FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Aplikace pro podporu výuky ATPG algoritmů
Student:	Bc. Martin Fabík
Vedoucí:	Ing. Martin Daňhel, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

Cílem diplomové práce je navrhnout a realizovat funkční webovou aplikaci pro podporu výuky předmětu NI-TSP.

Aplikace bude poskytovat zejména tyto funkcionality:
- přihlášení pomocí Shibboleth ČVUT,
- vytvoření a správa výukové místnosti,
- režimy výuky (trenažér studenta / vedená výuka),
- možnost uložení rozpracovaného postupu,
- animované krokované průchody testovacím algoritmem (D-Algoritmus, volitelně např. FAN či PODEM),
- možnost tvorby vlastních výukových obvodů s následnou možností nahrání do výukového prostředí v přenositelném formátu.

Postupujte v těchto krocích:
- proveďte rešerši a analýzu podobných aplikací,
- v návrhu zamýšlené aplikace se zaměřte především na případ použití ve výuce,
- navrhněte a implementujte serverovou část aplikace,
- navrhněte a implementujte klientskou aplikaci pro webové prohlížeče,
- vhodnými postupy otestujte a ověřte správnost implementace, zejména s ohledem na implementované algoritmy pro testování číslicového návrhu,
- zhodnoťte použitelnost výsledného prototypu aplikace, navrhněte způsob uvedení do budoucího provozu.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 15. srpna 2023 v Praze.

Diplomová práce

APLIKACE PRO PODPORU VÝUKY ATPG ALGORITMŮ

Bc. Martin Fabík

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Daňhel Martin, Ph.D.
9. května 2024

Odkaz na tuto práci: Fabík Martin. *Aplikace pro podporu výuky ATPG algoritmů.* Diplomová práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2024.

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Chtěl bych poděkovat především vedoucímu práce Ing. Martinu Daňhelovi, Ph.D., za motivaci a cenné připomínky, které mi poskytl v průběhu vypracování této závěrečné práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, avšak pouze k nevýdělečným účelům. Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 9. května 2024

Abstrakt

Cílem práce je analýza a vytvoření výukové pomůcky, která studentům i vyučujícím pomůže při vizualizaci a krokování základních algoritmů pro hledání testovacích vektorů pro logické obvody. V současné chvíli sice existují programy, které pomáhají hledat testovací vektory pro daný obvod, nicméně neexistuje nástroj, který by byl schopen představit práci takových algoritmů po jednotlivých krocích a vizuálně. Autor této práce se v úvodní části zaměřuje na rozbor stávajících možností, které se uplatňují ve výuce i jinde, a dále pak návrhem a realizací nové pomůcky, která studentům lépe přiblíží jednotlivé principy při nalézání testovacích vektorů.

Klíčová slova ATPG, simulace, D-algoritmus, logický obvod, logické hradlo, OpenSwoole

Abstract

The goal of this work is to analyze and create a teaching tool that will help students and teachers in visualizing and stepping basic algorithms for finding test patterns for logical circuits. At the moment, although there are tools that help to find test vectors for a given circuit, there is no such tool that is able to represent the working of such algorithms step by step and visually. The author of this thesis begins with analysis of existing options that are used in the classroom and even elsewhere, and then goes on to design and implement a new tool that will better expose students to the various principles of finding test patterns.

Keywords ATPG, simulation, D-algorithm, logical circuit, logical gate, OpenSwoole

Seznam zkratek

ATPG	Automatic test pattern generation
CLI	Command line interface
CUT	Circuit under test
DI	Dependency injection
DUT	Design under test
FAN	Fan-out oriented
IdP	Identity provider
PI	Primary input
PO	Primary output
SAML	Security Assertion Markup Language
SP	Service provider
SSO	Single Sign On
WS	WebSocket
WSL	Windows Subsystem for Linux
WSS	WebSocket Secure

Úvod

V současné době sice existují mnohé nástroje poskytující generování testovacích vektorů, takzvané ATPG nástroje (z anglického *Automated Test Pattern Generator*), které umí takové vektory najít a vyhodnotit pro kombinační obvody (například `Atalanta`[1], `FAN_ATPG`[2] a jiné), nicméně jedná se primárně o nástroje s rozhraním příkazové řádky (*CLI*), které tak neposkytují adekvátní vizuální reprezentaci vhodnou pro výuku na školách. Další nástroje používané ve výuce, jako například Java applet pro vizuální reprezentaci obvodů[3], jsou pro studenty názornější, avšak přináší s sebou další limitace, například nemožnost upravovat již předdefinované obvody.

V současnosti tak probíhá výuka převážně formou cvičení u tabule, kdy se však tento způsob jeví jako velmi náročný vzhledem k častým změnám, které se na testovaném obvodu projevují v průběhu testování. Následně je k dispozici samostatná úloha, kde studenti zkoušejí sami najít testovací vektory a mohou si na dostupných pomůckách ověřit své předpoklady.

V samotném úvodu se práce zabývá jak problematikou samotného generování testovacích vektorů (ATPG), tak aktuálními nástroji, které mohou výuku podpořit a studentům přiblížit, jak takový ATPG algoritmus funguje. Následně je představena analýza komplexního řešení pro výuku, jež by by umožnilo efektivnější vedení výukového cvičení za pomocí webové aplikace, která poskytne prostor nejen samostatnou práci, ale také usnadní společnou práci vyučujících a studentů nad jedním obvodem.

Práce obsahuje také popis návrhu architektury tak, aby bylo možné aplikaci snadno do budoucna rozvíjet a doplňovat o další možnosti. Je zde popsána jak implementace, tak ukázán způsob, jak je možné aplikaci rozšířit, jak aplikaci provozovat a v neposlední řadě poskytuje náhled na vytvořenou základní datovou sadu, která vychází zejména z příkladů aktuálně používaných na FIT ČVUT v Praze v předmětu NI-TSP.

Samotnou motivací pro vypracování této závěrečné práce tak byla snaha o zpřehlednění a zkvalitnění výuky v daném předmětu, stejně tak jako možnost vyzkoušet návrh moderní architektury pro provoz aplikace a ověřit výsledky takového návrhu v praxi. Aplikace je totiž sestavena nejenom ze samotného simulačního jádra, ale také z prostředků, které usnadní práci dalším vývojářům, kteří se budou podílet na rozvoji tohoto díla.

Kapitola 1

Problematika ATPG

Testování, nástroj k odhalení a diagnostice vzniklých poruch a chyb, je nedílnou součástí vývoje číslicového obvodu. Nicméně vzhledem k rozsáhlosti vyvíjených obvodů je prakticky nemožné připravovat testy pro vznikající obvody ručně. Již v roce 2005 se uvádělo, že vývojář logického obvodu poskytne pouze 70 až 75 % funkčních testů pro danou jednotku, přičemž toto číslo je nedostačující [3]. Z toho důvodu je zapotřebí nástroje, který je schopný systetizovat testy pro danou jednotku.
Algoritmy pro automatické generování testovacích vektorů (ATPG) byly prakticky představeny v roce 1966 s popisem D-algoritmu, který představoval formalizaci intuitivního přístupu k hledání takových vektorů. Od té doby prošly značným vývojem, nicméně základním kamenem pro dnes využívané ATPG algoritmy je stále D-algoritmus[4], který byl dále optimalizován a zdokonalován. V této kapitole je popsán zejména princip fungování tohoto algoritmu, jejich uplatnění a následný vývoj v této oblasti v průběhu času. Kapitola zároveň poskytuje přehled základních přístupů pro generování testovacích vektorů a simulaci poruchového modelu.
Vzhledem k zaměření této práce je tato část zaměřena na důkladný popis D-algoritmu a dále také na postupy užívané při simulaci testovacích vektorů. Další významné algoritmy jsou zmíněny, avšak nejsou dopodrobna vysvětlovány.

Hledání poruch v elektronických číslicových obvodech je vždy založeno na použití zjednodušeného modelu poruchy. Toto je dáno tím, že množství fyzikálních jevů, které mohou ovlivnit správnou funkci elektronického systému, je příliš mnoho a pro praktické využití by bylo téměř nemožné simulovat všechny. Z toho důvodu je nutné vybírat pro poruchový model takové poruchy, jež jsou nejpravděpodobnější a vhodně reprezentovatelné pomocí logických hodnot tak, aby bylo možné přístup pro testování dále automatizovat[5].

Mezi nejzákladnější a zároveň nejvíce využívané modely poruch patří modely logických poruch, jmenovitě trvalá jednička (t_1), trvalá nula (t_0) a porucha typu zkrat (z). Poruchy trvalých hodnot se souhrnně označují jako poruchy typu t. Tyto modely logické poruchy pokrývají většinu reálných jevů, které mohou vzniknout v klasických logických obvodech a tvoří základ pro testování logických číslicových obvodů. Dále představované algoritmy pracují s těmito modely a jsou navrženy pro hledání poruch na základě právě výše uvedených modelů. Je třeba také uvést, že se stále jedná pouze o pokus o přiblížení se realitě, avšak žádný model nemůže poskytnout dokonalý odraz fyzikální reality (jedná se vždy o zjednodušení a je třeba s tím takto počítat).

S těmito modely poruch pracují některé algoritmy zaměřené na hledání testovacích vektorů, které jsou dále podrobněji rozebrány v této kapitole. Základem pro zmíněné algoritmy je metoda **intuitivního zcitlivění cesty**, kdy vývojář manuálně hledá testovací vektory pro primární vstupy [1] testovaného obvodu [2] tak, aby se případná porucha projevila na výstupu [3] testovacího obvodu. Pro úplné pochopení je také nutné uvést přehled používaných termínu a jejich význam:

testovaný obvod, CUT/DUT Jedná se o číslicový obvod, který je podroben dalšímu zkoumání. Jako *DUT* označujeme obvod jako celek, který má své primární vstupy a výstupy a plní nějakou, libovolně složitou, logickou funkci. Obecně může být *DUT* více vstupů i výstupů.

primární vstup, PI Jedná se o vstup do číslicového obvodu. Tímto vstupem komunikuje obvod s ostatními komponenty systému a zajišťuje správné přijímání dat. Nejedná se o vstup do jednotlivého logického členu nebo dílčího obvodu systému, ale o vstup do systému jako celku.

primární výstup, PO Jedná se o výstup z číslicového obvodu jako z celku, analogově k primárním vstupům.

porucha, fault Takový stav, který zabraňuje části testovaného číslicového obvodu ve vykonávání správné a předem dané funkce. Může, ale nemusí se projevit změnou na jednom nebo více primárních výstupech. Pokud se projeví na primárním výstupu, hovoříme o **chybě**. Jedná se o poruchu na logické úrovni.

chyba, error Projev poruchy v takové míře, že je změněna hodnota výstupu testovaného obvodu.

zcitlivěná cesta, sensitive path Jedná se o nalezení takové cesty skrze logický obvod, které při vhodném nastavení primárních vstupů propaguje poruchu od místa vzniku až na primární výstup.

testovací vektor, test vector Takové ohodnocení primárních vstupů, které odhaluje zvolenou poruchu v logickém obvodu.

test Množina dvojic vektorů pro *primární vstupy* a *primární výstupy*. Neformálně lze říci, že *test* jsou všechna ohodnocení primárních vstupů a k nim odpovídající odezvy v bezporuchovém stavu.

krok testu, test step Jednotlivá dvojice ohodnocení pro primární vstupy a výstupy.

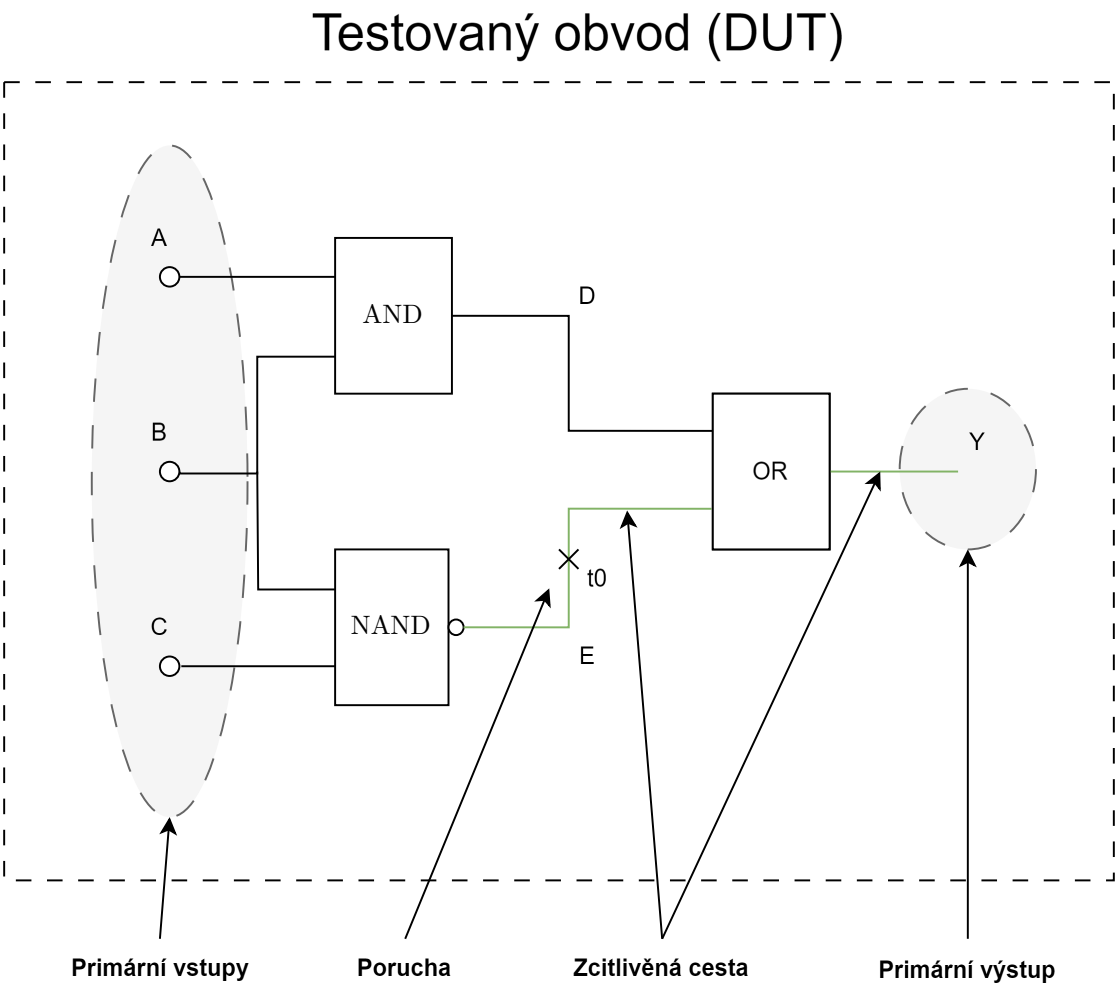
délka testu, test volume Počet kroků testu.

diagnostické pokrytí, fault coverage Vyjádření buď výčtem poruch detekovaných daným testem, nebo častěji poměr detekovaných poruch vůči celkovému počtu všech poruch.

efektivita testu, test effectiveness *(detekované a nedetekovatelné poruchy) / všechny poruchy*. Jedná se tedy o ukazatel celkové efektivity testu vzhledem k celkovým možnostem, které umožňuje obvod otestovat. Jako **nedetekovatelnou** označujeme takovou poruchu, která je například vykryta redundancí v obvodu, a tedy se neprojeví chybou na primárním výstupu.

Pro lepší pochopení některých výše uvedených pojmů je zde přiložena také ilustrace 1.1, kde jsou zmíněné termíny uvedeny. Pro úplnost je vhodné doplnit, že *testovací vektor* pro zvolenou ukázku a poruchu je $0, X, 0$ s očekávaným výsledkem *1*, nicméně v důsledku poruchy t_0 na vodiči E bude skutečný výsledek *0*. Občas se můžeme setkat s tím, že tento fakt je zapsán jako *1/0*.

[1] **PI**, z anglického **primary input**
[2] **DUT**, z anglického *design under test*, případně **CUT**, z anglického *circuit under test*
[3] **PO**, z angl. *primary output*



Obrázek 1.1 Ukázka jednotlivých pojmů použitých u ATPG algoritmů. Primární vstupy a výstupy jsou označeny v šedém pozadí, zcitlivěná cesta je znázorněna zelenou barvou a porucha je označena křížkem s popiskem t0, přičemž znamená poruchu typu *trvalá 0*. Testovací vektor pro zvolenou poruchu bude *(0, X, 0)* s očekávaným výsledkem *1* v bezporuchovém stavu.

Z hlediska zaměření testů je možné tyto dále rozlišit do dvou základních skupin. Tou první skupinou jsou testy funkční, které se ze své povahy zaměřují pouze na funkci testovaného obvodu, neuvažují jeho vnitřní strukturu a ohodnocují pouze na primární vstupy a výstupy. Výhodou je to, že k provedení testu stačí pouze předpis logické funkce, kterou má obvod vykonávat, a znalost vstupů a výstupů. Nevýhodou je pak praktická nemožnost optimalizovat testy právě na základě znalostí vnitřní architektury obvodu a s tím související pracnost při testování velkých obvodů.

Druhou skupinou jsou testy strukturní, které počítají se znalostí struktury testovaného obvodu. Strukturní test je schopen obsáhnout i poruchy na vnitřní struktuře a díky tomu je možné testy dále optimalizovat, minimalizovat, a pokud je odhalena porucha, prakticky dokáží napovědět, kde se daná porucha mohla vyskytnout.

1.1 Strukturní testy v číslicových obvodech

Všechny dále popsané metody uvažují **strukturní testy** číslicového obvodu. Funkční testy v této práci nejsou pokryty, jelikož rozsah práce je zaměřen právě na testy strukturní. V rámci generátorů testovacích vektorů existují různě pokročilé algoritmy. Přehled těchto algoritmů je uveden v tabulce 1.1[3].

Algoritmus	Rychlost	Rok publikace
D	1	1966
PODEM	7	1981
FAN	23	1983
TOPS	292	1987
SOCRATES	1 574	1988
Wiacukaiski	2 189	1990
EST	8 765	1991
TRAN	3 005	1993
Recursive learning	485	1995
Tafertshofer	25 057	1997

Tabulka 1.1 Přehled TPG algoritmů

Strukturní testy jako takové slouží k ověření správné funkce vnitřní logiky a jejich hlavní výhodou oproti testům funkčním je možnost diagnostiky poruchy, tedy nalezení místa, kde daná porucha mohla vzniknout. Velkým problémem je pak skutečnost, že bylo dokázáno, že deterministické generování testu pro logický obvod je **NP-úplný** problém, který má exponenciální složitost vzhledem k počtu primárních vstupů[3].

Z tohoto důvodu je dobré omezit běh ATPG algoritmu pouze na případy, kdy je to nezbytně nutné. Navíc bylo změřeno, že do určité míry je výhodnější generovat testovací vektory pseudonáhodně, jelikož míra pokrytí roste při tomto přístupu rychleji a hlavně nemá exponencionální složitost. Následně je použit deterministický TPG algoritmus, který nalezne testovací vektory pro obtížně detekovatelné poruchy[3].

V praxi to pak vypadá tak, že je nejprve spuštěn pseudonáhodný generátor testovacích vektorů, přičemž každý vektor je dále simulován pomocí *simulátoru poruch* a výstup z tohoto simulátoru, který obsahuje seznam detekovaných poruch, je zapsán na společný *seznam poruch*. Jakmile dosáhne pseudonáhodné generování stanovené hranice (například další náhodný vek-

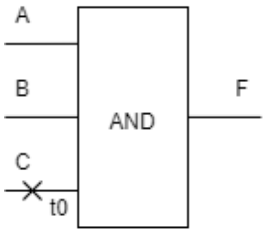
tor netestuje žádnou novou poruchu), přijde na řadu deterministický TPG algoritmus, který za pomocí seznamu poruch vybere stále neotestovanou poruchu a pokusí se pro ni najít testovací vektor. Pokud je úspěšný, tento vektor je opět simulován a seznam poruch je opět rozšířen o výstup simulace. Toto se opakuje do chvíle, než bude dosaženo požadované efektivity testu. Celkově je tento přístup označován jako *mixed-mode-testing* [3].

1.2 Metody generování testovacích vektorů

1.2.1 Intuitivní zcitlivění cesty

Jedná se o nejjednodušší a nejpřímější způsob, jak vygenerovat testovací vektory. Postup je jen velmi málo formalizován a spočívá v určení poruchy, kterou chceme v daném obvodu najít, a dále v nastavování vstupů daného hradla tak, aby se dané hradlo stalo pro poruchu propustným. Toho je dosaženo tak, že zajistíme takovou kombinaci vstupních hodnot logického hradla, aby případná změna na testovaném signálu zapříčinila změnu na výstupu za daným hradlem.

Tento klíčový prvek je pak důležitým pro dále uváděné algoritmy, jelikož tvoří základní stavební prvek v tom, jak je možno dále automatizovat tvorbu testů. Výše zmíněný postup nastavování vstupů u konkrétního hradla si můžeme ilustrovat na příkladu s třívstupovým hradlem AND s poruchou t_0 na vstupu C. V obvodu zaznačíme hradlo jako na obrázku 1.2.



Obrázek 1.2 Třívstupové hradlo AND s poruchou *t0* na vstupu C

Dále známe pravdivostní tabulku bezporuchového obvodu, jež je zachycena v tabulce 1.2. Pokud se však vyskytne porucha trvalá 0 na kterémkoli vstupu, hradlo již není schopno plnit svou funkci a výstup bude vždy 0. Abychom zcitlivěli cestu skrze toto hradlo, musíme najít ohodnocení zbývajících, bezporuchových vstupů tak, aby se změna *1/0* projevila i na výstupu. Z tabulky pravdivostních hodnot lze vidět, že takové ohodnocení je jen jedno, a to $1, 1, 1/0$, které propaguje změnu na výstup v podobě $1/0$.

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Tabulka 1.2 Pravdivostní tabulka hradla AND se třemi vstupy

Na příkladu výše vidíme, že postup, jakým hledáme citlivou cestu není zcela formalizován. Kýžená formalizace pro výpočet testovacích vektorů je pak poprvé představena v roce 1966 u J. Paula Rotha v podobě D-algoritmu[4].

1.2.2 D-algoritmus

Jedná se o algoritmus formalizující výše popsané intuitivní zcitlivění cesty za pomocí takzvaného D-kalkulu, který kromě základních logických hodnot přináší také hodnoty D (z anglického *dependant* nebo také *difference*), tedy hodnoty závislé na poruše. [4]. Všechny symboly používané v D-kalkulu jsou uvedeny v přehledné tabulce 1.3

Symbol	Význam
0	Logická 0
1	Logická 1
X	Na logické hodnotě nezáleží
D	Hodnota proměnné je závislá na poruše D hodnota znamená, že původní hodnota 1 je při poruše 0
$\overline{D}$	Hodnota proměnné je závislá na poruše $\overline{D}$ hodnota znamená, že původní hodnota 0 je při poruše 1

Tabulka 1.3 Symboly používané v D-kalkulu

Algoritmus vyžaduje rozšíření základních logických hodnot (navíc ještě s hodnotou X, tedy hodnotou, na které nezáleží) právě o nové symboly. Toho je docíleno pomocí jednoduchých pravidel:

- $0 \cap 0 = 0 \cap X = X \cap 0 = 0$
- $1 \cap 1 = 1 \cap X = X \cap 1 = 1$
- $X \cap X = X$
- $1 \cap 0 = D$
- $0 \cap 1 = \overline{D}$

Jak lze vidět z definice průniků výše, operace není asociativní, tedy záleží, který vektor hodnot vstupuje do operace na levé straně a který na pravé. Pro vektor $(0, 1, X, X) \cap (1, X, 0, 0)$ tak dostáváme výsledek $(\overline{D}, 1, 0, 0)$, při prohození operandů je výsledek $(D, 1, 0, 0)$. Jak si ukážeme později, algoritmus si s tímto dokáže poradit, jelikož jsou výsledky rozdílné pouze v inverzi hodnoty D.

Pro pochopení fungování je ještě nutné znát pojem *singulární pokrytí logického hradla*[4]. Pod tímto pojmem je prakticky ukryta minimalizace logické funkce pro výstupní hodnotu 1 a 0. Pokud máme například logické hradlo AND se třemi vstupy, bude vypadat minimální pokrytí tak, jak je uvedeno v tabulce 1.4

A	**B**	**C**	**F**
0	X	X	0
X	0	X	0
X	X	0	0
1	1	1	1

Tabulka 1.4 Symboly používané v D-kalkulu

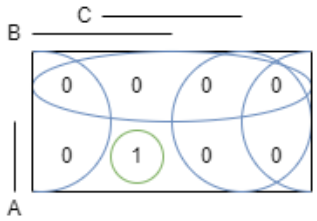
Toto singulární pokrytí můžeme získat pomocí minimalizace pravdivostní tabulky 1.2 v Karnaughově mapě, podobně jak je ilustrováno na obrázku 1.3. Z obrázku je patrné, že pro funkci $F(A, B, C) = 1$ je minimální logický výraz roven $F(A, B, C) = ABC$[5], odtud tedy vektor $(1, 1, 1, 1)$ do tabulky 1.4. Pokud provedeme minimalizaci pro $F(A, B, C) = 0$, pak je minimální logický výraz roven $F(A, B, C) = A + B + C$ a odtud zbývající vektory. Hodnotou X v singulárním pokrytí značíme, že na dané hodnotě nezáleží (proměnná se nevyskytuje u příslušného mintermu/maxtermu).

Pro označení skupin řádků ze singulárního pokrytí logické funkce se užívá α_0 pro řádky, jejichž výstup je 0, a α_1 pro řádky, jejichž výstup je 1. Pro poruchové hradlo se nejčastěji setkáváme s označením β_0, respektive β_1.

D-algoritmus jako takový pouze přijímá tabulku singulárního pokrytí. Jak tabulku singulárního pokrytí získat, algoritmus nespecifikuje. Je možné využít libovolný algoritmus, který je schopen takovou tabulku sestavit, například pomocí algoritmu Quine–McCluskey [6], nebo metodou představenou T. S. Rathorem [7].

[4] Někde je možné setkat se také s pojmem *minimální pokrytí*

[5] Zde se nemusíme omezovat pouze na hradlo AND, jelikož minimalizace funguje stejně pro jakoukoli pravdivostní tabulku.



Obrázek 1.3 Minimalizace pokrytí třívstupového hradla AND pomocí Karnaughovy mapy

1.2.2.1 Postup práce D-algoritmu

Algoritmus pracuje v pěti základních krocích, které jsou:

1. Vytvoření singulárního pokrytí a tabulky přenosových D-krychlí (*inicializace*)
2. Výběr poruchy, pro kterou budeme generovat krok testu
3. Vytvoření primitivní D-krychle poruchy
4. Propagace hodnoty D na primární výstup (*propagace*)
5. Nalezení ohodnocení primárních vstupů (*konzistence*)

Hlavním problémem tohoto algoritmu je, že porucha, stejně jako příslušné parametry, se volí vždy náhodně a tato náhodnost může zvýšit časovou náročnost algoritmu, protože častěji může docházet k situaci, kdy vznikne konfliktní ohodnocení na vodiči. V takovém případě je nutné se při výpočtu vrátit do bodu, kdy lze vybrat jiné ohodnocení, a postup opakovat. Tomuto přístupu se říká **backtracking**.

1.2.2.2 Primitivní D-krychle

V rámci výpočtu se pracuje s termínem **primitivní D-krychle** poruchy. Tento termín označuje takové ohodnocení vstupů a výstupů, které pro dané poruchové hradlo propaguje poruchu. Pro ilustraci můžeme uvažovat hradlo AND se třemi vstupy a poruchou t_0 na vstupu C, stejně jako na obrázku 1.2, a s minimálním pokrytím uvedeným v tabulce 1.4. K takovému hradlu následně sestavíme logickou tabulku pro chování s poruchou, jež je zobrazeno v tabulce 1.5

A	B	C	F
0	0	0	0
0	0	0	0
0	1	0	0
0	1	0	0
1	0	0	0
1	0	0	0
1	1	0	0
1	1	0	0

Tabulka 1.5 Pravdivostní tabulka třívstupového hradla AND s poruchou

Pokud provedeme minimalizaci, tedy nalezneme singulární pokrytí takové funkce, získáváme jeden jediný vektor, a to $(X, X, X, 0)$. Tento vektor tedy označíme jako součást množiny β_0. Nyní hledáme takový průnik s tabulkou singulárního pokrytí hradla v bezporuchovém stavu, aby se na výstupu objevila hodnota D, respektive $\overline{D}$. Z definice pro průnik v D-kalkulu vyplývá, že nemá smysl hledat průniky $\alpha_0 \cap \beta_0$ a $\alpha_1 \cap \beta_1$, jelikož na jejich výstupu bude vždy 0, resp. 1. Pro náš vektor tedy hledáme průnik vektorů $\alpha_1 \cap \beta_0$:

$$(1, 1, 1, 1) \cap (X, X, X, 0) = (1, 1, 1, D)$$

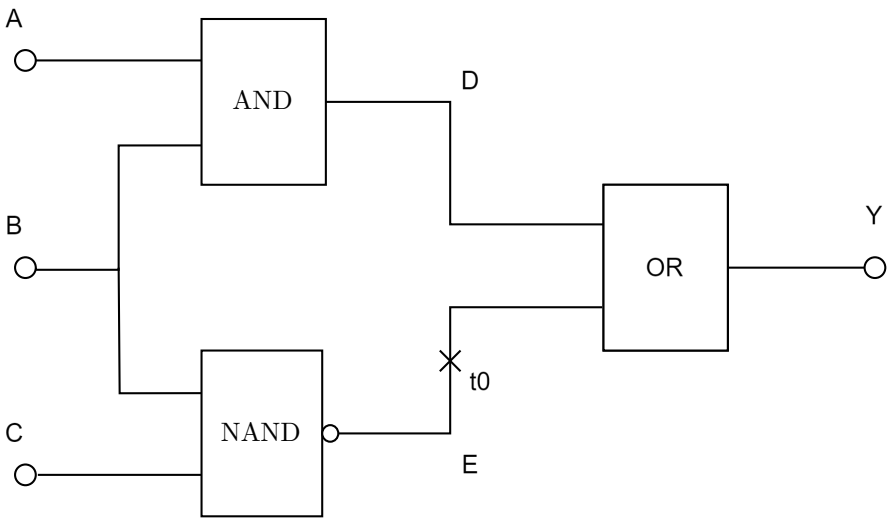
Primitivní D-krychle poruchy je tedy $(1, 1, 1, D)$.

1.2.2.3 Propagace poruchy

Další nezbytnou součástí algoritmu je propagace poruchy od poruchy směrem k výstupu. Propagace spočívá v nalezení takového ohodnocení zatím neohodnocených vstupů jednotlivých hradel na citlivé cestě, aby bylo dosaženo propsání hodnoty D $(\overline{D})$ až na některý z primárních výstupů. Není nutné poruchu propagovat na všechny výstupy, nicméně alespoň na jednom primárním výstupu se porucha musí projevit, aby byla detekovatelná.

K propagaci poruchy vyžaduje algoritmus tabulku přenosových krychlí. Ta se sestaví jako průnik všech dílčích přenosových krychlí pro jednotlivá hradla. Pro průnik jednotlivých hodnot se použijí pravidla popsána výše. Vznikne tak tabulka, která postihuje všechna hradla a jejich možné vstupy. Pokud budeme uvažovat příklad na obrázku 1.4, bude naše tabulka přenosových krychlí vypadat tak, jak je popsána v tabulce 1.6.

Ve chvíli, kdy by v průběhu propagace vznikla nekonzistence, provádí se *backtracking* a hledají se jiné možné d-krychle, které lze spojit. Pokud by žádná taková neexistovala, algoritmus skončí s tím, že se jedná o nedetekovatelnou poruchu.



Obrázek 1.4 Příklad pro D-algoritmus

	A	B	C	D	E	Y
d1	D	1		D		
d2	1	D		D		
d3		D	0		$\overline{D}$	
d4		0	D		$\overline{D}$	
d5				D	0	D
d6				0	D	D

Tabulka 1.6 Tabulka přenosových D-krychlí pro příklad na obrázku 1.4

Propagace se pak provádí postupným průnikem možných D-krychlí do chvíle, než je porucha (hodnota D, respektive $\overline{D}$) přítomna na primárním výstupu. Tento postup je zaznamenán v tabulce 1.7.

	A	B	C	D	E	Y	Komentář
tc^0		X	0		D		Injekce – primitivní D-krychle poruchy
$tc^1 = tc^0 \cap d6$		X	0	0	D	D	Šíření poruchy na výstup – D se objevuje na výstupu

Tabulka 1.7 Příklad šíření poruchy na výstup

1.2.2.4 Operace konzistence

Tímto krokem hledáme ohodnocení volných vodičů pomocí tabulky singulárního pokrytí (pro námi zvolený příklad je uvedena v tabulce 1.8) tak, abychom zajistili vstupní vektor, kterým poruchu dokážeme otestovat. Stejně jako v případě propagace poruchy i zde může nastat konflikt, který vyžaduje *backtracking*.

	A	B	C	D	E	Y
s1	1	1		1		
s2	0	X		0		
s3	X	0		0		
s4		1	1		0	
s5		0	X		1	
s6		X	0		1	
s7				1	X	1
s8				X	1	1
s9				0	0	0

Tabulka 1.8 Tabulka přenosových krychlí

Výstup, jak vypadá operace konzistence pro zvolený příklad, je znázorněn tabulkou 1.9

	A	B	C	D	E	Y	Komentář
tc^0		X	0		D		Injekce – primitivní D-krychle poruchy
$tc^1 = tc^0 \cap d6$		X	0	0	D	D	Šíření poruchy na výstup – D se objevuje na výstupu
$tc^2 = tc^1 \cap s2$	0	X	0	0	D	D	Konzistence – pro hradlo OR

Tabulka 1.9 Příklad šíření poruchy na výstup

Jelikož jsme jediným krokem ohodnotili všechny vodiče, algoritmus v této chvíli končí s tím, že nalezený krok testu $(0, X, 0) \rightarrow D$ a pokrývá poruchu t_0 na vodiči E.

1.2.2.5 Zhodnocení D-algoritmu

D-algoritmus položil dobrý základ pro další vývoj ATPG algoritmů. Nicméně jeho časová složitost, která je $O(2^s)$, kde s je počet signálů[6], z důvodu náhodného výběru signálu a proto, že v nejhorším případě bude nutné vybrat a nastavit každý signál v obvodu, vyvolala potřebu po efektivnějším algoritmu. D-algoritmus pak slouží jako reference pro další algoritmy, které se využívají pro generování strukturních testů. Algoritmy, které následovaly, využívají rovněž představený D-kalkulus, který se stal takřka standardem pro reprezentaci přenosu poruchy.

Z výše uvedených důvodů je patrné, jak důležité je pro další pochopení znalost D-algoritmu a jeho součástí, které jsou nutné pro správné vygenerování kroku testu. Je to také důvod, proč se tento algoritmus důkladně vyučuje na vysokých školách, například v předmětu NI-TSP na FIT ČVUT v Praze.

1.2.3 PODEM

Prvním úspěšným algoritmem, který navazoval na D-algoritmus, byl *PODEM* (z *Path Oriented Decision Making*), který byl představen v roce 1981 [8]. Potřeba zrychlit D-algoritmus přišla přirozeně, jelikož původní algoritmus je velice neefektivní, pokud se v obvodu vyskytují hradla XOR. To je způsobeno tím, že u XOR hradla není možné provést minimalizaci pokrytí, a také tím, že při procházení obvodu od poruchy na výstup a zpět k primárním vstupům velmi často nastává kolize a musí se provádět *backtracking*. A protože v tehdejší době byl vývoj obvodů se samoopravným mechanismem na vzestupu, přišel P. Goel s nápadem na nový algoritmus, který by nepostupoval od poruchy na výstup a zpět na vstup, ale zkoušel by nastavit ohodnocení primárních vstupů tak, aby se zvolená porucha projevila. Tím se omezil počet větvení z $O(2^s)$ na $O(2^n)$, kde s je počet signálů a n je počet primárních vstupů.

PODEM tak používá několik nových pojmů, které je dobré znát. Protože náhodný výběr primárních vstupů by nemusel být vždy efektivní, existuje více heuristik řízení výběru právě pomocí *pozorovatelnosti* a *řiditelnosti*. Nejčastěji zmiňovanými pojmy v souvislosti s tímto algoritmem jsou:

rozhodovací strom V kontextu PODEMu se jedná o binární rozhodovací strom sestavený z primárních vstupů. Respektive uzly rozhodovacího stromu jsou jednotlivé primární vstupy a hrana je reprezentovaná hodnotu 0 nebo 1.

řiditelnost Pravděpodobnost, že signál má danou hodnotu. Například pro hradlo AND je řiditelnost obou vstupů 50% pro obě logické hodnoty, nicméně výstup má pro hodnotu 1 řiditelnost $C(1) = 0,75$, a tedy analogicky $C(0) = 0,25$.

pozorovatelnost Pravděpodobnost, s jakou se daná hodnota objeví na daném signálu. K výpočtu je nezbytné znát jednotlivé řiditelnosti.

[6] Signál zde zahrnuje primární vstupy a výstupy společně s vnitřními vodiči v číslicovém obvodu. Pro příklad 1.4 se jedná o celkem 5 signálů.

První přístup je založen na obtížnosti excitovat poruchu a využívá *řiditelnosti* k tomu, aby zjistil, jak obtížné je nastavit které hradlo pomocí kterého primárního vstupu. Tento přístup nicméně ovlivňuje volbu poruchy, a tedy obecně nezefektivní práci s konkrétní poruchou. Druhý přístup je založen na *pozorovatelnosti* a snaží se nejprve najít *PO*, který je nejblíže poruše, a následně hledá, který *PI* nejvíce ovlivňuje daný výstup. Obecně se tak dá říci, že se snaží poruchu propagovat co nejjednodušeji.

Ze své definice je PODEM úplným algoritmem, jelikož pokud daný testovací vektor existuje, vždy jej nalezne. To je dáno tím, že vždy uvažuje všechny možné kombinace, jež mohou na primárních vstupech nastat. Obecně také k propagaci poruchy používá D-kalkulu se všemi pravidly, která jsou vysvětlena o kapitolu dříve.

1.2.4 FAN

Záhy po PODEM algoritmu byl představen algoritmus *FAN* (z *Fanout-Oriented Test Generation*) [9], který na PODEM navazuje a přináší vylepšení v podobě snahy o jednoznačné určení signálu, na základě již známých hodnot. Tedy ve chvíli, kdy nastavím hodnotu na PI a provedu její propagaci, FAN zpětně doplní unikátně určené hodnoty signálů. To následně pomáhá v průchodu obvodem, protože pokud nastane konflikt, nastane dříve a algoritmus může pokračovat jinde a jinak. Navíc se může stát, že tento zpětný průchod odhalí také hodnotu na nějakém jiném vstupu (čímž opět urychlí postup).

1.2.5 SOCRATES

Algoritmus *SOCRATES* (z *Structure-Oriented Cost-Reducing Automatic TESt patterns generation*) [10] přináší další vylepšení a rozšiřuje FAN o využití předpočítaných implikací a statické a dynamické učení. Tyto implikace představují vazby mezi jednotlivými signály v obvodu tak, bylo možné velmi rychle určit, jakou hodnotu vstupu je nutné zaručit pro jakou hodnotu na výstupu. Statické učení pak spočívá ve výpočtu takových implikací ještě před spuštěním algoritmu, kdežto dynamické učení tvoří seznam implikací postupně.

Tento algoritmus přinesl zásadní zrychlení (viz 1.1), jelikož implikace je nutné napočítat jen jednou a následné použití algoritmu nad stejným obvodem už pouze využívá tuto znalost.

1.3 Simulace testovacího vektoru

Simulace je vhodným doplněním při sestavování testu. Zatímco deterministicky vygenerovat krok testu je *NP-úplný* problém, ověření a simulace takového výstupu je problém spadající do kategorie *P* a pomůže nám navíc efektivně najít i další poruchy, které jsou odhaleny daným testovacím vektorem. Podobně jako pro generování testovacího vektoru existuje několik rozličných přístupů a algoritmů, tak i v rámci simulace testovacích vektorů existuje více přístupů[3].

Obecně pak každý takový algoritmus odpovídá následující specifikaci pro vstup:

- obvod,
- testovací vektor (případně více vektorů),
- poruchový model.

Algoritmus by pak měl zajistit následující výstup:

- poruchový výstup,
- detekované / nedetekované poruchy,
- pokrytí poruch.

Simulací automaticky získáme také tzv. *slovník poruch*, tedy jednoznačné určení, jaké poruše odpovídá jaká odezva. Možnými přístupy pro simulaci poruch jsou[11]:

- sériová simulace,
- paralelní simulace,
- deduktivní simulace,
- souběžná simulace.

Každý z výše uvedených přístupů má své výhody a nevýhody, zejména pak odlišné nároky na strojový čas a paměť. Jednotlivé přístupy jsou krátce představeny níže.

1.3.1 Sériová simulace

Sériová simulace je založena na velmi jednoduchém principu porovnávání odezvy bezporuchového a poruchového obvodu. Nejprve se spustí logická simulace bezporuchového obvodu, následně se vloží do obvodu porucha a simulace se spustí znovu. Odezvy simulací se mezi sebou porovnají a tím vznikne seznam detekovaných poruch. Výhodou je přímočará implementace a možnost simulovat prakticky jakoukoli poruchu nad jakýmkoli obvodem. Nevýhodou je pak časová náročnost tohoto přístupu.

1.3.2 Paralelní simulace

V rámci paralelní simulace je možné zvolit si mezi dvěma přístupy. První z nich byla uvedena v roce 1965 a nazývá se *Single Pattern Parallel Fault Propagation* (*SPPFP*) [12]. Jedná se o myšlenku, že je zbytečné testovat pouze jednu poruchu, jelikož všechny proměnné mají v architektuře procesoru nějakou šířku slova. S tímto přístupem se tak najednou testuje až $w - 1$ poruch, kde w je šířka slova (o jednu méně, protože stále musí být odsimulován bezporuchový stav). Výhodou je využití paralelismu, avšak nevýhodou je nemožnost využívat dominance poruch (vektor, který detekuje poruchu F_1 zároveň detekuje F_2), není možné předčasně simulaci ukončit (protože stále ještě není možné rozhodnout, zda jsme již narazili na poruchu) a z toho

vyplývající nutnost vždy simulovat až na všechny výstupy z obvodu. Navíc při testování a simulaci může signál obecně nabývat také hodnoty X, kterou ale není možné s tímto přístupem nijak reprezentovat.

Druhý přístup k paralelní simulaci se nazývá *Parallel Pattern Single Fault Propagation* (*PPSFP*) a byl představen Waicukauskim v roce 1986 [13]. Tento přístup nejprve simuluje bezporuchový obvod a následně je testováno w vstupních vektorů zároveň. Výhodou je opět rychlost tohoto přístupu. Navíc mohu ukončit simulaci ihned, pokud algoritmus zjistí, že se daná porucha nepropaguje na žádný primární výstup. Zůstává však problém, jak reprezentovat vícehodnotovou logiku.

1.3.3 Deduktivní simulace

Deduktivní simulace [14] se od paralelní liší především ve způsobu ukládání hodnot. Bezporuchový obvod je simulován pouze jednou a následně je ke každému signálu obvodu přiřazen seznam poruch, které jsou na takovém signálu detekovatelné. V danou chvíli tak simuluji pouze jeden testovací vektor a hlavní výhodou je, že se jedná o inkrementální algoritmus, kdy při každém dalším průchodu odrážím pouze změny v seznamu poruch.

1.3.4 Souběžná simulace

Principem souběžné simulace je testování pouze toho, co se liší od bezporuchového obvodu[15]. Algoritmus uvažuje poruchová hradla (což je rozdíl oproti předchozím způsobům, které se zaměřují na signály), které si udržují bezporuchový seznam hodnot vstupů a výstupů. Díky tomu je možné dosáhnout událostmi řízené simulace. Hlavní výhodou je poté rychlost odezvy na změny, protože mohu simulovat pouze změny stavů jednotlivých hradel.

Kapitola 2

Stávající řešení pro výuku

Kapitola se zabývá stávajícím stavem výuky ATPG algoritmů na FIT ČVUT v Praze v předmětu Testování a spolehlivost (NI-TSP). Slouží jako základ pro návrh aplikace, která má za cíl usnadnit výuku a umožnit studentům přehledně ukázat jednotlivé kroky algoritmů používaných ke generování testů pro logické obvody. Potřeba nové výukové pomůcky je zřejmá ze zhodnocení aktuálního stavu výuky, respektive náročnosti, kterou aktuální stav představuje.

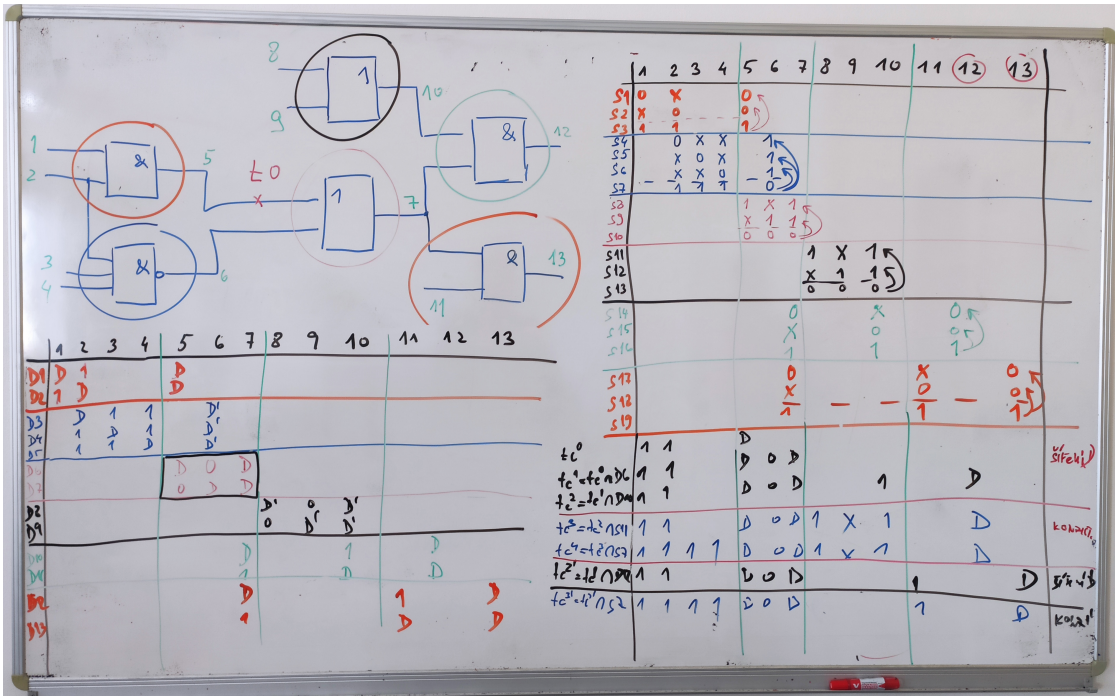
V současné době se v předmětu NI-TSP na FIT ČVUT vyučují již zmíněné strukturní ATPG algoritmy, tedy jmenovitě *D-algoritmus*, *FAN*, *PODEM* a *SOCRATES*, přičemž praktická výuka probíhá pouze u intuitivního zcitlivění cesty (jako teoretický základ) a D-algoritmu. Ostatní algoritmy nejsou vzhledem k časové náročnosti ve cvičeních probírány. Pro samostatnou práci jsou pak k dispozici nástroje Atalanta [1], implementující algoritmus *FAN*, a dále Java applet, který je součástí publikace [3]. Ani jeden z výše zmíněných nástrojů pak neumí krokovat svůj postup tak, aby studentům případně objasnil vnitřní fungování algoritmu.

První jmenovaný nástroj, *Atalanta*, poskytuje pouze CLI[1] rozhraní, přičemž vstup je obvod ve formátu `bench` a výstupem je seznam testovacích vektorů. S těmito informacemi studenti dále pracují, nicméně jedná se o pokročilejší zpracování dat, které nijak nevypovídá o tom, jak algoritmus pracuje uvnitř.

Druhým zmiňovaným nástrojem je Java applet, který už poskytuje vizuální rozhraní a disponuje více módy tak, aby studentovi přiblížil vztah mezi testovacím vektorem, seznamem poruch a naučil ho, jak zajistit propagaci a excitaci poruchy. Limitujícím faktorem tohoto řešení je omezený počet příkladů (není možné je upravovat).

Hlavní výukovou pomůckou je tedy klasická tabule, kdy vyučující nejprve nakreslí příslušný obvod a studenti se následně střídají u tabule a zakreslují jednotlivé kroky do tabulek, které jsou rovněž na tabuli. Vzhledem k povaze algoritmů, kdy se v nich často vyskytuje backtracking, tedy nutnost jít o krok zpět a zkusit jinou hodnotu, je tento způsob velice náročný, jelikož vyučující musí neustále hlídat konzistenci dat a zároveň s tím udržovat výklad. Ačkoli může být tento způsob velmi názorný a interaktivní pro malé obvody, jak je patrno z obrázku 2.1, již při relativně malém počtu hradel je situace lehce nepřehledná.

[1] command line interace



Obrázek 2.1 Výuka D-algoritmu na tabuli

Právě nepřehlednost a náročnost pro velké obvody vyvolala potřebu po názornějším a automatizovanějším přístupu, jelikož obvod, který se musí nakreslit na tabuli, se jednak nemusí vejít do vyhrazeného prostoru, a dále čas, který zabírá samotná příprava úlohy, lze využít efektivněji.

Dalším problémem je vizualizace, které hradlo se vztahuje ke kterému řádku z příslušných vnitřních struktur algoritmu. Ačkoli je na tabuli možné použít různé barev pro různá hradla, barev není neomezené množství a typicky je nutné některé opakovat, což dále přidává na nepřehlednosti.

V neposlední řadě je nutné, při snaze přiblížit studentům propagaci a excitaci poruchy, kombinovat různé barvy pro různé signály, což znamená, že je nutné na tabuli přemazávat jednou již nakreslené schéma. Všechny tyto faktory přispívají k potřebě výukové pomůcky, která by byla schopna rychlé projekce obvodu společně s možností krokování algoritmu.

Z pohledu studentů je pak téměř nemožné neustále udržovat přehled o tom, jak algoritmus postupuje a promítnout změny popsané na tabuli do sešitů. Pro studenta tak může být výklad matoucí a nepřehledný, a tento nedostatek lze odstranit právě vhodnou pomůckou pro výuku.

Kapitola 3

Analýza požadavků

Vzhledem k současnému stavu výuky, který byl popsán v předchozí kapitole, bylo rozhodnuto o vývoji výukové pomůcky, která by uspokojila alespoň některé potřeby vyučujících. Vývoj aplikace umožňující vizuální reprezentaci postupu jednotlivých algoritmů je předmětem této práce. Autor se v této kapitole zaměřuje na důkladný popis potřeb jak vyučujících, tak i studentů. V neposlední řadě návrh odráží potřebu ponechat systém otevřený pro budoucí modifikace tak, aby bylo možné projekt dále efektivně a smysluplně doplňovat například o další ATPG algoritmy..
Kapitola se postupně zaměřuje na popis funkčních a nefunkčních požadavků, rozpracování případů užití a obecně se zaměřuje na počáteční fázi vývoje aplikace, kdy bylo nutné stanovit si cíle, které bude práce splňovat.

Analýza požadavků vychází především z autorovy vlastní zkušenosti s výukou a dále z rozhovorů s vedoucím práce a vyučujícím předmětu *NI-TSP* v jedné osobě. Požadavky jsou zaměřeny na odstranění hlavních problémů, jež se vyskytují při výuce v souvislosti s ATPG algoritmy. Požadavky jsou rozděleny na funkční a nefunkční a následně jsou funkční požadavky přetaveny v případy užití.

3.1 Funkční požadavky

Aplikace bude poskytovat možnosti pro správu a vytváření projektů[1], přidělování přístupu jednotlivým uživatelům k daným projektům, možnosti pro seskupování projektů do skupin a v neposlední řadě možnost simulovat a krokovat zvolený algoritmus nad zvoleným obvodem. Aplikace bude dostupná pomocí webové stránky, nicméně samotná simulace a komunikace mezi jednotlivými uživateli bude zajištěna přes serverové rozhraní[2].

[1] projektem se rozumí návrh číslicového obvodu
[2] viz další kapitoly

3.1.1 Přihlášení uživatele [FP1]

Aplikace poskytne možnost autorizace pomocí jednotného přihlášení, které umožní studentům fakulty přístup do aplikace bez dodatečné registrace. Možnost prohlížet projekty a spouštět simulace bude dostupná pouze pro přihlášené uživatele.

3.1.2 Správa výukových skupin [FP2]

Pro udržení přehlednosti mezi jednotlivými návrhy logických obvodů bude možné organizovat tyto do skupin, které mohou představovat například osobní prostor uživatele nebo daný ročník vyučovaného předmětu. Aplikace umožní zobrazit uživateli pouze takové skupiny, kterých je členem, a umožní editaci skupiny a vytváření projektů ve skupině pouze uživatelům s oprávněním Správce.

Každá skupina je pro uživatele identifikovatelná pomocí svého názvu, který však nemusí být unikátní. V průběhu existence skupiny je možné ji editovat, měnit nastavení přístupu ke skupině jednotlivým uživatelům a skupinu může uživatel opustit, případně smazat. Skupina, jež nemá žádného člena, zaniká okamžikem opuštění posledního člena, který má roli správce.

3.1.3 Projekty [FP3]

V rámci jednotlivé skupiny bude možné vytvářet a udržovat projekty, které reprezentují číslicové obvody. Projekt bude možné označit uživatelsky definovanými štítky a k projektům bude možné přidělovat přístup obdobně, jako je tomu u výukových skupin.

Každý projekt bude mít svůj název, který však opět nemusí být unikátní, a to ani v rámci skupiny. Jednotlivé projekty si mohou prohlížet pouze uživatelé, kteří k nim mají patřičná oprávnění.

3.1.4 Editace projektu [FP4]

Projekt bude možné editovat, a to ve vizuálním editoru logických obvodů, jenž poskytne základní logická hradla, která může uživatel při návrhu využít. Jedná se o následující hradla:

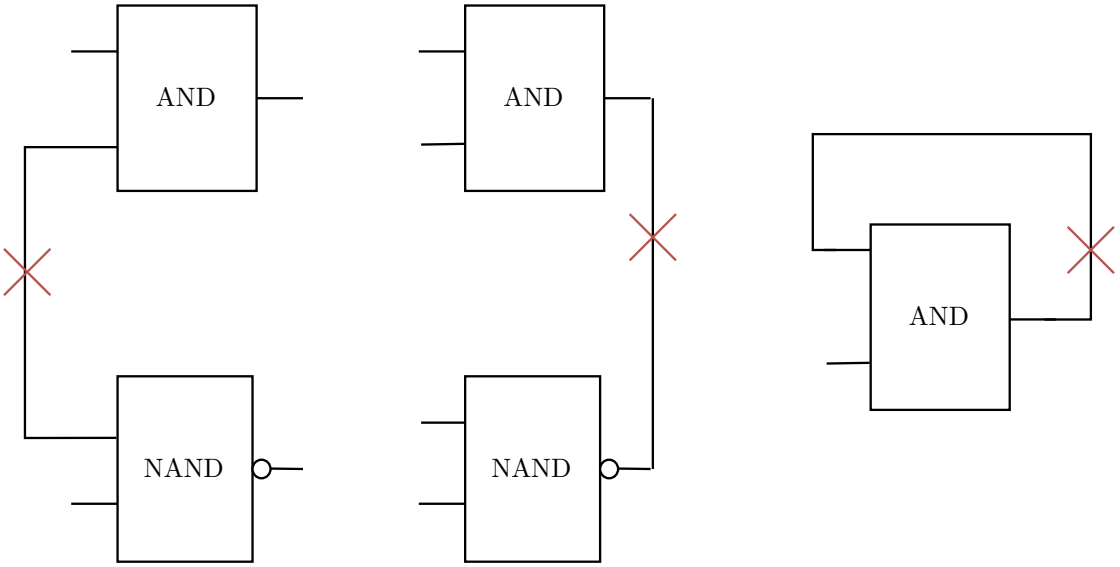
- AND
- OR
- NAND
- NOR
- XOR
- NOT

Vizuální editor umožní propojení logických hradel a definování primárních vstupů a výstupů z logického obvodu. V rámci editace bude možné vytvářet a přiřazovat uživatelsky definované

štítky. Editor poskytne základní validace tak, aby byla zajištěna konzistence a správnost návrhu. Mezi základní validační pravidla patří:

- Nemožnost propojení vstupu se vstupem a výstupu s výstupem hradla
- Každý vstup a každý výstup má napojen přesně jeden signál (větvení je zajištěno pomocí uzlů na vodiči)
- Navrhovaný logický obvod neobsahuje cyklus

Podrobněji jsou pravidla ilustrována na obrázku 3.1, který obsahuje příklady chybných.



Obrázek 3.1 Ukázka nesprávně zapojeného obvodu. Problémy v zapojení (zleva): propojení vstupu se vstupem hradla, propojení výstupu s výstupem hradla, cyklus v obvodu

3.1.5 Import obvodu [FP5]

Systém umožní import již existujícího obvodu do systému, a to ve formátu *bench*[16]. Tento formát se využívá při výuce a velice jednoduše se v něm popisují logické obvody díky přehledné a čitelné notaci. Díky možnosti importovat obvody do systému bude možné přenášet popis obvodu mezi různými, již využívanými, nástroji, například mezi Atalantou a touto aplikací.

3.1.6 Export obvodu [FP6]

Stejně jako import, umožní systém také možnost exportovat obvod ve formátu *bench*, tak aby byla zajištěna kompatibilita s jinými používanými nástroji.

3.1.7 Simulace průběhu algoritmu [FP7]

Aplikace po úspěšné validaci umožní spuštění simulace vytvořeného projektu a obvod uzamkne tak, aby nebylo možné provádět další úpravy. V tuto chvíli bude možné možné řídit simulaci a to především pomocí následujících voleb:

- Výběr algoritmu pro simulaci
- Spuštění/zastavení simulace
- Krokování simulace po blocích / krocích (tam i zpět)

Každou nově vytvořenou simulaci bude možné ponechat jako soukromou, případně bude možné zpřístupnit simulační prostředí pomocí jednorázového odkazu ostatním uživatelům aplikace. Tímto způsobem bude možné provozovat aplikaci v režimu výuky, jelikož práva manipulace s algoritmem bude mít pouze ten, kdo simulaci spustil. Uživatelé musí vždy navzájem vidět, kdo se simulace účastní a s jakými právy. Vlastník simulace může předat práva také některému z dalších účastníků, stejně tak může práva opět odebrat.

Simulace musí probíhat pro všechny účastníky v reálném čase a odezva řešení na jednotlivé podněty uživatele musí být minimální tak, aby se co nejvíce podpořila vzájemná spolupráce uživatelů.

3.1.8 Připojení uživatele k simulaci [FP8]

Uživatel musí mít možnost připojit se k simulaci, a to odkazem, nebo také kódem, který je možné sdílet mezi uživatele. Tímto způsobem bude umožněna spolupráce nad jedním simulovaným obvodem, jelikož aplikace musí zajistit stejný vizuální a podpůrný obsah algoritmu pro všechny uživatele bez rozdílu a ve stejném čase. Toto platí i v případě, že se uživatel připojí k simulaci později.

3.1.9 Vizualizace průběhu algoritmu [FP9]

Aplikace musí umět vhodně vizualizovat jednotlivé kroky simulovaných algoritmů. Pro D-algoritmus se bude jednat zejména o postupnou tvorbu tabulky minimálního pokrytí, přenosových D-krychlí, propagace a excitace poruchy. Aplikace musí umět zobrazovat libovolný vnitřní parametr algoritmu v každém kroku tak, aby byla uživateli poskytnuta vizuální zpětná vazba ohledně postupu. Tyto vnitřní stavy bude aplikace zobrazovat všem účastníkům simulace bez rozdílu.

3.2 Nefunkční požadavky

3.2.1 Podporované zařízení a prohlížeče

Vzhledem k povaze aplikace a plánovanému využití technologií budou podporovány pouze prohlížeče, které podporují využití *localStorage*, *web socketů*, a budou mít povolen *JavaScript*. K použití

aplikace je nutné stálé připojení k internetu bez výpadků. Bez těchto předpokladů není možné aplikaci správně využívat a nebude fungovat.

Rovněž je nutné zachovat minimální rozlišení aplikace, které činí 1920 na 1080 pixelů (FullHD). Zobrazení pod toto rozlišení mohou, ale nemusí fungovat, aplikaci však půjde v omezené míře používat dále.

3.2.2 Škálovatelnost

Aplikace bude připravena na snadné horizontální škálování v kubernetes clusteru tak, aby bylo možné dynamicky navýšit, případně ponížit využívané prostředky. Tato škálovatelnost by měla být zajištěna jak pro část simulační, tak pro části obsluhující klienta.

3.2.3 Jazykové mutace

Vzhledem k prvotnímu nasazení aplikace ve výuce na FIT ČVUT budou texty připraveny v českém jazyce. Aplikace nemusí podporovat snadné rozšíření o další jazyky.

3.2.4 Předpokládaná uživatelská zátěž

Zátěž na aplikaci je předpokládána nárazová, zejména v období výuky, kdy se očekává, že v rámci jedné simulace bude připojeno až 25 klientů, kteří mohou do simulace zasahovat a aktivně se jí účastnit. Mimo výuku se předpokládá využití studenty a vyučujícími v počtu jednotek návštěv denně.

3.2.5 Přihlášení uživatele

Přihlášení uživatele by mělo být stálé a mělo by být realizováno pomocí celofakultního přihlášení tak, aby nebylo nutné zadávat do aplikace jméno a heslo uživatele, ale aby si aplikace byla schopna tyto údaje automaticky obstarat. Uživatel tak bude své přihlašovací údaje zadávat pouze do jednotné přihlašovací služby.

3.3 Případy užití

Výše uvedené funkční požadavky jsou následně převedeny na případy užití, v nichž se již rozlišují jednotliví účastníci, kteří aplikaci používají. Pro tuto aplikaci byli rozlišeni dva hlavní účastníci, a to *přihlášený uživatel* a *nepřihlášený uživatel*, přičemž první jmenovaný se dále dělí na jednotlivé role ve skupině, které jsou popsány níže. Toto rozlišení pro aplikaci zcela postačuje a není nutné dále rozlišovat mezi vyučujícím a studentem – vyučující si stejně jako student může založit pracovní skupinu, do které následně studenty přidá. Vzhledem k předpokládané uživatelské zátěži je toto dostatečný způsob, jak řídit uživatelské přístupy v aplikaci.

Nepřihlášený uživatel si bude mít možnost vyzkoušet, jak vypadá editor obvodu se všemi jeho vlastnostmi, a stejně tak si bude moct vyzkoušet simulaci algoritmu v obvodu. Editor bude poskytovat také možnosti pro import a export obvodu z formátu *bench*. Nepřihlášený uživatel se bude moct navíc připojovat do simulací, nicméně pokud aplikace nebude mít data o identitě uživatele, bude vyzván, aby před vstupem do simulační místnosti doplnil své jméno. Toto je nutné, aby bylo zajištěno jednoznačné rozlišení mezi připojenými uživateli.

Přihlášený uživatel bude mít možnost navíc sdružovat své projekty do skupin, bude mít možnost si projekty označit štítky a editor umožní obvod uložit do aplikace pro pozdější použití. Rovněž na obvodu budou moci pracovat uživatelé společně[3]. Aplikace bude pro účely řízení přístupů k jednotlivým skupinám a projektům rozlišovat následující role, které budou nastavovat jako oprávnění skupiny:

- Vlastník
- Editor
- Návštěvník

V rámci simulace se pak ten uživatel, který simulaci spustí, stává *vlastníkem simulace* a ostatní účastníci, kteří se připojí později, jsou pouhými pozorovateli, pokud však nedostanou práva na ovládání simulace. Tím se pak mohou dočasně, případně trvale, stát *ovládajícím simulace*.

Celkově jsou pak případy užití zobrazeny na diagramu 3.2[4]

Pro úplnost je vhodné seznam případů užití zobrazit také v následujícím seznamu s k tomu náležící tabulkou 3.1, která znázorňuje pokrytí funkčních požadavků případy užití.

- UC1: Přihlášení
- UC2: Zobrazení dostupných skupin
- UC3: Zobrazení projektů skupiny
- UC4: Vytvoření nové výukové skupiny
- UC5: Úprava skupiny
- UC6: Odstranění skupiny
- UC7: Přiřazení uživatele ke skupině
- UC8: Přiřazení šítku k projektu
- UC9: Uložení projektu
- UC10: Tvorba projektu
- UC11: Import obvodu
- UC12: Export obvodu
- UC13: Spuštění simulace

[3] avšak nikoli současně
[4] vytvořený pomocí PlantUML

- UC14: Připojení k simulaci
- UC15: Opuštění simulace
- UC16: Krok vpřed v simulaci
- UC17: Krok o blok v simulaci
- UC18: Krok zpět v simulaci
- UC19: Změna role účastníka v simulaci
- UC20: Zobrazení všech účastníků simulace
- UC21: Výběr algoritmu pro simulaci
- UC22: Restart simulace
- UC23: Odstranění projektu

Funkční požadavek / Případ užití	**FP1**	**FP2**	**FP3**	**FP4**	**FP5**	**FP6**	**FP7**	**FP8**	**FP9**
UC1	✓								
UC2		✓							
UC3		✓	✓						
UC4		✓							
UC5		✓							
UC6		✓							
UC7		✓							
UC8			✓						
UC9			✓						
UC10				✓					
UC11					✓				
UC12						✓			
UC13							✓	✓	✓
UC14								✓	
UC15								✓	
UC16							✓		
UC17							✓		
UC18							✓		
UC19							✓	✓	
UC20								✓	
UC21							✓		
UC22							✓		
UC23			✓						

Tabulka 3.1 Pokrytí funkčních požadavků případy užití

Obrázek 3.2 Přehled případů užití

Kapitola 4

Návrh architektury

Vzhledem k požadavkům, které jsou nastíněny v předchozích kapitolách, a také vzhledem k faktu, že se nejedná o klasickou webovou aplikaci, ale hlavní důraz bude kladen na komunikaci v reálném čase, bylo nutné důkladně rozmyslet architekturu celkového řešení i jednotlivých komponent. V této kapitole je obsažen detailním návrhem takové architektury, aby bylo do budoucna možné ji snadno udržovat jak pro vývoj, tak pro nasazení v různých prostředích. Celkově je pak kladen důraz na izolaci jednotlivých komponent tak, aby bylo možné v budoucnu kdykoli takovou komponentu nahradit nebo upravit bez zbytečného vedlejšího dopadu na celkové řešení.
Kapitola tedy obsahuje jak návrh architektury obecně, tak také konkrétní řešení vývojového prostředí (tzv. local dev environment) a prostředí pro nasazení do Kubernetes clusteru, který byl zvolen jako cílové produkční prostředí. V této kapitole je lze najít také popis topologie cloud architektury.

Návrh architektury je vedle volby technologií velmi důležitým krokem, jelikož velmi silně ovlivňuje to, jakým způsobem se bude dál aplikace vyvíjet a jak snadno bude možné provádět zásahy do této aplikace. Tato kapitola je primárně zaměřena na podchycení architektury cílové aplikace jako celku„ a to pro následující prostředí:

- Vývojářské prostředí (local)
- Testovací prostředí (staging / feature)
- Produkční prostředí (prod)

Význam jednotlivých prostředí je objasněn v následujících kapitolách, přičemž výsledný návrh není omezen pouze na tyto tři výše jmenované možnosti, ale je snadno rozšiřitelný pro případné potřeby budoucího vývojového týmu. Výchozím předpokladem pro návrh architektury bylo, že aplikace bude kontejnerizovaná pomocí technologie Docker[17].

Motivací pro toto rozhodnutí byla snaha o co největší unifikaci mezi výše zmíněnými prostředími a snadná replikovatelnost případných problémů. Vzhledem k tomu, že Docker kontejner je izolovaná jednotka, která v sobě obsahuje vše potřebné pro běh aplikace, jež je v kontejneru zabalená, je možné případné pochybné chování aplikace velmi dobře cíleně zkoumat téměř bez nutnosti

ladit problémy lokálního stroje. Pokud porovnáme vývoj bez kontejnerů, kdy každý vývojář vyvíjí na svém stroji (*bare metal*), a vývoj v kontejnerizovaném prostředí, jako jasnou výhodu lze spatřit to, že odpadá nutnost vyřešit jednotlivé dílčí problémy (například verze podpůrných technologií, konfliktní verze jazyků, které jsou použity pro vývoj, atd...).

Dalším významným rozhodnutím, které navazuje na použití Docker kontejnerů, bylo rozhodnutí pro Kubernetes cluster[18] jakožto prostředí, do něhož bude probíhat nasazení (*deploy*) aplikace pro jiná než lokální prostředí. Toto rozhodnutí je opět motivováno snahou o vyšší abstrakci od konkrétních technologií použitých na serveru a izolaci samotné aplikace do dílčích jednotek. Díky takové izolaci je pak možné snadno replikovat nasazení aplikace a vytvářet takzvaná *feature prostředí*.

Všechna výše uvedená rozhodnutí byla provedena s ohledem na efektivní správu projektu v následujících letech a umožňují velmi snadný a pohodlný vývoj aplikace do budoucna jak u vývojáře, tak na serveru. Tato rozhodnutí jsou detailněji rozebrána níže.

4.1 Zvolené technologie a postupy

Jak již bylo zmíněno výše, pro tuto aplikaci bylo třeba zvolit hlavní technologie, které budou využity zejména pro produkční a staging nasazení (o významu jednotlivých prostředích pojednává kapitola 4.2), ale uplatní se také při lokálním vývoji tak, aby měl vývojář co nejvěrnější obraz produkce vždy k dispozici.

Koncepty, jakým způsobem bude aplikace vyvíjena a následně integrována do dalších prostředí jsou nastíněny v diagramu 4.1.

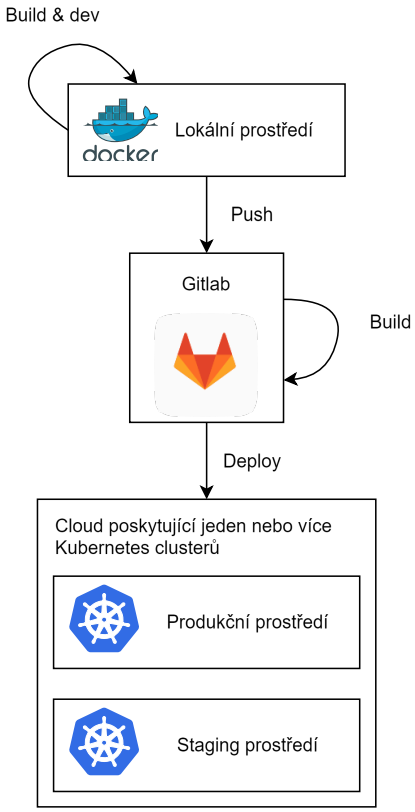
4.1.1 Kontejnerizace pomocí Dockeru

Nejprve bylo rozhodnuto o kompletní kontejnerizaci zvoleného řešení, jelikož izolace jednotlivých komponent aplikace je v současném dynamickém vývoji softwaru takřka nezbytností. Zejména kvůli stále rozmanitějšímu prostředí, které vývojáři používají (ať už se to týká operačních systémů, jejich verzí nebo jednoduše verzí jazyků, které mají vývojáři k dispozici) je nutné poskytnout spolehlivou a jednoduchou alternativu, jak projekty spravovat[19].

Existuje mnoho způsobů, kterými lze poskytnout unifikovaný přístup k vývoji aplikace, nicméně tím nejrozšířenějším způsobem je použití *aplikačních kontejnerů* technologie Docker. Tyto kontejnery obsahují pouze aplikaci, která typicky běží jako samostatný proces (a většinou je to jediný proces v rámci aplikačního kontejneru). Samotný Docker je pak možné využívat díky dříve rozvíjené technologii v linuxovém jádru.

Touto technologií jsou tzv. *user namespaces*, které jsou funkcemi linuxového jádra, které umí oddělit prostředky tak, že proces vidí vždy pouze a jen tu množinu prostředků, které náleží ke stejnému jmennému prostoru jako sám proces. Samotné prostředky pak mohou existovat ve více jmenných prostorech, nicméně proces jako takový je vždy přiřazen k jednomu konkrétnímu prostoru. Jmenných prostorů existuje více druhů, nicméně tato práce je nebude dále nijak rozebírat, jelikož pro pochopení jejich důležitosti postačí samotná rámcová znalost jejich existence.

Ačkoli *aplikační kontejnery* představil původně Docker, jejich specifikaci následně převzala



Obrázek 4.1 Provozování aplikace a proces vývoje

nezávislá organizace OCI[1], která zastřešuje specifikaci toho, jakým způsobem lze vytvořit *obraz (image)* aplikačního kontejneru a poskytuje také specifikaci běhového prostředí pro takové kontejnery.

Možností, jak vytvořit aplikační kontejner[2] a následně jej provozovat je tedy více, nicméně vzhledem ke kompaktnímu a velmi uživatlsky přívětivému prostředí Dockeru byla zvolena tato technologie. Docker poskytuje pro vývojáře velmi pohodlné spuštění aplikací pomocí mnoha integrovaých nástrojů a je možné ho provozovat na všech hlavních operačních systémech. Nejpohodlnější variantou použití je pak aplikace Docker Desktop, která však pro větší projekty vyžaduje placenou licenci[20], což však není na tomto projektu limitující.

4.1.2 Provozování aplikace

Jak vyplývá z kapitoly 4.1.1, pro samotnou izolaci aplikace a následný provoz bude potřeba spouštět *aplikační kontejnery*. Pro nasazení aplikace do staging a dalších sdílených prostředí je možné využít celou řadu jak komerčních řešení, například od Amazonu[3], Microsoft Azure[4] a dalších, tak open source, například samostatný Kubernetes cluster (který je možno provozovat nezávisle na cloud provideru)[18]. Pro lokální vývoj je pak možné používat nástroje jako *microk8s* nebo *Minikube*, které jsou schopné zpřístupnit Kubernetes cluster pro lokání vývoj[23]. Nevýhodou je pak fakt, že tyto nástroje vyžadují linuxové jádro a zprovoznění aplikace vývojářem na lokálním prostředí je tak náročnější na znalosti.

Vzhledem k faktu, že aplikace bude na všech sdílených prostředích provozována v podobě aplikačních kontejnerů a jelikož autor aplikace nechce vynutit jakéhokoli poskytovatele cloudových služeb, bylo rozhodnuto o využití Kubernetes clusteru jakožto běhového prostředí pro tyto kontejnery. Kubernetes umožňuje pohodlné nasazení, aktualizaci a správu běžících kontejnerů pomocí tzv. *manifestů*[24] a podporuje také mnoho nástrojů co se týče kontroly příchozích požadavků. Celkově je pak možné Kubernetes rozšiřovat o spousty dalších zdrojů, které pak umí spravovat pomocí jednotného API[5].

Aby pak nemusely být Kubernetes manifesty psány manuálně pro každé prostředí zvlášť a bylo možné je velmi efektivně spravovat a rozšiřovat, byl zvolen nástroj *Helm*, který slouží jako správce balíčků pro Kubernetes a je schopný jednotlivé aplikační kontejnery a jejich nasazení verzovat a provádět nad zvoleným Kubernetes clusterem[26]. Tento nástroj je velmi jednoduchý na užití a s jeho pomocí je možné řídit nasazení jak jednotlivých kontejnerů, tak souvisejících komponent do zvoleného clusteru[6].

Výše popsané technologie a postupy jsou vhodné pro sdílená prostředí, na kterých neprobíhá aktivní vývoj, jelikož je možné aplikační kontejner dopředu připravit a za běhu aplikace do něj není zasahováno. Pro lokální vývoj je však nutné mít možnost téměř okamžitě pozorovat změny v aplikaci, zvláště u webového vývoje, který je realizován pomocí interpretovaných jazyků a vývojář je zvyklý pozorovat odezvu ihned. Tohoto je možné dosáhnout pomocí nástroje Docker a vhodnou úpravou Dockerfile manifestu, který se postará o připojení zvoleného místního adresáře přímo do aplikace. Díky tomu se veškeré změny, které vývojář provede na svém zařízení ihned promítnou do kontejneru bez nutnosti tento kontejner znovu sestavit.

[1] Open Container Initiative
[2] Např. Buildah, Tekton Pipelines
[3] Amazon ECS, EKS[21]
[4] Mnoho dostupných možností pod Container Services[22]
[5] Dokonce existuje projekt Crossplane[25], který se zaměřuje na možnost nechat si poskytnout platformu jako službu pomocí Kubernetes manifestu
[6] Prakticky dokáže nasadit jakýkoli pro Kubernetes platný manifest

4.1.3 Správa kódu a podpůrné nástroje

Pro správu zdrojových kódů, stejně jako příslušných softwarových komponent[7] byl zvolen nástroj GitLab, jelikož jeho instance je provozována fakultou a je dostupný studentům FIT ČVUT. Nicméně aby bylo možné využít potenciálu tohoto nástroje naplno, bude nutné pro tento projekt zvolit veřejnou verzi dostupnou pro širokou veřejnost, jelikož práce bude využívat podpůrné nástroje pro práci s kontejnery přímo v GitLabu. Tyto nástroje zahrnují:

- Pipelines - potřebný pro kontinuální integraci a nasazení aplikací,
- Container registry - potřebný pro správu vytvořených aplikačních obrazů,
- Environments - potřebný pro správu dedikovaných vývojových prostředí v budoucnu.

Poslední dva zmiňované body nejsou dostupné ve fakultní instanci a tudíž by nebylo možné je využít. GitLab poskytuje určitý běhových počet minut pro bezplatné užití při sestavování a nasazování aplikací. Co se týče vlastního clusteru, ve kterém aplikace bude běžet, autor zvolil vlastní soukromý VPS server kvůli flexibilitě nasazení, nicméně do budoucna je možné využít například nasazení v prostředí CloudFIT za dodržení instalační příručky, která je součástí této práce jako příloha A. Popis samotného zprovoznění Kubernetes clusteru je nad rámec této práce a nebude mu zde věnována pozornost.

Pro lokální prostředí pak bude potřeba zajistit pohodlnou správu projektu a k tomuto účelu byl zvolen osvědčený nástroj Make, díky kterému je možné zaobalit komplexnější příkazy a postupy do jednoduchých instrukcí a vývojář tak opět nemusí řešit jak konkrétně musí interagovat s Dockerem a jinými nástroji. Detailnější použití vývojového prostředí vývojářem je popsáno v příloze C.

4.1.4 Rozdělení aplikace

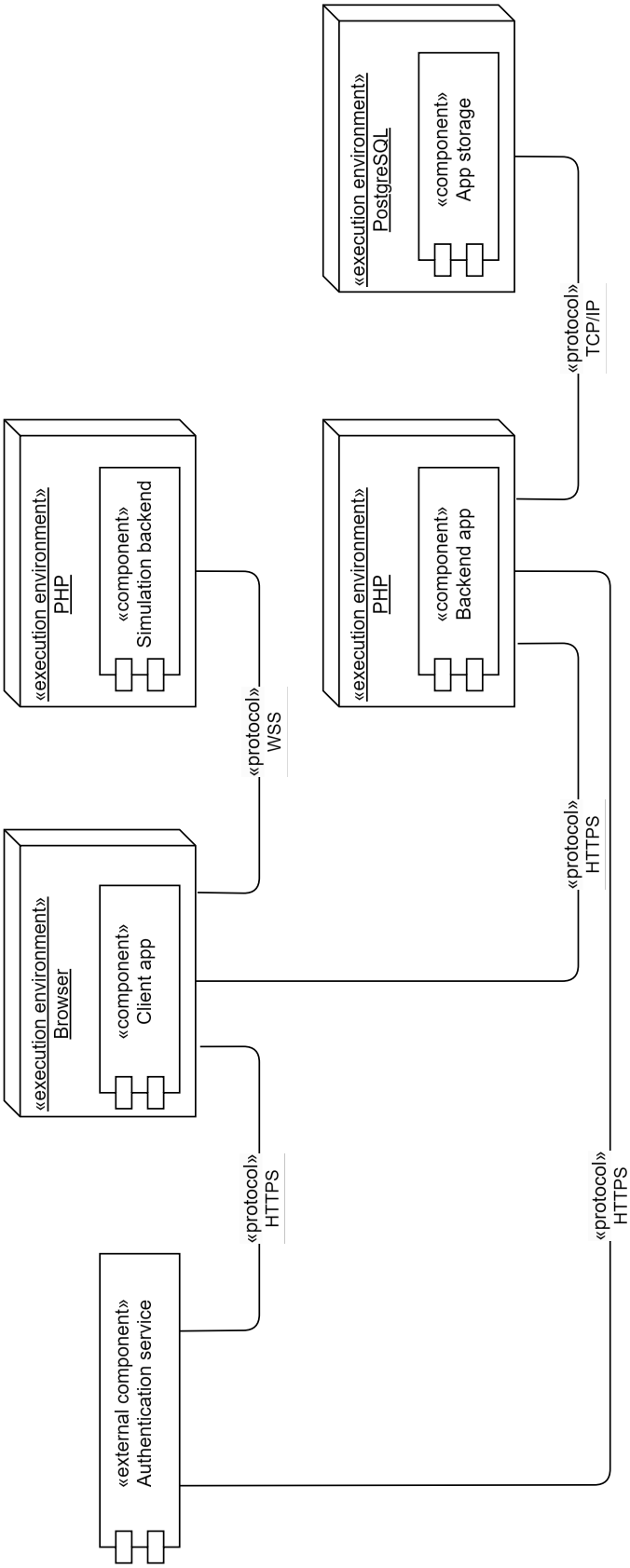
Jak již bylo nastíněno výše, architektura se zaměřuje na separaci jednotlivých oblastí a kontejnerizaci řešení pomocí běžně užívaných postupů. Aplikační návrh u nově vznikající aplikace by měl zohledňovat tuto volbu a díky tomu využít maximum z nabízených možností, které dané technologie nabízejí. Architektura vyvíjené aplikace tak respektuje tuto volbu a v maximální možné míře se snaží o využití tohoto faktu, ačkoli stále je zde prostor pro případná další zlepšení.

Samotný cílový stav rozdělení komponent v systému je zachycen na diagramu 4.2, přičemž jak je z diagramu patrné, aplikace bude rozdělena na tři hlavní části, kterými jsou:

- frontendová aplikace, sloužící jako hlavní rozhraní mezi klientem a zbytkem systému,
- backendová aplikace, která slouží pro zprostředkování uložených dat z databáze uživateli,
- simulační jádro, které poskytuje samotnou simulaci ATPG algoritmu a jeho dílčích kroků.

Mezi všemi těmito komponenty je plánován provoz výhradně zabezpečenými kanály. Tyto prvky aplikace cílí na to, aby byl provoz co nejjednodušší a vývoj jedné komponenty, díky použití předem definovaných rozhraní, byl co nejméně ohrožen vývojem na komponentě jiné. Zvolené technologie pro realizaci této architektury jsou popsány níže.

[7] Helm charty, Dockerfiles



Obrázek 4.2 Cílové rozdělení aplikačních komponent

4.1.4.1 Aplikační databáze

Vzhledem k požadavkům zadání bylo nutné zvolit databázovou technologii, která bude uživatelská data uchovávat. K tomuto účelu byla vybrána databáze PostgreSQL[27], nicméně požadavky na databázový provoz nejsou nikterak náročné a tedy v budoucnu nevylučuji možnost přechodu na jiný databázový engine. Databáze bude uchovávat pouze informace, neplánuje se využití pokročilejších funkcí.

4.1.4.2 Backend aplikace

Backend aplikace bude tvořit prezentační vrstvu nad databázovými daty a vystaví jednotné a předem definované API směrem ke klientské aplikaci. Vzhledem k povaze aplikace bude hrát roli pouze jako validátor vstupních data a nebude obsahovat příliš aplikační logiky. Prakticky veškeré zaměření tak bude pouze na načítání/ukládání požadovaných dat uživatelem a ověřování uživatelského přístupu pomocí autorizačních tokenů poskytnutých externí přihlašovací službou.

Vzhledem ke zkušenostem autora byl zvolen jazyk PHP společně s rozhraním GraphQl, které bude poskytováno klientské aplikaci. GraphQl bylo vybráno oproti REST rozhraní zejména z důvodu snadného získání dat o jednotlivých entitách, zejména vezmeme-li v úvahu, že účelem backendové aplikace je zde právě poskytnutí dat, která jsou jinak uložena v databázi. Hlavní rozdíly mezi *GraphQl* a *REST* přístupy jsou pak pospány v článku od IBM[28].

4.1.4.3 Klientská aplikace

Klientská aplikace by měla poskytovat uživatelské rozhraní, které bude komunikovat s ostatními komponenty celého systému. Klient jako takový pracuje v prohlížeči, který obstarává komunikaci se simulačním backendem (v případě aktivní simulace algoritmu) a získává data ohledně uživatelských skupin, projektů a schémat (v případě přihlášeného uživatele). Vzhledem k povaze aplikace, kdy je plánováno mnoho komponent znovu použít[8], a zkušenostem autora byl zvolen JavaScript framework *Vue 3*.

Vzhledem k požadavkům na odezvu v rámci simulace algoritmu bylo předem rozhodnuto o využití protokolu WebSocket[29], který zajišťuje oboustrannou komunikaci mezi klientem a cílovým serverem. Tato komunikace bude probíhat nezávisle na backendové aplikaci, jelikož veškerá data, potřebná pro spuštění simulace budou obsažena v první zprávě, kterou iniciuje klient. Toto je možné díky tomu, že navržené role v rámci simulačního prostoru jsou dány nikoli vlastnictvím daného návrhu, ale osobou, která daný návrh simuluje.

4.1.4.4 Simulační backend

Jedná se o hlavní část systému, jeliž zajišťuje samotnou simulaci poskytnutého obvodu. Tento backend není nijak napojen na databázi ani na autorizační rozhraní a tudíž nelimituje kohokoli v použití - pro spuštění simulace stačí dodržet předepsaný formát dat a komunikačního rozhraní. Stejně tak je možné jej nahradit případně jinou technologií, pokud bude potřeba, jelikož použité vnější rozhraní není technologicky specifické.

[8] například simulovat algoritmus může přihlášený i nepřihlášený uživatel, prvky editoru, atp...

Pro vývoj této komponenty byl opět zvolen jazyk PHP, především kvůli znalosti autora a také kvůli novým možnostem, které poskytuje samotný jazyk od verze 8 a framework OpenSwoole[30], který poskytuje moderní API pro práci v distribuovaném prostředí a tedy splňuje požadavky pro tuto práci.

4.1.4.5 Autentizační služba

Jakožto služba, která bude poskytovat autentizaci, byl zpočátku vybrán *Shibboleth ČVUT*, který poskytuje přihlášení veškerým identitám na univerzitě. Tato autorizační služba funguje pomocí protokolu *SAML 2.0*, který je široce využíván zejména pro sdílení informací o uživateli[31]. Nicméně vzhledem k plánovanému využití ve výuce na FIT ČVUT, a také vzhledem k možnosti anonymního přístupu k simulaci, bez nutnosti přihlášení, byl nakonec zvolen pohodlnější a přímočařejší přístup pomocí *OAuth2*[32].

I přes toto rozhodnutí je v aplikaci zachována možnost autentizace skrze službu *Shibboleth*[9], nicméně pro lokální vývoj a následné nasazení bylo využito práce OAuth2.0 autorizačního serveru, který poskytuje fakulta v podobě samoobsluhy[33].

4.2 Provozovaná prostředí

Jak již bylo nastíněno výše, v plánu je více vývojových prostředí pro projekt, přičemž každé plní odlišnou roli ve vývojovém cyklu aplikace a je tak uzpůsobené konkrétním potřebám dané fáze. V této sekci je nástin toho, jakou roli taková prostředí plní a jaké jsou na ně kladeny požadavky. Detailně je zde také popsáno, jakým způsobem jsou prostředí vystavěna a jaké nástroje poskytují.

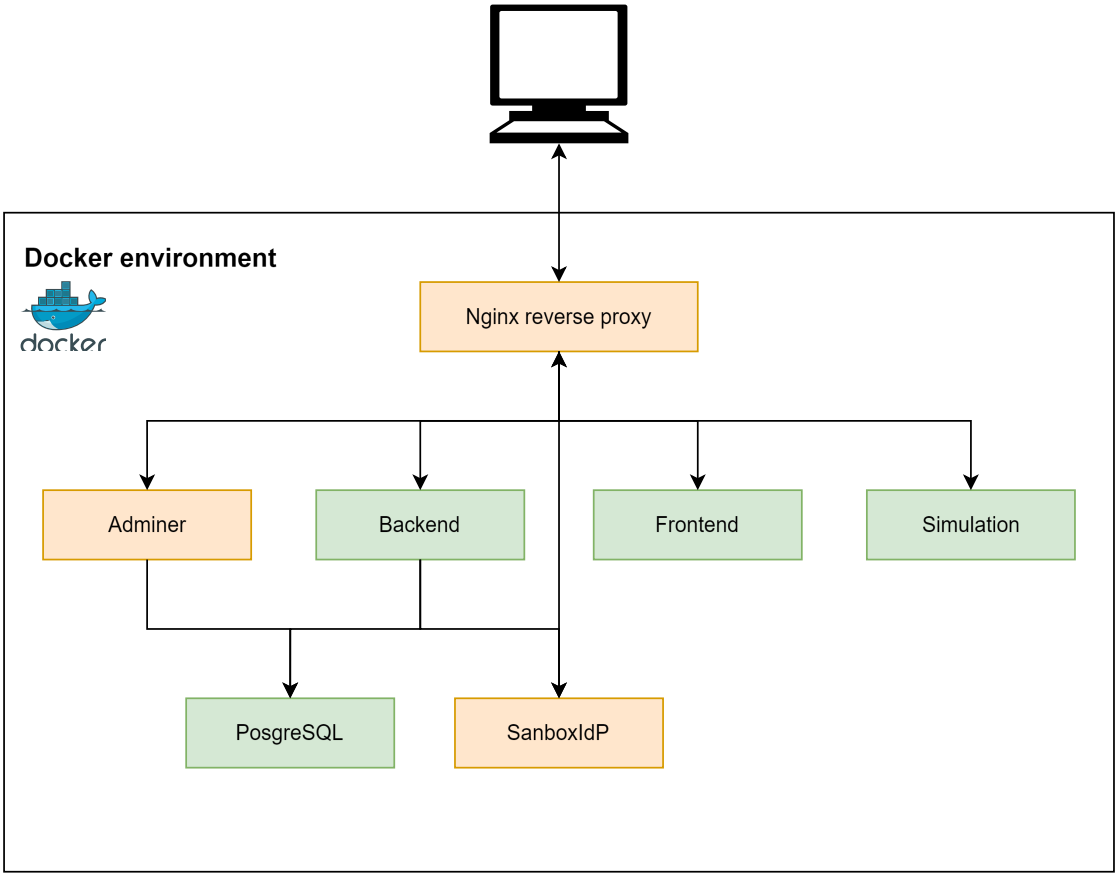
4.2.1 Lokální vývojové prostředí

Cílem lokálního vývojového prostředí je umožnit vývojáři efektivní práci s projektem a potřebnými technologickými závislostmi projektu. Toto prostředí by mělo poskytovat dostatečnou volnost při vývoji, přičemž by nemělo vývojáře nijak omezovat v jeho možnostech vyzkoušet technologie nové, zkoušet jiné verze použitých technologií, případně kompletně aplikaci přestavět. Na druhou stranu je nutné zajistit co největší unifikaci prostředí s dalšími vývojáři a produkčním prostředím tak, aby se eliminovaly případné chyby způsobené nekompatibilním užitím verzí.

Jedním z klíčových aspektů, který hraje roli v efektivním vývoji softwaru, je také rychlost, s jakou je vývojář schopen dané prostředí spustit a provádět v něm změny. V neposlední řadě se také nesmí zapomenou na rychlost odezvy, kterou aplikace dokáže poskytnout v takovém při provedení změny. Volba technologií pro lokální prostředí, jak již bylo zmíněno v kapitole4.1, zejména pak nástroj Docker, umožňuje právě takovou míru zapouzdření technologií pro lokální vývoj, který poskytuje jak potřebnou unifikaci, tak požadovanou flexibilitu pro vývojáře.

Architekturu aplikaci je tedy třeba vhodně upravit tak, aby zajišťovala výše zmíněné možnosti. Takto upravenou architekturu je možné postihnout podobně, jako je tomu na obrázku 4.3. V ilustraci jsou zeleně podbarveny kontejnery, které budou následně sestaveny také pro další prostředí. Oranžově jsou pak zaznačeny ty kontejnery, které slouží jako podpůrný nástroj pro vývoj.

[9] Respektive pomocí jakéhokoli autorizačního serveru poskytující SAML2.0 rozhraní



Obrázek 4.3 Layout docker prostředí pro lokální vývoj

Technologie nginx byla použita, jelikož je velmi využívána také pro nasazení v Kubernetes clusteru[34]. Dále se zde nachází nástroj pro procházení databáze, *Adminer*[35], který je velmi pohodlný na použití a je velmi jednoduché jej provozovat. V neposlední řadě obsahuje vývojové prostředí také kontejner *SandboxIdP*, který poskytuje SAML2.0 rozhraní pro testování autentizace. Tento sandbox využívá projekt *SimpleSAMLphp*, který zde vystupuje v roli poskytovatele identity (*IdP*)[36].

Toto prostředí je možné spouštět efektivně spouštět pomocí utility `docker-compose`, společně zaobalené do jednoduchých příkazů nástroje `make`. Cílem je, aby potencionální vývojář nemusel řešit jaké verze nástrojů použít a jediné, co potřeboval, byl funkční Make, bash a Docker. následné spuštění vývojového prostředí bude možné provést jediným příkazem `make up`.

Aby bylo zajištěno rychlé synchronizace projektu s běžícím prostředím na zařízení vývojáře, není projekt pevně součástí obrazu aplikace, ale adresář dílčí aplikace je vždy připojen k běžícímu kontejneru. Soubory je nutné připojit pod stejným `USER_ID` a `GROUP_ID`, stejně jako běžící procesy v kontejneru bude potřeba nastavit na tyto identifikátory tak, aby případné změny souborů nezpůsobily nekonzistenci v právech s těmito soubory manipulovat.

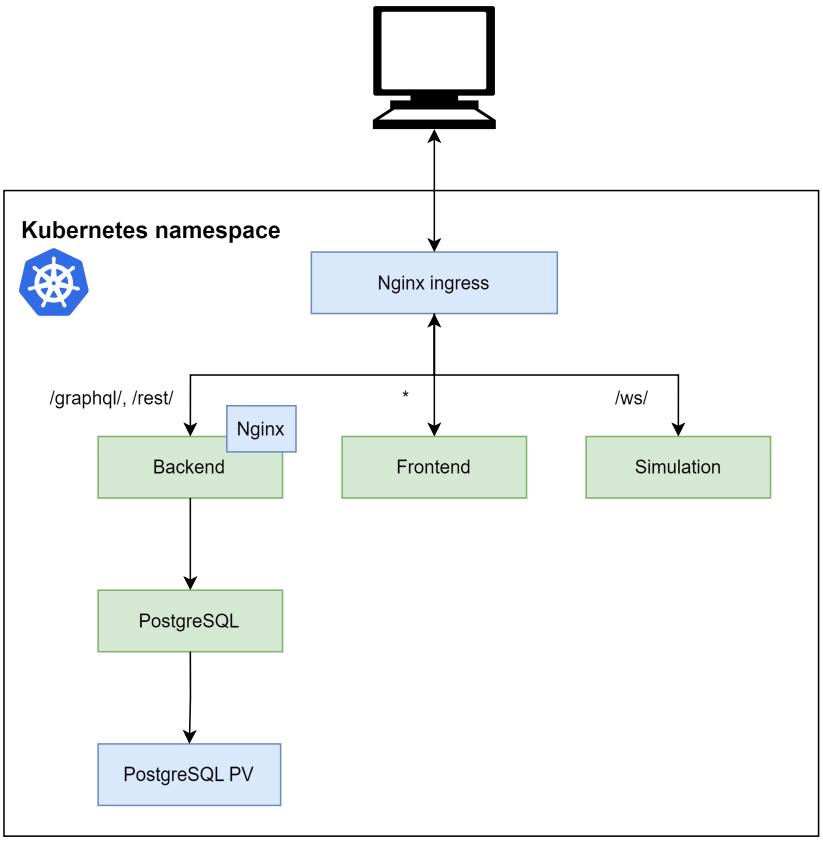
4.2.2 Produkční, staging a feature prostředí

Druhým, velmi důležitým prostředím ve vývoji softwaru je takzvané *staging* prostředí, které slouží k integraci práce vývojářů a následnému společnému testování. Staging prostředí by mělo co nejvíce odpovídat prostředí produkčnímu a jeho architektura je tomu přizpůsobena. Po úspěšném otestování je projekt nasazen do prostředí produkčního. Volitelným, avšak velmi užitečným přístupem jsou také tzv. *Feature Environments*. Ty umožňují každému vývojáři spuštění vlastního odděleného prostředí pro testování aplikace za pomocí konfigurace velmi podobné té produkční.

V rámci této práce je zpracována podpora pro všechny tři typy prostředí, přičemž všechny využívají nástrojů v bezplatné verzi *GitLabu* a dalších. Základem pro tato prostředí je připravený Kubernetes cluster (jeho spuštění není součástí této práce), do kterého probíhá nasazení pomocí Helm chartů pro jednotlivé aplikace zvlášť. To, o které prostředí se jedná je pak jednoduše rozpoznatelné pomocí Kubernetes namespaces.

Vývoj aplikací, potřebných pro tento projekt je soustředěn pouze na větev master, přičemž každý *commit* umožní manuální nasazení na *feature prostředí* a každý *tag* pak bude znamenat automatické nasazení na staging prostředí s volitelným, manuálním nasazením do prostředí produkčního. Součástí pipeline bude samozřejmě příprava obrazu aplikace tak, aby bylo možné ji spustit v clusteru.

Architektura nasazení v clusteru je pak zachycena v ilustraci 4.4, přičemž je stejná pro všechny tři výše zmíněné prostředí - jediným rozdílem je pak doména, ze které se přistupuje k aplikaci. Zeleně jsou opět označeny obrazy, které jsou společné pro vývoj aplikace. Modře jsou pak zaznačeny objekty poskytnuté samotným Kubernetem pro nasazení aplikace.



Obrázek 4.4 Layout kubernetes prostředí

Kapitola 5

Implementace

Tato část se zabývá implementací jednotlivých aplikací, které jsou popsány v kapitole 4.1.4, tedy frontend, backend a simulační aplikace. nejprve jsou navrhnuta jasně daná komunikační rozhraní mezi jednotlivými aplikacemi, následně je provedena analýza pro každou funkční část. Po analýze následuje samotný popis implementace a vnitřní architektury výsledné aplikace tak, aby měl čtenář komplexní přehled o vnitřním fungování jednotlivých komponent.
V neposlední řadě je zde také uveden postup při tvorbě architektury lokálního a cílového prostředí tak, aby byly respektovány poznatky z předchozích kapitol a byla zajištěna vysoká míra abstrakce a znovupoužitelnosti. Tato kapitola tak poskytuje široký pohled na aplikaci jako celek tak, aby bylo možné si představit, co obnáší vývoj, provoz a údržba celkového řešení.

Implementace této aplikace je mírně nestandardní v tom, že se vyžaduje WebSocket pro obousměrnou a okmažitou komunikaci mezi serverem a klientem. Z tohoto důvodu je možné říci, že se nejedná o klasickou webovou aplikaci, jejichž vývoj je častý. Tento fakt byl využit při návrhu architektury, kdy bylo rozhodnuto o oddělení simulačního jádra řešení do samostatné aplikace. Vzhledem k tomu, že se tím stalo řešení distribuovanou aplikací, bylo nutné nejprve důkladně navrhnout rozhraní mezi jednotlivými komponenty.

5.1 Návrh rozhraní a datových formátů

Jeden ze základních kroků k dosažení úspěšné implementace je zcela jistě navržení a dodržování jasně definovaných rozhraní, které poskytne spolehlivý vztažný bod pro další vývoj. Z tohoto pohledu autor vnímá jako nesmírně důležité se věnovat jednotlivým formátům, komunikačním rozhraním a vzorům použitým při následné implementaci.

5.1.1 Popis číslicového obvodu

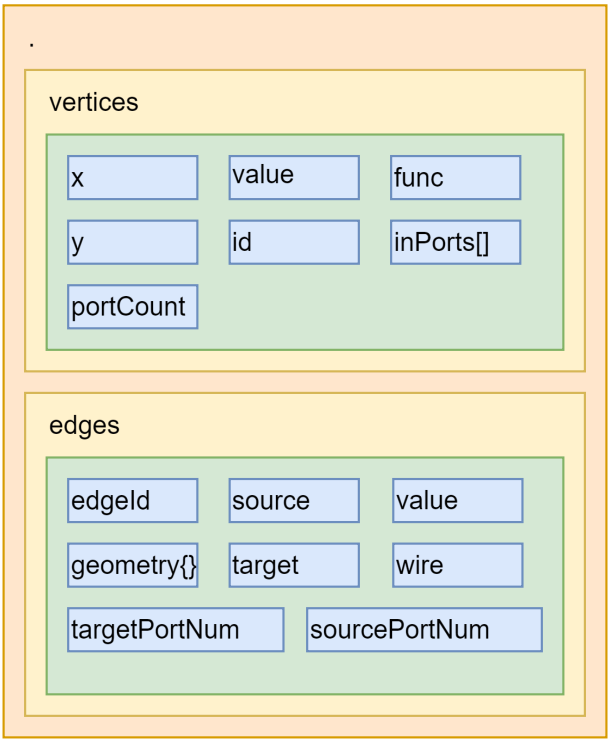
Vzhledem ke zaměření aplikace na číslicové obvody bylo nutné navrhnout formát, který by byl schopen postihnout takový číslicový obvod a zároveň zachoval následující:

- logickou strukturu obvodu, tedy které hradlo je umístěno na kterém vodiči, jaké má hradlo vstupy a výstup, kam tyto vstupy a výstupy vedou,
- vizuální strukturu, tedy jak vypadá rozmístění obvodu na virtuálním plátně a jak a kudy jsou vedeny vodiče.

Tento problém si lze abstrahovat na problém popisu grafové struktury, kdy jednotlivá hradla si lze představit jako vrcholy orientovaného grafu (kde se signál šíří po směru orientace hrany) a propojení mezi hradly jako hrany v tomto grafu. Drobným problémem této struktury je fakt, že na virtuálním plátně jsou propojení jednotlivých vodičů realizovány jako uzly v elektrickém obvodu a tento uzel je přirozeně také vrcholem.

Pro vyřešení tohoto problému byl navržen nový formát přenosu dat o logickém obvodu, který využívá právě principu struktury v orientovaném grafu, kdy ve formátu jsou přítomny jak hrany, tak vrcholy na samostatné úrovni, nicméně samotná hrana/vodič může mít jako zdroj/cíl mimo vrchol také jinou hranu. Díky tomu je možné velmi snadno pokrýt situace, kdy se výstup hradla větví a z vodiče skrze uzel vychází jiný vodič do jiného cíle.

Rozhodnutí pro takové řešení padlo z důvodu snadného popisu obvodu. Bylo by jistě možné uzly reprezentovat jako samostatné vrcholy, nicméně toto není nutné a implementačně by se pak jednalo o další úroveň složitosti, kterou by bylo nutné brát v úvahu. Samotný formát lze ilustrovat hierarchicky na diagramu 5.1. Význam jednotlivých polí je pak popsán v tabulce 5.1 pro vrcholy a 5.2 pro hrany/vodiče.



Obrázek 5.1 Formát pro popis číslicového obvodu v aplikaci

Název pole	Popis
x	pozice x na kreslícím plátně
y	pozice y na kreslícím plátně
portCount	počet vstupů daného hradla
value	označení daného hradla
id	interní identifikátor objektu
func	daným hradlem realizovaná funkce
inPorts[]	seznam hran, které jsou připojeny k hradlu

Tabulka 5.1 Popis a význam jednotlivých polí pro vrcholy ve společném datovém formátu

Název pole	Popis
edgeId	interní identifikátor hrany
source	interní identifikátor zdrojového objektu (hrany nebo vrcholu)
target	interní identifikátor cílového objektu (hrany nebo vrcholu)
value	označení hrany/vodiče
geometry	objekt popisující geometrický tvar hrany
targetPortNum	použito, pokud je cílovým objektem vstupní port hradla – označuje pak jeho pořadí
sourcePortNum	použito, pokud je zdrojovým objektem vstupní port hradla – označuje pak jeho pořadí
wire	uživatelské označení vodiče v obvodu

Tabulka 5.2 Popis a význam jednotlivých polí pro hrany ve společném datovém formátu

Interní identifikátory v popisovaném formátu slouží především k efektivnímu vzájemnému odkazování mezi objekty. Jak je možné si rovněž povšimnout, pole *inPorts* u vrcholů není nezbytně nutné, jelikož informace v něm obsažené je možné vyčíst z popisu hran. Nicméně pole bylo ve formátu ponecháno pro budoucí snazší odkazování při zpracování formátu.

Detailnímu popisu pole *geometry* se v této práci nebudu dále zabývat, pro detailnější náhled je možné prozkoumat libovolný exportovaný obvod z aplikace. Aby bylo možné s obvodem pracovat, bylo nutné zavést také některá omezení týkající se možného použití hran. Například aby byl číslicový obvod platný a simulovatelný, není možné propojit výstup jednoho hradla s výstupem hradla druhého. Za tímto účelem bylo zavedeno několik omezení:

1. není možné aby hrana vedla od výstupu k výstupu,
2. z toho vyplývá, že hrana musí být vedena buď z hrany jiné nebo ze vstupu hradla,
3. není možné vést spojení hrana-hrana
4. na pořadí zdroj-cíl nezáleží - pro simulaci bude signál šířen od výstupu hradla rovnoměrně

Díky těmto omezením a tomuto formátu je možné popsat jak vizuální, tak logické vlastnosti obvodu. Takový popis obvodu bude dále sloužit jak pro uložení formátu do databáze, tak pro spuštění a vedení simulace. Rovněž bude možné formát použít pro export a opětovný import obvodu do aplikace. Pro úplnost je zde také uveden příklad formátu ve výpisu kódu 5.1.1. Tento příklad reprezentuje jedno hradlo AND se dvěma vstupy, oba připojené na vstup a výstup propojený na výstup *OUT*.

Výpis kódu 5.1 Ukázka společného formátu pro popis číslicového obvodu

```
{
  "vertices": {
    "Cell#6": {
      "x": 450,"y": 180,"value": "&","id": "Cell#6",
      "func": "and","inPorts": ["Cell#15","Cell#16"],
      "portCount": 2
    },
    "Cell#13": {
      "x": 790,"y": 230,"value": "OUT","id": "Cell#13",
      "func": "o","inPorts": ["Cell#6"]
    },
    "Cell#15": {
      "x": 180,"y": 190,"value": "A","id": "Cell#15",
      "func": "i"
    },
    "Cell#16": {
      "x": 180,"y": 220,"value": "B","id": "Cell#16",
      "func": "i"
    }
  },
  "edges": {
    "Cell#8": {
      "edgeId": "Cell#8","source": "Cell#15",
      "target": "Cell#6","targetPortNum": 0, "wire": "A"
    },
    "Cell#9": {
      "edgeId": "Cell#9","source": "Cell#16",
      "target": "Cell#6","targetPortNum": 1, "wire": "B"
    },
    "Cell#10": {
      "edgeId": "Cell#10","source": "Cell#6",
      "target": "Cell#13","targetPortNum": 0, "wire": "OUT"
    }
  }
}
```

5.1.2 GraphQl rozhraní

Pro přístup k uživatelsky vytvořenému obsahu bylo zvole GraphQl rozhraní, které poskytuje dostatečnou flexibilitu co se týče možnosti dotazování dat jak na straně serveru, tak na straně klienta. GraphQl, na rozdíl od REST rozhraní, nedefinuje jednotlivé zdroje a jejich výstupní formát striktně. Namísto toho definuje množinu položek, které je možné dotazovat, přičemž daná položka se může skládat s více vnořených položek, či dokonce vytvářet kruhovou referenci[28].

Další nesmírnou výhodou je možnost deklarovat direktivy jak pro zpracování na serveru, tak na straně klienta a díky tomu automatizovat nebo značně usnadnit práci se společnými objekty. Například lze pomocí direktivy `resolver` rovnou odkazovat na funkci, která bude mít na starosti render dané položky ve schématu, či direktivou `authorization` omezit přístup k dané položce pouze pro uživatele s daným oprávněním. Uvedené direktivy jsou pouze příkladem a nejsou součástí standardu, nicméně ukazují na značnou flexibilitu tohoto dotazovacího jazyku.

Všechny dotazy, které pomocí *GraphQl* probíhají jsou vedeny buď jako *query*, nebo jako *mutation*. Rozdíl je v tom, že *query* obecně nemění data, pouze je dotazuje a naproti tomu *mutation* slouží k úpravě uložených dat. Pokud bychom měli zvolit analogii k *REST* rozhraní, **query** budou představovat všechny možné *GET* dotazy a *mutation* budou představovat dotazy *POST, PUT a DELETE*.

Schéma jako takové je pak jednoduché vizualizovat v diagramu a existuje spousta online nástrojů, které zvládnou vizualizaci schématu provést, jako například GraphQl Voyager[37], Výsledné schéma je pak možné procházet do detailu. Schéma pro tuto aplikaci pro dotazování dat je zachyceno na ilustraci 5.2[1].

Pro úplnost je vhodné zmínit, že *GraphQl* rozhraní, na rozdíl od *RESTu*, funguje spíše jako dotazovací jazyk a tomu odpovídá také struktura odpovědi. Namísto HTTP kódů, které využívá REST k indikaci možných problémů při zpracování požadavku, jsou veškeré chyby umístěny přímo do odpovědi v podobě pole *error*. Toto pole je pak nutné parsovat pro získání více informací o chybě a není skoro nijak předepsán formát takových chyb.

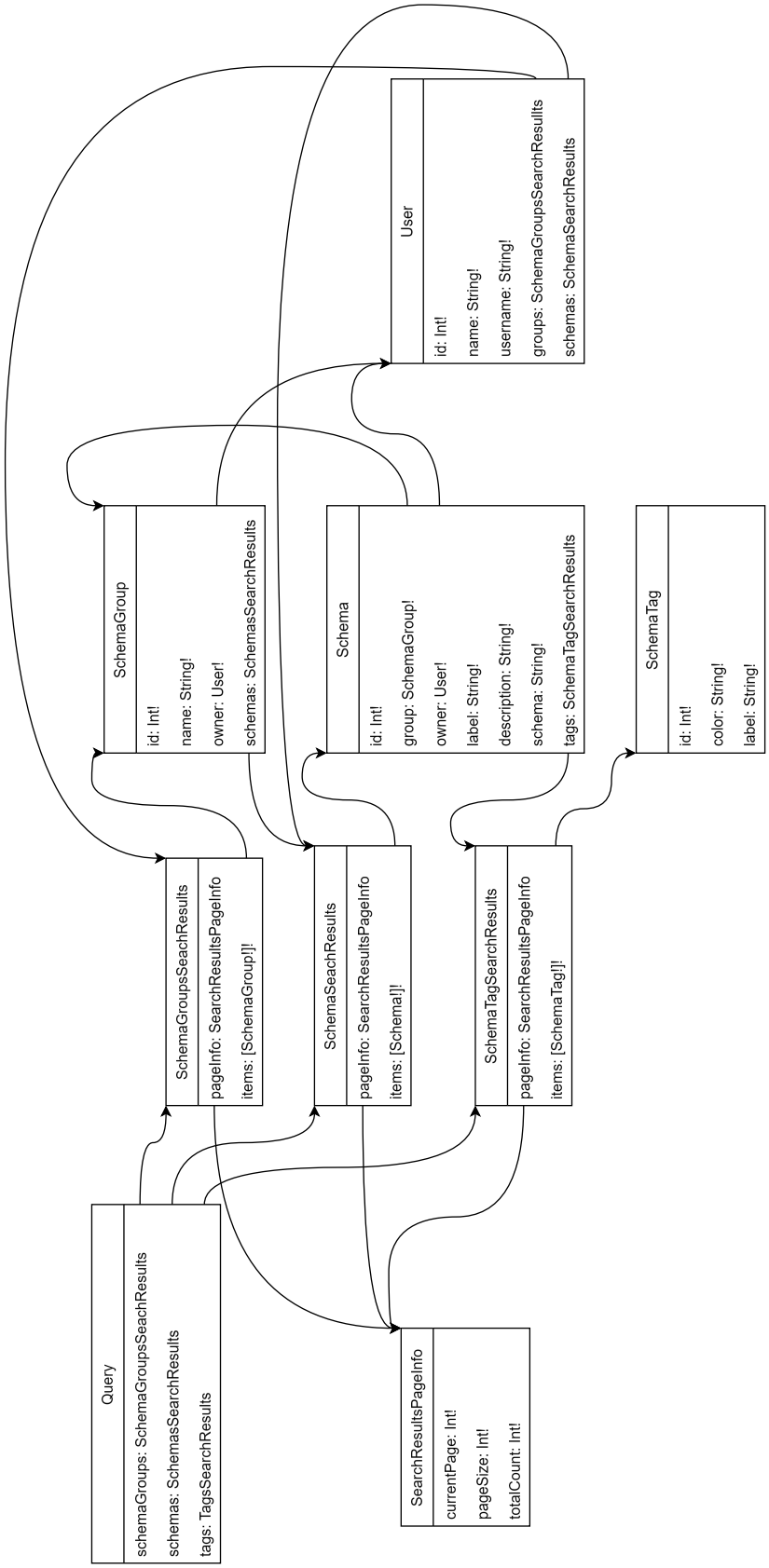
Dalším rozdílem je přístupový bod, endpoint, který je možné použít. *REST* definuje každý zdroj na samostatné, jednoznačně identifikovatelné adrese, zatímco *GraphQl* poskytuje jeden jediný endpoint, typicky */graphql*, nad kterým jsou prováděny všechny dotazy. Požadavek je tedy odesílán buď jako query string, nebo jako tělo požadavku a server se musí postarat o jeho zpracování do požadovaného výstupu.

S výše uvedeným je třeba počítat při následné implementaci, avšak existuje spousta jak serverových, tak klientských knihoven, které umožnují práci s tímto rozhraním, takže výsledné použití pro pogramátora je posléze velice jednoduché a intuitivní.

5.1.3 Simulační rozhraní

Simulace algoritmu jako taková, jak vyplývá z ilustrace 4.2, bude probíhat jako samostatná aplikace, která k tomuto účelu bude poskytovat otevřené rozhraní protokolu WSS[2]. V této sekci je popsána struktura zasílaných zpráv mezi klientem a serverem a rovněž stavy, ve kterých se simulace může nacházet.

[1] Vytvořeno pomocí nástroje drawio.com[38]
[2] WebSocket protocol over https



Obrázek 5.2 Vizualizace navrhovaného GraphQl schématu (Query)

Ačkoli protokol *WS* nijak nespecifikuje, jak má vypadat tělo zprávy zasílané mezi klientem a serverem, pro účely této aplikace bude vhodné se omezit na prostý formát *JSON*, který je možné následně zpracovávat hojně dostupnými nástroji. Jelikož každý algoritmus, který bude simulován lze popsat stavovým diagramem, vychází návrh rozhraní z tohoto faktu a poskytuje následující rozhraní pro komunikaci klient → server:

operation opearce, kterou má server v danou chvíli provést.

data volitelná data, která si přeje klient zaslat na server. Upřesňují danou operaci a umožňují serveru v přesnějším postupu.

V případě komunikace server → kliento je pak situace obdobná s tím rozdílem, že se rozlišuje chybová a úspěšná odpověď. V případě chybové odpovědi obsahuje zpráva následující pole:

code indikuje, zda je zpráva úspěšná nebo ne. Pro neúspěšnou zprávu se jedná o hodnotu 400.

error.message zpráva upřesňující chybu, která nastala. tato zpráva bude dále zobrazena klientovi.

V případě úspěšně provedené operace je pak odpověď následující:

code indikuje, zda je zpráva úspěšná nebo ne. Pro úspěšnou zprávu se jedná o hodnotu 200.

operation popisuje, co se na serveru stalo za operaci. Klient na základě této hodnoty rozhodne, co dále v odpovědi očekává a jak s ní naloží.

libovolná další data každá jednotlivá operace, která může na serveru nastat, může přidat další data do odpovědi. Formát a jejich klíče jsou pak nespecifikované.

Na základě těchto tří rámcových zpráv pak bude probíhat veškerá komunikace mezi klienty a serverem. Obecně platí, že chybové zprávy jsou zasílány pouze uživateli, který operaci požadoval a úspěšné odpovědi jsou šířeny na všechny klienty (tzv *broadcast*). Server může v některých případech ukončit spojení, pokud se jedná o závažnou chybu [3] a klient by měl být schopný na tento stav adekvátně zareagovat.

5.1.3.1 Přípustné operace

Obecně existuje pouze omezená množina operací, které je možné provést při přístupu na server. Každou z těchto operací provádí uživatel pod jednou ze tří rolí. Tyto role jsou následující:

visitor Pouhý návštěvník simulace. Jsou mu zasílány změny stavů, stejně jako všem dalším rolím, nicméně nemůže do simulace nijak zasahovat.

operator Může manipulovat se simulací algoritmu jako takovou, nicméně nemá pravomoce na zavření simulační místnosti, restartu simulace, výběr algoritmu a další.

owner Vlastník simulační místnosti. Může se simulací provádět cokoli, včetně změny rolí jiných uživatelů.

[3] například pokus o připojení do neexistující simulační místnosti

Operace, kterou mohou jednotliví účastníci vykonávat jsou uvedeny níže. Každá operace vyžaduje určitý stupeň oprávnění, což je zde také zachyceno:

create Vytvoří simulační místnost s daným obvodem a nastaví aktuálního uživatele jako vlastníka simulační místnosti. Není třeba mít jakákoli práva. Jako vstup jsou požadována data o simulovaném obvodu ve společném formátu, který byl popsán v kapitole 5.1.1.

join Připojí se k již vytvořené simulační místnosti s právy *visitor*. K provedení operace je třeba poskytnout jednoznačný identifikátor simulační místnosti.

reset Zastaví simulaci a vymaže její vnitřní stav. Po této operaci bude možné opět spustit novou simulaci, třebaže s jiným algoritmem. Není třeba žádný další vstup – server již obvod zná. Je nutné mít oprávnění *owner* pro vykonání této operace.

set_role Danému uživateli přiřadí zvolenou roli. K provedení je nutné mít oprávnění *owner* a jako vstup je nutné uvést identifikátor uživatele, jehož oprávnění mají být změněna a roli, na kterou má být povýšen/ponížen.

start_simulation Zahájí simulaci obvodu, který je v simulační místnosti. Požadovaným vstupem je identifikátor algoritmu, který má být simulován a volitelně porucha, která bude simulována jako první. Je nutné mít oprávnění *owner* pro vykonání této operace.

step_forward Pro již spuštěnou simulaci učiní krok vpřed. Nejsou potřeba žádné další data a je nutné mít oprávnění *operator*.

step_backward Učiní krok zpět na předchozí stav simulace. Nejsou potřeba žádné další data a je nutné mít oprávnění *operator*.

step_block_forward Provede skok o celý jeden blok vpřed (například rovnou vyplní celou tabulku singuárního pokrytí namísto jednotlivého procházení hradel). Nejsou potřeba žádné další data a je nutné mít oprávnění *operator*.

select_fault Vybere simulovanou poruchu nad obvodem. Tato operace lze provést pouze pokud to algoritmus umožňuje. Jako vstup je nutné uvést popis poruchy a je nutné mít oprávnění *operator*.

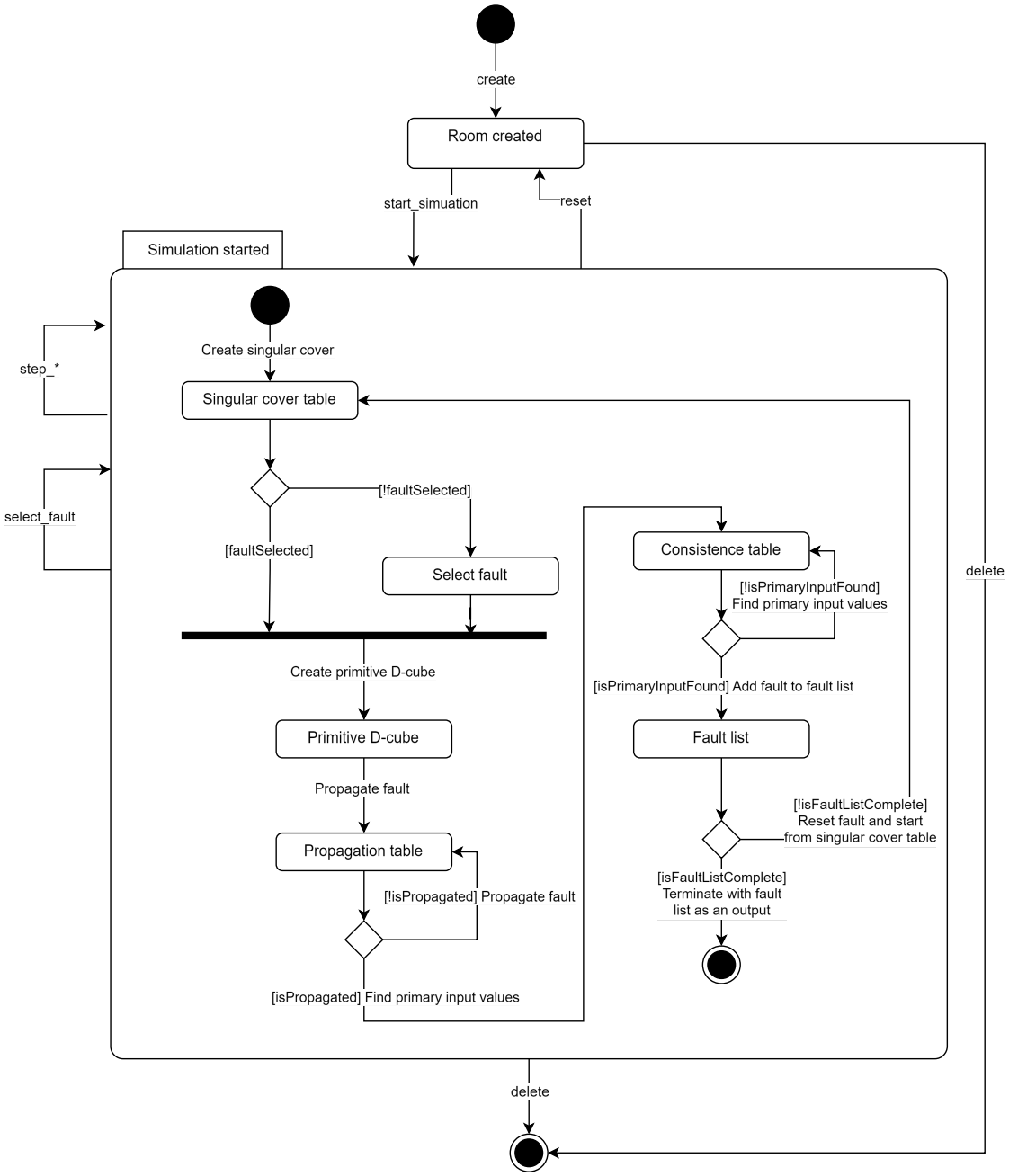
query_state Dotáže se serveru na stav simulace v daném bodě. Server vrátí požadovaný stav, případně chybovou hlášku, pokud takový neexistuje. Není potřeba mít žádné další oprávnění, tato operace je přístupná komukoli v simulační místnosti.

Vzhledem k potřebě udržovat spojení se simulační aplikací aktivní, je zde připravena ještě operace *ping*. Tato operace je požadována klientem v pravidelných intervalech, takže prohlížeč uživatele ponechá spojení aktivní. Tato operace je prázdná a nic nedělá.

5.1.3.2 Stavy simulace

Pomocí operací výše se pak simulace dostává do stavů, které se dají reprezentovat pomocí stavového automatu. Tento automat je zachycen na diagramu 5.3 a je ze něj patrné, které příkazy/operace změní vnitřní stav simulace a které ne. Na uvedené ilustraci jsou také zachyceny vnitřní stavy D-algoritmu[4], nicméně implementace jako taková bude navržena tak, aby umožnila simulaci libovolného algoritmu.

[4]Vizualizace D-algoritmu je zde velmi zjednodušená - diagram nezahrnuje řešení konfliktů backtrackingem pro nekonzistentní hodnoty. Nicméně pro ilustraci toho, jak je navržena komunikace se simulační aplikací je tento příklad dostatečný.



Obrázek 5.3 Vizualizace stavů simulace, včetně ukázkového D-algoritmu

V rámci zjednodušení diagramu jsou všechny operace, které začínají na *step_* sdruženy do jednoho přechodu.

V příkladu jsou uvedeny pouze operace, které mění stav simulace. Operace *join* a *set_role* nijak nemění stav simulace, pouze mění práva uživatelů. Ilustraci je možné použít také k vysvětlení rozdílu mezi operací *step_forward* a *step_block_forward*. První zmíněná provede krok algoritmu o nejmenší možnou jednotku, tedy algoritmus může vnitřně setrvat v daném stavu, pouze se změní vnitřní proměnné, se kterými pracuje. Druhá operace pak způsobí, že se algoritmus posune na další stav (například zde by to byl skok *Propagation table* → *Consistence table*), uživatel tak dostane výsledek rovnou, bez nutnosti procházet dílčí kroky. Tato funkce je vhodná zejména pro větší obvody, kdy jsou některé kroky uživateli zřejmé a potřebuje si ozřejmit jen určitou část algoritmu.

5.1.4 Autentizace

Autentizaci je možné provádět na různých aplikacích a různými způsoby, nicméně ne všechny způsoby jsou bezpečné pro klienta. Návrh autentizace a další práce s autorizačními tokeny, které uživatel obdrží je popsán právě v této kapitole. Jelikož je pak autentizace prováděna pouze u backendu aplikace a stejně tak tam je nutno dotazovat se na doplňující informace k uživateli na základě autorizačního tokenu (jméno a identifikátor), bylo rozhodnuto, že hlavním držitelem tohoto tokenu bude právě backend aplikace.

Přístupové tokeny budou následně uloženy v podobě HTTP only cookie[39]. Klientská aplikace tak nebude mít možnost s tokeny manipulovat, jelikož HTTP only cookies jsou dostupné pouze při provádění zabezpečených dotazů na server. Server tak bude mít informaci o tom, jaký uživatel požadavek vyvolal a bude tak schopen adekvátně odpovědět.

Veškeré HTTP požadavky, které by souvisely s autentizací uživatele pak budou prováděny výhradně skrze backend aplikaci, která se postará o celé zpracování požadavku, získání identity z autorizačního serveru a přesměrování uživatele zpět do klientské aplikace. Vzhledem k faktu, že klientská a serverová aplikace spolu budou komunikovat skrze *GraphQl* rozhraní, tak ve chvíli, kdy nastane chyba autorizace, bude tento fakt komunikován klientovi jako *GraphQl* error a bude z aplikace odhlášen. Další přihlášení bude nutné pro pokračování v použití aplikace.

5.2 Implementace autentizačních mechanismů

V prvotní fázi projektu bylo uvažováno o implementaci přihlášení pomocí *SSO Shibboleth*[40] skrze protokol *SAML2*. Za tímto účelem byl do backendové aplikace integrován modul *simpleSAMLphp* v režimu *Service Provider*[41]. Tento modul poskytuje nástroje k připojení aplikace na *identity provider (IdP)* službu v rámci ČVUT v Praze. Ještě před samotným napojením na službu bylo zhotoveno *mock*[5] prostředí jako samostatný kontejner tak, aby bylo možné si integraci pohodlně vyzkoušet. Tento krok byl nutný zejména z toho důvodu, že nebylo možné dohledat cvičné prostředí, které by poskytovala univerzita.

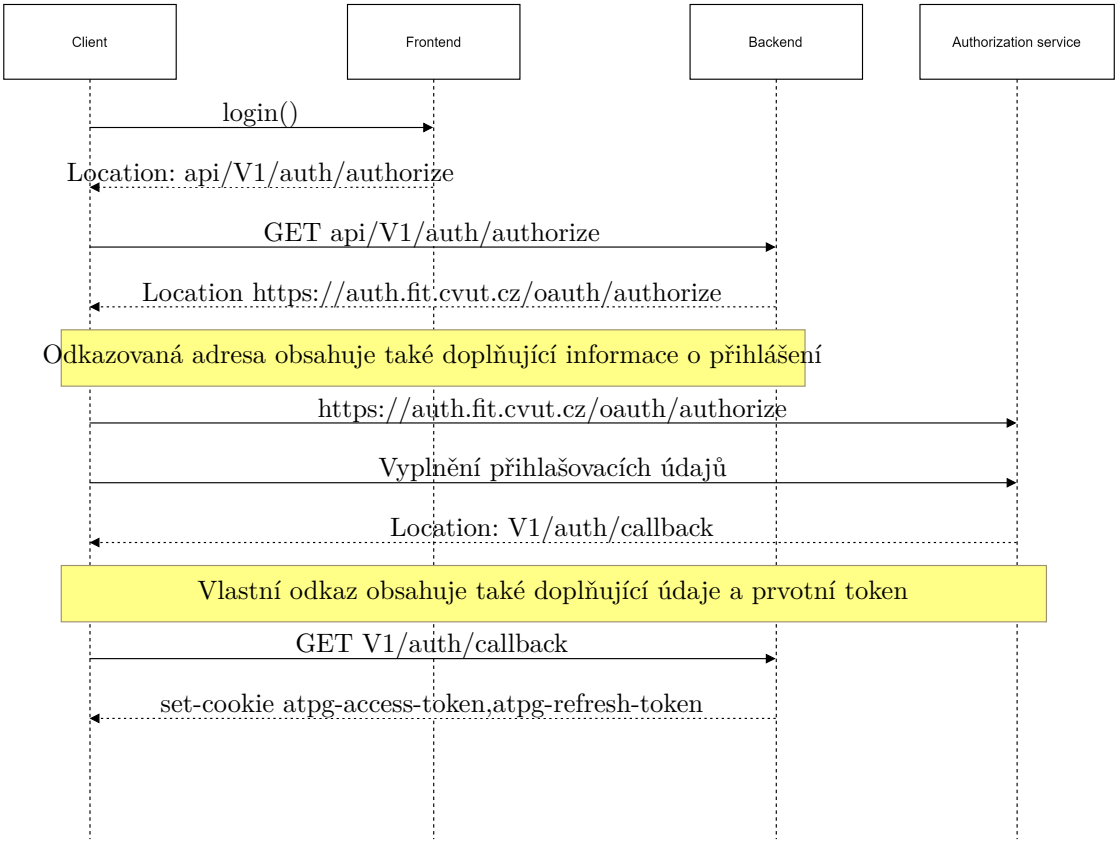
Samotný vývoj autentizace pomocí SAML2.0 protokolu byl proveden jako rozšiřující modul pro backendové prostředí. Díky tomu bylo možné v budoucnu vyměnit technologii pro přihlášení,

[5]Prostředí, které pouze nahrazuje funkci cílového systému. Poskytuje kompatibilní rozhraní a umožňuje testovat propojení bez nutnosti ostré integrace.

což se ukázalo jako dobrá volba. SSO Shibboleth, respektive IdP, neumožňuje samoobsluhu pro vytváření a registraci služeb, které mohou tuto technologii využívat. Jakožto implementátor je totiž nutné nejprve připravenou konfiguraci registrovat u daného IdP a až poté je možné využívat jeho služeb.

Z toho důvodu bylo v průběhu implementace rozhodnuto o využití modernějšího přístupu k autentizace uživatele pomocí standardu *OAuth 2.0*[33], který je podporován na FIT ČVUT v Praze velmi dobře. K vytvoření nového služby, která bude využívat přihlášení jednotným heslem ČVUT v Praze, je vytvořena samoobsluha[42], která umožňuje uživateli velmi pohodlnou správu a vytváření aplikací.

Vznikl tak druhý autorizační modul, který je, stejně jako *SAML 2.0*, možno vypnout dle potřeby v cílovém nasazení. Vzhledem k potřebám ve výuce bylo rozhodnuto, že přihlášení skrze *OAuth 2.0* plně pokryje požadavky, které jsou na aplikaci kladeny a je tak možné tuto náhradu provést. Do budoucna však stále zůstává možnost využít jednotného přihlášení srze *Shibboleth*, tento modul je v aplikaci stále přítomný.



Obrázek 5.4 Implementované OAuth 2.0 flow pro získání access tokenu a refresh tokenu

Na ilustraci 5.4 lze vidět jakým způsobem je implementován postup pro získání přístupového a obnovovacího tokenu. Oba tokeny jsou drženy nezávisle na backendové aplikaci u klienta v podobě HTTP-only cookies a jsou předávány serverové části aplikace pouze, pokud je požadavek na takovou komunikaci. Aplikace na serveru posléze ověřuje platnost přístupového tokenu na poskytnuté adrese *https://auth.fit.cvut.cz/oauth/check_token*.

5.3 Backendová aplikace

Samotná backendová aplikace, která má na starosti správu uložených dat a řízení přístupu k nim, je psána v jazyce PHP bez použití specifického frameworku. Volba nevyužít některý z již zaběhnutých řešení byla provedena s ohledem na minimalizaci velikosti výstupního balíčku pro běh v cílovém prostředí. Aplikace však využívá některé knihovny třetích stran, například pro práci s databází, parsování *GraphQl* schéma, a další.

Byl připraven jednoduchý systém pro správu instalovaných balíčků v aplikaci, kdy je možné jednotlivé balíčky vypnou a zapnout bez nutnosti je odinstalovat a měnit systém. K tomuto účelu bylo použito knihovny `PHP-DI`[43], která umožňuje vytváření definic dependency injection (DI) kontejneru za běhu aplikace z více souborů. Každý modul tak poskytne výchozí konfiguraci závislostí a systém se pak postará o jejich zavedení. Jakmile je konfigurace zavedena, je možné ji v aplikaci volně používat.

Zároveň je možné oddělit konfigurace používané jednotlivými moduly podle oblasti použití, takže aplikace může poskytovat různou konfiguraci například pro `webapi` rozhraní a jinou pro `GraphQl` rozhraní. Struktura dílčího modulu pak typicky vypadá následovně:

```
Module
├── etc
│   ├── di.php ..... Globální konfigurace DI kontejneru
│   └── module.php ..... Základní popis modulu včetně jeho identifikátoru
├── Model ..... Datové modely použité v aplikaci
└── Setup ..... Skripty, které jsou použity při zavádění aplikace
```

Konfigurace, který modul je zapnut a který ne je pak umístěna v souboru `/etc/modules.php`, kde je seznam modulů a k nim příznak, zda mají být zavedeny do systému. V ostatních ohledech je vývoj aplikace shodný s vývojem jiných PHP aplikací, tedy jako balíčkovací systém společně je využit nástroj *Composer*, který poskytuje také autoloading aplikačních tříd[44].

5.3.1 GraphQl rozhraní

Pro komunikaci skrze *GraphQl* rozhraní byla vybrána knihovna `webonyx/graphql-php`[45], která poskytuje vhodný základ pro efektivní vývoj schémat pro toto rozhraní. Knihovna umožňuje mnohé rozšíření, čehož bylo hojně využito například v oblasti vlastní direktivy pro vynucení autorizace.

Byl implementován vlastní kontext resolver, který umí do požadovaného dotazu dosadit objekt, který reprezentuje právě přihlášeného uživatele. Díky tomu je pak velmi snadné dohledat příslušné projekty a skupiny, které se tohoto uživatele týkají, jelikož v každém místě aplikace je jednotné rozhraní. Představené rozšíření základního kontextu je snadno rozšiřitelné o další položky, jelikož je psáno s ohledem na univerzální použití do budoucna.

Podobně byl představen resolver pro direktivy přítomné v GraphQl schématu, který se stará o vykonávání příslušných direktiv, které se ve schématu vyskytují. Aplikace takto podporuje dvě direktivy a to:

Výpis kódu 5.2 Ukázka zápisu GraphQl schéma

```
1
2  extend type Query {
3      schemaGroups(search: SearchInput): SchemaGroupsSearchResults
4      @resolver(class: "GetGroupList")
5      @authorization
6  }
7
8  extend type Mutation {
9      # Group operations
10     createGroup(name: String!): SchemaGroup
11     @resolver(class: "CreateGroup")
12     @authorization
13 }
14
15 type SchemaGroupsSearchResults implements SearchResults {
16     pageInfo: SearchResultPageInfo!
17     items: [SchemaGroup!]!
18 }
19
20 type SchemaGroup implements SearchResultItem {
21     id: Int!
22     name: String!
23     owner: User!
24     @resolver(class: "ObjectOwner")
25     schemas(search: SearchInput): SchemasSearchResults
26     @resolver(class: "GroupSchemas")
27 }
```

resolver direktiva, která slouží k popisu třídy, která se postará o získání hodnoty pro danou položku,

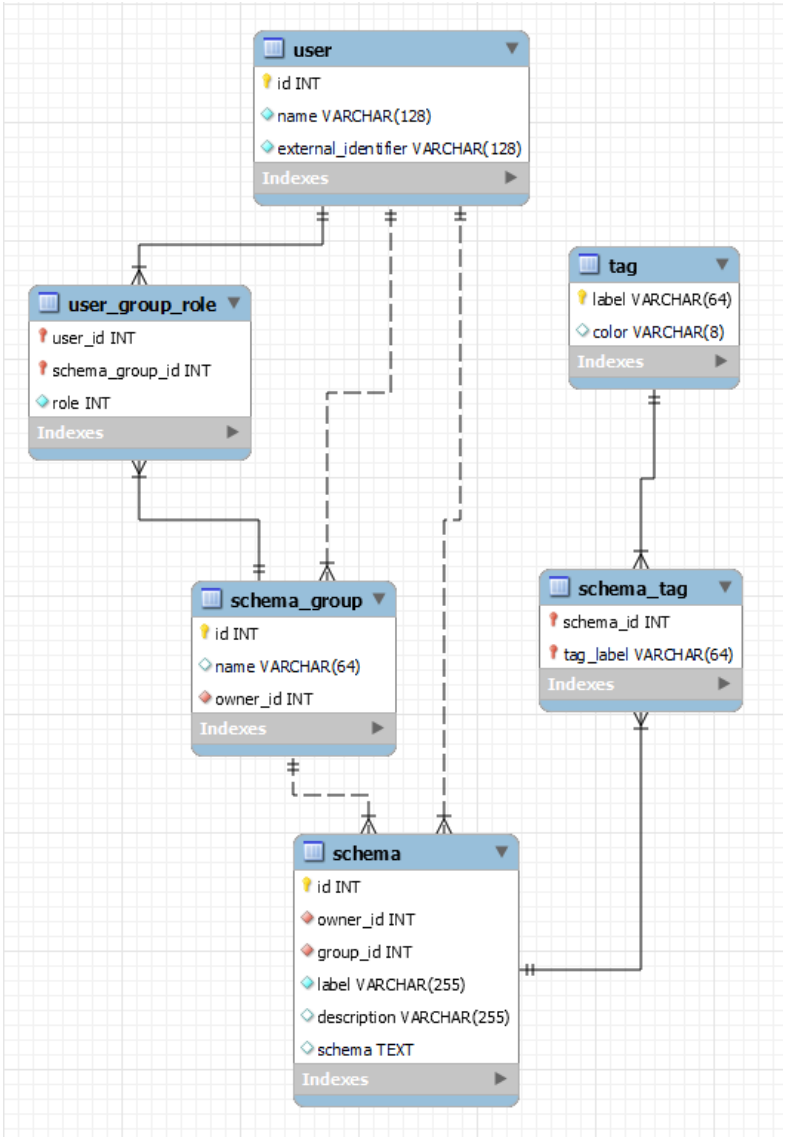
authorization zajistí, že k dané položce bude možné přistoupit pouze pod podmínkou přihlášení.

Samotné schéma, které bude aplikace využívat, je popsáno na ilustraci 5.2, pro představu je však vhodné uvést ještě ukázku kódu, jak vypadá zápis části tohoto schématu, viz výpis kódu 5.3.1.

Za pomocí takto rozšířených možností pro tvorbu rozhraní bylo implementováno schéma v plném rozsahu, včetně operací pro vytváření a úpravu objektů.

5.3.2 Databázové schéma

Databázové schéma pro vyvíjenou aplikaci je velmi podobné komunikačnímu schématu. Schéma je zachyceno na ilustraci 5.5 a bylo vytvořeno v nástroji MySQL Workbench[46]. Datové schéma podporuje všechny položky, které jsou potřebné pro správnou funkci aplikace na straně klienta.



Obrázek 5.5 Databázové schéma aplikace

5.3.3 Zpracování dat

Jelikož backendová aplikace sama o sobě pouze zprostředkovává přihlášení uživatele a následně slouží čistě jako mezivrstva, mezi databází a klientskou aplikací, byl návrh architektury poměrně přímočarý. Pro správu data byl zvolen návrhový vzor Repository[47], který slouží jako správce dat a poskytuje tak metody pro správu databázových objektů. Každý repozitář v aplikaci pak obsahuje následující metody:

get vrátí požadovaný objekt na základě jednoznačného ID

getList vrátí seznam objektů na základě vstupních parametrů

save uloží poskytnutý objekt do databáze

delete smaže zadaný objekt z databáze

5.4 Klientská aplikace

Klientská aplikace slouží k prezentaci dat a výsledků simulace algoritmu uživateli. Byla psána za pomocí následujících nástrojů:

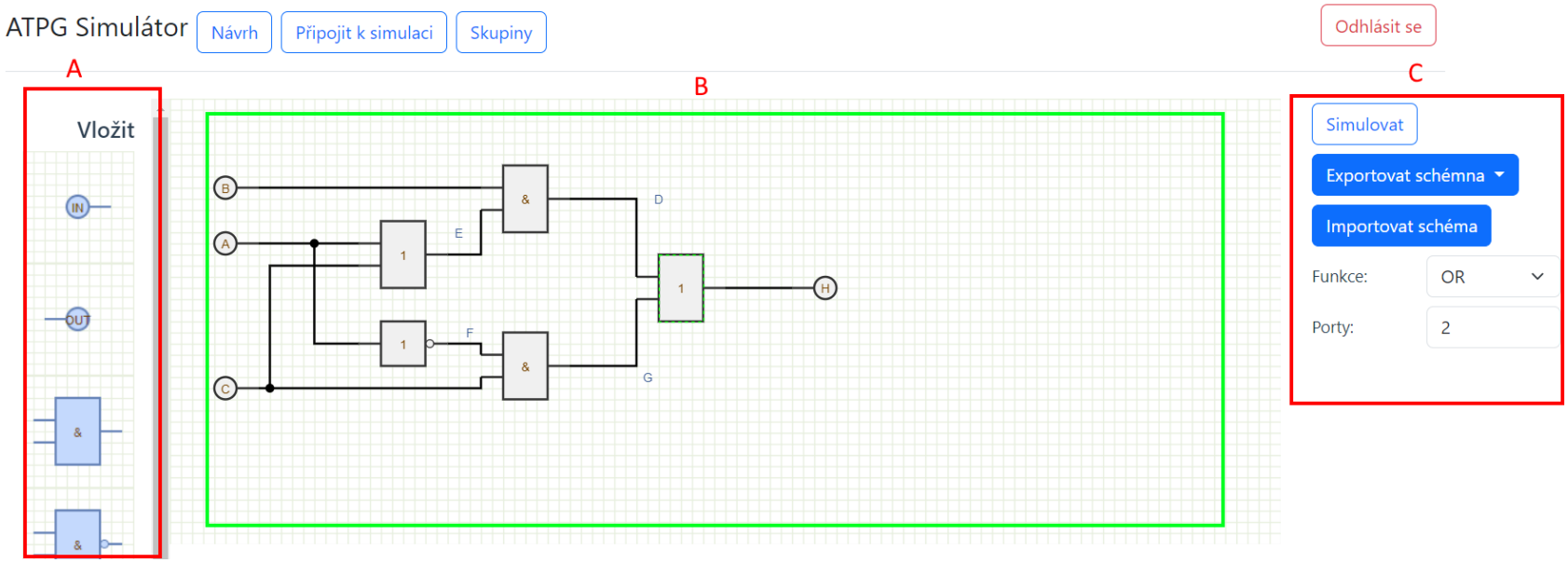
1. frameworku *Vue 3*[48] pro výkoný kód,
2. knihovny *Vue Apollo GrapQl*[49] jako klient pro komunikaci s GraphQl rozhraním backendové aplikace,
3. knihovny *maxGraph*[50] pro vykreslování a editaci obvodů,
4. sady stylů *Bootstrap 5*[51] pro stylování a pozicování prvků stránky.

Jako první byl zahájen vývoj na samotném kreslícím nástroji, který je postaven na již zmíněné knihovně *maxGraph*, přičemž bylo nutné tuto knihovnu značně přizpůsobit potřebám této aplikace. Pro přihlášeného i nepřihlášeného uživatele je k dispozici kreslící plátno, které umožňuje vkládání jednotlivých prvků obvodu na plátno.

Aplikace využívá npm.js verzovací nástroj, pomocí kterého jsou instalovány doplňující balíčky. Pro běh aplikace při vývoji je použit nástroj *Vite*[52], který poskytuje pro frontendový vývoj velmi užitečné nástroje, zejména pak hot reload pro okamžitý náhled komponent při změně zdrojového souboru. Pro nasazení do staging a produkčního prostředí je poté aplikace, opět pomocí nástroje *Vite*, kompilována do statických stránek.

5.4.1 Kreslící plátno

Vzhled plátna je zachycen i s vysvětlujícími popisky na ilustraci 5.6. Podporovaná hradla, společně se vstupy a výstupy, je možné vkládat přetažením z levého nástrojové panelu. Po zahájení vkládání chycením prvku a jeho tažením, je uživateli ukázána pomocí vodítka poloha, kde bude prvek umístěn. Tyto vodítka jsou dále zobrazena, pokud je nějaký prvek přemisťován.



Obrázek 5.6 Kreslící plátno aplikace
A: levý panel nástrojů, obsahuje prvky, které je možné do obvodu vložit
B: samotné kreslící plátno obsahující číslicový obvod
C: pravý panel nástrojů, obsahující možnosti pro práci se schématem a změnu vlastností vybraných objektů

V rámci implementace vyvstalo několik problémů, přičemž největší se týkal větvení vodičů. Nakonec byl tento problém vyřešen omezením, které je kladeno na uživatele, pokud chce takové větvení nakreslit – je nutné táhnout vodič z dosud neobsazeného vstupního portu hradla. Veškeré prvky je možné volně přemisťovat pomocí tahu myši při stisknutém levém tlačítku myši.

Plátno není nijak omezeno co se týče velikosti – uživatel může volně přesouvat objekty a pro posun náhledu je možné stisknout pravé tlačítko myši a přesunout se tak pohledem na jinou část obvodu.

V pravé části od kreslícího plátna je k dispozici panel nástrojů, které slouží k upřesnění vlastností jednotlivých objektů. Pro hradla lze vybrat příslušnou funkci, kterou bude plnit, a počet portů, které budou na vstupu hradla[6]. Pro vodiče a vstupní a výstupní porty je pak k dispozici možnost pojmenování vodičů, přičemž toto pojmenování musí být jednoznačné a nezaměnitelné s jiným vodičem ve stejném obvodu.

K dispozici je rovněž možnost pro import a export schéma, přičemž podporovány jsou dva formáty: *bench*, který využívají pro práci i jiné nástroje, a formát *atpgv*, který vznikl pro tuto aplikaci a jeho schéma je ilustrováno na obrázku 5.1. V neposlední řadě obsahuje panel možnost simulovat obvod, který spustí simulační prostředí.

5.4.2 Simulační prostředí

V tomto prostředí vidí uživatel stejný obvod, jako při návrhu, avšak není možné jej nijak editovat a měnit jeho parametry. Veškeré změny, které by se týkaly vzhledu, musí být iniciovány ze simulační aplikace, přičemž zde jsou stavy pouze renderovány. Vzhled prostředí je zachycen na ilustraci 5.7. Jednotného vzhledu bylo dosaženo použitím stejných komponent pro kreslící plátno a pro simulační prostředí, což framework Vue pohodlně umožňuje.

V tomto prostředí probíhá komunikace se simulační aplikací, která poskytuje data o tom, jak vypadají vnitřní stavy algoritmu pro daný krok. Uživatel vidí v seznamu kroků, umístěném ve spodní části obrazovky, jak probíhá průchod algoritmem a může si příslušný mezistav prohlédnout (ikona oka na řádku kroku). Simulační aplikace posílá mezistavy ve formě tabulek tak, aby bylo možné je prezentovat uživateli na výstupu v přehledné a snadno pochopitelné formě.

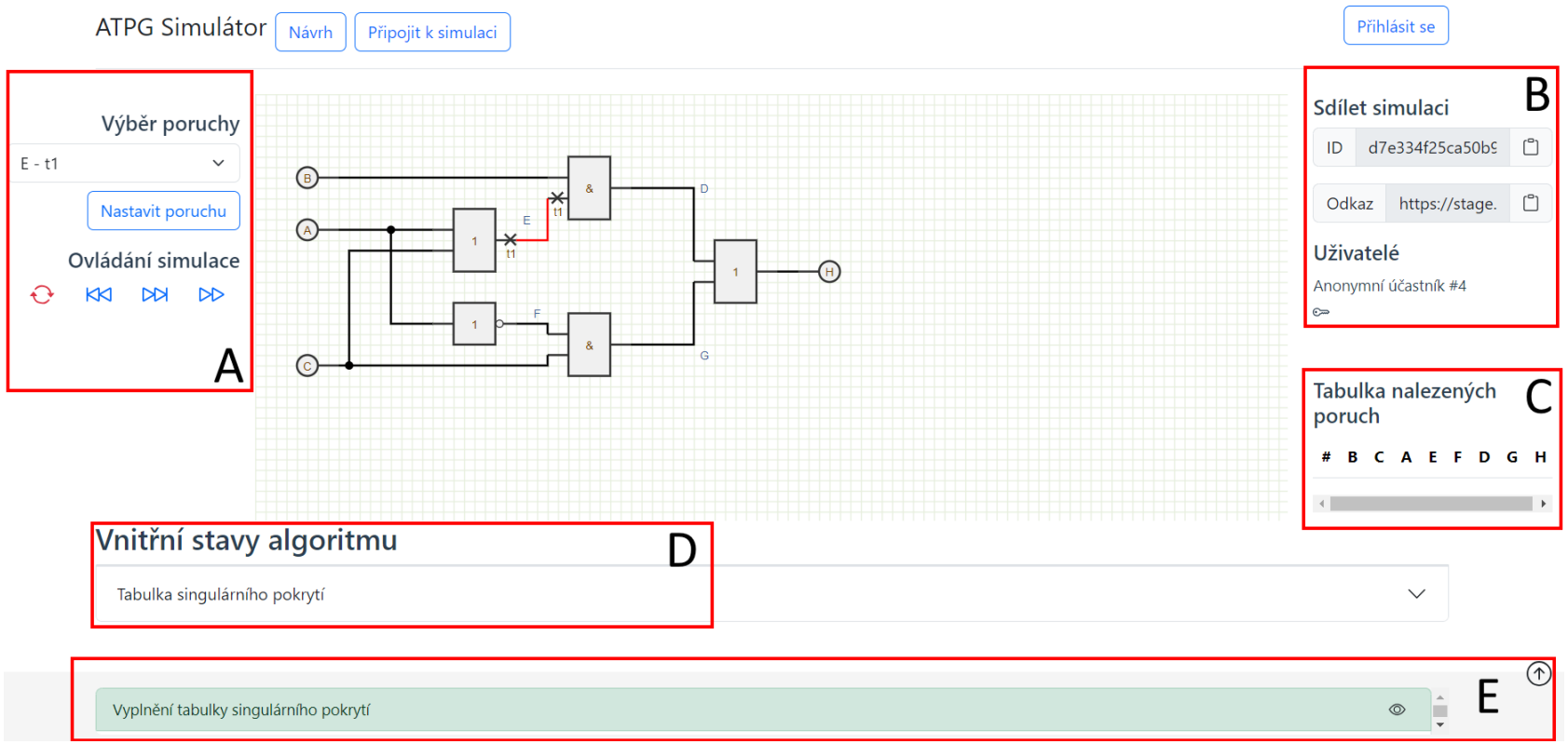
Aplikace umí také zobrazit stav typu *overlay*, který slouží k podchycení změny na daném vodiči. Aktuálně se využívá změny barvy vodiče a vykreslení ikony poruchy na vodiči, nicméně v aplikaci je pro budoucí použití zabudována také podpora pro vykreslování stavů jednotlivých hradel.

Prostředí na straně klienta nedefinuje žádným způsobem, jakou barvou a kdy bude který vodič obarven – toto je čistě v gesci simulační aplikace, která pro to musí poskytnout data. Rovněž si klientská aplikace neudržuje seznam všech stavů, které simulace nabývá, jelikož by takové uchovávání mohlo znamenat nemalé paměťové nároky.

Namísto toho je komunikace obstarána pomocí celých vnitřních stavů, přičemž pokud je takový vnitřní stav zpřístupněn uživateli, musí mít na straně simulační aplikace *renderer*, který ho převede do serializovatelné podoby a na straně klienta opět *renderer*, který prezentuje serializovaná data, poslána skrze WebSocket komunikaci, uživateli.

Díky takovému návrhu komunikace není samotná implementace vázána na konkretní algorit-

[6]Tato funkce není podporována pro hradlo XOR a NOT.



Obrázek 5.7 Simulační okno aplikace
Kreslící plátno se nachází uprostřed
A: levý panel nástrojů, obsahuje prvky, kterými se řídí simulace
B: pravý panel nástrojů, obsahující možnosti pro řízení přístupů uživatelů
C: vnitřní stavy algoritmu zobrazené v pravém panelu – pro D-algoritmus je to tabulka nalezených poruch a seznam netestovatelných poruch
D: vnitřní stavy algoritmu zobrazené pod plátnem, typicky pro velké tabulky – pro D-algoritmus se jedná o všechny ostatní stavy
E: notifikace o událostech v simulaci – uživatel má možnost tuto lištu rozbalit a vybrat z ní stav, který si může následně prohlédnout

mus, což je důležité budoucím rozvoji, například rozšíření o *FAN* a *PODEM* algoritmy. Rozšíření o další algoritmus, pokud nebude třeba změnit styl vykreslování hodnot, je tak pouze otázkou zásahu do simulační části aplikace a nevyžaduje změnu na straně klienta. To s sebou přináší výhody v podobě snadnějšího následného vývoje a údržby.

Veškerá data o obvodu, který uživatel vytváří a simuluje jsou uchovávána v *localStorage* prohlížeče. Uživatel má díky tomuto řešení možnost se vrátit k rozpracovanému obvodu později nebo v případě náhlého výpadku, zavření prohlížeče, případně jiné, neočekávané události, nepřijde o svá data.

5.4.3 Uživatelská administrace

Uživatelská administrace poskytuje přihlášeným osobám přehled všech skupin, do kterých náleží. Aplikace opět spoléhá na data z backendové části řešení, které obsahují vše potřebné k překreslení obsahu. Díky návrhu autentizace, která využívá HTTP-only cookies není nutné nijak řešit přidání tokenu k příslušným *GraphQl* dotazům. Zamezuje se tak i případným bezpečnostním problémům, které mohou manipulací citlivých dat v aplikaci vzniknout.

Administrace je členěna na tři hlavní stránky:

1. přehled skupin – z této stránky je možné vytvořit novou skupinu nebo otevřít některou ze skupin zobrazených v seznamu,
2. přehled projektů – po otevření skupiny se zobrazí podobný přehled, ovšem pro projekty,
3. editace projektu – podobná obrazovka jako pro kreslení obvodu, avšak obsahuje více kontrolních prvků (umožňuje přidat popis a štítky k obvodu).

Pokud by nastal během prohlížení aplikace problém s přihlášením (vyprší *access* i *refresh* token), je uživatel odhlášen, přesměrován na úvodní stránku a vyzván k opětovnému přihlášení. Data o obvodu jsou opět uchována v paměti prohlížeče a po přihlášení je uživateli umožněno se vrátit k rozpracované činnosti.

5.5 Simulační aplikace

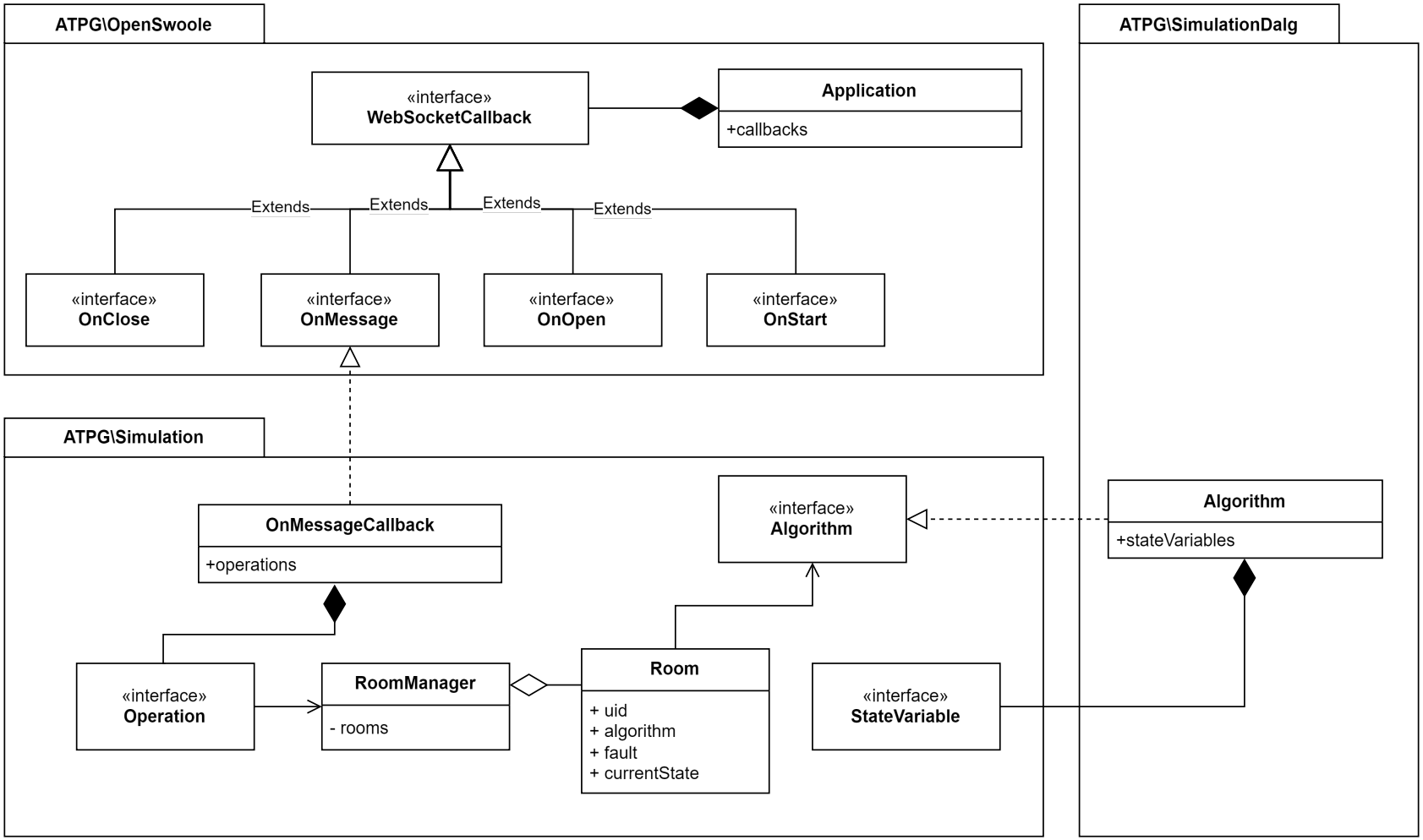
Simulační aplikace je postavena, podobně jako backendová vrstva, v jazyce PHP za použití stejné kostry, jako byla popsána v kapitole 5.3, obsahuje však jiné moduly. Hlavní změnou je, že aplikace jako celek běží v režimu HTTP serveru za pomocí technologie OpenSwoole[30]. Při vývoji byl kladen důraz na nezávislost na konkrétním algoritmu a rozšiřitelnost do budoucna o další operace.

Při spuštění aplikace je tak v hlavním vlákně využíváno několik obsluh událostí, které jsou pomocí návrhového vzoru *Pool*[53] udržovány oddělené a je možné je snadno doplnit o další, případně vyměnit za definice jiné. Navíc událost `Message` opět přijímá *Pool* operací, které jsou podporovány pro ovládání simulace (seznam takových operací viz 5.1.3.1).

Návrh simulačního jádra je zachycen na diagramu 5.8. Tento diagram je značně zjednodušený tak, aby bylo možné jej v práci prezentovat, nicméně poskytuje dobrý přehled o tom, jak je

uvnitř aplikace vystavěna. V komponentě *OpenSwoole* lze vidět deklarace možných callbacků pro události webserveru, které jsou následně realizovány v komponentě *Simulation*. Ta obsahuje deklaraci základních komponent potřebných pro abstrakci jednotlivých instalovaných algoritmů. Teprve komponenta *SimulationDalg* je implementován D-algoritmus. Třídy a rozhraní použitá pro interní potřeby v rámci jednotlivých balíčků byly vynechány.

Díky této mnohavrstvé separaci je případná implementace dalšího algoritmu velmi jednoduchá, jelikož ze strany aplikace vyžaduje pouze implementaci rozhraní `Algorithm` z komponenty `Simulation` a dále se vývojář zabývá pouze implementací tohoto algoritmu. Nemusí řešit jak dostane stav na simulační plátno ke klientovi – o toto se postará platforma automaticky, pokud bude dodrženo předepsaného formátu dat.



Obrázek 5.8 Zjednodušený diagram vnitřní architektury simulační aplikace

Kapitola 6

Testování

Samotná implementace nového řešení pro výuku prošla také testovací fází, ve které se odhalily a doladily některé nedostatky. Existují různé testy podle účelu a toho, kdo se jich účastní. V této části práce je zevrubně popsáno, jaké testovací procesy jsou při vývoji aplikace použity a které kroky byly provedeny k co nejoptimálnějšímu vývoji.

Samotný vývoj aplikace s sebou často přináší také mnohé změny, které mohou pozměnit, či případně zcela omezit funkčnost nějakého aplikačního celku. Z toho důvodu bylo v průběhu vývoje přijato několik opatření, aby se případným problémům zamezilo. Mezi tyto opatření patří zavedení jednotkového testování pro simulační část aplikace, pro kterou je klíčové správné nalezení hodnot pro dílčí kroky algoritmu a dále využití nástroje *Postman*[54], který umožňuje zavedení API testování.

V závěru implementace bylo provedeno také uživatelské testování s vyučujícím předmětu NI-TSP, které dalo mnohé podněty a návrhy na vylepšení aplikace a přineslo pohled na použití aplikace ze strany cílového uživatele.

6.1 Jednotkové testování

Pro simulační jádro bylo zavedeno jednotkové testování, které umožňuje automatický test aplikace při každém nasazení, případně změně u vývojáře. Za tímto účelem je využito nástroje *PHPUnit*[55] ve verzi 10. Konfigurace pro tento nástroj byla zapsána tak, aby univerzálně pokryla případné další testy do budoucna.

Jednotkové testování se zaměřilo primárně na ověření správnosti hodnot pro singulární pokrytí hradel a dále na správnost hodnot pro přenosové D-krychle. Samotný test se spouští přímo z vývojového prostředí a za tímto účelem je připraven příkaz `make test`.

6.2 API testování

Pro testování navrhovaného *GraphQl* rozhraní bylo použito nástroje *Postman*[54], ve kterém byla připravena kolekce, která umožňovala spouštění automatických sekvencí, které následně pomocí funkce post-script tohoto nástroje testovali správnost odezvy daného prostředí. Postman jako takový umožňuje také export kolekce do přenositelného formátu a spouštění pomocí příkazové řádky, nicméně tato možnost nebyla využita.

6.3 Uživatelské testování

V závěru implementace proběhlo uživatelské testování s vyučujícím předmětu NI-TSP, tedy uživatelem, na kterého cílí tato aplikace. Testování probíhalo formou úkolů uživatele a sledování odezvy a míry, do jaké byl uživatel orientován při používání frontendové aplikace, zejména kreslícího plátna a simulační části aplikace.

Uživatelské testování odhalilo problémy, které byly zapracovány do výsledného řešení. Seznam úkolů, které uživatel dostával, jeho odezva a výsledné změny v implementaci na základě této odezvy, jsou uvedeny níže v podobě podsekcí. Kompletní původní záznam z testování je součástí této práce jako příloha.

Nakreslete libovolný obvod

Plné znění zadání: Nakreslete libovolný obvod, který bude obsahovat alespoň tři hradla a libovolný počet vstupů a jeden výstup.

Odezva uživatele

Orientace na plátně dobrá, ovládací prvky pro vložení hradel dosažitelné bez problémů a jednoduše rozpoznatelné typy hradel. Obvod nakreslil, nicméně nepropojil vstupy a výstupy s hradly.

Provedené změny

Důvodem, proč uživatel nespojil vstup a výstup s hradly byl předpoklad, že jakmile umístím vstup nebo výstup tak, aby se dotýkal již existujícího vodiče/výstupu, bude propojen. Tento poznatek byl zapracován do aplikace a nyní, pokud je vstup nebo výstup jakéhokoli objektu při vložení do schématu bezprostředně navazující na jiný volný vstup nebo výstup, dojde k propojení vodičem a užival pak nemusí manuálně doplňovat propojení.

Pojmenování vodičů

Plné znění zadání: Nyní prosím pojmenujte vodiče nakresleného obvodu.

Odezva uživatele

Ovládací prvky byly snadno dosažitelné a uživatel nejevil problémy při plnění tohoto úkolu.

Přidání hradla a větvení

Plné znění zadání: Přidejte čtvrté hradlo, které bude napojeno na původní obvod způsobem, který vytvoří větvení.

Odezva uživatele

Přidání hradla uživatelem bylo bezproblémové, zadání bylo splněno z pohledu uživatele bez problému. Aplikace vykazovala drobné problémy při přichycení vodičů k hradlu (odezva nebyla pro uživatele uspokojivá).

Provedené změny

Při testování vyvstala chyba, která se projevila špatným označením vodiče. Vodiče v aplikaci mají být jako celek, bez ohledu na větvení, od výstupu libovolného hradla na libovolný počet vstupů jiných hradel. Vodiče, které sousedí se vstupem/výstupem přejímají označení takového vstupu/výstupu.

Uvedená chyba, kdy byl vodič neoznačen, přestože sousedil s výstupem, byla opravena a nyní se v aplikaci neprojevuje.

Zahájení simulace

Plné znění zadání: Zahajte simulaci navrženého obvodu a vyberte algoritmus, kterým se bude simulovat.

Odezva uživatele

V počáteční fázi byly ovládací prvky snadno dosažitelné, uživatel se snadno orientoval a zahájení simulace jako takové zvládl bez problému. Náročnější byla orientace v ovládacích prvcích samotné simulace, kdy byl uživatel zmatený, které tlačítko znamená kterou akci. Následně proběhlo krokování algoritmu, u kterého byl sice prováděn postup vpřed, nicméně uživatel si nevšiml změn stavů algoritmu. To bylo dáno tím, že nové stavy se na monitoru uživatele vykreslovali mimo zobrazovanou plochu a tedy nenastala žádná vizuální odezva na provedený krok.

Ačkoli by se mohlo zdát, že problém se týkal pouze vnitřních stavů vykreslených mimo zobrazení prohlížeče, uživatel nezaznamenal ani doplnění poruchy do tabulky nalezených poruch. Byl tak ponechán zdánlivému přesvědčení, že již není co dále krokovat. Navíc nebyla zaznamenána ani změna poruchy, pro který se hledá testovací vektor.

Dalším problémem byl s vizuálním zobrazením tabulky nalezených poruch, kdy tato tabulka nebyla zobrazena celý. Uživatel tak neviděl celý obsah a jeho práci to tak dále ztížilo.

Provedené změny

Do aplikace byla přidána funkce sledování jednotlivých kroků simulace s tím, že jejich přehled je vykreslován ve spodní části obrazovky, která je na fixní pozici a tedy je vždy vidět. Seznam provedených kroků je vykreslován postupně s tím, jak probíhá simulace a je možné si daný krok zobrazit i zpětně. Samotnou plochu, na kterou jsou vykreslovány názvy kroků je možné zvětšit tak, aby ukazovala vždy alespoň šest kroků.

K ovládacím prvkům aplikace byl pak doplněn popisek, který se zobrazí při najetí myši. Díky tomu je uživatel informován i textem, ne jen vizuálně, co dělá které tlačítko v simulaci. Tabulka testovacích poruch byla opravena tak, aby nezalamovala hodnoty a zobrazovala se vždy celá, případně s posuvníkem tak, aby bylo možné ji zobrazit celou.

Restart simulace

Plné znění zadání: Restartovat provedenou simulaci, vybrat jinou poruchu než v předchozím úkolu a krokovat algoritmus až do konce.

Odezva uživatele

Vzhledem k detailnímu seznámení uživatele s testovacím prostředím nečinil tento úkol problém.

Připojení uživatele k simulaci

Plné znění zadání: Pošlete odkaz na simulační místnost.

Odezva uživatele

Uživatel sice nalezl příslušné pole, které obsahovalo odkaz na připojení, nicméně vzhledem k absenci popisku nedůvěřoval tlačítku na zkopírování obsahu pole do schránky. Po připojení dalšího uživatele do simulační místnosti bylo nutné vysvětlit jednotlivé ikonky za jménem uživatele.

Provedené změny

Stejně jako v předchozích bodech spočívala hlavní změna v úpravě vizuálního zobrazení prvků. Byly doplněny popisky k jednotlivým ikonkám a přidány odezvy, v podobě notifikace v dolní části obrazovky, na zkopírování odkazu a ID simulace do schránky.

Sledování změn

Plné znění zadání: Sledujte jak druhý uživatel krokuje poruchu a popište co se stalo.

Odezva uživatele

Uživatel byl orientován a byl schopen popsal přesně co se stalo. Nicméně bylo navrhnuto řešení, které již bylo popsáno výše, v podobě stavového řádku, který bude zobrazován vždy.

Kapitola 7

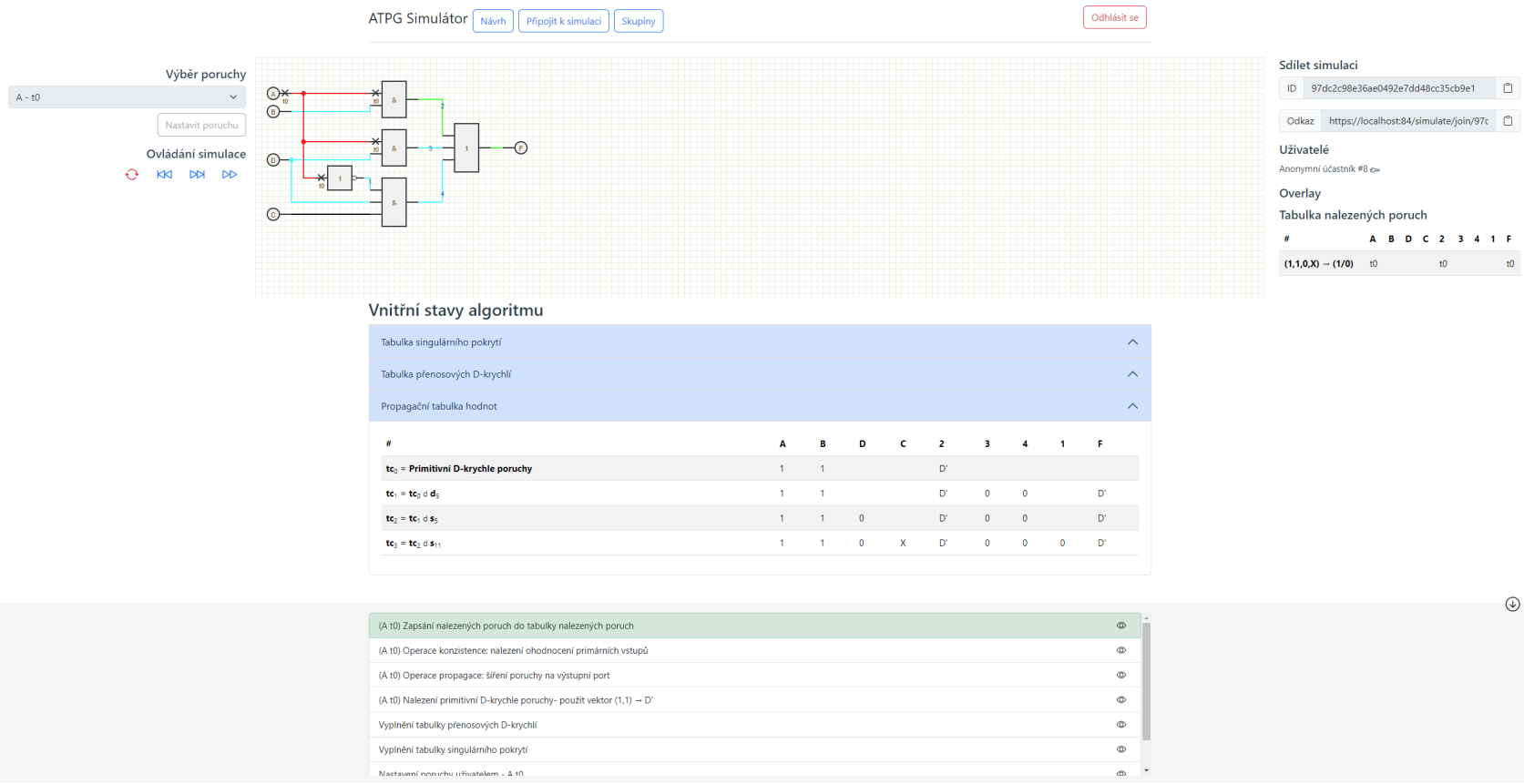
Základní datová sada

Součástí této práce je také základní datová sada, která obsahuje často používané příklady z výuky. Kapitola poskytuje přehled vytvořených obvodů pro možné okamžité užití ve chvíli, kdy uživatel chce vyzkoušet funkčnost aplikace.

Jednotlivé datové soubory, které obsahují popis obvodů kompatibilní se simulační částí této aplikace, jsou k dispozici jako příloha k této práci. Uživatel může nahrát tyto soubory do aplikace pomocí funkce import na kreslícím plátně. V základní sadě existuje celkem 7 obvodů, pričemž každý z nich je možné simulovat v aplikaci. Příklad, jak taková simulace může vypadat je uveden na ilustraci 7.1.

Na ilustraci lze ve spodní části obrazovky vidět stavový řádek s vyznačeným průchodem algoritmu. Jednotlivé vodiče obvodu jsou barevně znázorněny podle funkce toho, jak algoritmus procházel obvod. Konkrétně se jedná o tyto barvy:

- červená: poruchový vodič – na tomto vodiči se nachází porucha, pro kterou se snažíme nalézt tenstovací vektor,
- zelená: zcitlivěná cesta – přes tyto vodiče se šíří signál s hodnotou D, respektive D',
- tyrkysová: vodiče, které nabývají hodnoty na základě operace konzistence,
- černá: vodiče, na jejichž hodnotě nezáleží, nebo na ně algoritmus zatím nenarazil.



Obrázek 7.1 Příklad simulace obvodu na simulátoru - obvod 5 z testovací sady

Kapitola 8

Závěr

Výsledkem práce je funkční aplikace pro vizualizaci postupu ATPG algoritmů. Aplikaci se podařilo vyvinout společně s vizuálním nástrojem pro samotnou tvorbu logických obvodů, včetně možnosti importu a exportu do formátu *bench* a vlastního popisného formátu obvodu *atpgv*, používaném na fakultě informačních technologií ČVUT v Praze v předmětu NI-TSP. Uživatel má možnost si jednotlivé obvody uložit, organizovat do skupin a dále označovat štítky pro lepší přehlednost.

V průběhu realizace byla problematická zejména implementace autentizace pomocí protokolu *SAML2* skrze portál *SSO Shibboleth*, jelikož neexistuje způsob, jakým přihlásit aplikaci k možnosti využívat tuto technologii automaticky. I přes tento problém se však podařilo vyvinout základ přihlášení skrze tuto službu a pro další vývoj byla nalezena adekvátní náhrada v podobě autorizačního serveru FIT, který poskytuje samoobsluhu pro tvorbu aplikací používající *oAuth2.0* protokol.

Další výzvou byl samotný návrh architektury řešení, kdy nakonec byla zvolena cesta dekompozice architektury na samostatné aplikace, které se mohou mnohem lépe a cíleně zaměřit na jednotlivé dílčí implementace. Dále pak může být každá jedna dílčí část projektu snadno sestavena do příslušného Docker image a nasazena na cílový server. Kontejnerizace pomocí nástroje Docker bylo taktéž hojně využito při lokálním vývoji a výsledný projekt mimo jiné obsahuje také nástroje pro práci na lokálním stroji vývojáře, zahrnující kompletní vývojové prostředí právě pomocí Docker kontejnerů.

Pro usnadnění a snadnou správu vytvořené aplikace na serveru bylo zvoleno vytvoření popisu pro Kubernetes prostředí v podobě šablon nástroje Helm. Tyto šablony je možné použít pro nasazení aplikace na cílový server, například pomocí automatických pipeline nástroje GitLab, které jsou taktéž součástí projektu.

Práce splnila svůj hlavní cíl, a to posloužit jako dobrý základ pro budoucí rozvoj v rámci školního vyučování. Architektura aplikace umožňuje také snadné rozšíření o definice dalších algoritmů. Toho bylo docíleno cílenou separací prezentační, datové a simulační vrstvy, kdy předávané vstupy jsou unifikovány a nezáleží tak na simulovaném algoritmu. Pro implementaci dalšího algoritmu je tak nutné dodržet pouze předepsané rozhraní.

Obrovským přínosem může být vytvořený kreslící nástroj, který může být dále dekomponován

a používán jako samostatná komponenta pro kreslení logických obvodů. Tato dekompozice nebyla provedena přímo v této práci a je zde otevřen prostor pro další rozvoj.

Celkově lze říci, že zadání práce bylo dostatečně splněno, jelikož se povedlo sestavit aplikaci, která poskytuje možnosti jak pro jednotlivce, tak pro skupinovou výuku, umožňuje přenos a uložení rozpracovaného obvodu. Prototyp aplikace se podařilo spustit v Kubernetes prostředí tak, že je přístupný veřejnosti a funguje bez zjevných obtíží. Aplikaci je možné nasadit a provozovat v cloudovém prostředí bez zásadnějších úprav (nutná je pouze počáteční konfigurace), pokud by však byla snaha o zprovoznění na samostatném serveru, neexistuje technologické omezení, které by tomu bránilo.

Budoucí rozvoj projektu by se mohl týkat hlavně rozšíření o další *ATPG* algoritmy, jako je například již zmiňovaný *FAN*, *PODEM* nebo třeba *SOCRATES*.

Příloha A

Instalační příručka

Tato instalační příručka slouží jako návod ke zprovoznění různých částí aplikace. Cílem je poskytnout přehled jednotlivých komponent a jak je používat pro počáteční instalaci.

A.1 Požadované prostředí

Pro správný běh aplikace je nutné poskytnout Kubernetes cluster, který umožní využití alespoň 2 GB místa pro persistentní uložiště, které je potřebné pro databázi aplikace. Dále je nutné mít zprovozněný *ingress* a *egress* controller, který umožní příchozí komunikaci na zvolenou doménu, ve které poběží aplikace, a odchozí komunikaci na doménu autorizačního serveru. V případě, že nebude poskytnut Kubernetes cluster, ale instalace bude probíhat na samostatný server, prosím pokračujte na sekci A.3 této přílohy.

Kubernetes cluster musí umožňovat prostup do veřejné sítě[1] a poskytovat správné DNS záznamy pro komunikaci s internetem.

Obecně je pak nutné mít pro zvolenou doménu vygenerován příslušný certifikát a k němu privátní klíč.

A.2 Zprovoznění automatické build & deploy pipeline

K dispozici je připravená funkční pipeline pro všechny tři aplikace a platformní kód. Tato pipeline vyžaduje nástroj GitLab s docker runnerem, který je schopný spustit docker image *docker:24.0.7* a vyšší. Tento image je potřebný pro build aplikace.

Pro zprovoznění této pipeline je nutné vytvořit v nástroji GitLab následující strukturu:

`atpg-vision` ..... Skupina, ve které jsou umístěny jednotlivé projekty. Název se může lišit.
- `platform` ..... Projekt platformního repozitáře.
- `backend` ..... Projekt backendové aplikace.

[1] To je potřeba zejména pro komunikaci s autorizačním serverem

```
└── frontend ..... Frontendová aplikace.
└── simulation ..... Simulační aplikace.
```

Tato struktura následně usnadní konfiguraci jednotlivých proměnných prostředí. Proměnné prostředí konfigurované na úrovni skupiny (v případě, že stage a produkční prostředí se nacházejí na stejném Kubernetes clusteru) jsou pak uvedeny v tabulce A.1.

Proměnná	Popis
KUBE_CERTIFICATE_AUTHORITY_DATA	certifikát kubernetes clusteru – je nutný pro správné přihlášení servistního účtu do prostředí a provádění změn (deploy)
KUBE_CLUSTER	název clusteru pro Kubernetes
KUBE_SERVER	IP adresa kubernetes control plain serveru
KUBE_USER_ACCOUNT	název servisního účtu v Kubernetes
KUBE_USER_TOKEN	access token servisního účtu v Kubernetes
OAUTH20_GRANT	Název OAuth2.0 grantu – pro tuto aplikaci se jedná o hodnotu *urn:ctu:oauth:umapi.read*
OAUTH20_URL_ACCESS_TOKEN	OAuth2.0 token URL
OAUTH20_URL_AUTHORIZE	OAuth2.0 URL autorizace
OAUTH20_URL_RESOURCE_OWNER_DETAILS	OAuth2.0 URL pro získání informací o vlastníkovi aktuálního tokenu
PROJECT_NAMESPACE	základní název prostoru, ve kterém bude projekt umístěn – za tento název se ještě přidává identifikátor prostředí, tedy *stage*, *production*, atd...

Tabulka A.1 Popis jednotlivých společných proměnných pro konfiguraci build a deploy pipeline

Pro platformní aplikaci je pak vhodné vyplnit následující proměnné:

Proměnná	Popis
DOMAIN	doména, na které výsledná aplikace poběží – toto je nutné pro správnou konfiguraci ingress zdroje
TLS_CRT	TLS certifikát pro zabezpečení komunikace na doméně.
TLS_KEY	privátní klíč ke zvolenému TLS certifikátu

Tabulka A.2 Popis jednotlivých proměnných pro konfiguraci platform repozitáře

Frontendová aplikace obsahuje následující proměnné:

Proměnná	Popis
VITE_AUTHORIZE_URL	URL adresa backendového endpointu pro přihlášení – hodnotou je *..doména../api/V1/auth/authorize*
VITE_GRAPHQL_HOST_URL	URL adresa GraphQl rozhraní – hodnotou je *..doména../graphql*
VITE_LOGOUT_URL	URL adresa pro odhlášení uživatele – hodnotou je *..doména../api/V1/auth/logout*
VITE_WSS_HOST_URL	URL adresa pro websocket simulace – hodnotou je *..doména../ws/*

Tabulka A.3 Popis jednotlivých proměnných pro konfiguraci frontend repozitáře

Backendová aplikace vyžaduje následující konfiguraci:

Proměnná	Popis
DOMAIN	zvolená cílová doména, na které poběží aplikace
OAUTH20_CLIENT_ID	client_id pro přístup k autorizačnímu serveru
OAUTH20_CLIENT_SECRET	client_secret pro přístup k autorizačnímu serveru

Tabulka A.4 Popis jednotlivých proměnných pro konfiguraci backend repozitáře

Proměnné je možné konfigurovat jak na skupině, tak na jednotlivých projektech. Konfigurace na projektu umožňuje nastavit hodnoty proměnných per prostředí. Ty jsou v základu připraveny dvě: *staging* a *production*. Po konfiguraci a spuštění pipeline na všech prostředích bude provedena instalace projektů do zvoleného Kubernetes clusteru.

Po nasazení bude aplikace fungovat v omezeném režimu a bude přístupná pouze pro nepřihlášené uživatele. Pro plné zprovoznění je třeba na daném prostředí spustit příkaz `bin/atpg db:upgrade` v aplikačním podu backend aplikace, kontejneru `fpm`.

A.3 Instalace na čistý server

Pokud se rozhodnete pro instalaci na server bez využití Kubernetes clusteru a bez docker images, bude nutné zajistit, aby pro backendovou část bylo nainstalováno následující:

1. PHP ve verzi alespoň 8.2
2. Rozšíření PHP: pdo, pdo_pgsql, bcmath, soap, xsl, zip, sockets, exif
3. Webový server – nginx nebo Apache
4. Volitelně php-fpm jako fastcgi rozhraní

Pro simulační část aplikace je vyžadováno následující:

1. PHP ve verzi alespoň 8.2
2. Rozšíření PHP: openswoole

Frontendová část aplikace vyžaduje libovolný webový server, který dokáže zajistit směrování požadavků na jediný soubor, *index.html*, pokud neexistuje jiný statický soubor, který by odpovídal požadavku. V případě samostatné instalace je nutné zajistit směrování požadavků tak, jak je ilustrováno na obrázku 4.4.

Příloha B

Uživatelská příručka

Pro ovládání aplikace uživatelem slouží tato uživatelská příručka. Příručka provede uživatele základními možnostmi aplikace a vysvětlí, jak se chovají některé ovládací prvky.

B.1 Uživatelská administrace

Přístup do uživatelské administrace je zajištěn skrze tlačítko "Skupiny"v hlavním menu, které se zpřístupní po přihlášení do aplikace. Tlačítko "Přihlásit se"je umístěno v pravém horním rohu aplikace a pro přihlášeného uživatele je místo něj zobrazeno tlačítko "Odhlásit se".

Po kliknutí na tlačítko "Skupiny"se uživateli nabídne seznam skupin, ve kterých je členem a může je tak prohlížet. Rovněž je při pravém horním rohu umístěna možnost tento seznam filtrovat a vytvořit novou skupinu. Pod výpisem je pak umístěno stránkování, které umožní procházení delších seznamů. Po možnosti "Vytvořit skupinu"má uživatel možnost si tuto skupinu pojmenovat a po jejím založení se stává jejím vlastníkem.

Při výběru skupiny ze seznamu je opět nabídnut seznam, tentokrát však projektů. Každý projekt je označen názvem a volitelně štítkem, který je možné opět filtrovat pomocí nabídky v pravém horním rohu. Pokud jsem v roli vlastníka skupiny, pak mám možnost navíc editovat jednotlivé projekty kliknutím na tlačítko s ikonkou tužky u příslušného řádku. Pokud mám roli nižší, mohu obvod pouze simulovat a to skrze tlačítko s ikonkou blesku.

Editace projektu poskytuje totožnou kreslící plochu jako bude popsána níže. Navíc jsou zde možnosti pro uložení projektu, spuštění simulace a editace názvu a popisu projektu.

B.2 Kreslící plocha

Při editaci projektu přihlášeným uživatelem, stejně jako pro nepřihlášeného uživatele pod tlačítkem "Návrh"v menu aplikace, je dostupná kreslící plocha. Ta obsahuje prostor pro tvorbu obvodu situovaný do střední části obrazovky a dva postranní panely. Levý panel slouží pro vkládání prvků

do obvodu a obsahuje podporovaná hradla a vstupní a výstupní porty. Pravý panel je určen pro ovládací prvky specifické pro uživatele a obsahuje minimálně:

1. tlačítko ”Simulovat”
2. tlačítko ”Exportovat schéma”
3. tlačítko ”Importovat schéma”

Exportovat i importovat je možné ve formátech *bench* a *atpgv*. Pro přihlášeného uživatele, pokud přistoupil ke kreslící ploše skrze uložený projekt, je pak k dispozici nabídka se změnou jména projektu, popisku a štítku projektu a možnost tyto změny uložit. Veškeré změny, které uživatel provede na kreslící ploše jsou průběžně ukládány do paměti prohlížeče, takže kdyby nastal nějaký problém, například by uživatel omylem zavřel okno, je možné schéma obnovit opětovným přístupem na stránku.

Samotná kreslící plocha se ovládá převážně myší, přičemž funkce jednotlivých tlačítek jsou následovné:

1. držet pravé tlačítko myši mimo prvek: přesun plátna
2. držet levé tlačítko myši: výběr prvků
3. stisk levého tlačítka myši nad prvkem: výběr daného prvku
4. držet a táhnout tlačítko myši nad vybraným prvkem: posun prvku po plátně

Při používání kreslící plochy fungují standardní klávesové zkratky *ctrl+c* a *ctrl+v* pro práci se schránkou. Dále je možné použít klávesu *delete* pro smazání vybraného prvku.

B.2.1 Editace prvku obvodu

Po vybrání jednotlivého prvku[1] obvodu se zpřístupní nabídka editace vlastností tohoto prvku. Pro hradla je to výběr funkce, kterou bude plnit a počet portů, které budou na vstupu. Pro vodiče, vstupy a výstupy se pak jedná o označení, které ponesou.

Platí, že po změně funkce hradla a počtu jeho portů budou odpojeny všechny stávající vodiče a místo původního hradla bude vloženo hradlo nové. S tímto je třeba počítat, jelikož nelze zachovat stávající propojení.

Při změně označení vodiče je nutné tuto volbu potvrdit kliknutím mimo příslušný formulářový prvek nebo nejlépe klávesou *enter*. Platí, že dva vodiče nemohou nést stejný název a pokud se uživatel pokusí o pojmenování vodiče stejně, jako je již pojmenován jiný, zobrazí se chybová hláška a změna se neprovede.

[1] Prvkem se rozumí vstup, výstup, hradlo nebo vodič

B.2.2 Kreslení vodičů

Kreslení vodičů má svá pravidla, která jsou vysvětlena v této sekci. Vodiče je možné kreslit pouze z vstupního nebo výstupního portu nějakého hradla, vstupu nebo výstupu. Nelze mít neukončený vodič visící do prázdna – vždy musí být spojen s nějakým vstupním/výstupním portem. Není možné na stejný vstup/výstup umístit více vodičů. Pokud se v obvodu vyskytuje větvení, pak je nutné jen nakreslit následovně:

- propojit vstupní port hradla s výstupním portem hradla jiného,
- propojit výstupní port dalšího hradla s dříve nakresleným vodičem.

Při kreslení vodičů je možné přidávat jednotlivé průchozí body a to následujícím postupem:

- klikněte na port, ze kterého chcete vodič vést,
- klikněte na místo, který má vodič procházet (lze i opakované pro přidání více bodů),
- klikněte na port, do kterého má vodič vést.

Při provedení tohoto postupu bude mít vytvořený vodič odpovídající tvar. Vodiče je možné po nakreslení dále upravovat pomocí tahu myši při stisknutém levém tlačítku nad kontrolním bodem vodiče. Tyto body jsou zobrazeny jako zelené úchyty na vodiči.

Pro vodiče platí jistá omezení, které zahrnují následující:

1. nelze propojit porty stejného typu (vstup-vstup nebo výstup-výstup),
2. nelze vytvořit takový vodič, který by způsobil cyklus v obvodu,
3. nelze propojit již obsazený port.

B.3 Simulace obvodu

Pro spuštění obvodu je nutné, aby byly splněny následující podmínky, jinak server odmítne simulaci spustit:

- obvod je v uzavřen, tedy všechny porty jsou napojeny na vodiče,
- každý vodič má svůj název,
- názvy vodičů jsou jednoznačné.

Každý uživatel, který do simulace vstoupí nabývá následujících rolí:

1. návštěvník — může pouze prohlížet, jak vypadá průchod algoritmem a není nijak oprávněn do simulace zasahovat

2. operátor — může zasahovat do simulace výběrem poruchy a krokovat ji, ale nemá možnost ji restartovat ani vybrat algoritmus
3. vlastník — navíc může restartovat simulaci, vybrat algoritmus a měnit oprávnění ostatním uživatelům

Jakmile je simulace spuštěna, zobrazí se uživateli stejný obvod jako při návrhu, nicméně nelze v něm provádět žádné změny. V levém panelu nástrojů se nyní nachází ovládací prvky simulace a to:

1. výběr poruchy — pokud to algoritmus dovolí a uživatel má oprávnění *operátor* a vyšší, může vybrat poruchu pro obvod
2. výběr algoritmu — pokud není spuštěna simulace konkrétním algoritmem a uživatel má oprávnění *vlastník* a vyšší, může vybrat algoritmus a začít simulaci obvodu tímto algoritmem,
3. ovládání probíhající simulace — pokud je vybrán algoritmus, je možné jej krokovat vpřed i vzad. Toto je možné pro uživatele s oprávněním *operátor* a vyšší.

Pro každého uživatele je pak zobrazen seznam přihlášeným uživatelů a jejich oprávnění. Pro vlastníka simulace je navíc zobrazen také unikátní odkaz pro připojení dalších účastníků. Tento odkaz obsahuje 32 bytů dlouhý náhodný řetězec, takže je zajištěna unikátnost přístupu.

Pod plátnem, na kterém je zobrazen simulovaný obvod a v pravé liště nástrojů se vykreslují vnitřní stavy simulace. Tyto stavy se doplňují průběžně a je tak možné sledovat, jak algoritmus hledá cestu obvodem. Pokud stav obsahuje také vizualizaci, je tato vizualizace provedena změnou barvy příslušného prvku přímo v reprezentaci simulovaného obvodu na plátně.

Ve spodní části obrazovky se pak nachází seznam událostí. Každý událost nese svůj název a na řádku s událostí je možné vybrat náhled na stav algoritmu během dané události. Tento náhled se aktivuje kliknutím na ikonku oka u příslušného stavu. Seznam událostí je také možné rozbalit pomocí šipky umístěné v pravém dolním rohu. Během prohlížení události není možné do simulace nijak vstupovat a uživatel sice bude dostávat aktualizace o nových stavech, ale ty se nebudou aktivně zobrazovat. Mód prohlížení stavu je tak vhodnou pomůckou při zpětné kontrole průchodu a pochopení, jak algoritmus funguje.

B.3.1 Připojení k simulaci

Uživatel má možnost připojit se k simulaci pomocí jedinečného odkazu, který mu poskytne jiný uživatel. Před vstupem do simulační místnosti je vyzván, pokud není již přihlášen, aby zadal své jméno. Teprve po zadání jména mu bude umožněn vstup do simulační části aplikace.

Alternativou k unikátnímu odkazu, ačkoli se jedná o preferovanou volbu, je použití tlačítka ”Připojit k simulaci”v horním menu. Toto tlačítko zobrazí formulář, kde je možné zadat ID simulační místnosti a uživatelovo jméno. Po vyplnění těchto údajů a kontrolou správnosti na straně serveru je uživatel připojen k simulační místnosti.

Příloha C

Manuál vývojového prostředí

Vývojové prostředí je připraveno v adresáři *platform*, společně s jednotlivými aplikacemi. Pro zprovoznění vývojového prostředí je nutné mít nainstalován nástroj Docker, `docker-compose`, `make` a podporované prostředí. Mezi podporovaná prostředí spadá *WSL2*[1] a samotný Linux. Dále je nutné mít uvolněný TCP port 84, na který se aplikace bude bindovat.

Jakmile uživatel splňuje požadavky na lokální prostředí, je možné jej spustit pomocí příkazu `make init`. Tento příkaz spustí build docker images, takže budou připraveny na spuštění. Jakmile tento příkaz doběhne, je možné pokračovat dále pomocí `make up`, přičemž po tomto kroku by se měly spustit kontejnery, ve kterých běží aplikace. Pro úplné dokončení instalace je nutné dodržet také následující kroky:

- vstoupit do kontejneru *backend* a vykonat příkaz *composer install*
- vstoupit do kontejneru *backend* a vykonat příkaz *bin/atpg db:upgrade*
- vstoupit do kontejneru *simulation* a vykonat příkaz *composer install*
- vytvořit soubor *apps/backend/.env* a vyplnit hodnoty potřebné pro OAuth autorizaci a spojení na databázi, vše podle šablony *apps/backend/.env.template* nebo analogicky podle instalační příručky
- vytvořit soubor *apps/frontend/.env* a vyplnit hodnoty URL služeb, vše podle šablony *apps/backend/.env.template*, případně pro vývojové prostředí je možné využít *apps/backend/.env.example*

Pokud uživatel dodrží tento postup, mělo by se zpřístupnit vývojové prostředí, které mimo základní služby obsahuje také nástroj **Adminer**, dostupný na adrese *https://localhost:84/adminer/*. Z nástrojů pro ovládání projektu pak obsahuje příkazy:

- make init – spustí sestavování images pro jednotlivé aplikace
- make up – spustí projekt pomocí docker-composer
- make down – spustí docker-compose down

[1] Windows Subsystem for Linux

- make cleanup – spustí docker-compose down a promaže všechny použité prostředky
- make test – spustí Unit testy v simulačním kontejneru

Bibliografie

1. LEE, HK; HA, Dong S. *On the generation of test patterns for combinational circuits*. 1993. Tech. zpr. Technical Report.
2. *FAN_ATPG* [online]. [B.r.]. [cit. 2024-05-08]. Dostupné z: `https://github.com/NTU-LaDS-II/FAN_ATPG`.
3. NOVÁK, Ondřej. *Handbook of testing electronic systems*. Prague: Czech Technical University, 2005. ISBN 80-01-03318-X.
4. ROTH, J. Paul. Diagnosis of Automata Failures: A Calculus and a Method. *IBM Journal of Research and Development*. 1966, roč. 10, č. 4, s. 278–291. Dostupné z DOI: `10.1147/rd.104.0278`.
5. HLAVIČKA, Jan. *Diagnostika a spolehlivost číslicových systémů*. Prague: Czech Technical University, 1978.
6. QUINE, W. V. The Problem of Simplifying Truth Functions. *The American Mathematical Monthly*. 1952, roč. 59, č. 8, s. 521–531. Dostupné z DOI: `10.1080/00029890.1952.11988183`.
7. RATHORE, T. S. Minimal Realizations of Logic Functions Using Truth Table Method with Distributed Simplification. *IETE Journal of Education*. 2014, roč. 55, č. 1, s. 26–32. Dostupné z DOI: `10.1080/09747338.2014.921412`.
8. GOEL, P. An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits. *IEEE Transactions on Computers*. 1981, roč. C-30, č. 3, s. 215–222. Dostupné z DOI: `10.1109/TC.1981.1675757`.
9. FUJIWARA; SHIMONO. On the Acceleration of Test Generation Algorithms. *IEEE Transactions on Computers*. 1983, roč. C-32, č. 12, s. 1137–1144. Dostupné z DOI: `10.1109/TC.1983.1676174`.
10. SCHULZ, M.H.; TRISCHLER, E.; SARFERT, T.M. SOCRATES: a highly efficient automatic test pattern generation system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 1988, roč. 7, č. 1, s. 126–137. Dostupné z DOI: `10.1109/43.3140`.
11. PARKES, S.; BANERJEE, P.; PATEL, J. A parallel algorithm for fault simulation based on PROOFS. In: *Proceedings of ICCD '95 International Conference on Computer Design. VLSI in Computers and Processors*. 1995, s. 616–621. Dostupné z DOI: `10.1109/ICCD.1995.528932`.
12. SESHU, Sundaram. On an Improved Diagnosis Program. *IEEE Transactions on Electronic Computers*. 1965, roč. EC-14, č. 1, s. 76–79. Dostupné z DOI: `10.1109/PGEC.1965.264063`.

13. WAICUKAUSKI, John A; LINDBLOOM, Eric; IYENGAR, Vijay S; ROSEN, Barry K. Transition Fault Simulation by Parallel Pattern Single Fault Propagation. In: *ITC*. 1986, s. 542–551.

14. ARMSTRONG, D.B. A Deductive Method for Simulating Faults in Logic Circuits. *IEEE Transactions on Computers*. 1972, roč. C-21, č. 5, s. 464–471. Dostupné z DOI: 10.1109/T-C.1972.223542.

15. ULRICH, E. G.; BAKER, T. Concurrent simulation of nearly identical digital networks. *Computer*. 1974, roč. 7, č. 4, s. 39–44. Dostupné z DOI: 10.1109/MC.1974.6323496.

16. *BENCH Format Manual* [online]. [B.r.]. [cit. 2024-05-01]. Dostupné z: https://sportlab.usc.edu/~msabrishami/benchmark-project/bench.html.

17. DOCKER INC. *Docker overview* [online]. [B.r.]. [cit. 2024-05-01]. Dostupné z: https://docs.docker.com/get-started/overview/.

18. *Kubernetes Components* [online]. [B.r.]. [cit. 2024-05-01]. Dostupné z: https://kubernetes.io/docs/concepts/overview/components/.

19. *Containerized Applications Overview* [online]. [B.r.]. [cit. 2024-05-01]. Dostupné z: https://www.datadoghq.com/knowledge-center/containerized-applications/.

20. DOCKER INC. *Overview of Docker Desktop* [online]. [B.r.]. [cit. 2024-05-01]. Dostupné z: https://docs.docker.com/desktop/.

21. AWS. *Running Applications in Containers* [online]. [B.r.]. [cit. 2024-05-01]. Dostupné z: https://docs.aws.amazon.com/whitepapers/latest/develop-deploy-dotnet-apps-on-aws/running-applications-in-containers.html.

22. MICROSOFT AZURE. *Container Services* [online]. [B.r.]. [cit. 2024-05-01]. Dostupné z: https://azure.microsoft.com/en-us/products/category/containers.

23. *Kubernetes and cloud native operations report 2021* [online]. 2021. [cit. 2024-05-01]. Dostupné z: https://juju.is/cloud-native-kubernetes-usage-report-2021.

24. *Managing Workloads* [online]. [B.r.]. [cit. 2024-05-01]. Dostupné z: https://kubernetes.io/docs/concepts/workloads/management/.

25. *The cloud native control plane framework* [online]. [B.r.]. [cit. 2024-05-01]. Dostupné z: https://www.crossplane.io/.

26. *The package manager for Kubernetes* [online]. [B.r.]. [cit. 2024-05-01]. Dostupné z: https://helm.sh/docs/.

27. *PostgreSQL: The World's Most Advanced Open Source Relational Database* [online]. [B.r.]. [cit. 2024-05-01]. Dostupné z: https://www.postgresql.org/.

28. *GraphQL vs. REST API: What's the difference?* [Online]. [B.r.]. [cit. 2024-05-01]. Dostupné z: https://www.ibm.com/blog/graphql-vs-rest-api/.

29. I. FETTE, A. Melnikov. *The WebSocket Protocol* [Internet Requests for Comments]. RFC Editor, 2011-12. RFC, 6455. RFC Editor. ISSN 2070-1721. Dostupné také z: https://www.rfc-editor.org/rfc/rfc6455.

30. *Powering the next-generation microservices and application* [online]. [B.r.]. [cit. 2024-05-01]. Dostupné z: https://openswoole.com/.

31. *What is SAML 2.0?* [Online]. [B.r.]. [cit. 2024-05-01]. Dostupné z: https://auth0.com/intro-to-iam/what-is-saml.

32. *What is OAuth 2.0?* [Online]. [B.r.]. [cit. 2024-05-01]. Dostupné z: https://auth0.com/intro-to-iam/what-is-oauth-2.

33. *OAuth 2.0* [online]. [B.r.]. [cit. 2024-05-01]. Dostupné z: https://help.fit.cvut.cz/dev/oauth2.html.

34. *NGINX Ingress Controller* [online]. [B.r.]. [cit. 2024-05-01]. Dostupné z: https://docs.nginx.com/nginx-ingress-controller/.
35. *Adminer - Správa databáze v jednom PHP souboru* [online]. [B.r.]. [cit. 2024-05-01]. Dostupné z: https://www.adminer.org/cs/.
36. *SimpleSAMLphp as an Identity Provider* [online]. [B.r.]. [cit. 2024-05-01]. Dostupné z: https://simplesamlphp.org/samlidp/.
37. *GraphQl Voyager* [online]. [B.r.]. [cit. 2024-05-01]. Dostupné z: https://graphql-kit.com/graphql-voyager/.
38. *draw.io* [online]. [B.r.]. [cit. 2024-05-01]. Dostupné z: https://www.drawio.com/.
39. *Using HTTP cookies* [online]. [B.r.]. [cit. 2024-05-01]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies.
40. *SINGLE SIGN ON (JEDNOTNÉ PŘIHLAŠOVÁNÍ) ČVUT* [online]. [B.r.]. [cit. 2024-05-08]. Dostupné z: https://ist.cvut.cz/nase-sluzby/single-sign-on/.
41. *SimpleSAMLphp as a Service Provider* [online]. [B.r.]. [cit. 2024-05-08]. Dostupné z: https://simplesamlphp.org/samlsp/.
42. *Start building apps on CTU APIs!* [Online]. [B.r.]. [cit. 2024-05-08]. Dostupné z: https://auth.fit.cvut.cz/manager/index.jsf.
43. *PHP-DI: The dependency injection container for humans* [online]. [B.r.]. [cit. 2024-05-08]. Dostupné z: https://php-di.org/.
44. *Composer: A Dependency Manager for PHP* [online]. [B.r.]. [cit. 2024-05-08]. Dostupné z: https://getcomposer.org/.
45. *graphql-php* [online]. [B.r.]. [cit. 2024-05-08]. Dostupné z: https://github.com/webonyx/graphql-php.
46. *MySQL Workbench* [online]. [B.r.]. [cit. 2024-05-08]. Dostupné z: https://www.mysql.com/products/workbench/.
47. *Repository design pattern* [online]. [B.r.]. [cit. 2024-05-08]. Dostupné z: https://designpatternsphp.readthedocs.io/en/latest/More/Repository/README.html.
48. *The Progressive JavaScript Framework* [online]. [B.r.]. [cit. 2024-05-08]. Dostupné z: https://vuejs.org/.
49. *Vue Apollo GraphQL* [online]. [B.r.]. [cit. 2024-05-08]. Dostupné z: https://v4.apollo.vuejs.org/.
50. *maxGraph* [online]. [B.r.]. [cit. 2024-05-08]. Dostupné z: https://github.com/maxGraph/maxGraph.
51. *Build fast, responsive sites with Bootstrap* [online]. [B.r.]. [cit. 2024-05-08]. Dostupné z: https://getbootstrap.com/.
52. *Vite: Next Generation Frontend Tooling* [online]. [B.r.]. [cit. 2024-05-08]. Dostupné z: https://vitejs.dev/.
53. *Pool design pattern* [online]. [B.r.]. [cit. 2024-05-08]. Dostupné z: https://designpatternsphp.readthedocs.io/en/latest/Creational/Pool/README.html.
54. *Postman API platform* [online]. [B.r.]. [cit. 2024-05-08]. Dostupné z: https://www.postman.com/home.
55. *PHPUnit: The PHP Testing Framework* [online]. [B.r.]. [cit. 2024-05-08]. Dostupné z: https://phpunit.de/index.html.

Obsah příloh

- `readme.txt` — stručný popis obsahu média
- `platform` — Adresář s kódem platformy aplikace
  - `apps`
    - `backend` — Zdrojové kódy pro backendovou aplikaci
    - `frontend` — Zdrojové kódy pro frontendovou aplikaci
    - `simulation` — Zdrojové kódy pro simulační aplikaci
- `text` — Textová část práce
  - `src` — zdrojová forma práce ve formátu LaTeX
  - `thesis.pdf` — text práce ve formátu PDF
- `video` — Záznam z uživatelského testování
- `sampledata` — Základní datová sada