přijímá argumenty `--foo` obsahující celočíselnou hodnotu a přepínač `--bar`.

```
1 {
2     "setup" : [ "docker-compose up" ],
3     "between_runs" : [ "docker-compose restart" ],
4     "scenarios" : [
5         {
6             "name" : "testFoo",
7             "timeout" : 15,
8             "arguments" : {
9             "--foo" : "10",
```

[4] https://docs.docker.com/compose/

```
10              "--bar" : ""
11              }
12          },
13          {
14              "name" : "testBar",
15              "timeout" : 5,
16              "arguments" : {
17              "--bar" : ""
18              }
19          }
20      ],
21      "cleanup" : [ "docker-compose down" ]
22  }
```

Výpis 6.3: Příklad souboru obsahující testovací scénáře sloužící pro automatizaci testů.

6.5 Distribuce balíčku

Celý výše popsaný balíček nástrojů je volně dostupný ve veřejném repositáři `GitHub`[5] a také na přiloženém médiu. Všechny postupy pro práci s tímto balíčkem jsou popsané v souborech `README.md` vždy uložených ve složce s ním související komponentou. Popis instalace balíčku je v kořenovém adresáři.

Všechno potřebné pro instalaci tohoto balíčku je konfigurováno pomocí nástroje `CMake`[6]. Pro sledování závislostí je využívána knihovna `CMLIB`[7].

6.6 Testování komponent

Všechny komponenty balíčku StateSmurf mají pro každou svou třídu definovanou sadu jednotkových testů implementovaných pomocí nástroje `GoogleTest`. Tyto testy se nachází v podsložce `test/` každé z komponent. Pro jejich konfiguraci je potřeba použít `CMake` přepínač `-DBRINGAUTO_TESTS=ON`. Samotné testy se spouštějí příkazem `ctest`.

Dále je součástí balíčku aplikace `SmurfExampleApp`, která slouží k testování celého systému StateSmurf, ale také jako opora pro integraci a ukázka použití jeho jednotlivých komponent. Tato aplikace obsahuje v podsložce `test/` všechny náležitosti pro použití automatických testů StateSmurf.

[5] https://github.com/Melky-Phoe/StateSmurf
[6] https://cmake.org/
[7] https://github.com/cmakelib/cmakelib

Kapitola 7

Nasazení a testování

Testování balíčku `StateSmurf` probíhalo na softwaru partnerské společnosti BringAuto. Integrace do těchto systémů probíhala téměř od začátku vývoje balíčku a komponenty byly často ovlivněny specifickými požadavky konkrétního softwaru. Všechny tyto specifické požadavky jsem se snažil řešit co nejvíce univerzálně, aby byl balíček nástrojů použitelný i mimo partnerskou společnost.

7.1 Nástroj Virtual vehicle utility

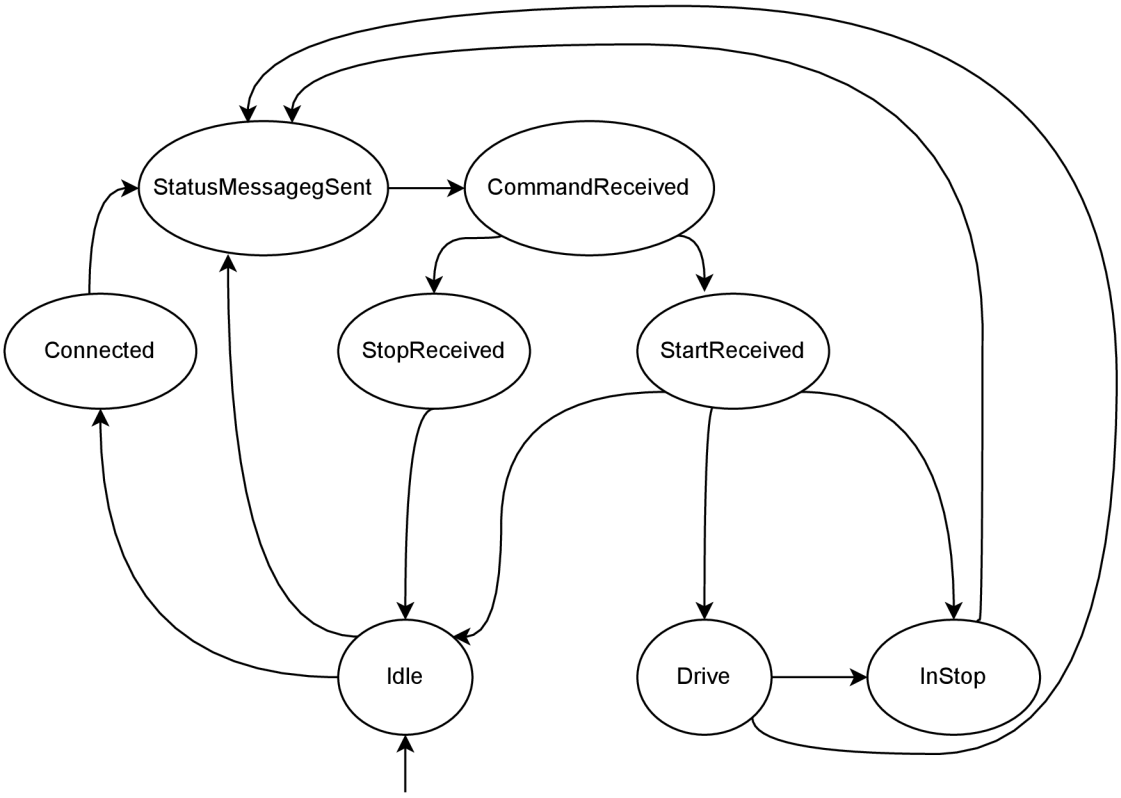
První software, do kterého byl StateSmurf integrován, je `Virtual vehicle utility`. Jedná se o software pro simulaci autonomního vozidla partnerské společnosti. Virtual vehicle utility simuluje přesné chování vozidla za účelem testování zbylých části systému `BringAuto Fleet`, což je nástroj sloužící pro správu autonomních vozidel. Vozidlo komunikuje se serverem `BringAuto Fleet Management` (v textu bude dále používána zkratka BAFM, uživatelské rozhraní serveru je na obrázku 7.2), od kterého vozidlo přijímá zprávy, obsahující mise. Každá mise obsahuje seznam zastávek v pořadí, ve kterém má dané vozidlo tyto zastávky projet. Program simuluje jízdu vozidla do určené zastávky po předem definované cestě. Přitom posílá na server stavové zprávy. Ty obsahují údaje o poloze vozidla, rychlosti, momentálním stavu a kapacitě baterie.

Detailní popis komunikačního protokolu lze najít na veřejném úložišti Google partnerské společnosti[1]. V tomto protokolu je zastaralý a zjednodušený stavový diagram pro virtuální vozidlo, o kterém bylo při integraci zjištěno, že je nedostačující. Tento diagram nijak nezohledňuje komunikaci vozidla se serverem a neobjevoval tedy závady způsobené chybou komunikace, která je pro vozidlo zásadní. Nový stavový diagram softwaru je na obrázku 7.1. Tento diagram je ta část, podle které se bude tento software testovat nástrojem `StateSmurf`.

7.1.1 Chování virtuálního vozidla

Vozidlo začíná ve stavu `Idle`. Po připojení k serveru BAFM přechází do stavu `Connected`. Dále odesílá stavovou zprávu na server a přechází do stavu `StatusMessageSent`, po přijetí odpovědi od serveru přejde do stavu `CommandReceived`. Odpověd zpracuje a podle obsahu přejde do jednoho ze dvou stavů, `StopReceived` ze kterého přechází do stavu `Idle`, nebo `StartReceived` ve kterém dále podle své polohy a obsahu mise rozhodne, do kterého ze

[1] https://drive.google.com/drive/folders/1ZE9VRs86QtP6GqTJBl6vRJLmkh1lTEc5



Obrázek 7.1: Stavový diagram aplikace Virtual vehicle utility

stavů `Idle`, `Drive`, nebo `InStop` přejde. Poté vozidlo znovu odešle stavovou zprávu a cyklus se opakuje znovu od stavu `StatusMessageSent`.

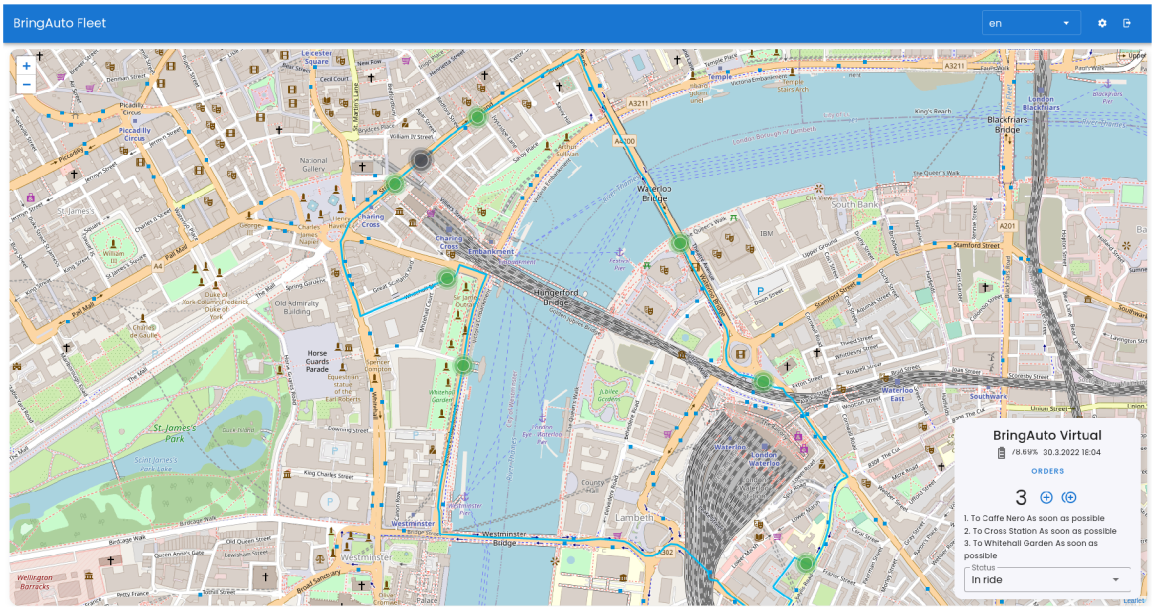
7.1.2 Postup při integraci

V této podsekci je popsán postup integrace balíčku StateSmurf do Virtual vehicle utility a přehled problémů, které při integraci nastaly.

Integrace do virtuálního vozidla začala hned po vytvoření komponent DiagramSmurf a TransitionSmurf. Již při integraci bylo objeveno spoustu chyb jak v těchto komponentách, tak ve virtuálním vozidle. Zpočátku po zalogování informace o přechodu stavu TransitionSmurf logoval také potvrzení, že ke změně stavu došlo. Toto způsobovalo zahlcování logů redundantními informacemi a vedlo ke snížení přehlednosti. V této fázi integrace správnost běhu byla ověřována pouze manuálním čtením výstupních logů.

Když se po úpravách softwaru v různých běžích aplikace přestaly vyskytovat neúspěšné přechody, začala se správnost logů kontrolovat pomocí SmurfEvaluatoru. Zde však často nastával problém, kdy aplikace nezaznamenala stejný průběh přechodů stavů pro každý běh se stejnými vstupy. Důvodem chyby bylo spouštění testovacího BAFM ve virtuálním prostředí `Docker` zároveň s aplikací Virtual vehicle. Více podrobností o spouštění testů je v sekci 7.1.3. Vzhledem k různému vytížení procesoru trvalo různou dobu, než BAFM reportoval aplikaci jeho misi a tedy vozidlo na začátku čekalo různou dobu. Z tohoto důvodu byl celý běh zpožděn a průchod stavy nebyl ekvivalentní, přestože logika běhu byla stejná.

Tyto chyby vedly k nápadu agregovat stavy do `superstavů`. Každý superstav je nějaký cyklus ve stavovém diagramu. Přechody stavového diagramu popsaného v 7.1 jsou takto



Obrázek 7.2: **BringAuto Fleet Management** sloužící pro přehled o flotile autonomních vozidel a vytváření objednávek.

interpretovatelné mimo jiné jako superstavy `ČekáníBezMise`, `Jízda`, `ČekáníVZastávce`. Superstavy agreguje až SmurfEvaluator, etalon tedy zůstává stejný, tedy dostatečně podrobný pro manuální validaci běhu a debugging. Díky tomuto přístupu k porovnávání lze chování aplikace ověřit bez falešně negativních testovacích případů zaviněných zpožděním. Zároveň nebyl nalezen případ, kdy by agregace superstavů skryla chybu v běhu.

7.1.3 Průběh testů

Aby se mohlo chování Virtual vehicle utility testovat, bylo potřeba nejdříve připravit prostředí. Na přípravu prostředí slouží nástroj `Bringauto Etna`. Jedná se o virtuální vývojovou platformu, která pomocí `Dockeru` spouští všechny komponenty potřebné pro autonomní vozidla. Může spouštět také samotný Virtual vehicle utility, to však pro účely tohoto testování není vhodné.

Pro automatizaci testů byl vytvořen skript v jazyce `Python` popsaný v sekci 6.4. Ten při přípravě prostředí spouští nástroj Bringauto Etna pomocí příkazu `docker-compose`, podobně jak je popsáno ve výpisu 6.3. Poté spouští scénáře pro tři typy misí na třech různých tratích. Celková doba běhu těchto testů je přibližně 10 minut.

7.1.4 Nalezené chyby

Při integraci bylo nalezeno několik chyb. Mezi nejzásadnější chyby patří to, že vozidlo používalo funkce pro asynchronní komunikaci. Komunikační protokol vozidla se serverem však funguje jako synchronní komunikace typu požadavek/odpověď. Vozidlo se dále pokoušelo komunikovat dříve než se připojilo. Další chybou byl souběh, kdy byla zároveň ve dvou vláknech odesílaná stavová zpráva s různým obsahem. Tyto zprávy poté vedly ke špatnému zpracování na serveru BAFM a virtuálnímu vozidlu se tím změnila mise.

7.2 Integrace do aplikace Modulog

Balíček `StateSmurf` byl také nasazen do aplikace Modulog. Ze zpětné vazby o integraci bylo zjištěno, jaké chyby StateSmurf dokáže odhalit a také získal nějaké podněty pro vylepšení. Celá zpětná vazba je v příloze B.

Nejkritičtější chybou, kterou StateSmurf odhalil, byl souběh při uvolňování paměti, při kterém docházelo k chybě paměti a aplikace byla ukončená se zprávou `SEGFAULT`. Další chyby se týkaly například nepozornosti při odlaďování kódu, kdy některé části kódu byly zakomentovány a programátor na ně zapomněl.

V nedostatcích byla hlavně vytknuta nutnost předávat instanci třídy TransitionSmurf do jednotlivých funkcí, které ji chtějí využít, nebo také práce s řetězci. Nápravu těchto nedostatků mám v plánu implementovat co nejdříve, protože je to dle mého názoru největší nedostatek tohoto balíčku.

Kapitola 8

Závěr

Tato práce se zabývá problematikou použití stavového diagramu pro testování programů. Cílem této práce bylo vytvořit balíček nástrojů pro vyhodnocení správnosti běhu systému na základě stavového grafu. Tento cíl byl splněn. Problematika stavových diagramů a diskrétních grafů je popsána v prvních dvou kapitolách. Pro definici stavového diagramu a jeho přechodů byla implementována knihovna DiagramSmurf. Při běhu systému se stavy ukládají v podobě textových záznamů pomocí nástroje TransitionSmurf. Přehrání nahraných stavů a jejich vyhodnocení je prováděno pomocí SmurfEvaluator. Výsledný systém navíc implementuje nástroj pro automatizaci testování. Veškeré zdrojové kódy a plakát reprezentující výsledky této práce jsou uloženy na přiloženém médiu. Jako míru splnění cíle považuji nasazení systému StateSmurf do dvou různých systémů, ve kterých odhalila několik chyb, které žádný jiný dříve použitý testovací nástroj neodhalil.

Výsledný systém je přínosný pro jeho použití v partnerské společnosti BringAuto, kde byl prozatím nasazen jen do jedné části pro správu flotily autonomních vozidel, ale bude v budoucnu nasazen do všech zbylých komponent. Systém je navíc volně dostupný na portálu GitHub s open-source licencí a věřím, že až se prokáže býti užitečný i ve zbylých komponentách, bude využíván i jinými společnostmi. Přínosem této práce pro mě byla zkušenost s vývojem komplexního systému, který je tvořený několika komponentami, které spolu spolupracují. Dále jsem nabyl zkušenosti s hledáním správných algoritmů, které vykonají požadovaný úkol v nejrychlejším čase.

Při psaní této technické zprávy jsem objevil několik nedostatků, které plánuji v systému StateSmurf vylepšit. Prvním návrhem na zlepšení je vytváření instance třídy TransitionSmurf podle návrhového vzoru singleton (jedináček), díky kterému by byla odstraněna nutnost předávat instanci této třídy do všech částí programu. TransitionSmurf by také mohl ukládat stavový diagram v podobné formě, jak to dělá SmurfEvaluator při hledání cyklů, což by snížilo výpočetní nároky pro přechody stavů. Dalším návrhem je vytvoření společného rozhraní pro komponenty DiagramSmurf a TransitionSmurf, díky kterému bude kód programu pracovat pouze s jedinou knihovnou, čímž se zjednoduší práce s integrací systému StateSmurf. Navíc toto rozhraní přispěje k modularitě celého systému. Dále plánuji přepsat knihovnu DiagramSmurf do moderní `C++20`, která umožní vytvářet graf již při kompilaci programu a méně zatěžovat systém při běhu.

Literatura

[1] *All You Need to Know about State Diagrams* [online]. Visual Paradigm [cit. 2022-4-23]. Dostupné z: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/about-state-diagrams/.

[2] *Asymptotic Analysis: Big-O Notation and More* [online]. Parewa Labs Pvt [cit. 2022-4-11]. Dostupné z: https://www.programiz.com/dsa/asymptotic-notations.

[3] *Googletest Primer* [online]. Google Inc. [cit. 2022-05-04]. Dostupné z: https://google.github.io/googletest/primer.html.

[4] *The Valgrind Quick Start Guide* [online]. Valgrind™ Developers [cit. 2022-05-04]. Dostupné z: https://valgrind.org/docs/manual/quick-start.html#quick-start.intro.

[5] Beck, K. *Test-driven development: by example*. 1. vyd. Addison-Wesley Professional, 2002. ISBN 0-321-14653-0.

[6] Braica, P. *State Machines, in C++* [online]. Code Project, listopad 2016 [cit. 2022-4-25]. Dostupné z: https://www.codeproject.com/Articles/1155557/State-Machines-in-Cplusplus.

[7] Demel, J. *Grafy a jejich aplikace*. Vyd. 1. Praha: Academia, 2002. ISBN 80-200-0990-6.

[8] Graham, D. *Foundations of software testing : ISTQB certification*. Fourth edition. Andover, Hampshire: Cengage, 2020. ISBN 978-1-4737-6479-8.

[9] Honzík, J. M. *ALGORITMY IAL Studijní opora* [online]. 14-N. Brno, leden 2014 [cit. 2022-04-09]. Dostupné z: https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FIAL-IT%2Flectures%2FOpora-IAL-2014-verze-14-N.pdf&cid=13948.

[10] Hopcroft, J. E., Motwani, R. a Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 0321455363.

[11] Johnson, D. B. Finding All the Elementary Circuits of a Directed Graph. *SIAM Journal on Computing*. 1975, sv. 4, č. 1, s. 77–84. DOI: 10.1137/0204007.

[12] Křena, B. a Kočí, R. *Úvod do softwarového inženýrství IUS Studijní opora* [online]. Brno, prosinec 2010 [cit. 2022-05-03]. Dostupné z: https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FIUS-IT%2Ftexts%2FIUS_opora.pdf&cid=13367.

[13] KŘÍŽ, P. *Etalony. Etalon je charakterizovaný takto: Druhy etalonů:* [online]. DocPlayer.cz, 2017 [cit. 2022-4-12]. Dostupné z: https://docplayer.cz/39748531-Etalony-etalon-je-charakterizovany-takto-druhy-etalonu.html.

[14] LAFRENIERE, D. *State Machine Design in C++* [online]. Code Project, únor 2019 [cit. 2022-4-25]. Dostupné z: https://www.codeproject.com/Articles/1087619/State-Machine-Design-in-Cplusplus-2.

[15] MEDUNA, A. a LUKÁŠ, R. *Formální jazyky a překladače IFJ Studijní opora* [online]. Brno, leden 2006, revize 2009-2015 [cit. 2022-04-20]. Dostupné z: https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FIFJ-IT%2Ftexts%2FOporaIFJ.pdf&cid=13981.

[16] NAGAPPAN, M. a VOUK, M. A. Abstracting log lines to log event types for mining software system logs. In: *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. 2010, s. 114–117. DOI: 10.1109/MSR.2010.5463281.

[17] NETZER, R. H. B. a MILLER, B. P. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems*. 1992, sv. 1.

[18] OLINER, A., GANAPATHI, A. a XU, W. Advances and Challenges in Log Analysis: Logs Contain a Wealth of Information for Help in Managing Systems. *Queue*. New York, NY, USA: Association for Computing Machinery. dec 2011, sv. 9, č. 12, s. 30–40. DOI: 10.1145/2076796.2082137. ISSN 1542-7730.

[19] RUSHDI, A. a ALSALAMI, O. Multistate Reliability Evaluation of Communication Networks via Multi-Valued Karnaugh Maps and Exhaustive Search. In:. únor 2021, s. 114–136. DOI: 10.9734/bpi/aaer/v1/2370E. ISBN 978-93-90149-84-1.

[20] STALLMAN, R. M. a PESCH, R. H. *Using GDB: A Guide to the GNU Source-level Debugger: GDB Version 4.0, July 1991*. Free software foundation, 1991.

[21] TARJAN, R. Depth-first search and linear graph algorithms. *SIAM journal on computing*. SIAM. 1972, sv. 1, č. 2, s. 146–160.

[22] TARJAN, R. Enumeration of the elementary circuits of a directed graph. *SIAM Journal on Computing*. SIAM. 1973, sv. 2, č. 3, s. 211–216.

[23] TIERNAN, J. C. An efficient search algorithm to find the elementary circuits of a graph. *Communications of the ACM*. ACM New York, NY, USA. 1970, sv. 13, č. 12, s. 722–726.

[24] VANÍČEK, J., PAPÍK, M., PERGL, R. a VANÍČEK, T. *Teoretické základy informatiky*. 1. vyd. Praha: Kernberg, 2007. ISBN 978-80-903962-4-1.

[25] WEINBLATT, H. A new search algorithm for finding the simple cycles of a finite directed graph. *Journal of the ACM (JACM)*. ACM New York, NY, USA. 1972, sv. 19, č. 1, s. 43–56.

[26] WEST, D. B. et al. *Introduction to graph theory*. Prentice hall Upper Saddle River, 2001.

Příloha A

Johnsonův algoritmus pro hledání cyklů v orientovaném grafu

```
1  begin
2      integer list array A_k(n), B(n); logical array blocked(n); integer s;
3      logical procedure CIRCUIT (integer value v);
4          begin logical f;
5              procedure UNBLOCK (integer value u);
6                  begin
7                      blocked(u) := false;
8                      for w∈B(u) do
9                          begin
10                             delete w from B(u);
11                             if blocked(w) then UNBLOCK(w);
12                         end
13                 end UNBLOCK
14             f := false;
15             stack v;
16             blocked(v) := true;
17     L1:     for w ∈ A_k(v) do
18                 if w = s then
19                     begin
20                         output circuit composed of stack followed by s;
21                         f := true;
22                     end
23                 else if ¬blocked(w) then
24                     if CIRCUIT(w) then f := true;
25     L2:     if f then UNBLOCK(v)
26             else for w ∈ A_k(v) do
27                     if v ∉ B(w) then put v on B(w);
28             unstack v;
29             CIRCUIT := f;
30         end CIRCUIT;
31     empty stack;
32     s := 1;
```

```
33      while s < n do
34          begin
35              A_k := adjacency structure of strong component K with least
36                     vertex in subgraph of G induced by {s, s+1, ..., n};
37              if A_k ≠ ∅ then
38                  begin
39                      s := least vertex in V_k;
40                      for i ∈ V_k, do
41                          begin
42                              blocked(i) := false;
43                              B(i) := ∅;
44                          end;
45      L3:             dummy := CIRCUIT(s);
46                      s := s+1;
47                  end
48              else s := n;
49          end
50  end;
```

Výpis A.1: Johnsonův algoritmus na hledání cyklů popsán v pseudokódu [11]

Příloha B

Zpětná vazba

Zpětná vazba z integrace balíčku StateSmurf do aplikace Modulog:

Knihovna StateSmurf je velmi užitečná pro integrační testy jiných aplikací. V mé aplikaci pomohla najít několik chyb, které mohly být v budoucnu kritické a také upozornila na to, že jsem během vývoje zakomentoval části kódu pro ladící účely, bez kterých kód nefungoval přesně podle specifikace.

Její implementace do již existujícího řešení nebyla také nějak složitá, pouze pár drobností bylo komplikovanějších, protože jsem knihovnu nasazoval v době jejího nedokončeného vývoje. Zatím vidím nevýhodu pouze v tom, že při integraci je potřeba vytvořit instanci třídy, která je následně nutná předávat do všech míst aplikace, kde se budou měnit stavy. To u velké aplikace může být poměrně zdlouhavé. Také mi chybí možnost prázdné implementace - aktuálně je potřeba v produkci pomocí maker odstraňovat všechna místa, kde je StateSmurf využit, protože se využívá pouze pro testování. Do budoucna by bylo lepší udělat verzi, kde se pomocí přepínače zvolí, jestli jsou testy aktivovány či nikoliv a na základě toho se při překladu vygenerují těla funkcí nebo ne (v aplikaci poté testovací aplikace zůstane, ale všechna volání její funkcí nebudou nic dělat).