FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Webová aplikace pro spolehlivostní analýzu
Student:	Bc. Daniel Vrátil
Vedoucí:	Ing. Martin Daňhel, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Webové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

1. Prostudujte a implementujte spolehlivostní analýzu pomocí metody stromů poruch.
2. Začleňte tuto metodu do existující webové aplikace pro spolehlivostní modely, kterou jste vyvinul v rámci bakalářské práce v roce 2022.
3. Rozšiřte webovou aplikaci o následující funkcionality:
   a. Implementujte historii změn umožňující krokování vzad a vpřed.
   b. Integrujte aktualizace z reálného dvouletého nasazení v předmětu NI-TSP, zahrnující administraci aplikace (uživatele, projekty, modely).
   c. Propojte aplikaci se školním autentizačním systémem.
   d. Přidejte spolehlivostní model: Stromy poruch.
   e. Upravte zadávání vstupu tak, aby bylo možné vkládat spolehlivostní parametry v různých formátech (MTBF, FIT, intenzita poruch, pomocí výrazu).
   f. Umožněte tvorbu vnořených modelů pro Stromy poruch.
4. Aktualizujte výpočetní šablonu ve Wolfram Mathematice na základě výše uvedených změn.
5. Otestujte navržené řešení na reálných příkladech železničního zabezpečovacího zařízení v porovnání se stávajícími aplikacemi (SHARPE, ZUSIM) specializujícími se na spolehlivostní analýzu.

Elektronicky schválil/a Ing. Jaroslav Kuchař, Ph.D. dne 13. prosince 2023 v Praze.

Diplomová práce

WEBOVÁ APLIKACE PRO SPOLEHLIVOSTNÍ ANALÝZU

Bc. Daniel Vrátil

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

Obsah

Seznam obrázků

Seznam tabulek

Rád bych poděkoval v první řadě vedoucímu práce Ing. Martinovi Daňhelovi Ph.D. za skvělé vedení práce a odbornou pomoc při vypracování práce této i prací souvisejících. Také chci poděkovat své rodině za jejich neustálou podporu nejen při vypracování této práce, ale i po celou dobu studia.

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. Dále prohlašuji, že jsem s Českým vysokým učením technickým v Praze uzavřel licenční smlouvu o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona. Tato skutečnost nemá vliv na ust. § 47b zákona č. 111/1998 Sb., o vysokých školách, ve znění pozdějších předpisů.

V Praze dne 9. května 2024

Abstrakt

Tato diplomová práce se zabývá návrhem a implementací webové aplikace, která umožňuje vytvářet spolehlivostní analýzy metodou stromu poruch. Hlavním cílem práce je vytvořit dostupný nástroj pro tvorbu komplexní spolehlivostní analýzy, který však veškeré výpočty provádí externě v matematickém systému Wolfram Mathematica. Jedná se rozšíření obhájené bakalářské práce, která umožňovala základní práci s Markovskými modely a blokovými diagramy. Zaměřuje se především na cílovou skupinu malých týmů a studijních projektů. Adresuje problémy nalezené při testování a přináší jeden z nejpoužívanějších modelů: strom poruch. Dále přidává několik nových funkcí zaměřených na zjednodušení a zrychlení analýzy. V závěru je aplikace vyzkoušena a porovnána s dosavadními nástroji na zadání železničního přejezdu vycházející z reálného prostředí.

Klíčová slova Nástroj pro Spolupráci, PHP, React, Spolehlivostní Analýza, Strom Poruch, Webová Aplikace, Wolfram Mathematica

Abstract

This thesis deals with the design and implementation of a web application that allows to create reliability analyses using the fault tree method. The main goal of the thesis is to create an accessible tool for creating a complex reliability analysis, but which performs all calculations externally in Wolfram Mathematica. This is an extension of the defended bachelor thesis, which allowed basic work with Markov models and reliability block diagrams. It is mainly aimed at the target group of small teams and study projects. It addresses problems found in testing and introduces one of the most used models: the fault tree. It also adds several new features aimed at simplifying and speeding up analysis. Finally, the application is tested and compared with existing tools for specifying a level crossing based on a real environment.

Keywords Collaboration Tool, PHP, React, Reliability Analysis, Fault Tree, Web Application, Wolfram Mathematica

Seznam zkratek

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
BDD	Binary Decision Diagram
CRUD	Create, Read, Update, Delete
DNF	Disjunktní Normální Forma
FIT	Failures In Time
FTA	Fault Tree Analysis
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force
JSON	JavaScript Object Notation
JWT	JSON Web Token
MCS	Minimal Cut Sets
MTBF	Mean Time Between Failures
MTTF	Mean Time To Failure
MPS	Minimal Path Sets
ORM	Object Relational Mapping
PHP	PHP Hypertext Processor
PNG	Portable Network Graphics
RAMS	Reliability, Availability, Maintanance, Safety
REST	Representational State Transfer
SSO	Single Sign-on
STI	Single Table Inheritance
SVG	Scalable Vector Graphics
URI	Uniform Resource Identifier
UUID	Universally Unique Identifier
WM	Wolfram Mathematica
WSL	Wolfram Script Language

Kapitola 1
Úvod

V současné době jsou v průmyslové praxi jedny z nejpoužívanějších spolehlivostních modelů stromy poruch (FTA - Fault Tree Analysis). Jedná se o spolehlivostní metodu jejíž vznik je datován do 60. let minulého století, kdy se hojně využívala k identifikaci rizik při návrhu systému kontroly startu raket. Mimo to se tato metoda stále vyvíjí a postupně se rozšířila do oblasti jako je železnice, letectví, sledování bezpečnosti v elektrárnách a dokonce i do vesmírného programu, tedy do oborů, ve kterých je klíčovým faktorem bezpečnost (safety). Analýza metodou stromů poruch se využívá k identifikaci možných rizik, která by mohla nastat při provozu systémů kritických na bezpečnost.

V komerčním prostředí se FTA provádí v komplexních a často drahých nebo interních aplikacích. K těm se jednotlivec nebo malý tým zabývající se spolehlivostní analýzou jen těžko dostane. Při hledání dostupné alternativy zjistíte, že výběr vyhovujících aplikací je velice omezen. Nástroje, které lze získat pro potřeby výuky nebo vývoje malého týmu, jsou zastaralé a na nových systémech nemusí být plně funkční. Společně s chybějící podporou práce ve skupině více lidí nezbývá moc jiných možností, než provádět analýzu ručně nebo pouze za pomoci grafického editoru. Proto v této práci vyvíjím novou aplikaci pro spolehlivostní analýzu. Ta přináší řešení problémů stávajících dostupných nástrojů.

1.1 Motivace

Nedostatek dostupných nástrojů pro studijní a menší projekty společně s rozsahem oboru spolehlivosti je pro mě jedna z největších motivací této práce. Aplikace, která by splnila požadavky cílové skupiny má potenciál být opravdu využívanou v reálném produkčním prostředí. Výsledek mé práce tak může pomoci studentům naučit se určité aspekty spolehlivostní analýzy nebo menším týmům při tvorbě svého projektu z oblasti spolehlivosti. Teoretická oblast zabývající se spolehlivostními modely je velice rozsáhlá. Existuje mnoho spolehlivostních modelů, některé z nich mají navíc rozšíření o specifické bloky. Takové modely se pak používají ve specifických případech spolehlivostní analýzy. Rozsah této práce může pokrýt jen jejich zlomek. Možné pokračování této práce by pak nabízelo mnoho příležitostí, jak aplikaci dále rozvíjet a zlepšovat.

1.2 Cíle práce

Tato práce má za cíl přinést nástroj pro zpracování spolehlivostní analýzy, který je připraven pro produkční použití. Umožní tak cílové skupině malých týmu nebo jednotlivců, především z řad studentů, vypracování analýzy. A to pomocí nového nástroje kompatibilního s dnešními

systémy za pomoci moderních technologií. Základ práce vychází z původní verze webové aplikace pro spolehlivostní modely vytvořené v rámci mé obhájené bakalářské práce na FIT ČVUT v Praze. Tato práce tedy bude rozšířením a vylepšením původního řešení o nové modely, funkce nebo grafické prvky. Jedním z hlavních bodů zadání je rozšíření sady modelů o stromy poruch, jeden z nejpoužívanějších spolehlivostních modelů v průmyslovém použití. S tím souvisí další rozšíření v podobě kvantitativní a kvalitativní analýzy nebo propojení více modelů do jednoho. Mezi další rozšíření napříč modely patří například historie změn, propojení s fakultním autorizačním systémem, export do obrázkových formátů a samozřejmě opravené problémy nalezené při testování aplikace.

Před začátkem vývoje, ale i v jeho průběhu, probíhalo testování původní verze webové aplikace na cvičeních předmětu *Testování a Spolehlivost* na FIT ČVUT v Praze. To odhalilo několik zásadních problémů s rychlostí a přívětivostí analýzy, které budou v rámci této diplomové práce opraveny.

1.3 Popis kapitol

První kapitola je věnována základům spolehlivosti. Vysvětluje význam spolehlivosti v praxi, popisuje její rozdělení na specifické části a představuje spolehlivostní analýzu. Hlavní část kapitoly je však věnována stromům poruch, jakožto jednomu z nejpoužívanějších spolehlivostních modelů, o který je webová aplikace rozšířena. Tato sekce seznamuje čtenáře s významem stromu poruch, jeho základními i rozšiřujícími bloky a popisuje základní výpočetní parametry. Kapitola slouží jako teoretické vysvětlení všech termínů a výpočtů, které jsou v aplikaci využívány ke spolehlivostní analýze.

Další kapitolou je analýza všech dalších technologií, nástrojů a postupů, které jsem před nebo při vývoji aplikace potřeboval. Úvod kapitoly tvoří analýza nástrojů, které stromy poruch nebo jiné plánované funkce obsahují. Pro každý nástroj jsem našel několik výhod a nevýhod, na které jsem při použití nebo konzultacích narazil. Účelem bylo přenést některé z výhod do výsledné aplikace a naopak se vyvarovat jejich chybám. Jedním z těchto aplikací byla i původní verze webové aplikace, ze které v této práci vycházím. Všechny poznatky jsem využil pro definování požadavků aplikace. Kapitola obsahuje výčet funkčních, nefunkčních požadavků i několika konkrétních případů užití, které má aplikace za cíl splnit. Kapitolu uzavírá analýza jednotlivých využitých technologií, která byla značně ovlivněna původním řešením, ale ani tak se nevyhnula změnám.

Následující kapitola postupně popisuje návrh a implementaci konkrétních funkcí, o které byla aplikace rozšířena nebo upravena. První popsaná funkce jsou již několikrát zmíněné stromy poruchy. Podrobně popisuji nejen základní bloky a práci s novým modelem, ale především takové specifické vlastnosti jako jsou přechodové a vnořené bloky nebo algoritmy kvalitativní analýzy. Následují změny, které se týkají všech modelů jako přepracování vstupu v podobě matematického výrazu. Dále historie provedených změn, administrace pro správce systému nebo úpravy stávajících funkcí.

Předposlední kapitola je věnována testování, nasazení. Popisuje míru testování a průběh nasazení nové verze. Obsahuje překážky, na které jsem narazil při nasazení na produkční prostředí a aktualizaci původní verze aplikace. Důležité bylo provést nasazení tak, aby zůstala zachována veškerá data uživatelů z předchozí verze a provoz na předmětu *Testování a Spolehlivost* mohl pokračovat.

V poslední kapitole jsem se soustředil hlavně na verifikaci aplikace za pomoci zadání z reálného prostředí. Účel verifikace je ukázat, že aplikace vrací pro konkrétní sadu modelů stejné výsledky jako aplikace zmíněné v analýze. Smysl této kapitoly byl provést ověření na příkladu, který se bude co nejvíce přibližovat reálnému průmyslovému použití. Proto jsem při tvorbě zadání i průběhu ověřování spolupracoval s vedoucím práce, který má v tomto oboru praktické zkušenosti.

Kapitola 2

Spolehlivostní analýza metodou stromu poruch

Hlavním rozšířením webové aplikace bude nový typ spolehlivostního modelu. Před technickou analýzou je důležité popsat téma spolehlivosti, spolehlivostní analýzy a modelů, které budou později v práci zpracovány.

2.1 Spolehlivost a spolehlivostní analýza

„Obecná vlastnost objektu spočívající ve schopnosti plnit požadované funkce při zachování hodnot stanovených provozních ukazatelů v daných mezích a v čase podle stanovených technických podmínek.“ Dle [1] je právě tato věta definicí spolehlivosti. I přes svou definici je spolehlivost poměrně obecný termín a není vždy jednoznačný. Právě z toho důvodu je spolehlivost komplexní vlastnost objektu nebo zařízení. Dále ji lze rozdělit na bezporuchovost, životnost, skladovatelnost a další. S prvními dvěma přímo pracuje mnoho typů spolehlivostní analýzy včetně později rozebírané FTA (Fault Tree Analysis). Definice popisuje technické podmínky, ty představují vnější vlivy, které mohou zařízení ovlivnit. Pokud tedy nejsou dodrženy specifikované technické podmínky a zařízení není schopno plnit své funkce, neříká to nic o jeho spolehlivosti. Zachováním hodnot stanovených provozních ukazatelů autor říká, že by spolehlivé zařízení mělo udržovat své stanovené ukazatele. V praxi se tak může jednat o rychlost, spotřebu nebo teplotu zařízení.

Situace, kdy je objekt (systém, zařízení, součástka, ...) schopen plnit požadované funkce se nazývá bezporuchový stav, v opačném případě se jedná o poruchový stav. Z počátku je typicky objekt v bezporuchovém stavu a v čase se po objevení poruchy může přesunout do poruchového stavu. Poté již v poruchovém stavu zůstane nebo nastane oprava (automatická či manuální), tím se rozlišují objekty na obnovované a neobnovované. Spolehlivostní analýza spočívá ve sledování objektu nebo skupiny objektů za účelem zkoumání spolehlivostních ukazatelů a její optimalizace vzhledem k nákladům nejen na tvorbu objektu, ale i provoz celého systému.

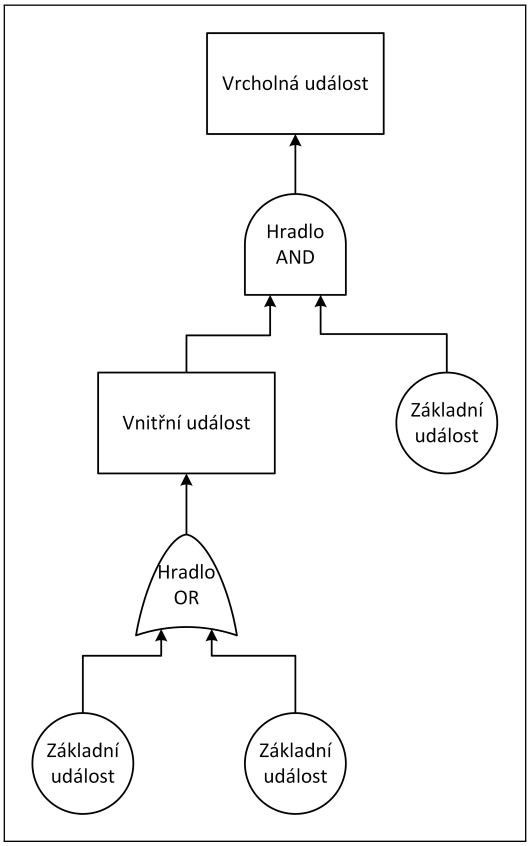
2.2 Úvod do spolehlivostní analýzy

V současné době jsou v průmyslové praxi jedny z nejpoužívanějších spolehlivostních modelů stromy poruch. Jedná se o spolehlivostní metodu jejíž vznik je datován do 60. let minulého století, kdy se hojně využívala k identifikaci rizik při návrhu zařízení v elektrárnách. [2] Mimo to se tato metoda stále vyvíjí a postupně se rozšířila do oblasti jako je železnice, letectví, dokonce i do vesmírného programu, tedy do všech oborů, ve kterých je klíčovým faktorem bezpečnost (sa-

fety). Analýza metodou takzvaných statických stromů poruch se využívá k identifikaci a analýze možných rizik, která by mohla nastat při provozu systémů kritických na bezpečnost. Výsledkem této analýzy jsou mimo identifikace rizik například tyto parametry: intenzita poruch (failure rate) nebo dostupnost (availability), které číselně popisují vlastnosti navrženého zařízení. Tato forma analýzy se nazývá kvantitativní. Alternativou je kvalitativní analýza, která zkoumá zařízení z více sémantického pohledu. Soustředí se například na identifikaci slabších míst systému, která by měla být pro bezpečnější výsledek posílena.

2.3 Stromy poruch

Strom poruch (příklad je znázorněn na obrázku 2.1) je logický diagram zobrazující závislosti mezi poruchami systému. Každý strom má za cíl popsat jednu hlavní událost, která může mít za fatální následek pro systém. Stromovou strukturou popisuje, jaké k ní vedou poruchy menšího rázu. Kompletní proces tvorby stromu, jeho optimalizace a zkoumání výsledků se nazývá analýza stromu poruch. Analýzu lze rozdělit na kvalitativní a kvantitativní, oběma těmto postupům je věnovaná část této kapitoly. Stromy poruch používají logická hradla, mezi základní hradla patří hlavně OR a AND. Ty ve stromu představují závislosti výstupních poruch na vstupní. OR má dva a více vstupních signálů a propaguje ho, pokud je alespoň jeden z nich poruchový. AND naopak propaguje poruchu jen, pokud jsou poruchové všechny vstupy. Pro správnost výpočtu se předpokládá, že jsou všechny vstupy hradla AND nezávislé. [3]



Obrázek 2.1 Jednoduchý strom poruch se základními prvky. Směrem ze shora se skládá z vrcholné události, která typicky označuje poruchu modelované části systému. Níže je hradlo AND, do kterého vstupují dvě různé události, základní a vnitřní. Ta se dále rozpadá na dvě menší základní událostí spojené hradlem OR.

FTA se používá převážně v oborech, ve kterých je klíčovým faktorem bezpečnost. Používá se k získání spolehlivostních parametrů (RAMS). Tyto parametry se v průběhu analýzy optimalizují, aby bylo dosaženo co nejlepšího výsledného zařízení nasazeného do provozu.

Bezporuchovost (Reliability) je pravděpodobnost správné funkce komponenty během daného časového období za daných provozních podmínek.

Dostupnost (Availability) systému je pravděpodobnost, že systém bude v daný okamžik správně fungovat.

Udržovatelnost (Maintainability) je ukazatel o tom, kdy se předpokládá jakýsi servis, celého zařízení, nebo jeho kritických částí.

Bezpečnost (Safety) je vlastnost systému, která zajistí, že nebude ohrožovat lidský život ani životní prostředí. [4] [5]

2.3.1 Kvantitativní analýza stromů poruch

Kvantitativní analýza je proces, při kterém se předvídá spolehlivost analyzovaného systému (zařízení) v určitém čase a za určitých podmínek. Konkrétně se odvozují číselné spolehlivostní parametry. Ty je možné určit na základě stavby sytému a spolehlivostních parametrů jeho částí. Cílem analýzy je odvodit parametry plánovaného běhu systému. Na základě získaných dat dále rozhodovat, zda analyzované zařízení bude odpovídat předem definovaným požadavkům na bezpečnost a spolehlivost. Výsledná analýza může později sloužit jako jeden ze vstupů do praktické tvorby a testování systému. [1] [4]

2.3.1.1 Základní parametry

Základním parametrem je intenzita poruchy navrhovaného zařízení nebo také pravděpodobnost, že v určitém časovém úseku přestane pracovat podle očekávání. Z intenzity lze dále odvodit další parametry jako například předpokládanou dobu bezporuchového běhu, životnost nebo dobu po které je nutné zařízení kontrolovat či opravit.

Tyto i další parametry jsou součástí zmíněné kvantitativní analýzy a lze je získat analýzou stromů poruch. FTA reprezentuje zařízení jako strom událostí, logických hradel a pomocných bloků. Strom, jehož kořen tvoří kořenová událost (například porucha celého zařízení), jejíž parametry nás nejvíce zajímají. Níže se v jednotlivých hladinách stromu střídají události a logická hradla až po listy, které vždy tvoří základní události. Pokud je cílem nalezení přesného číselného výsledku libovolných parametrů, je nutné znát tyto hodnoty pro všechny základní události. Nejpoužívanějšími logickými hradly jsou OR a AND, ale je možné využít libovolné logické hradlo, které má na vstupu n binárních hodnot a na výstupu jednu binární hodnotu. Následně lze z takto namodelovaného stromu postupně mechanicky dopočítat parametry všech událostí, včetně té kořenové. Při výpočtu se postupuje od základní události výše, jelikož parametry události závisí jen na událostech a hradlech v podstromu, jehož je kořenem. [1]

2.3.1.2 Single-time

V závislosti na typu hodnot se analýza dále dělí dva *single-time* a *continuous-time*. *Single-time* přístup nebere v potaz změnu systému v čase. Pracuje se s fixním časovým obdobím, ve kterém může každá část selhat nejvýše jednou. Pro správný výpočet je předpokladem nezávislost všech základních událostí. [6]

Vstupním parametrem všech základních událostí je poté pravděpodobnost poruchy v definovaném časovém okně s hodnotou z intervalu $[0, 1]$. Spolehlivost systému potom představuje pravděpodobnost, že nenastala vrcholná událost stromu. Nebo-li doplněk k pravděpodobnosti poruchy vrcholné události. [6]

2.3.1.3 Continuos-time

Předchozí *single-time* přístup zobecňuje celý proces a neuvažuje změny systémů v čase. V praxi je však běžnější, že části systému s časem mění své vlastnosti. Typicky jsou například starší součástky náchylnější k poruše. *Continuos-time* stromy poruch dokáží tuto skutečnost obsáhnout a poskytují tím přesnější odhad chování systému. Na rozdíl od *single-time* se pravděpodobností poruchy zadává jako funkce v čase s hodnotou v intervalu $[0, 1]$. ($\mathbb{R}^+ \rightarrow [0, 1]$). V praxi je často možné aproximovat inverzí exponenciálního rozdělení a počítat s parametrem *lambda*. V některých případech může být vhodné použít rozdělení jiného typu.

V případě *continuos-time* stromů poruch lze k poruchám zavést i opravy částí systému. Podobně jako u poruch je oprava také pravděpodobností funkcí v čase, jejíž počátek nastane v době poruchy. Platí zde i častá aproximace exponenciálním rozdělením.

Bezporuchovost v *continuos-time* stromu poruch představuje pravděpodobnost, že systém bude bezporuchový i po uplynutí určitého času. Pro výpočet spolehlivosti v určitém čase, lze *continuos-time* převést do *single-time*, převodem pravděpodobnosti poruch v daném časovém rámci.

Dostupnost systému je veličina vázána na určitý čas a určuje, zda je v tomto čase systém dostupný. Pro převod z *continuous-time* do *single-time* stačí nahradit poruchové rozdělení zadané základním událostem právě dostupností v určitém čase.

2.3.1.4 Výpočet spolehlivostních parametrů

Hlavním cílem spolehlivostní analýzy stromů poruch je dopočítání finálních spolehlivostních parametrů v závislosti na základních blocích, jejich provozní parametry nebo další spolehlivostní ukazatele jsou známé a jsou zadány jako vstup modelu. Pro výpočet parametrů vrcholné události se postupuje směrem zespoda nahoru, tedy od základních událostí postupně směrem k vrcholné události. Události jsou napojeny na určitý typ hradla, ty určují v jakých situacích se událost nebo porucha propaguje ve stromu výše.

Výpočet lze provést v různých jednotkách a formách vstupu. Jedním z nich je nastavení pravděpodobnosti poruchy, značeno P. V mnoha případech se ale pracuje s doplňkem této hodnoty, tedy pravděpodobností bezporuchového stavu, který se značí R. Přičemž platí, že jedna je doplňkem druhé, tedy:

$$P(t) = 1 - R(t). \tag{2.1}$$

Jednou z často počítaných hodnot je MTTF (Mean Time to Failure). Tato hodnota uvádí střední dobu do první poruchy, což je v praxi důležitá informace, která říká, za jak dlouho se systém v průměru porouchá. Výpočet je následující:

$$MTTF = \int_0^\infty R(t)dt \tag{2.2}$$

Vstupem základních událostí stromu poruch ale nejčastěji není samotná pravděpodobnost, častěji se uvádí hodnoty, které dodává výrobce jednotlivých součástek nebo jinak spočítané hodnoty v podobě intenzity poruch v čase nebo počet poruch za určitý čas. Z této hodnoty se pravděpodobnost řídí podle toho, zda jde o single-time nebo continuos-time, v praxi je samozřejmě častější případ, kdy se v čase pravděpodobnost mění. V tom případě je řízena nějakým pravděpodobnostním rozdělením a nejčastěji se jedná o exponenciální rozdělení s parametrem λ, tedy:

$$R(t) = e^{-\lambda * t} \tag{2.3}$$

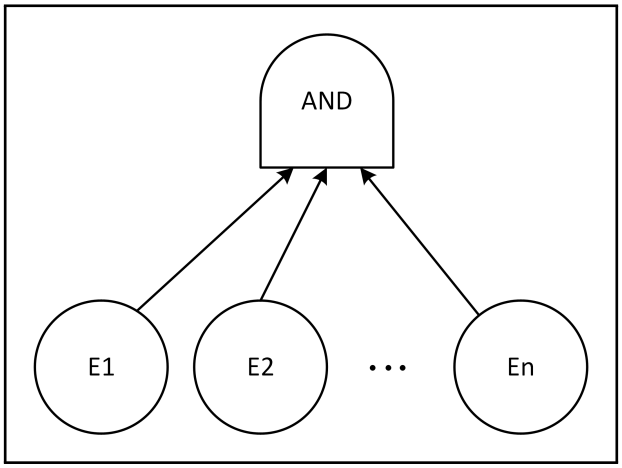
V samotném modelu do výpočtu vstupují společně s událostmi také hradla. Základními hradly jsou AND a OR. První jmenované označuje případ, kdy pro propagaci poruchy výše musí nastat porucha na všech vstupních událostech.

Pokud se ve výpočtu nachází pravděpodobnost poruchy, pak se výsledek pro hradlo AND (obrázek 2.2) rovná součinu pravděpodobností poruchy všech vstupních událostí:

$$P_{AND}(t) = P_1(t) * P_2(t) * ... * P_n(t), \tag{2.4}$$

naopak pravděpodobnost bezporuchového stavu se rovná doplňku hodnotě součinu doplňků hodnot vstupních událostí:

$$R_{AND}(t) = 1 - (1 - R_1(t)) * (1 - R_2(t)) * ... * (1 - R_n(t)). \tag{2.5}$$



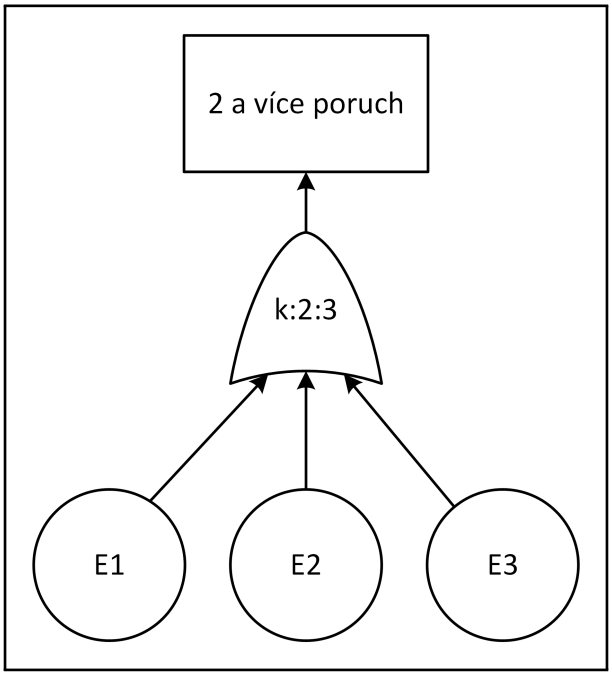
Obrázek 2.2 Hradlo AND s n vstupními událostmi. Výsledná pravděpodobnost poruchy resp. pravděpodobnost bezporuchového stavu je vypočítána pomocí vzorce 2.4 resp. 2.5

Hradlo OR značí přesně opačnou situaci, pro propagaci poruchy stačí, aby selhala alespoň jedna vstupní událost. Pravděpodobnost poruchy je tedy stejná jako pravděpodobnost bezporuchového stavu hradla AND a naopak.

$$P_{OR}(t) = 1 - (1 - P_1(t)) * (1 - P_2(t)) * ... * (1 - P_n(t)), \tag{2.6}$$

$$R_{OR}(t) = R_1(t) * R_2(t) * ... * R_n(t). \tag{2.7}$$

Typický příklad použití takového hradla je kontrola funkce pomocí majority. Vyskytuje-li se v systému nějaká kritická komponenta, která provádí výpočet, pak je možné použít tři takové komponenty najednou a pokud alespoň dvě dávají stejný výsledek, považuje se tato část systému za funkční. Dalším hradlem, který ovlivní výpočet, ale nevyskytuje se tak často, ke K-N hlasovací hradlo (obrázek 2.3). Na vstupu má číslo k od 1 do n (počet vstupů) a představuje logickou funkci, která je pravdivá v případě, že je alespoň k vstupů pravdivých. Přestože je toto hradlo nahraditelné kombinací AND a OR hradel, velmi zjednoduší celý strom poruch. Nahrazení takového hradla vyžaduje vytvoření jednoho OR hradla a do něj bude vstupovat $\binom{n}{k}$ AND hradel, kde každé z nich bude obsahovat jednu kombinaci k vstupních událostí. Typický příklad použití takového hradla je kontrola funkce pomocí majority. Kritická komponenta provádějící výpočet se duplikuje a za správný výsledek se považuje takový, který vrátí většina těchto shodných komponent.



Obrázek 2.3 Voting hradlo, které propaguje poruchu při alespoň 2 vstupních poruchách ze 3 vstupních událostí.

2.3.2 Kvalitativní FTA

Kvalitativní metoda je druhým typem analýzy stromů poruch, která si za cíl klade detekci silných a slabých míst analyzovaného zařízení. Slouží k nalezení takových částí, které je možné optimalizovat nebo více zabezpečit proti poruše. Strom je speciálním typem grafu a je tedy možné pro tuto analýzu využít grafové algoritmy pro hledání takových množin událostí, které mohou vést k vrcholné poruše. Pro nalezení takových slabých a silných míst se používá hledání minimálního řezu a minimální cesty v grafu. K provedení analýzy slouží Monte Carlo simulace nebo deterministické metody. [6] [7]

V průběhu analýzy je snaha typicky systém zlepšovat tak, aby byl co nejodolnější vůči poruchám. V průběžný návrh zařízení se hledají slabá místa a možnosti, jak je odstranit. K tomu se hodí hledání minimálního řezu (množin minimálních řezů) v grafu. Řez grafu vrací takové události, jejichž současná porucha automaticky vede k poruše kořenové události. Neboli pro každý řez platí, že pokud nastaly všechny jeho události, pak nutně nastala i kořenová událost. Po nalezení takových řezů stačí dopočítat pravděpodobnosti výskytu bloků v řezu a soustředit se na jejich snížení.

Obecně existují tři základní postupy kvalitativní analýzy, dle [6]. První se nazývá *minimal cut sets*, který hledá množiny základních událostí, jejichž současná porucha, způsobí poruchu kořenové události. *Minimal path sets* funguje přesně opačně. Je nutné zajistit, že kořenová událost není poruchová a hledají se takové množiny funkčních základních událostí, které to zaručí. Obě tyto metody je možné algoritmicky zpracovat v aplikaci a zobrazit uživateli v grafické podobě přímo v modelu.

Poslední uvedený postup se nazývá *Common cause failures*, ten zkoumá spíše neočekávané chyby, které mohou nastat například po duplikaci některé komponenty, stejnou komponentou od stejného výrobce. Pokud totiž bude tato komponenta trpět na nějaké vady ve výrobě, pak tím mohou být postiženy obě komponenty. Jedná se o jakési zavedení událostí, které naruší nezávislost základních událostí. Jelikož tyto situace závisí na externích faktorech a nejsou přímo

součástí modelu. Vzhledem ke své povaze není možné tento typ analýzy provést automaticky a v tomto případě musí analytik postupovat rozšířením modelu jinou cestou.

Kapitola 3
Analýza

Analýza práce se skládá z analýzy dosavadních řešení, popisu plánovaných rozšíření webové aplikace a zpracování podkladů k implementaci takových rozšíření. Hlavním bodem zadání je rozšíření o stromy poruch. Ty samozřejmě nejsou v původní webové aplikaci, ale jsou zpracovány jinými, existujícími a praxí ověřenými aplikacemi. Při vyzkoušení těchto aplikací se může objevit mnoho poznatků, které bude vhodné přenést do webové aplikace. Další zadané požadavky budou specifikovány při analýze dosavadní aplikace. K tomu budou přidány další požadavky, které vznikly v průběhu testování předchozí verze webové aplikace. Všechny budou specifikovány ve funkčních a nefunkčních požadavcích a vybrané z nich budou popsány podrobněji a jejich analýza bude využita v další kapitole zabývající se implementací těchto požadavků.

3.1 Existující aplikace pro FTA

FTA se řadí mezi jednu z nejpoužívanějších metod užívanou pro účely spolehlivostních nebo bezpečnostních analýz. Proto také existují nástroje, které je možné pro analýzu využít. V komerčním prostředí se FTA provádí v komplexních a často drahých nebo interních aplikacích. K těm se jednotlivec nebo malý tým zabývající se FTA problematikou těžko dostane. Při hledání dostupné alternativy zjistíte, že výběr vyhovujících aplikací je velice omezen. Nástroje, které lze získat pro potřeby výuky nebo vývoje malého týmu, jsou zastaralé a na nových systémech nemusí být plně funkční. Společně s chybějící podporou práce ve skupině více lidí nezbývá moc jiných možností, než provádět analýzu ručně nebo pouze za pomoci grafického editoru. Níže jsou vybrané aplikace analyzovány a výstupy budou později použity pro návrh, implementaci, závěrečné testování a porovnání s webovou aplikací.

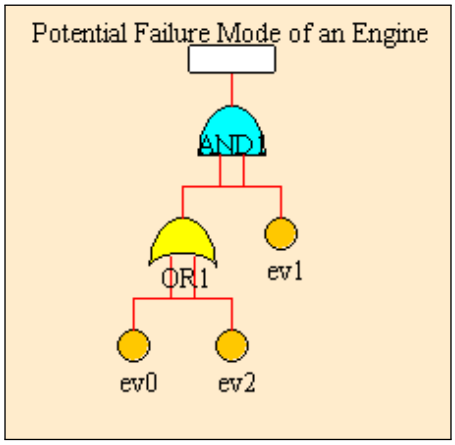
3.1.1 SHARPE

Jedná se desktopovou multiplatformní aplikaci, vytvořenou na DUKE University v roce 2002. Pro účely této práce byla použita verze 1.3.1 na Microsoft Windows XP. Po instalaci do systému se spustí okno s nabídkou ovládacích prvků. Dále se vytvoří projekt společně s prvním modelem. SHARPE nabízí poměrně velkou škálu modelů, které lze použít pro spolehlivostní analýzu. Zde se budu zabývat především stromy poruch. Po výběru se otevře nové okno s hlavní událostí. Tvorba modelu (obrázek 3.1) probíhá tak, že se vždy přidá hradlo určitého typu a počet vstupních událostí, které můžu dle potřeby rozšiřovat a měnit. Ke každé události nebo hradlu lze možné přidat text, který slouží jako podrobnější popis.

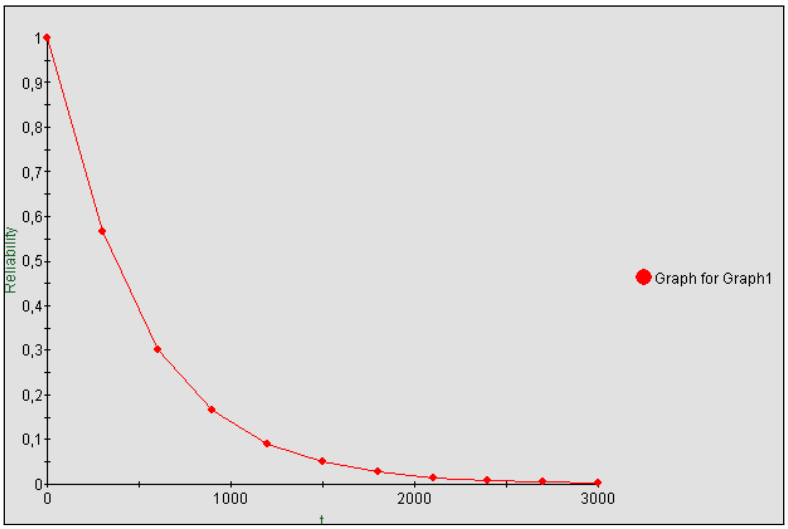
SHAPRE poskytuje výsledek kvantitativní analýzy vygenerováním reportu v podobě textu a grafu (obrázek 3.2), k tomu je nejprve nutné zadat všechny povinné spolehlivostní parametry

všem základním událostem. Bez tohoto kroku nelze report vygenerovat. Nejčastěji se spolehlivostní parametry zadávají jako parametry nějakého statistického rozdělení, SHARPE nabízí výběr mezi několika různými rozděleními. Výsledný report lze vygenerovat textově nebo jako vstup pro externí nástroje (například Wolfram Mathematica). Použití externích nástrojů značně rozšiřuje možnosti další práce s výslednými daty (tvorba grafů, změna parametrů, sdílení, ...). Po dokončení návrhu přichází uložení do souboru. Pokud je potřeba provádět analýzu ve více lidech, nezbývá nic jiného, než stromy rozdělit, pracovat paralelně a poté je propojit, aplikace neumožňuje přímou kooperaci více uživatelů.

SHARPE bylo původně vytvořeno pro starší verze Microsoft Windows. Na dnešních verzích operačního systému se mohou objevit různé chyby. Při analyzování práce s SHARPE jsem narazil na problém, kde na Microsoft Windows 11 nebylo možné vygenerovat výsledek modelu v žádné formě. Z toho důvodu jsem použil starší Microsoft Windows 10 a postupně jsem přešel zpět na Microsoft Windows XP, kde aplikace fungovala stabilně.



Obrázek 3.1 Příklad jednoduchého stromu poruch vytvořeného v programu SHARPE.



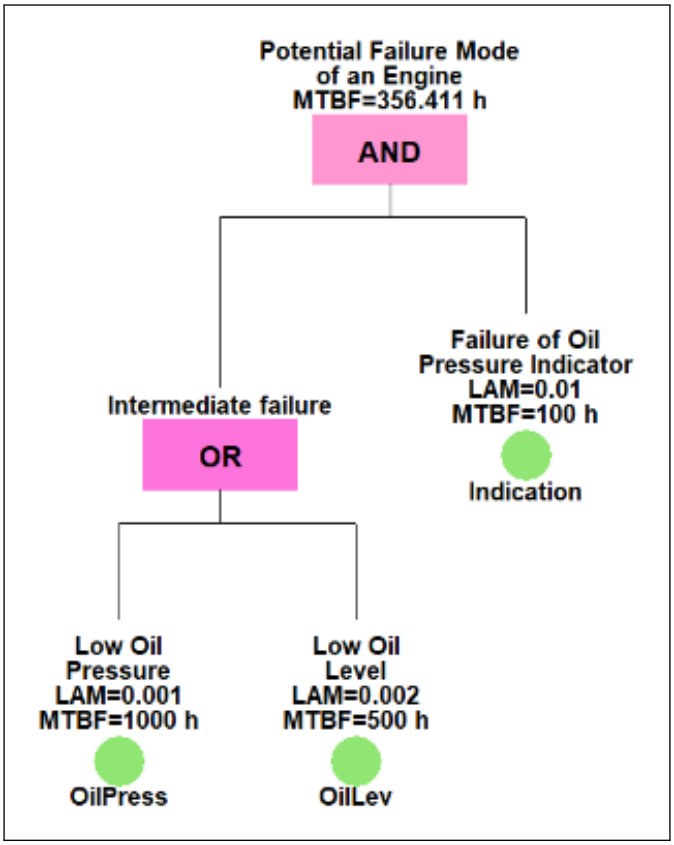
Obrázek 3.2 Vizualizace pravděpodobnosti bezporuchového stavu v čase programem SHARPE.

3.1.2 Zusim

Interní aplikace firmy Siemens, která není mimo Siemens běžně dostupná. Vzhledem k tomu ji není možné využít cílovou skupinou. V této práci je uvedena především pro své využití na reálných projektech v sektoru železnice. Slouží tak jako dobrý benchmark pro verifikaci výsledků a analýzu požadavků, které mohou uživatelé obdobných aplikací mít.

Opět jde o desktopovou aplikaci pro platformu Windows. Má stálou podporu a funguje správně i na novějších verzím operačního systému. Pro účely této práce byla použita verze 30.2.0 z roku 2016. Navržena především pro práci jednotlivce. Při spolupráci v týmu je potřeba, podobně jako v případě SHARPE, složitější strom poruch rozdělit na podstromy a na každém z nich pracovat jednotlivě. Poté sdílet exportované soubory přes externí úložiště.

Zusim dokáže kromě FTA, zobrazené na obrázku 3.3, provádět i analýzu Markovských řetězců. Pro tvorbu stromu poruch je možné využít základní události a hradla, včetně voting hradla. V rámci projektu umí propojit více modelů přechodovými bloky a vytvořit tak komplexní strom poruch složený z několika jednodušších. Vstupní parametry je možné zadávat ve více formách, *lambda* hodnota, MTBF nebo FIT. Výsledek propisuje přímo do modelu, ale naopak neumožňuje širší report, pouze vypočítanou hodnotu ve zvolených jednotkách. Jejich správnost je ověřena na mnoha již vyvinutých projektech. Tím však analýza končí a bez úpravy modelu je není možné dále měnit.



Obrázek 3.3 Příklad jednoduchého stromu poruch vytvořeného v programu ZUSIM.

3.2 Dosavadní stav webové aplikace

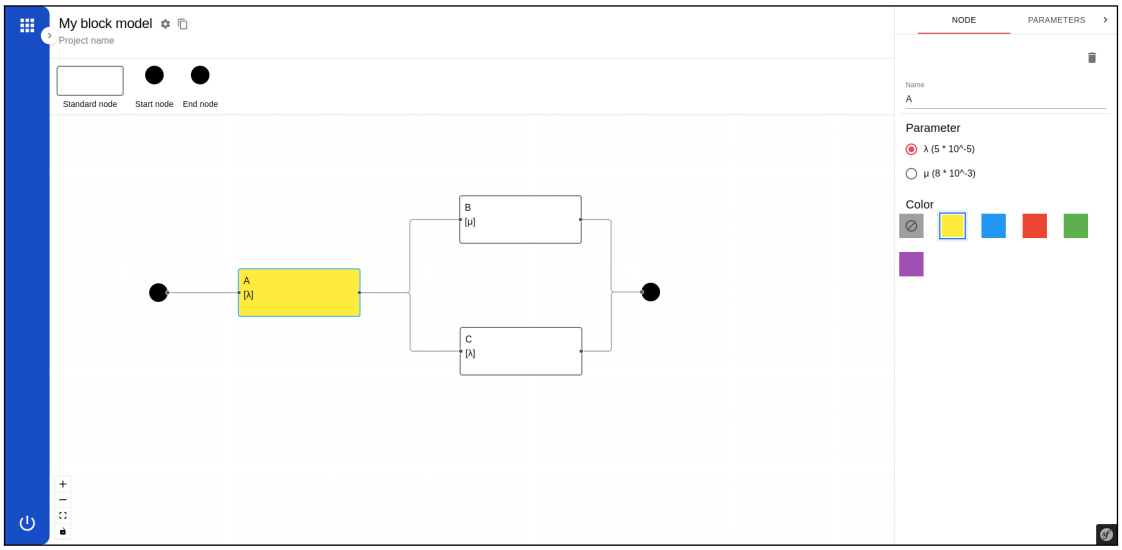
Tato práce navazuje na mou obhájenou bakalářskou práci z roku 2022. [8] V průběhu analýzy je důležité zanalyzovat dosavadní stav webové aplikace, které mohou vyústit v nové požadavky pro zlepšení výsledné aplikace. Aplikace se soustředí na komplexnější spolehlivostní analýzu pomocí dvou typů modelů a textové dokumentace. Soustředí se přitom na dobré uživatelské rozhraní, aby byla analýza oproti ostatním aplikacím jednodušší a příjemnější na práci jednotlivce nebo malého týmu.

Při testování finální aplikace se objevilo několik problémů a situací, ve kterých aplikace není příliš intuitivní. Při práci na větších projektech se tak uživatel setká s několika překážkami, které mohou práci znepříjemnit. Vzhledem k tomu, že je jedním z hlavních úkolů aplikace předat uživateli jednoduchý a přívětivý nástroj, měli by být tyto problémy odstraněny.

Plánovaným využitím webové aplikace je mimo jiné použití při výuce předmětu Testování a spolehlivost na FIT ČVUT. Z tohoto důvodu musí být umožněno studentům přihlášení pomocí fakultního účtu. Fakulta umožňuje podobným aplikacím toto přihlášení využít pomocí technologie OAuth na adrese *auth.fit.cvut.cz*. Webová aplikace by tedy měla tuto formu přihlášení umožnit.

3.2.1 Testování aplikace

Při testování aplikace ve výuce předmětu *Testování a spolehlivost* se velmi brzy objevil problém související s registrací a správou uživatelů. Prvním krokem, který jsem potřeboval udělat bylo přidat přítomné studenty do systému a ke společnému projektu. To však nebylo možné udělat ve chvíli, kdy neexistoval jejich účet ani je nešlo přidat nijak hromadně, navíc jsem neměl žádnou možnost zjistit jejich uživatelská jména přímo v aplikaci. Musel jsem tedy počkat na jejich registraci, zeptat se na uživatelské jméno a až poté je ručně po jednom přidat. Vzhledem k tomu, že se jedná o plánovaný případ užití, považuji to za nedostatek, který by měla rozšířená aplikace adresovat. S tím souvisí nejen správa uživatelů, ale také jejich projektů.



Obrázek 3.4 Spolehlivostní analýza RBD modelu pomocí původní verze webové aplikace.

Po založení nového projektu nebo modelu a tvorbě složitějšího aplikace kromě přidání elementů (bloků a hran) vyžaduje zadání vstupní hodnoty pro určité elementy. U blokových modelů

se jedná o intenzitu poruchy, u markovských se používá pravděpodobnost přechodu. Příklad blokového modelu ve webové aplikaci je zobrazen na obrázku 3.4. Uživatel je tak omezen na tyto vstupní hodnoty, které není možné zadat v alternativních formách, což ostatní aplikace často umožňují. Problémy může uživatelům způsobovat forma zadávání, obzvláště ve větších modelech. Vstupní hodnoty se zadávají pomocí parametrů, nejprve je definován parametr názvem a hodnotou ve formátu $a*10^b$. Většina hodnot zadaných výrobci součástek a norem tento formát používá, nicméně v některých případech musí uživatel hodnoty přepočítat a až poté je zadat do aplikace. Parametry na sobě nemohou záviset, pokud se vytvoří parametr $\lambda = 3 * 10^{-3}$ a následně druhý parametr μ roven $2 * \lambda$, musí být definovány postupně. Nejprve se zadá $\mu = 6 * 10^{-3}$ a při každé změně parametru λ se ručně změní i μ. Pokud je v modelu několik na sobě závisících parametrů, může být náročné je udržovat a pro každou novou hodnotu vytvořit nový pojmenovaný parametr. To pochopitelně prodlužuje dobu, po kterou trvá tvorba modelu.

Při práci s obdobnými webovými aplikacemi je často zvykem využití standardních klávesových zkratek pro určité akce. Například pomocí klávesy *DELETE* smazat označenou část modelu nebo *CTRL+Z/Y* pro vrácení či opakování poslední akce. Tyto dvě funkce zde chybí a uživatel je tak nucen smazat elementy otevřením podrobností a stiskem tlačítka pro smazání. Poté už není schopen takovou změnu vrátit a může tak přijít o část své práce.

3.3 Požadavky na rozšíření aplikace

Před návrhem aplikace je nutné specifikovat konkrétní a ověřitelné požadavky, které musí finální aplikace obsahovat. Jelikož je cílem práce rozšířit již existující aplikaci, nebudou zde požadavky, které jsou již implementovány a zmíněny v předchozí bakalářské práci. Všechny požadavky vycházejí buďto ze zadání, analýzy existujících nástrojů nebo komunikace s vedoucím práce, který má v tomto ohledu praktické zkušenosti. Dále budou rozděleny na funkční a nefunkční požadavky. Následně je představeno několik případů užití, které krok po kroku popisují několik běžných scénářů použití aplikace. Při návrhu tak bude mnohem zřetelnější, jaká jsou od rozšířené webové aplikace očekávání.

3.3.1 Funkční požadavky

F1 - Autentizace účtem FIT ČVUT
Uživatelé se mohou zaregistrovat a přihlásit fakultním účtem FIT ČVUT.

F2 - Historie úprav
Při práci s modelem mohou uživatelé pomocí klávesových zkratek vrátit dříve provedené úpravy nebo je provést znovu.

F3 - Komplexní vstupní hodnoty
Vstupní hodnoty je umožněno zadávat ve formě matematického výrazu. Výraz může obsahovat standardní matematické operátory, základní funkce (logaritmus, mocninu, průměr) nebo proměnnou. Ta je aliasem pro jiný matematický výraz sdílený všemi bloky v modelu. Parametry mohou být zadány ve více formách (intenzita poruch, MTBF, FIT).

F4 - Zadání vstupní hodnoty výstupem jiného modelu
Uživatel může zadat vstupní hodnotu parametru zvolením jiného modelu. Vstupní hodnota se bude rovnat výstupní hodnotě zvoleného modelu. Tím je umožněno modely kombinovat a vnořovat.

F5 - Administrační sekce pro správu systému
Administrátoři aplikace mají plnou kontrolu nad systémem. V samostatné sekci, dostupné pouze pro ně, mohou provádět CRUD operace nad všemi uživateli, projekty, modely a dokumentacemi.

F6 - Převod modelu na obrázek
Uživatel může vytvořený model exportovat do obrázkového formátu. Na výběr jsou alespoň dva formáty: PNG a SVG.

F7 - Model stromu poruch
Aplikace dokáže provést spolehlivostní analýzu třetím modelem, stromem poruch. Model obsahuje všechny bloky nutné pro provedení analýzy, ale také pokročilé přechodové bloky. Ty je možné propojit s dalším stromem poruch nebo jeho částí.

F8 - Kompletní export stromu poruch do Wolfram Mathematica notebooku
Výsledek stromu poruch může být exportován do notebooku Wolfram Mathematica. Notebook obsahuje všechny části, které jsou pro výpočet potřebné, včetně výpočtu modelů, na kterých je výstup závislý, grafů a komentovaného výstupu.

F9 - Automatické uspořádání bloků pro všechny obsažené modely
Aplikace zvládne sama automaticky uspořádat bloky. Každý typ modelu je uspořádán podle standardních zvyklostí z literatury.

3.3.2 Nefunkční požadavky

N1 - Provedení spolehlivostní analýzy složitých modelů se stovkami bloků
Aplikace je schopna provést analýzy všech typů modelů, které jsou sestaveny až ze stovek různých bloků.

N2 - Spolupráce až 15 uživatelů
Aplikace zvládá všechny funkce modelů s až 10 aktuálně připojenými uživateli bez chyb a znatelného snížení výkonu.

N3 - Uživatelsky přívětivé rozhraní
Aplikace by se měla soustředit na zjednodušení a zrychlení analýzy. Uživateli by neměla vystavovat zbytečné překážky, místo toho přinést takové rozhraní, které neopakuje chyby podobných systémů a příjemně se s ní pracuje.

3.3.3 Případy užití

UC1 - Registrace účtem FIT ČVUT
1. Uživatel je studentem FIT ČVUT a již vlastní fakultní účet. Případ užití začíná na úvodní stránce webu. V aplikaci nemusí vyplňovat žádné přihlašovací údaje a zvolí přihlášení pomocí fakultního účtu.

2. Je přesměrován na autentizační stránky fakulty, kam zadá své přihlašovací údaje. Po potvrzení a úspěšném přihlášení je přesměrován zpět do aplikace. Nyní už se nenachází na úvodní stránce, ale na stránce se seznamem projektů.

3. Pokud se jednalo o první přihlášení, všechna data o uživateli, jako jeho *username* jsou předem vyplněna, uživatel se tak nemusí starat o jejich zadávání a může aplikaci ihned okamžitě používat.

UC2 - Vytvoření nového stromu poruch
Uživatel se nachází detailu projektu, kde může mít již vytvořené nějaké modely. Na této stránce je zobrazeno tlačítko pro vytvoření nového modelu, zobrazí se formulář, který obsahuje pole pro název modelu a jeho typ. Jedním z typů modelu je strom poruch. Po jeho zvolení, vyplnění názvu a potvrzení je model vytvořen a uživatel přesměrován do detailu modelu, se kterým může dále pracovat.

UC3 - Návrat změn provedených při tvorbě modelu

1. Případ užití začíná v detailu modelu stromu poruch. Ten již obsahuje nějaká hradla a události. Uživatel se zde nachází sám, bez dalších připojených uživatelů.

2. Uživatel pracuje na analýze a modifikuje model. Poté smaže jedno z hradel, které má vstupní i výstupní hrany. Následně si tuto úpravu rozmyslí a chce ji vrátit. Po stisku určené klávesové zkratky se hradlo společně se smazanými hranami znovu objeví v modelu a uživatel může pokračovat tak, jako kdyby hradlo nikdy nesmazal.

3. Analýza dále probíhá a uživatel provedl řadu změn, mezi nimi bylo vytvoření událost, změna názvu, přesun elementu a další akce, které může vyvolat. Podobně jako v předchozím kroku si poslední úpravu rozmyslel a vrátil zpět. Spolu s ní vrátil i další dvě úpravy, které vracet nechtěl. Pomocí další klávesové zkratky provede znovu vrácené akce, ty se provedou přesně tak, jako dříve a může pokračovat v analýze.

UC4 - Export modelu do obrázkového formátu

1. Tento případ užití je důležitý pro situace, kdy potřebuje uživatel model v podobě obrázku vložit do nějakého článku, dokumentace nebo jen někomu odeslat. Začíná v detailu modelu připraveného k převodu na obrázek.

2. Uživatel otevře nabídku pro export do obrázku. Jako první zvolí, zda má mít výsledek transparentní pozadí či nikoliv. Následně zadá velikost okraje v pixelech. A nakonec zvolí jeden z dostupných formátů.

3. V závislosti na prohlížeči a operačním systému může vybrat umístění souboru. Po krátkém generování obrázku se obrázek stáhne do uživatelova zařízení.

UC5 - Zadávání vstupu matematickým výrazem

1. Uživatel provádí analýzu a nachází se v detailu modelu. Strom poruch je vytvořený a nyní chce zadat vstupní hodnoty základním událostem.

2. Dvojklikem na událost otevře její nabídku. Nabídka obsahuje několik nastavení, důležitá je vstupní hodnota intenzity poruch.

3. Uživatel zadá neplatný matematický výraz (např. $*2*3$). Aplikace výraz zvýrazní a zobrazí chybovou hlášku.

4. Uživatel opravuje výraz a připisuje proměnnou (např. $2 * \lambda$). Výraz je platný, ale proměnná nemá přiřazenou hodnotu, proto se chybová hláška mění na varování a zvýrazněna je pouze λ.

5. Uživatel otevře nabídku s proměnnými a definuje novou proměnnou s názvem λ a hodnotou $10\hat{}(-3)$. Po návratu do nabídky původní varovaní zmizí a výraz je tedy bez problému.

6. Nyní přejde do nabídky jiné události a může stejně jako u předchozí zadat matematický výraz s proměnnou λ.

UC6 - Acyklické vnořené stromy poruch

1. Uživatel se nachází v detailu projektu a zpracovává analýzu systému složeného z jedné hlavní komponenty a dvou podpůrných.

2. Nejprve vytvoří první strom poruch pro podpůrnou komponentu A a vyplní vstupní hodnoty. Nyní vytvoří hlavní komponentu se dvěma základními událostmi. První události zadá vstupní hodnotu přebranou z výstupu komponenty A. Komponenta A je tedy vnořena do hlavní komponenty.

3. Uživatel vytvoří strom poruch pro podpůrnou komponentu B a v jedné základní události zadá chybně intenzitu poruch převzatou z výstupu hlavní komponenty.

4. Následně se vrátí zpět k hlavní komponentě a ke druhé základní události nastaví intenzitu poruch převzatou z komponenty B. V tuto chvíli spustí generování notebooku a aplikace vrací chybu, jelikož je mezi vnořenými stromy vytvořen cyklus a není možné vygenerovat výsledek.

5. Po kontrole uživatel chybu opraví a v komponentě *B* zadá intenzitu výrazem. Opětovné vygenerování notebooku uspěje a bude obsahovat výpočet pro všechny tři modely s důrazem na výsledek hlavní komponenty.

UC7 - Správa aplikace administrátorem

1. Administrátor začíná na úvodní stránce s přihlášením. Hlavní administrátorský účet není vhodné provázat s fakultním účtem, proto se přihlásí pomocí účtu vytvořeném v aplikaci. Zadá přihlašovací jméno (e-mail), heslo a dostane se do aplikace.

2. V menu má tento účet navíc sekci administrace. Jako první přidá nového uživatele, poté mu vytvoří projekt a jeden model.

3. Administrátor přejde do sekce *Uživatelé* a zobrazí si výpis. Z výpisu zkontroluje, že uživatel již existuje a může mu tak přiřadit nový projekt.

4. Uživatel je vytvořen a proto se přesune do sekce *Projekty* a na formulář pro vytvoření nového projektu. Zadá název projektu, zvolí autora a vytvoří projekt.

5. Vytvoření projektu se dostane do administračního detailu projektu, kde opět pomocí formuláře vytvoří nový model, zadá stejné údaje jako při vytváření modelu za uživatele a potvrdí. Projekt i model je připraven a až se uživatel přihlásí, může začít na modelu pracovat.

UC8 - Spolupráce více uživatelů

1. V tomto případě figurují dva uživatelé, oba již mají účty v aplikaci. První uživatel vytvoří projekt a model standardním procesem.

2. Poté přejde na detail projektu a přiřadí druhého uživatele k projektu s oprávněním k úpravě projektu.

3. Oba nyní přejdou do detailu modelu, kde pracují na analýze. Jeden z uživatelů provede změnu v modelu (přidání bloku, odebrání hrany, změna názvu bloku), ihned po jejím potvrzení serverem se stejná změna projeví i u druhého uživatele bez jakékoliv akce z jeho strany.

3.4 Technologie

Webová aplikace, která je cílem této práce vychází z již existující aplikace vytvořené v předešlé práci. Z toho důvodu je většina technologií již použita a bez přepracování aplikace není možné některé technologie nahradit. V této kapitole jsou všechny použité technologie popsány a komentovány důvody, proč bude nebo nebude oproti původní aplikace nahrazena.

3.4.1 Serverové technologie

PHP

PHP je open-source skriptovací jazyk na straně serveru. Jeho původ sahá do roku 1994, kdy byl vytvořen v programovacím jazyce C a původně sloužil jako souhrn podpůrných skriptů na webové stránce. [9] Od jeho vzniku prošel dlouhým vývojem a nynější podporovaná verze PHP 8.x se používá se převážně pro vývoj webových stránek a aplikací, ale i ostatních projektů, včetně grafických uživatelských rozhraní. [10] Výhodou použití PHP je jednoduchá integrace do HTML dokumentu. Vzhledem k ostatním jazykům je relativně jednoduchý na pochopení. A nakonec má širokou základnu vývojářů, knihoven a frameworků (především webových). I kvůli těmto výhodám se dle webu [11] jedná o nejpoužívanější jazyk na webových stránkách.

Serverová část původního projektu poskytuje standardní rozhraní pro klientskou část. PHP bylo vybráno z důvodu jednoduchosti a předchozím zkušenostem s jazykem i dalšími knihovnami, které práci velmi zjednoduší. Cílem této práce je rozšířit aplikaci o několik nových funkcí

a přinést tak uživatelům obecně lepší nástroj pro spolehlivostní analýzu. Žádný z požadavků není v rozporu se zvoleným programovacím jazykem a proto v aplikaci zůstane i nadále.

Symfony

Symfony je uznávané, stabilní vývojové prostředí. Inovativní a snadno použitelné pracovní prostředí díky integraci řešení z jiných prostředí, jako je *Dependency Injection* nebo vlastních řešení jako například *Web Profiler*. Symfony je nejčastěji používáno pro vývoj středních a velkých webových aplikací. Pro ty existuje mnoho různých rozšiřujících řešení specificky připravených právě pro integraci se Symfony. Jedná se o pragmatický nástroj připravený pro požadavky na moderní webové aplikace s dlouhodobou podporou. Pro efektivnější využití může být strukturován pomocí vzoru MVC (Model-View-Controller). Jedná se o architektonický vzor, který vývojářům pomáhá navrhovat webové stránky nebo webové aplikace strukturovaně s oddělenými odpovědnostmi. [12] [13] [14] [15]

Nahrazení frameworku by způsobilo přepis podstatné části aplikace. Finální aplikace bude pravděpodobně rozšířena o několik serverových funkcionalit, jako je parsování matematického výrazu nebo autentizace účtem třetí strany. Nic z toho není pro tak komplexní nástroj jako je Symfony samozřejmě problém, naopak velká škála rozšíření pro tvorbu REST API nebo napojení na službu Mercure je dostatečným argumentem pro zachování frameworku.

PostgreSQL

PostgreSQL je výkonný, open-source objektově-relační databázový systém s historií sahající do roku 1986. Proslavil se svou osvědčenou architekturou, spolehlivostí a integritou dat, nabízející bohatou funkcionalitu a rozšiřitelnou open-source platformu. Od roku 2001 splňuje standard ACID a běží na hlavních operačních systémech. Podporuje různé typy dat, nabízí robustní nástroje pro zachování integrity dat a umožňuje vývojářům vytvářet odolné prostředí. Databáze poskytuje pokročilé funkce, jako jsou sofistikované indexování a kompilace výrazů za běhu. [16]

Podobně jako u předchozích technologií bylo PostgreSQL zvoleno jako databázový systém již v předchozí práci. Z pohledu nároků na databázový systém není webová aplikace pro spolehlivostní modely příliš náročná a všechny potřebné funkce splňuje PostgreSQL bez problému. Důležitá je také návaznost knihovny pro objektově relační mapování (ORM) na databázový systém, Symfony používá knihovnu Doctrine, která má plnou integraci na tento PostgreSQL.

Mercure

Mercure je open-source řešení pro zprostředkování komunikaci více systémů v reálném čase, které klade důraz na rychlost a spolehlivost. Slouží jako moderní alternativa k *WebSocket* a podobným knihovnám, Mercure bezproblémově integruje streamování a asynchronní posílání zpráv do REST a GraphQL API. Přidává další vrstvu nad protokol HTTP, poskytující nativní podporu pro moderní webové prohlížeče, mobilní aplikace a zařízení Internetu věcí. [17] [18]

Mercure na serveru spustí takzvaný Hub, přes který probíhá veškerá komunikace. Každý klient se zaregistruje k danému tématu (topic) a inicializuje obousměrné připojení na Mercure Hub. Pokaždé, když některý z klientů pošle zprávu na Hub, ten reaguje odesláním stejné zprávy na všechny ostatní klienty. Autentizace probíhá pomocí JWT (JSON Web Token)[19], kterými lze omezit různé formy přístupu, například přístup pouze pro čtení a omezený na specifická témata.

Mercure je nyní v aplikaci stěžejním bodem umožňující spolupráci více uživatelů zároveň. Při prvotní integraci jsem byl se službou velmi spokojen a to převážně s jednoduchou formou autentizace, která je plně připravena k použití nebo formátem zpráv, který je možné nadefinovat naprosto libovolně. Pro požadavky na synchronizaci klientů tato služba zcela dostačuje a vyhovuje.

3.4.2 Klientské technologie

TypeScript

TypeScript je silně typovaný programovací jazyk postavený na jazyku JavaScript. Přidává bezpečnější a robustnější nástroje. Přináší statickou kontrolu typů, která detekuje chyby před spuštěním programu, a zlepšuje spolehlivost kódu tím, že upozorňuje na neočekávané chování a snižuje tak pravděpodobnost chyb v programu. Kód napsaný v TypeScriptu lze převést na JavaScript, což zajišťuje kompatibilitu na různých platformách, včetně prohlížečů a *Node.js*. [20]

React

React je open-source JavaScript knihovna pro tvorbu uživatelských rozhraní. Je oblíbená pro svůj přístup založený na komponentách. Jsou to znovu použitelné části kódu zobrazující pouze určitou část rozhraní (např. tlačítko, blok textu, ...). React používá deklarativní styl, kde popíšete, jak chcete, aby UI vypadalo, a React se efektivně postará o aktualizace. Díky tomu je kód snadněji pochopitelný a udržovatelný. Podle [21] je React nejpoužívanější knihovnou pro vývoj klientských aplikací. [22] [23] [24]

React je jedním ze základních knihoven, které se v klientské části aplikace objevují. Tvoří tak základ celého přístupu, na kterém je postaven zbytek aplikace. Stejně tak stěžejní knihovna React Flow, která se stará o vykreslení a manipulaci s modelem. React je proto v tuto chvíli velmi těžko nahraditelný a v aplikaci zůstane i pro nové funkce.

Redux

Redux je open-source JavaScript knihovna navržená speciálně pro prediktovatelné řízení stavu aplikace [25]. Nejčastěji se používá pro tvorbu uživatelských rozhraní společně s knihovnami jako React nebo Angular. Redux centralizuje stav aplikace, kde ukládá a spravuje většinu dat. To zjednodušuje správu komplexních aplikací. Celkově tím Redux pomáhá vyvíjet komplexní a škálovatelné aplikace pomocí jednotného přístupu ke správě stavu.Vynucuje predikovatelný způsob aktualizace stavu aplikace. Toho se dosahuje pomocí předem definovaných akcí, které popisují změnu. Ke každé akci je zadefinována funkce, která podle typu akce data aktualizuje. [26] Celkově tím Redux pomáhá vyvíjet komplexní a škálovatelné aplikace pomocí jednotného přístupu ke správě stavu.

Při tvorbě klientské části aplikace velmi často nastala potřeba přidat data do společného stavu aplikace, sdíleného mezi více nezávislými komponentami. V případě použití Reduxu je nutné v každém takovém případě vytvořit několik nových akcí, funkcí a obecně přidat hodně kódu. S přibývajícími požadavky se kód stává více složitějším vhodným řešením může být změna knihovny.

Zustand

Zustand je knihovna pro správu stavu v React aplikacích. Poskytuje jednoduchý a flexibilní způsob, jak tento stav spravovat. Zustand je postavený na React Context API a využívá moderní funkce JavaScriptu, jako jsou proxy a generátory. Zjednodušuje vytváření a aktualizaci komplexních datových struktur, umožňuje zpracovávat asynchronní akce a sdílet stav mezi komponentami. Je navržen tak, aby byl snadno použitelný a lehký, s minimálním API, které lze rychle zvládnout. [27] [28]

Vzhledem k tomu, že je Zustand jednodušší a méně komplexní než Redux, mělo by dojít ke zjednodušení většiny funkcí pro správu stavu aplikace. Aplikace se tam stane čitelnější a jednodušší na údržbu nebo případné změny, které dle požadavků do této části aplikace jistě zasáhnou.

3.4.3 Autorizace účtem třetích stran

Při analýze autentizace pomocí třetích stran je nutné vycházet z požadavků aplikace. Jediným požadavkem, který se týká autentizace je požadavek *F1*. Způsob implementace se musí řídit podle požadavků autentizačního systému FIT ČVUT. Fakulta poskytuje veškeré informace týkající se napojení aplikací na webové stránce: auth.fit.cvut.cz. Dokumentace popisuje pouze napojení přes autentizační protokol OAuth2.

3.4.3.1 OAuth 2.0

OAuth 2.0 je průmyslovým standardem pro autorizaci. Zaměřuje se na jednoduchost pro vývojáře klientských aplikací a zároveň poskytuje specifické autorizační toky pro webové aplikace, desktopové aplikace, mobilní telefony a další zařízení. OAuth specifikace a její rozšíření jsou vyvíjeny pracovní skupinou IETF OAuth. OAuth 2.0 slouží k umožnění přístupu k určitým skupinám dat pro třetí stranu. [29] [30]

OAuth 2.0 využívá pro autorizaci přístupové tokeny (Access Token), což je datový blok, který zajišťuje oprávnění k přístupu k datům uživatele. Nejčastěji se pro přístupové tokeny používá JSON Web Token (JWT), ten dokáže přímo s tokenem přenést i některá data , včetně hlavičky obsahující hashovací algoritmus, datum expirace a další. Nicméně standard nespecifikuje žádný konkrétní formát. OAuth 2.0 specifikuje více procesů, jak může autorizace probíhat, každý server si vybere ten, který nejvíce vyhovuje jeho použití. Níže popíšu proces, který implementuje fakultní systém, tedy ten na který bude aplikace využívat.

Nejprve je nutné vysvětlit termíny několik termínů, které OAuth používá:

Resource Owner - Uživatel, který povoluje aplikaci třetí strany přístup ke svým datům.

Client - Klientem je aplikace třetí strany, která žádá po uživateli přístup k datům. Předtím, než data získá musí být přístup schválen uživatelem (Resource Owner). Každý klient se musí předem zaregistrovat, autorizační server pro každého klienta vydá identifikátor a *secret*. Ty jsou odeslány při inicializaci a autorizační server tak přesně ví, jaký klient data požaduje.

Resource server - Server, který drží data uživatelů a přijímá požadavek od klienta (Client).

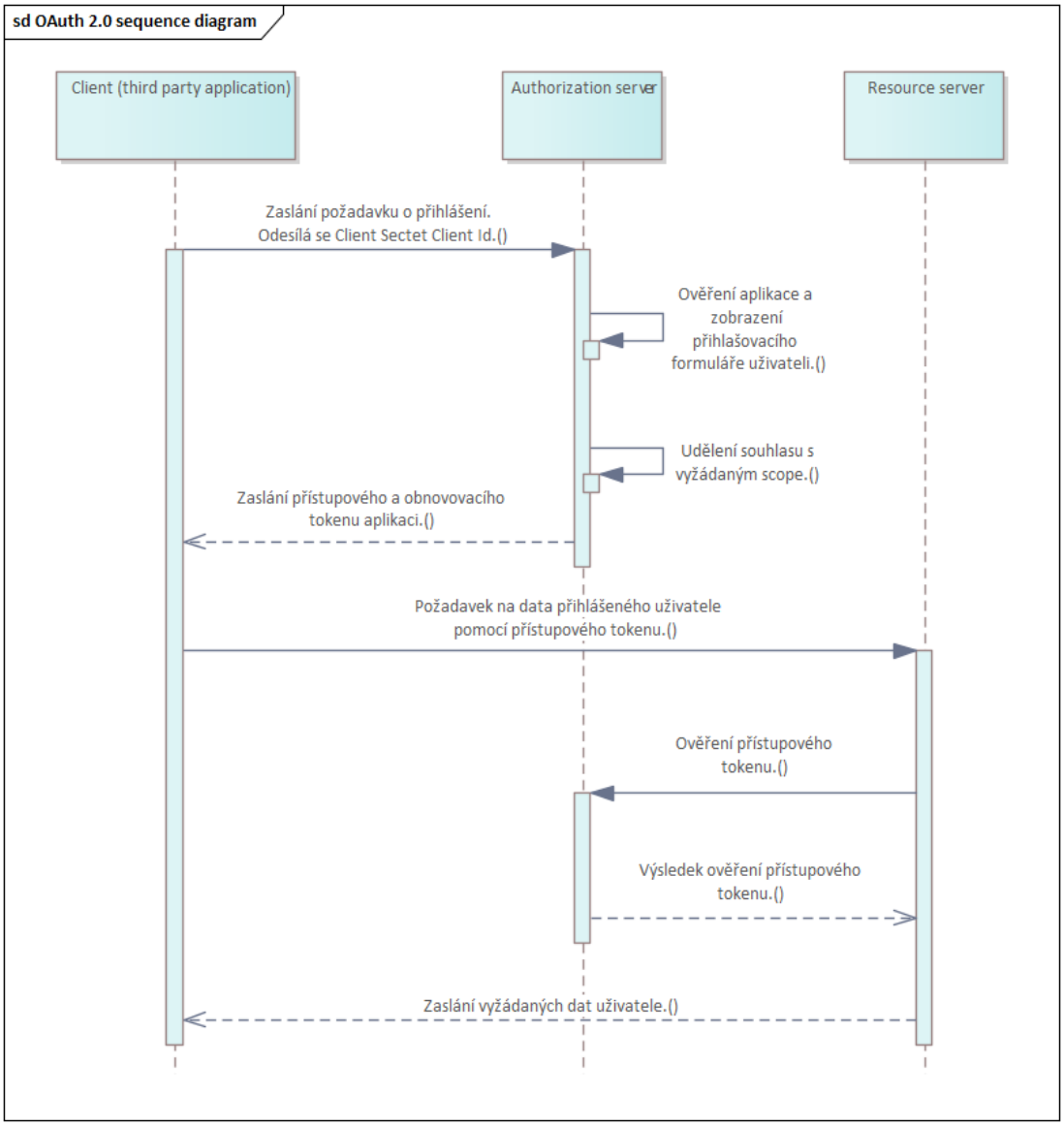
Authorization server - Server přijící požadavek k získání přístupového tokenu a po úspěšném požadavku vrací token klientovi (Client). typicky obsahuje dva přístupové body, jeden pro samostatné přihlášení a potvrzení přístupu uživatelem (Resource Owner) a druhý pro vyřízení požadavku od klienta (Client).

Scope - Kategorie dat, ke kterým Client žádá přístup. Před schvalováním by měl uživatel přesně vědět, jaký *scope* bude zobrazen (např. email, kontaktní údaje, profilový obrázek). V případě FIT ČVUT může být scope pro uživatelské údaje, známky, závěrečné práce a další.

Systém *auth.fit.cvut.cz* používá pro získání přístupového tokenu proces zahrnující autorizační kód (Authorization Code) a později také obnovovací token (Refresh Token). Komunikaci inicializuje *Client*, ten odešle identifikátor, *secret*, *scope* a URI pro přesměrování. URI bude použita v případě úspěšného i neúspěšného přihlášení, aplikace tím označuje místo, které dokáže zpracovat výsledek vrácený autorizačním serverem a dále zobrazí uživateli. Autorizační server uživateli zobrazí nějakou formu přihlášení (nejčastěji přihlašovací formulář) a po přihlášení ho nechá potvrdit přístup k požadovaným datům (scope).

V případě úspěšného přihlášení a potvrzení přístupu se na dříve zaslanou URI odešle autorizační kód. Zasláním tohoto tokenu zpět na autorizační server získá *Client* přístupový token a obnovovací token. Ten už se používá pro získání dat uživatele nebo provedení akcí na *Resource Server*. Každý přístupový token by měl být platný jen po omezenou dobu, bez dalšího procesu by byl, po jejím uplynutí, uživatel odhlášen a celý proces autorizace by začal znovu. Aby k tomu

nedošlo, dostal Client ještě obnovovací token, ten má typicky delší dobu platnosti než přístupový token a používá se pro získání nového přístupového tokenu s prodlouženou dobou platnosti. Většina serverů zároveň s tím zruší platnost všech dosavadních tokenů. Celý tento proces je graficky popsán na sekvenčním diagramu na obrázku 3.5. Diagram popisuje komunikaci mezi jednotlivými aktéry v čase. [30] [31]



Obrázek 3.5 Sekvenční diagram přihlášení uživatele pomocí OAuth 2.0. Diagram je vytvořen v programu Enterprise Architect.

Kapitola 4

Návrh a implementace

Zadání webové aplikace vychází z původní již vytvořené webové aplikace. Hlavním úkolem této práce je rozšířit aplikaci o nový model, který je složitější než dva dosavadní. Dalšími úkoly jsou zaměřené na přidání nových funkcí, které se soustředí převážně na zjednodušení práce cílovým uživatelům a z původní aplikace tak vytvoří plnohodnotné řešení připravené k použití na produkčních projektech. Tato kapitola popisuje kroky k dosažení tohoto cíle. Jednotlivé sekce se týkají refaktorizace zdrojového kódu, změn v databázi, přidání nového modelu a dalších.

4.1 Stromy poruch

Největší implementačním rozšířením aplikace je přidání nového modelu pro stromy poruch. Oproti dosavadním se jedná o komplexnější a také nejpoužívanější model. Aplikace byla navržena tak, aby přidáním nového modelu nebylo nutné přepisovat dosavadní implementaci. Hlavní změnou tak bude vytvoření nových tříd, které se napojí na dosavadní workflow aplikace. Vzhledem ke komplexnosti stromů poruch bude nutné do workflow přidat další kroky, aby byla FTA kompletní.

4.1.1 Bloky stromu poruch

Oproti blokovému nebo markovskému modelu se z pohledu implementace liší hlavně počtem různých bloků. Jejich počet se v literatuře může různit, některé zdroje uvádí pouze základní bloky nutné pro provedení analýzy, jiné zdroje naopak popisují několik pokročilejších bloků, které do modelu přivádějí větší komplexitu. Aplikace by měla být navržena takovým způsobem, aby v prvním kroku zvládala minimálně všechny bloky, které jsou pro FTA nutné, a zároveň ji bylo možné rozšířit o jakékoliv komplexnější bloky. Všechny bloky je možné zařadit do následujících skupin.

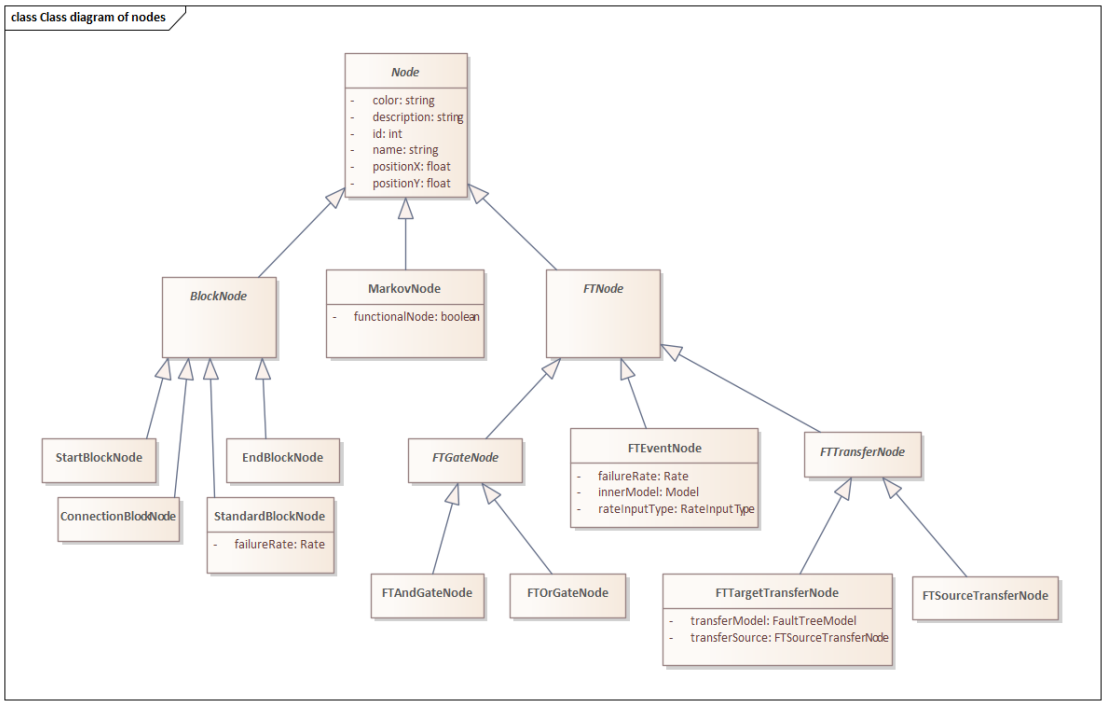
Události
: První jsou události reprezentující nějakou skutečnou událost (nejčastěji poruchu) na modelovaném systému nebo zařízení. Pokud je událost také základní (tzn. nemá žádné potomky) měla by obsahovat vstupní parametr pravděpodobnosti výskytu této události.

Hradla
: Další skupinou jsou hradla. Každé hradlo má několik vstupních hran a jednu vstupní hranu. Na typu závisí pravděpodobnost propagace poruchy ze vstupu na výstup. Základními a nejpoužívanějšími hradly jsou AND a OR, mimo tyto dvě je možné použít libovolné logické

hradlo. Jedním z v praxi používaných hradel je *Voting* hradlo, které propaguje chybu v případě, že nastane alespoň k vstupních poruch.

Sémantické bloky

Poslední skupinou jsou bloky, které nemají přímo vliv na výsledek modelu. Naopak mají spíše sémantický význam a přidávají modelu další úroveň komplexity. Za zmínku stojí přechodové bloky, které umožňují rozdělit strom poruch na několik menších a těmito bloky je propojit. Modely jsou potom přehlednější a jednoduše lze použít stejný podstrom opakovaně.



Obrázek 4.1 Diagram tříd všech bloků napříč všemi spolehlivostními modely. Diagram je vytvořen v programu Enterprise Architect.

Implementace bloků navazuje na dosavadní řešení a výsledek je v podobě diagramu tříd na obrázku 4.1. Všechny bloky v aplikace napříč všemi modely mají jednu společnou entitu *Node*. Bloky, které jsou specifické pro jeden model patří pod abstraktní třídy daného modelu (např. *MarkovNode*). Dále pak dědí konkrétní třídy, které reprezentují samotné bloky. Výhodou tohoto provedení je, že každá entita bloku může mít vlastní třídní proměnné a metody, které mají smysl pouze pro určité bloky. Například u stromů poruch může mít událost zmíněnou pravděpodobnost výskytu poruchy, což u hradel nebo sémantických bloků nedává smysl. Každá konkrétní třída musí nutně obsahovat jednoznačný typ, podle kterého klientská aplikace o jakou třídu se jedná a zobrazí uživateli správnou grafickou komponentu.

Výhoda stromu tříd je, že kdykoliv přesně vím, o jaký blok se jedná a k jakému modelu může patřit. Když uživatel vytváří nový blok, musí server požadavek validovat. Jedna z validací kontroluje právě příslušnost bloku k určitému modelu. Například chci zabránit tomu, aby uživatel přidal hradlo OR do blokového modelu. To lze jednoduše ověřit tak, že je vytvářený blok instancí abstraktní třídy *BlockNode*.

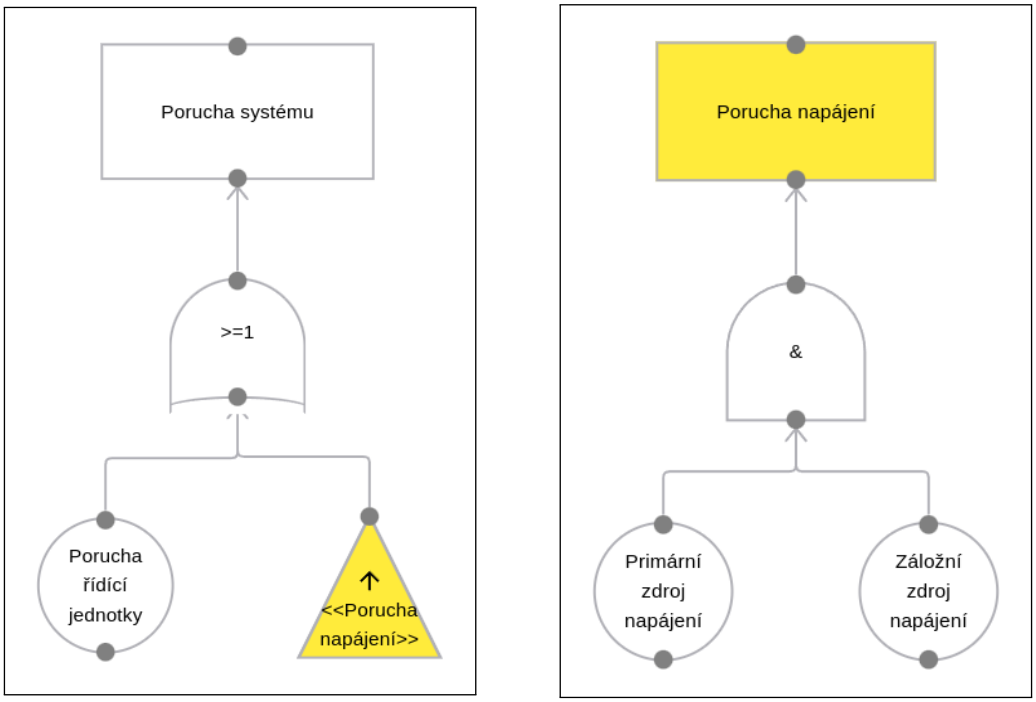
Podobně funguje validace a výpočet před exportem do Wolfram Mathematica notebooku. Pomocí stromu tříd server pozná, o kterou se jedná třídu a na základě toho může filtrovat nebo pracovat s parametry specifickými pro určité bloky.

Při implementaci jsem však narazil na nevýhodu tohoto návrhu, kterou je důležité zmínit. Jednou z vlastností entit v ORM je synchronizace PHP třídy a záznamem v databázi. V tomto případě od sebe třídy nějaké atributy dědí, což má vliv na tabulky v databázi. *Doctrine* v tomto případě nedovoluje měnit typ třídy, která je již perzistentní. Prakticky je tak nutné při každé změně třídy smazat původní záznam a vytvořit nový. Tato funkce se může hodit například pro hradla stromu poruch, pokud existuje hradlo, a cílem je změnit jeho typ bez ztráty jeho vstupních a výstupních parametrů.

Ve stromu poruch jsou kromě bloků také hrany, jsou orientované a nenesou žádná dodatečná data (narozdíl od markovského modelu). Nicméně se k nim v aplikaci přistupuje stejně jako k blokům. Také existuje hlavní třída *Edge*, ze které dědí všechny ostatní typy hran a každá konkrétní třída nese jednoznačný typ hrany. Speciální hrany se používají pouze v markovských modelech, jelikož obsahují pravděpodobnosti přechodu mezi dvěma stavy.

4.1.2 Přechodové bloky

Přechodové bloky již byly zmíněny, slouží k rozdělení modelu na menší části a jejich následné spojení. Místo tvorby jednoho velkého stromu s desítkami bloků, stačí vytvořit jeden hlavní s přechodovými bloky, které povedou na menší stromy. Umožňují i připojení stejného podstromu opakovaně, tudíž nemusí být stejný strom modelován několikrát, ale stačí pouze jeden. Přechodové bloky se značí pomocí trojúhelníků a jejich nasměrování nahoru nebo dolů určuje, zda se jedná o zdroj nebo cíl přechodu (znázorněno na obrázku 4.2).



Obrázek 4.2 Příklad použití přechodových bloků - Obrázky zobrazují oddělené modely v rámci jednoho společného projektu. První strom poruch označuje obecnou poruchu systému a druhý zobrazuje pouze poruchu napájení. Modely používají přechodový blok k logickému propojení. Při kalkulaci spolehlivostních parametrů prvního z nich vznikne kompletní výpočet obou stromů poruch. Zároveň je možné vygenerovat kalkulaci poruchy napájení samostatně bez druhého modelu. Bloky přechodu jsou na obrázcích zvýrazněny žlutou barvou. Oba obrázky byly vytvořeny exportováním modelu do obrázkového formátu bez použití externího nástroje.

Přechody je možné vytvářet v rámci jednoho modelu. Uživatel jednoduše paralelně vytvoří dva stromy, kde jeden z nich bude mít na místě kořenového elementu zdrojový přechod. Nicméně vzhledem k tomu, že má k dispozici nástroj, který udržuje více modelů v jednom projektu, není důvod mu neumožnit rozdělit stromy do samostatných modelů. Uživatel si tak sám zvolí, jestli stromy rozdělí do samostatných pojmenovaných modelů nebo je nechá v jednom společném modelu.

Pokud je strom propojen napříč více modely, je pro propojení nutné nahradit kořenovou událost přechodovým blokem. Potom není možné tento model spočítat i samostatně. To lze vylepšit přidáním možnosti vybrat nejen zdrojové přechodové bloky, ale také kořenové bloky všech ostatních stromů poruch. Při výběru musí být možné rozeznat o jakou variantu se jedná. Sestavení celého stromu může probíhat stejně.

Taková funkce vnáší do aplikace určitá omezení a pravidla. Znamená to, že nyní nejsou modely samostatné a na to je potřeba myslet při validaci i výpisu výsledků.

- Při validaci je nutné zkontrolovat existenci přechodu, na který odkazuji.
- Validace musí detekovat cyklus mezi přechodovými bloky.
- Při validaci musí server nejprve propojit všechny modely, poté rekurzivně nahradit přechodové bloky podstromem a výsledek počítat na celém získaném stromu.
- Bloky ve dvou odlišných modelech mohou být stejně pojmenovány, aplikace musí před vygenerováním upravit názvy tak, aby byli unikátní.

4.1.3 Vnořené modely

Funkce vnořených modelů je podobná přechodovým blokům. Základní bloky stromu poruch mají vstupní hodnoty pravděpodobnosti výskytu poruchy. Při tvorbě složitějších modelů je lze využít tak, aby pravděpodobnost výskytu poruchy základního bloku byla odvozena z výsledku jiného modelu. Tato funkce je zde nazvána jako vnořený model. Určitá pravidla jsou společná s přenosovými bloky. V neposlední řadě je nutné zkontrolovat existenci zdrojového modelu nebo detekovat cyklus.

Při návrhu této funkce jsem pracoval se dvěma variantami. Z pohledu uživatele se liší pouze ve vygenerovaném WM notebooku, nicméně mají konsekvence i do průběhu zpracování.

Jednotný strom - Postup by byl stejný jako u přechodových bloků. Při generování stromu by se základní blok choval stejně jako cíl přechodu a zdrojem by byl kořenový blok vybraného modelu. Ve WM notebooku by tak byl jeden společný vzorec pro oba propojené stromy poruch. Vzhledem k tomu, že by bylo velmi jednoduché stejnou funkci nahradit přechodovým blokem, nepřinesla by příliš nového a nejspíš by ani nebyla implementována.

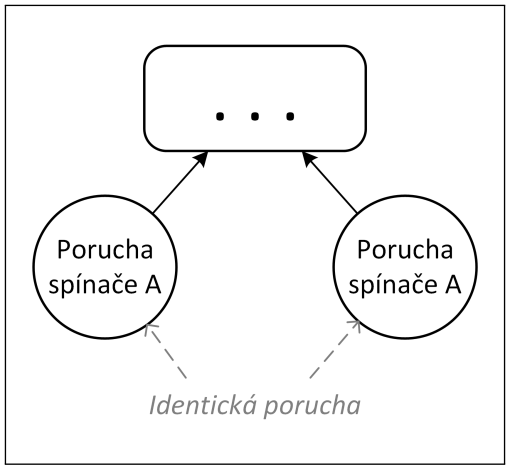
Oddělené stromy - Postup s oddělenými stromy je založen na myšlence, že do základního bloku chci v ideálním případě doplnit pouze číselnou hodnotu. Stejně tak bych to mohl udělat pomocí standardního postupu, ale v tomto případě uživatel ušetří výpočet a aktualizaci při změně zdrojového modelu. Problém je výpočet hodnoty, v tuto chvíli neprobíhají na serveru žádné matematické výpočty, pouze se strom transformuje na rovnice pro WM. Aplikace tedy dopředu neví, jaká hodnota vstupuje do základního bloku. Ta se spočítá až ve vygenerovaném notebooku. Nyní je potřeba vyřešit způsob, jakým spočítat více modelů v jednom notebooku a výsledky propsat do hlavního výpočtu.

Jelikož se jedná o notebook, není problém vygenerovat libovolný počet rovnic. Uživatel by měl mít stále přehled o tom, co se v daném notebooku nachází, který strom je ten hlavní a kde najde výpočty navazujících modelů. Pro tento účel lze použít určité funkce přímo WM notebooku. Hlavní model závisí na všech vnořených, ale stejná závislost může být i mezi jednotlivými

vnořenými modely (nesmí vzniknout cyklus). Aplikace jako první sestaví pořadí vnořených modelů, ve kterém jsou následně zapsány do notebooku. Všechny jsou obaleny inicializační buňkou, notebook zařídí vykonání všech inicializačních buněk před ostatními buňkami. Navíc je notebook odlišuje podbarvením, aby je uživatel lépe rozlišil. Navíc ke každému modelu aplikace vygeneruje nadpis a ve výchozím stavu celý výpočet zabalí. Pokud uživatel nepotřebuje zkoumat výpočty ve vnořených modelech, kromě nadpisu si jich ani nemusí všímat.

4.1.4 Kvalitativní analýza

Kvalitativní analýza je oproti kvantitativní méně exaktní, není možné vyjádřit kvalitu návrhu jasně porovnatelnými čísly. Liší se tedy už samotný přístup k analýze. Jedním ze způsobů může být zkoumání stromu poruch grafovými algoritmy, kterými se mohou hledat silná a slabá místa zařízení. Do těchto algoritmů mohou, ale i nemusí, být zaneseny další vstupy v pohodě spolehlivostních parametrů základních bloků. Analytik má tedy k dispozici další vrstvu dat, kterou může zkoumat a optimalizovat. Pod kvalitativní analýzu spadá také sledování závislostí mezi jednotlivými událostmi. Aby standardní strom poruch fungoval správně a výsledné parametry odpovídali realitě, musí být všechny základní bloky nezávislé. Pokud by byla tato podmínka porušena, musela by se do výpočtu tato závislost zanést. Například by se tak mohlo stát v případě znázorněném na obrázku 4.3, kde dvě základní události odpovídají stejné události v reálném prostředí, nebo by byli ovlivněny jinou událostí nezavedenou do stromu poruch.



Obrázek 4.3 Část stromu poruch s porušenou nezávislostí základních událostí. V případě, že je spínač A pouze jeden, je nezávislost porušena vždy. Pokud by se jednalo o dva stejné spínače, nezávislost by mohla být narušena v případě nějakých vnitřních vad spínače.

V analýze této práce byly zmíněny tři základní způsoby, jak lze analýzu provádět, jedním z nich byl *Common cause failures*, ten zkoumá hlavně neočekávané chyby, které mohou nastat, jako je duplikace stejné komponenty, která může mít skrytou vadu a ovlivnit tak reálný výsledek. Tyto chyby lze těžko implementovat do části, kde uživatelé vytváří spolehlivostní modely. Pro tento účel mnohem lépe poslouží dokumentační část aplikace, kde mohou uživatelé tyto potenciální problémy zmínit, diskutovat a dokumentovat. Ve většině případů se uvádí v podobě tabulky.

Z pohledu návrhu a implementace jsou mnohem zajímavější zbylé dva postupy, *minimal cut sets* resp. *minimal path sets*. V obou případech se jedná o samostatné postupy, které hledají slabá resp. silná místa stromu poruch. Podrobně rozepíši jen první z nich, protože úprava na druhý je triviálního charakteru. Pro nalezení MCS lze využít více algoritmů: *Boolean Manipulation*, *Binary Decision Diagram*, *Modularization method* a další.

Boolean Manipulation

První z algoritmů, které jsem zvažoval pro implementaci vychází z myšlenky, že každé hradlo lze reprezentovat nějakou logickou funkcí n vstupních proměnných s jedním výstupem, kde vstup tvoří základní události a ostatní hradla. Každou takovou logickou funkci je možné přepsat do logického výrazu, které se dají kombinovat, zjednodušit a dále upravovat dle pravidel logických výrazů. Algoritmus tedy všechny objekty označí logickou proměnnou a postupuje shora dolů nebo zespoda nahoru tak, aby postupně vytvářel logický výraz celého stromu poruch. Důležitým bodem algoritmu je v každém kroku (hloubce stromu) převést výraz na disjunktní normální formu (DNF). Potom každá konjunkce logického výrazu je ekvivalentní s MCS.

Tento algoritmus je poměrně jednoduchý, pouze využívá vlastnosti logických hradel a logických výrazů. Nicméně jsem se při prvním zkoumání obával především převodu logického výrazu na DNF. Takový algoritmus by jistě bylo možné naprogramovat, nicméně mi tato metoda přišla lepší převážně pro manuální analýzu, než pro integraci do aplikace.

Binary Decision Diagrams (BDD)

Další možností získání MCS je převedení stromu poruch do BDD. Definován je jako orientovaný acyklický graf reprezentující logickou funkci $f : x_1, x_2, ..., x_n - > \ 0, 1$. Všechny vnitřní uzly grafu jsou označeny proměnnou x_i, který reprezentuje i-tou základní událost. Mají vždy přesně dva potomky, kde levý potomek reprezentuje proměnnou $x_i = 0$, naopak pravý potomek $x_i = 1$. Listy BDD jsou označeny nejsou označeny proměnnými, ale přímo hodnotami 0 nebo 1. Pro nalezení všech MCS stačí pouze najít všechny listy, které jsou rovny 1 a proměnné po cestě k vrcholu stromu tvoří vždy jednu množinu. Naopak *minimal path sets* (MPS) lze získat stejným způsobem, akorát místo hodnoty 1 je použita hodnota 0. Tímto postupem vyjdou vždy stejné výsledky, jelikož záleží na pořadí proměnných při tvorbě BDD. Z toho důvodu je přidán poslední krok, který vrátí jen takové množiny, které neobsahují jiné, menší nalezené množiny.

V popisu algoritmu je vynechána jedna důležitá část a tou je algoritmus pro převod stromu poruch na BDD. Znovu je více způsobů, ale pro implementaci byla zvolena metoda „Rauchy Approach“ a to převážně pro její přímočarý a lehce pochopitelný postup. Pro sestavení BDD se nejdříve zadefinuje funkce *ite* (if-then-else): $ite(X, f_1, f_2)$, která pro $X = 1$ vrací funkci f_1 a pro $X = 0$ funkci f_2. Tato funkce se používá rekurzivně podle stavby stromu. Hradla zde hrají roli operátoru mezi dvěma *ite* funkcemi: $ite(X, f_1, f_2) \ < OP > \ ite(X, g_1, g_2)$, kde se $< OP >$ nahradí AND nebo OR.

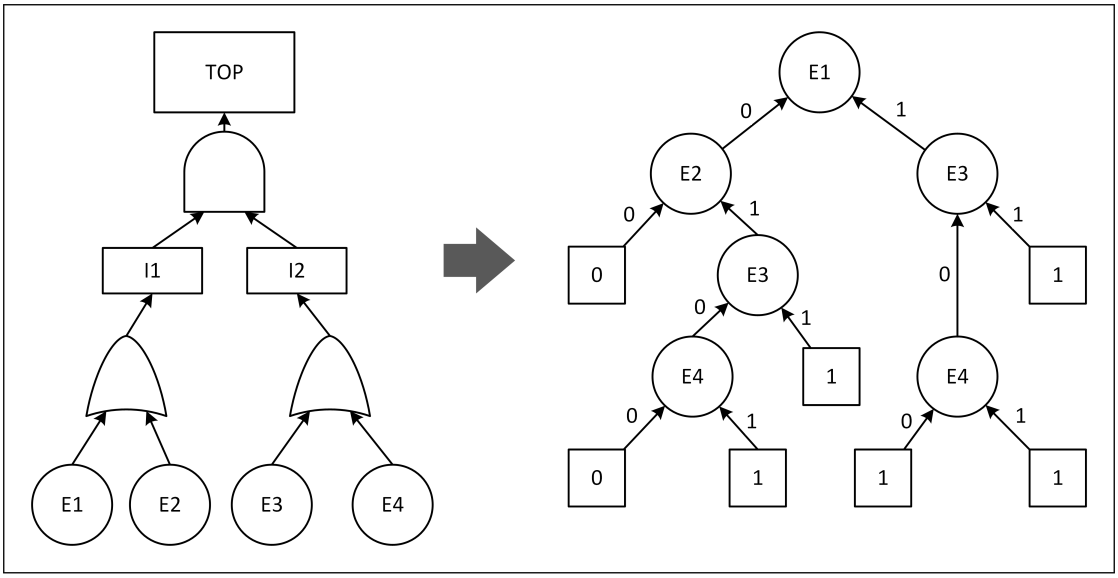
Na obrázku 4.4 je ukázán konkrétní příklad. Strom obsahuje čtyři základní události, dvě hradla OR vedoucí do událostí *I1* a *I2*. Vrchol stromu je zakončen hradlem AND vedoucím do hlavní události označené *TOP*. Metoda převodu bude rozepsána postupně zezdola nahoru a zleva doprava.

[32]

$$
\begin{aligned}
I1 &= E1\ OR\ E2 = ite(E1, 1, 0)\ OR\ ite(E2, 1, 0) \\
&= ite(E1, 1, ite(E2, 1, 0)) \\
I2 &= E3\ OR\ E4 = ite(E3, 1, 0)\ OR\ ite(E4, 1, 0) \\
&= ite(E3, 1, ite(E4, 1, 0)) \\
\mathbf{TOP} &= I1\ AND\ I2 = ite(E1, 1, ite(E2, 1, 0))\ AND\ ite(E3, 1, ite(E4, 1, 0)) \\
&= \mathbf{ite(E1, ite(E3, 1, ite(E4, 1, 0)), ite(E2, 1, 0))}
\end{aligned}
\tag{4.1}
$$

Z rovnice pro *TOP*, se bude odvíjet strom pro BDD, každá funkce $ite(x_i, f_1, f_2)$ značí jeden uzel x_i a funkce f_1 je levý podstrom a f_2 je pravý podstrom BDD.

Metodu BDD pro nalezení MCS a MPS jsem považoval za vhodnou vzhledem k jednoduchosti na implementaci, jelikož se jedná pouze o převod stromu z jednoho tvaru na druhý a nemusí se zavádět algoritmy pro práci s logickými výrazy. [6]



Obrázek 4.4 Převod základního stromu poruch se čtyřmi základními událostmi na BDD. Pořadí událostí bylo určeno podle jejich označení (E1, E2, E3, E4). Diagram byl vytvořen pomocí programu Microsoft Visio.

4.2 Matematické výrazy

V průběhu modelování se téměř v každém typu modelu objeví situace, kdy uživatel potřebuje zapsat číselnou hodnotu. V markovském modelu například jako pravděpodobnost přechodu, v blokovém modelu jako pravděpodobnost selhání bloku, ve stromu poruch jako pravděpodobnost poruchy a nebo libovolnou další číselnou hodnotu, se kterou se počítá nebo ji třeba jen zobrazuje. Tato funkce nebyla dle analýzy vhodně navržena a uživatelé naráželi na několik problémů se zadáváním parametrů. Proto jsem se rozhodl celý tento proces od základu přepracovat. Vyhledal jsem příklady webových nástrojů, které nějakou podobnou formu zápisu umožňují, některé z nich jsou *Wolfram Alpha*, *iMathEQ* nebo *math.microsoft.com*.

Cílem je umožnit uživateli zapisovat parametry tak, aby byl co nejméně omezen. V ideálním případě by měl mít možnost zapsat jakýkoliv matematický výraz, aby nemusel před zapsáním do aplikace svá data přepočítat do specifického formátu. V tomto směru je nejjednodušší formát zápisu samostatné textové pole, kam uživatel napíše libovolný výraz. Aplikace by v tomto případě měla podporovat základní matematické operace ($+$, $-$, $*$, $/$) a funkce (logaritmus, exponenciální funkce, odmocnina, ...). V původním řešení toto přinesly proměnné, které se přiřazovali určitým elementům modelu. Výhodou bylo sdílení kontextu mezi více elementy. Proměnné by měly figurovat i v novém řešení, nejlépe jako součást matematického výrazu. Řešení uživateli umožní zadefinovat například proměnnou $\lambda = 0.005$ a poté ji sdílet ve více elementech: $A = 2 * \lambda$ a $B = \lambda/2$.

Úkolem aplikace je poté zvolit ideální formát zápisu a pomocné funkce, které uživateli usnadní zápis výrazu. Na dříve zmíněných stránkách pro zápis výrazů jsem pozoroval více typů zápisu,

od jednoduchého textu, který lze zapsat do klasického textového pole, až po komplexní řešení s grafickými prvky, jako je integrál, složené zlomky, odmocniny a další. S tím souvisí otázka interpretace a parsování výrazů. Nejprve jsem se rozhodl vyzkoušet nějaká existující řešení, která by bylo možné k tomuto účelu využít.

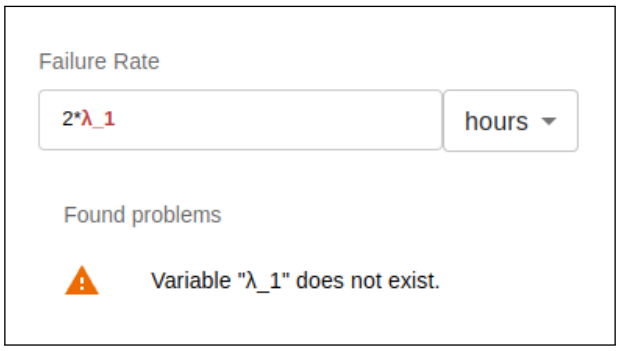
Math Parser (mossadal/math-parser) je první knihovna, kterou jsem zvažoval použít. Při použití s jednoduchými matematickými výrazy působila velmi dobře a postupně jsem ji integroval do aplikace. Výhodou knihovny je převod do syntaktického stromu, který je možné před vyhodnocením upravovat a umožňuje tak různá rozšíření knihovny o nové funkce. Na první pohled splňuje i požadavek na proměnné. Problém nastal ve chvíli, kdy jsem chtěl pracovat s proměnnými obsahující více, než jeden znak. Knihovna tuto funkci implicitně neumí a nepodařilo se mi ji dostatečně rozšířit. Vzhledem k tomu, že proměnné zadávají uživatelé, jedná se o příliš velké omezení.

Math Executor (nxp/math-executor) je alternativou, která si stále udržuje podporu. Matematický výraz nejprve rozdělí na tokeny jako čísla, operátory, funkce nebo proměnné. Operátorů a funkcí umí implicitně více než 60 a poskytuje jednoduché rozhraní pro integrací vlastní operátorů i funkcí. Problém s proměnnými, který měla předchozí knihovna zde není problém, proto jsem ji integroval do aplikace. Při vývoji jsem i přesto narazil na nepříjemnost, kterou knihovna nezvládla. Jednalo se o případ, kdy jsem zadával název proměnné písmenem z řecké abecedy. Tokenizer, který knihovna používá pro převod řetězce na posloupnost tokenů nepočítal s písmeny řecké ani české abecedy. Pro opravu této chyby bylo nutné rozšířit implementaci třídy Tokenizer a přidat podporu pro tyto znaky. Pro implementaci jsem tedy zvolil tuto knihovnu.

4.2.1 Validace matematického výrazu

Matematický výraz aplikace dokáže zpracovat, nyní se musí implementace dostat na klientské straně k uživateli. Po zadání výrazu do textového pole musí proběhnout validace. Tu může obvykle provést klientská i serverová strana. Na straně serveru dochází k parsování výrazu, tudíž je snazší přesunout veškerou validaci tam.

Formát zadávání výrazu nemusí být pro všechny uživatele jasný, klientská aplikace by měla v nějaké formě uživatele informovat o pravidlech výrazu, jako jsou dostupné operátory nebo zápis funkcí. Pokud je výraz zadán chybně, aplikace by neměla bránit v uložení a propagaci změny ostatním uživatelům, místo toho ji uloží a provede validaci. Její výstup by neměl být pouze logická hodnota, ale podrobnější popis toho, jaké chyby výraz obsahuje.



Obrázek 4.5 Varování v případě, že se ve výrazu objeví nedefinovaná proměnná. V tomto případě se jednalo o proměnnou λ_1, která nemá přiřazenou žádnou hodnotu.

Validační hlášky jsou rozděleny na 2 kategorie, chyby a varování. Vážnější kategorie značí, že není možné výraz zpracovat a při výpočtu vrátí aplikace chybu. Varování značí, že se nejedná

o chybu při parsování výrazu, ale je nalezen nějaký problém související s výrazem. Tato situace je znázorněna na obrázku 4.5. Některé hlášky se týkají pouze určité části výrazu, například nedefinovaná proměnná se týká jen názvu proměnné. Takové chyby a varování jsou označeny přímo ve výrazu a uživatel může lépe identifikovat problém a opravit jej. Pomocí standardního textového pole není možné efektivně této funkce docílit. Rozhodl jsem se proto využít knihovnu *Quill.js*, která přidává textové pole s rozšířenými funkcemi. Text uvnitř pole zobrazuje pomocí HTML schématu, díky tomu lze libovolně upravovat pomocí kaskádových stylů. Volba právě této knihovny byla jednoduchá, jelikož je stejná knihovna použita pro tvorbu dokumentace a v aplikaci už je integrována.

4.2.2 Převod do externího formátu

V aplikaci už může uživatel matematický výraz zadat, validovat a uložit a dokonce převést na rovnice pro FTA. Jako další krok je potřeba zobrazit finální výsledek rovnic. Tento krok neprovádí přímo aplikace, ale Wolfram Mathematica. Uložený výraz a sestavené rovnice jsou v takovém formátu, aby byly použitelné pro *Math Executor*. To se pochopitelně liší od formátu, který přebírá Wolfram Mathematica. Proto nelze rovnice čistě zapsat do notebooku, ale nejprve provést konverzi. Za tímto účelem jsem vytvořil třídu *WolframScriptTokenizer*, která přijme parsované tokeny z matematického výrazu a převede je do formátu pro WM notebook. Zajímavostí je, že WM používá vlastní formát pro speciální znaky ve tvaru *\[Symbol]*. Například všechny písmena řecké nebo české abecedy musejí být specificky překonvertovány a aplikace opravdu obsahuje tabulku speciálních znaků.

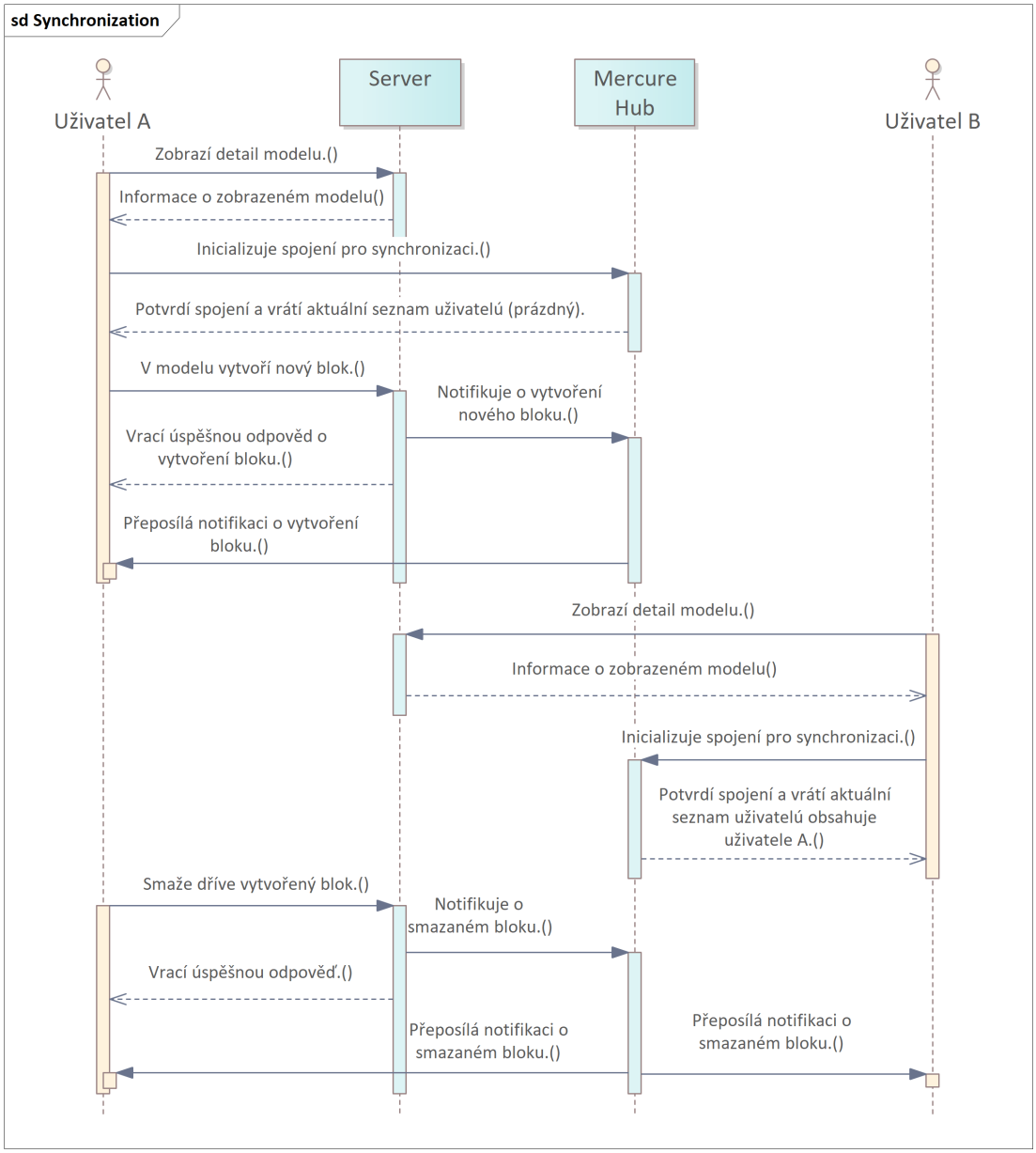
4.3 Komunikace mezi serverem a klientem

V průběhu používání aplikace musí posílat klientská část požadavky na server. Ten všechna data validuje, uloží do databáze a vrátí nově vytvořená či upravená data zpět v odpovědi. Takový požadavek může trvat několik set milisekund a během této doby čeká klient na výsledek akce. Mnoho webových aplikacích v tomto mezidobí zobrazí načítání a počkat na odpověď. Ostatně i v této aplikaci je to v mnoha případech využito. Při práci s modelem je to však jinak. Pokud klient například posune nějaký blok modelu, pak se před odesláním liší data v prohlížeči a na serveru, potažmo v databázi. Pokud by aplikace čekala na výsledek, uživatel by posunul blok, ten by se vrátil a po obdržení odpovědi znovu posunul. Aplikace by se stala nepoužitelnou. Z toho důvodu si musí data udržovat lokálně a se serverem se v průběhu používání synchronizuje.

Dosavadní řešení spočívalo v tom, že funkce, která vykonávala akci (např. posun bloku) dostala nové i staré souřadnice. Aplikace nejprve nastavila nový stav bloku lokálně a odeslala požadavek na server. Pokud byl požadavek úspěšný, znovu aktualizovala stav bloku podle souřadnic v odpovědi. V případě neúspěšného požadavku vrátila blok do původního stavu.

Při větším testování se zjistilo, že tento postup není špatně, ale skrývá jeden nepříjemný problém. Pokud totiž někdo v aplikaci provádí změny rychle za sebou, rychleji než přijde zpět odpověď serveru, tak bloky zvláštně přeskakují z jednoho místa na druhé. Důvodem byla aktualizace po přijetí odpovědi. Pokud uživatel přesunul blok z bodu A do bodu B, blok se přesunul a odeslal požadavek. Před přijetí odpovědi byl blok přesunut do bodu C a blok se opět přesunul a odeslal se požadavek. Poté se vrátila první odpověď a proběhla aktualizace na bod B a následně se vrátila druhá odpověď a nastal přesun zpět na bod C. Proto nastalo zmíněné probliknutí bloku.

Pozdní aktualizace po přijetí odpovědi má smysl ve chvíli, kdy server doplní data, která klient nemá, bohužel ale způsobuje toto chování. Z toho důvodu jsem aktualizaci odebral a některé dotazy upravil tak, aby na serveru nedocházelo ke změněn dat, která mění klient. Kompletní průběh komunikace mezi klientskou aplikací, serverem a *Mercure Hub* je popsán formou sekvenčního diagramu na obrázku 4.6. Ten popisuje průběh komunikace dvou klientů a serveru v čase.



Obrázek 4.6 Sekvenční diagram, zaznamenávající inicializaci a průběh komunikace mezi klientem, serverem a Mercure Hub při spolehlivostní analýze. Diagram byl vytvořen v programu Enterprise Architect.

4.4 Historie provedených změn

V dnešní době existuje nespočet aplikací, které nám usnadňují psaní textu, tvorbu designu nebo grafů a mnoho dalších pracovních aktivit. Při všech těchto činnostech uživatel někdy udělá chybu. Jak nejlépe chybu opravit, než se vrátit do chvíle, kdy vznikla. Většina zmíněných nástrojů přesně s tímto počítá a proto mají funkce, které vrátí nebo znovu provedou poslední úpravy. Při testování se tato funkce jevila jako velmi postrádaná. Rozhodl jsem se ji tedy přidat.

Historie funguje celkem jednoduše, pokaždé, když uživatel provede nějakou akci, tak se uloží.

Poté ji uživatel vrátí a to může opakovat až do první provedené akce. Naopak, pokud se vrátil až příliš daleko, může vrácené akce znovu provést. V případě, že po vrácení provede novou akci, smaže se zásobník dopředných akcí, jelikož už nejsou validní. Zde dochází ke komplikaci, aplikace není pouze lokální, přistupovat k ní mohou další uživatelé a klient tak nemá kontrolu nad každou provedenou akcí.

4.4.1 Analýza řešení

V tuto chvíli jsem si nebyl jistý, jak takovou funkci implementovat, rozhodl jsem se tedy analyzovat řešení jiných aplikací, které mají historii a zároveň jsou přístupné pro více uživatelů ve stejnou chvíli. Našel jsem aplikace Google Docs a Whimsical. První jmenovaná sice nezobrazuje modely, ale to v tomto případě není příliš důležité. Zkoušel jsem s nimi pracovat z pohledu více uživatelů a všiml jsem si společného řešení. Vracet je možné pouze lokální změny.

Pro představu, uvádím dva uživatele *A* a *B*, kteří pracují na stejném Google dokumentu. *A* napíše slovo *Hello* a poté chvíli připíše *World*. Udělal tak dvě akce, každá na jedno slovo. Pokud by vrátil akci (pomocí *Ctrl+Z*), smazalo by se slovo *World*. Místo toho *B* přidal slovo *Ahoj*. Pokud by nyní uživatel *A* vrátil akci, pořád by smazal pouze slovo *World*. Do jeho historie se uložily pouze jeho úpravy. Co když jeho úpravy už nejsou validní? Uživatel *B* smaže slovo *World*. *A* při vrácení úpravy nemá co smazat, protože je slovo už smazané, dokument tak jeho úpravu přeskočí a smaže rovnou *Hello*. Whimsical funguje úplně stejně jako Google Docs. Na tomto příkladu je hezky vidět, jak přesně historie funguje, uživatelé jsou na tento systém zvyklí a proto jsem se ji rozhodl implementovat stejně.

4.4.2 Implementace prvního řešení

Implementačně se tato funkce týká pouze klientské strany aplikace. Prvním krokem byla identifikace akcí, na které chci tuto funkci aplikovat. Jednoznačně se musí jednat o akce, které primárně ovlivňují vzhled modelu, tedy cokoliv co se zobrazí v grafické podobě musí mít zpětnou akci. Dále jsem se rozhodl přidat i proměnné, tudíž se jedná primárně o entity *ModelObject* (*Node*, *Edge*) a *Variable*.

Ke způsobu implementace jsem měl v první fázi různé možnosti. První možností je ukládání stavu celého modelu. Před každou akcí by se uložil aktuální stav, tedy bloky, hrany a proměnné. Pokud by se chtěl uživatel vrátit, tak by se pouze načetl uložený stav. V tomto způsobu jsem viděl komplikace se synchronizací a změnami proběhlými na serveru. Pokud by byl uložen celý stav, zpětná aktualizace by mohla změnit data upravená jinými uživateli. Další možností je implementovat API tak, aby pro každou akci existovala zpětná akce. Jelikož se úprava objevuje pouze lokálně, server je nemusí nijak evidovat. Klient tedy řídí celý proces tak, že před každým požadavkem uloží starý a nový stav objektu, společně s akcí, kterou provádí. Akci provede lokálně, počká na odpověď a potom ji uloží do historie.

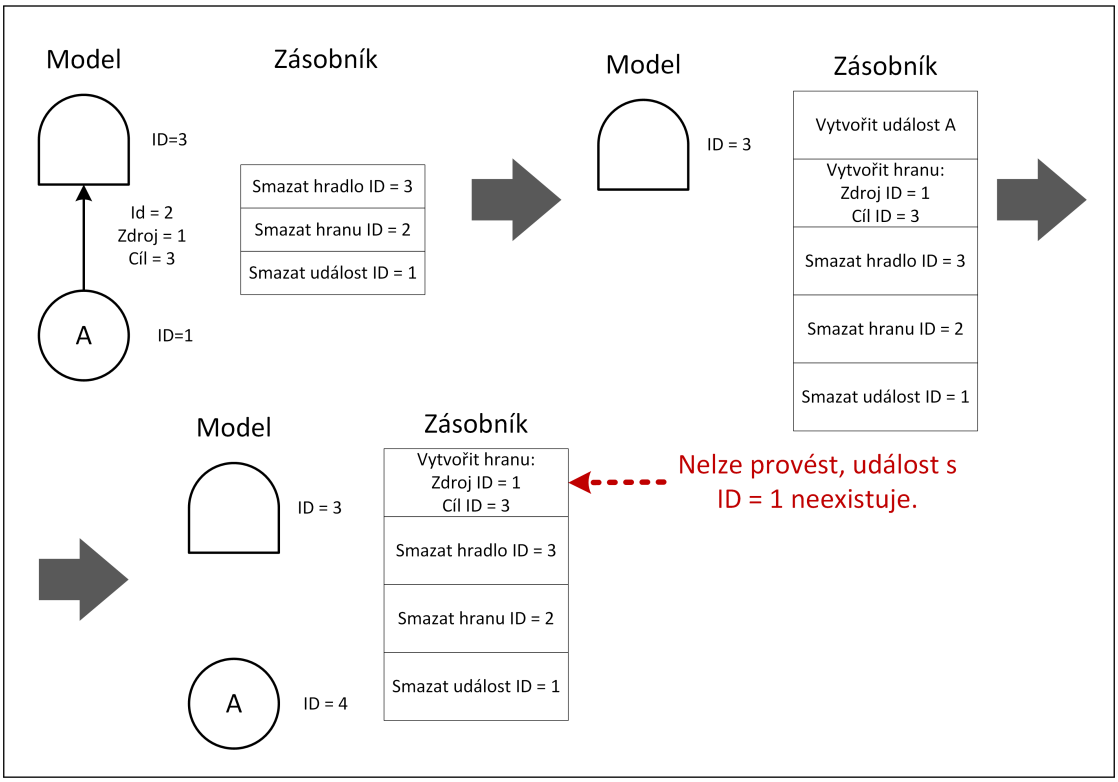
Implementace historie funguje tak, že aplikace má dva zásobníky, jeden pro krok zpět (*undo*) a druhý pro krok dopředu (*redo*). Pokaždé, když uživatel provede úspěšnou akci, přidá se opačná akce do zásobníků *undo* a nakonec se vyprázdní zásobník *redo*. Po stisku *Ctrl+Z* najde nejnovější platný krok zpět, provede se a opačná akce (stejná jako původní) se uloží do zásobníku *redo*. Dopředný krok funguje přesně opačně, po spuštění (*Ctrl+Y*) se vezme akce z *redo*, spustí a opačná akce se uloží do *undo*.

4.4.3 Testování a nalezení problému

Po dokončení implementace jsem prováděl testování, vypracoval jsem jednoduché modely s dvěma připojenými uživateli a testoval funkci kroků zpět a vpřed. Přitom jsem objevil chybu, kterou jsem z počátku neviděl. Pro její popis nejdříve vysvětlím kontext situace. Každá entita v aplikaci

je identifikována číselným *id*, které je entitě přiřazeno při persistenci do databáze. Pro zobrazení potřebuje knihovna React Flow nějaký identifikátor, ten je vytvořen náhodně a následně automaticky aktualizován. Od této chvíle se všude používá stejný identifikátor. Jakmile je entitě identifikátor přiřazen, není možné ho změnit ani dopředu nastavit, jediným způsobem je vytvořit novou entitu, které bude automaticky přiřazen nový identifikátor. Nyní popíši problém na konkrétním situaci. Graficky je tento problém znázorněn na obrázku 4.7.

Je dán strom poruch, uživatel vytvoří novou základní událost *A*, hradlo AND a propojí je hranou. Událost dostane přiděleno *id=1*, hradlo *id=2* a hrana *id=3*. Informace o propojení bloků je uložena v objektu reprezentujícím hranu, zdrojový blok je tedy událost *A (id=1)* a cílový blok hradlo AND *(id=2)*. V zásobníku pro historii jsou tyto tři akce v opačném pořadí. Jako další krok smaže hranu, což přidá do zásobníku novou akci v podobě vytvoření hrany z bloku *id=1* do bloku *id=3*. Dále smaže také událost *A*, která v tuto chvíli nemá žádnou vstupní ani výstupní hranu a do historie se zapíše vytvoření události *A*. Vrátí poslední akci, jako první jde na řadu vytvoření událost *A*, které se přiřadí nové *id=4*. Pokračuje dalším krokem zpět, vytvoří se hrana, která má jako zdroj událost s *id=1*, což pochopitelně selže, protože takový událost neexistuje.



Obrázek 4.7 Příklad situace, ve které první implementovaná verze nefunguje. Každý krok popisuje aktuální model a zásobník pro kroky zpět.

4.4.4 Řešení problému

Dlouho jsem hledal řešení této situace, zvažoval jsem více možností včetně nastavení smazaného identifikátoru. Nakonec jsem dospěl ke složitějšímu řešení, které zároveň zjednoduší nastavování identifikátorů na straně klienta. Všechny zpětné akce jsou ukládané pouze lokálně a není potřeba je odesílat na server. Proto jsem každému objektu přidal unikátní lokální identifikátor v podobě UUID. Důvodem je nejen snazší identifikace objektů, které zatím nejsou persistentní, ale hlavně zachování kontextu v historii. Ve chvíli, kdy se objekt vrátí v odpovědi a uloží do lokální paměti,

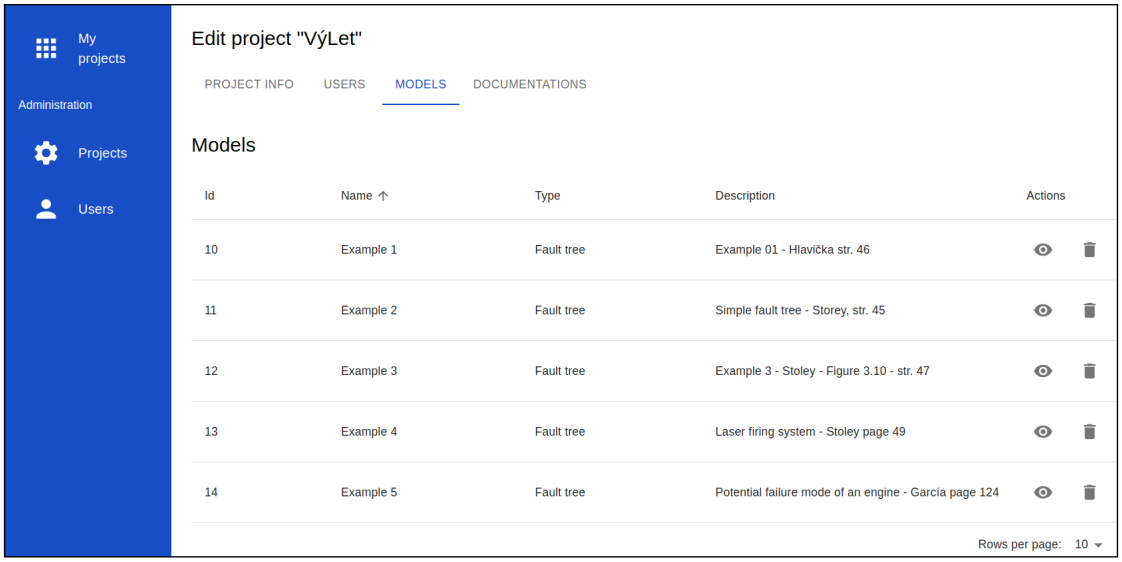
vytvoří se mu lokální identifikátor a zapíše se do tabulky společně s globálním identifikátorem. Od této chvíle se všude použije pouze lokální identifikátor. Všechny objekty, které na sebe odkazují, se budou odkazovat pouze pomocí lokálních identifikátorů. Převod zpět na globální identifikátory proběhne až ve chvíli, kdy se bude objekt posílat zpět na server. Tato vrstva je od zbytku aplikace oddělena a žádná vrstva pod ní nemusí s globálními identifikátory nijak pracovat.

Popíši stejnou situaci, která způsobovala problémy s novým řešením. Uživatel vytvoří událost *A* s *id=1* a *localId=l1*, hradlo AND s *id=2* a *localId=l2* a nakonec hranu s *id=3* a *localId=l3*. Zdrojem hrany je *localId=l1* a cílem *localId=l2*. Smaže hranu, událost *A* a tu vrátí. V historii byla uložena událost s *localId=l1*, na serveru se znovu vytvořila a bylo jí přiřazeno *id=4*, v tuto chvíli se aktualizuje tabulka lokálních identifikátorů a *localId=l1* koresponduje s *id=4*. Dalším krokem pro vrácení je hrana, jejíž zdroj a cíl je popsán pomocí lokálních identifikátorů, aplikace nalezne globální identifikátory, odešle je v požadavku a server ho tak přijme.

4.5 Administrace

V původní aplikaci chyběla možnost správy aplikace univerzálním administrátorem. Při testování se to projevovalo zbytečnými prodlevami při přidávání uživatelů k projektům a nemožností správy projektů bez přihlašovacích údajů konkrétních uživatelů. Jedním z funkčních požadavků je docílit takového stavu, kdy bude mít správce systému možnost najít libovolného uživatele nebo projekt a spravovat ho tak, jak může samotný autor.

Nejprve je nutné takového správce vytvořit a zařídit příslušné oprávnění. K tomuto účelu není příliš dobré mít uživatele, který se autentizuje přes třetí stranu, pokud by tato možnost přihlášení vypadla, administrátor by neměl přístup do systému, což se nesmí stát. Proto by měl být i nadále přístupný přihlašovací a registrační formulář, minimálně pro tento typ uživatelů. Účet jednoho správce by měl existovat již při prvním nasazení aplikace, aby bylo možné systém spravovat ještě před registrací prvního uživatele.



Obrázek 4.8 Administrační sekce projektu s názvem *VýLet* - Správce vidí v tomto přehledu projektu všechny jeho modely, může si je zobrazit a upravit nebo smazat. V jednotlivých záložkách si může zobrazit podobné informace o dokumentacích, přiřazených uživatelích nebo projektu.

Správcovský účet může být pojat více způsoby, zde jsem se rozhodl vytvořit klasický účet a přidat mu speciální uživatelskou roli označenou jako *ROLE_ADMIN*. Server tedy po autentizaci velmi jednoduše pozná, že se jedná o správce. Vzhledem k tomu je nutné povolit přístup

na všechny akce, které by měl vykonávat. Například se jedná o zobrazení seznamu všech uživatelů a projektů a samozřejmě CRUD (Create, Read, Update, Delete) operace nad těmito entitami.

Server je tedy na administraci připraven, nyní zbývá klientská část aplikace. Ta pracovala pouze s jedním typem uživatele, proto nebylo příliš nutné dělat dodatečné ověřování přístupu vzhledem k oprávnění. To bylo přidáno tak, že ve chvíli, kdy zobrazuje nějaké komponenty určené pro správce, nejprve se doptá na svou roli a poté je zobrazí či nikoliv. Využívá se to například při zobrazení administrační sekce v menu.

Samotná sekce pro správu webu je oddělena, nekombinují se tedy administrační a uživatelské stránky. Ty jsou zaměřeny hlavně na zobrazení seznamu uživatelů a projektů. Na obrázku 4.8 je detail administrace konkrétního projektu. V případě projektů má správce možnost projekt vytvořit, upravit nebo smazat. Dále může u projektu řídit práva pro přístup k projektu, přiřazovat nebo odebírat uživatele a dokonce změnit stupeň oprávnění. Co se týče modelů, tak k nim má plný přístup, což se týká i detailu a generování WM notebooku.

4.6 Autentizace účtem FIT ČVUT v Praze

Plánovaným případem použití aplikace je výuka na FIT ČVUT v Praze v předmětu *Testování a Spolehlivost*. Aplikace bude dostupná studentům na vyzkoušení praktické analýzy spolehlivostních modelů. Pro takové použití vyžaduje fakulta možnost přihlášení pomocí fakultního účtu. Nyní aplikace obsahuje jen standardní přihlašovací a registrační formulář a proto musí být autentizace implementována.

Fakulta umožňuje implementaci fakultního účtu do aplikace třetích stran pomocí OAuth služby. Pro napojení na fakultní službu je nejprve nutné svoji aplikaci na službě zaregistrovat. K tomu slouží stránka *https://auth.fit.cvut.cz/manager/index.jsf*, pro registraci stačí zadat název projektu a v něm zaregistrovat konkrétní instanci aplikace s URI pro přesměrování. Fakultní služba vygeneruje identifikátor a klíč aplikace, ty budou později použity pro navázání spojení. V mém případě jsem vytvořil aplikace dvě, produkční a vývojovou.

Po registraci OAuth aplikace zbývá napojení webové aplikace. K napojení je využit balíček knihoven *knpuniversity/oauth2-client-bundle* připraven přímo pro Symfony. Integrace se rozpadla na několik kroků:

Konfigurace klíčů v Symfony - Do konfiguračního souboru se pomocí proměnných prostředí přidají klíče získané z fakultní služby.

Vytvoření OAuth uživatele - Knihovna přidává rozhraní *ResourceOwnerInterface*, které obsahuje uživatelská data získaná z OAuth služby.

Napojení na OAuth službu - Knihovna obsahuje *AbstractProvider* třídu, do které se pomocí připravených metod zadají informace používané při OAuth komunikaci. Například URI pro inicializaci autorizace, URI pro přístup k tokenu nebo scope aplikace. Všechny tyto informace jsou popsány v dokumentaci fakultní OAuth služby.

Rozhraní pro klientskou část aplikace - Přidat napojení integrace na klientskou aplikaci. Klient při přihlášení posílá požadavek na server, který následně spustí proces autentizace.

4.7 Export do Wolfram Mathematica notebooku

Při provádění spolehlivostní analýzy webovou aplikací je kalkulace spolehlivostních parametrů aktuálního modelu jednou z klíčových funkcí celého procesu. Veškeré výsledky kvantitativní analýzy se nyní nachází převážně ve Wolfram Mathematica notebooku. To s sebou nese několik problémů. Wolfram Mathematica je externí software, takže uživatel ji musí mít nainstalovanou a připravenou k použití. Do toho spadá i aktivace, jelikož je WM licencovaný software, nelze

ho spustit bez předchozí aktivace. Tato skutečnost vytváří vyšší nároky na klienta webové aplikace.

Proces generování funguje v původní verzi aplikace tak, že uživatel nejprve vytvoří model, zadá všechny hodnoty a nechá aplikaci vygenerovat finální notebook. Na serveru se připraví šablona, doplní se hodnoty z modelu a vygeneruje se skript spustitelný nástrojem WolframScript. Sestavený script se spustí a jedním z posledních příkazů je uložení WM notebooku, který se následně vrátí klientovi v odpovědi a umožní tak stažení souboru do počítače. Při generování notebooku se objevil problém s licencí. WolframScript dokáže spustit klasický kód, nicméně pro vygenerování WM notebooku potřebuje mít takzvaný *FrontEnd*. Čímž je myšlena instance Wolfram Mathematica. Vzhledem k tomu, že celý proces se odehrává na serveru, musí tam být Wolfram Mathematica nainstalována, včetně licence.

Při implementaci bylo možné pomocí školního autentizačního klíče aktivovat WM na serveru a umožnit tak automaticky generovat kompetní WM notebooky. Později se změnila forma autentizace na SSO (Single Sign-on), které funguje pouze v grafickém prostředí a není tak možné provést aktivaci.

Tuto formu generování je tedy nutné změnit. Vzhledem k tomu, že každý klient musí mít pro zobrazení notebooku svoji instanci WM aktivovanou, rozhodl jsem jsem se přenést generování na stranu klienta. Až po tvorbu skriptu ze šablony funguje postup naprosto stejně. Po vytvoření skriptu se však nespouští WolframScript, namísto toho se vrátí rovnou klientovi, který provede generování lokálně a vygeneruje WM notebook.

4.8 Změny v relačním modelu

Se změnami a rozšířením aplikace se pojí i nutná změna pohledu na schéma dat v databázi. V této části budou popsány hlavní změny schématu a související témata, jako například migrace dat v produkčním prostředí.

Na relační schéma mají největší vliv dva zpracované požadavky, kterými jsou: nahrazení parametrů matematickými výrazy a přidání stromů poruch. V původním relačním modelu se veškeré parametry ukládali do tabulky *Parameter*, ta se následně relací propojila s hranou (*Edge*) nebo blokem (*Node*). Pro matematické výrazy je zbytečné udržovat oddělenou tabulku, protože je lze uložit pouze ve formě textu a nemají žádné relace na ostatní tabulky. Tabulka *Parameter* bude tedy odebrána a nahrazena sloupcem v každé tabulce, která obsahovala relaci.

To souvisí s další větší změnou v databázi. Při rozšiřování o stromy poruch přibyli další sloupce do tabulky *Node*. Rozhodl jsem se pojmout tento problém jiným způsobem a vytvořil jsem obecnou tabulku (entitu) pro všechny zobrazitelné prvky, pojmenovanou *ModelObject*. Z této entity následně dědí původní třídy *Node*, *Edge* a jejich další podtřídy. Pro převod dědičnosti do relačního modelu se v původní aplikaci vždy požívala varianta *Single Table Inheritance* (STI). Celý strom podtříd byl tak převeden do jedné společné tabulky, které obsahovala sloupce všech podtříd. S přibývajícími sloupci se zvyšuje počet prázdných sloupců, protože pro každý řádek mají smysl pouze některé sloupce. Proto jsem se rozhodl pro změnu a převod na *Class Table Inheritance*, každá entita tak bude mít vlastní tabulku, kde související řádky mají stejný primární klíč a každá tabulka obsahuje pouze vlastní sloupce.

Přidáním stromů poruch se značně rozšíří dědičnost entity *Node* o hradla, události a přechody. Všechny relace na tabulku *Parameter* byly nahrazeny třemi sloupci, kde jeden je matematický výraz, další jednotka a poslední je pole chyb v matematickém výrazu. Další změny už nebyli zapotřebí a ostatní tabulky relačního modelu zůstaly stejné.

Kapitola 5
Testování a nasazení

Tato kapitola obsahuje nejprve stručný popis testování určité části aplikace. Dále je blíže popsán průběh konkrétního nasazení na fakultní server, na kterém probíhalo testování původní verze aplikace. A nakonec jsou popsány všechny další změny, které byly pro aktualizaci aplikace na novou verzi nutné. Součástí testování mělo být i ověření aplikace na praktickém příkladu z praxe, nicméně se jedná o dostatečně velkou a samostatnou část práce. Nachází se tedy v samostatné následující kapitole.

5.1 Testování

Průběh implementace probíhá vždy po určitých částech. Každá implementovaná část aplikace je odlišně složitá. Ne vždy je také na první pohled jasné, zda provedená změna nebo nová funkce funguje tak, jak se od ní očekává. Z toho důvodu je pro takové části aplikace vhodné připravit testy, aby bylo možné ověřit, zda funguje správně či ne. Aplikace se dělí na serverovou a klientskou část, kde každá z nich má svoje specifické funkce, které je vhodné testovat. V této podkapitole jsou představeny formy testů a jejich využití na konkrétních příkladech aplikace.

5.1.1 Unit testy

Základní kategorií testů, které se používají při vývoji nejen webových aplikací jsou unit testy. Jedná se o proces, kde je testována nejmenší část kódu. Například se jedná o vybranou funkci nebo třídu. Cílem testu je izolovat několik malých částí aplikace a ověřit, zda fungují podle očekávání. Tyto testy je možné psát jak před, tak po implementaci testované funkce. [33]

K tomuto účelu se na serverové části používá knihovna *PHPUnit* [34]. Standardní knihovna pro testování aplikací v jazyce PHP. Její použití je velmi jednoduché, nejprve se knihovna nainstaluje standardním postupem pomocí nástroje *Composer*. V adresáři `tests` se vytvoří nová třída, která dědí z knihovní třídy *TestCase*. Taková třída může obsahovat libovolný počet metod, které testují kód, jediným kritériem je, že název metody musí být ve formátu *test¡Název Metody¿*, aby se odlišili testovací metody od ostatních. Pro spuštění testu pak stačí jen spustit příkaz `php bin/phpunit [Název třídy]` a test se provede.

Konkrétně jsou v aplikaci takto otestovány třídy, které představují důležitou aplikační logiku aplikace. Unit testy tedy testují vyhodnocování matematických výrazů. Jejich validaci a správný převod mezi formáty. Aplikace by tak neměla mít chybu mezi zápisem matematického výrazu a exportem stejného výrazu do WM notebooku. Dále jsou testovány algoritmy pro převod ze stromu poruch na BDD a následně výstup kvalitativní analýzy.

5.2 Nasazení na fakultní server

Jelikož je plánováno aplikaci i do budoucna využívat pro výuku na FIT ČVUT v Praze, je nutné ji nasadit na produkční prostředí. K tomu je potřeba nejprve vybrat prostředí, na kterém může být aplikace nasazena a provozována.

5.2.1 CloudFIT

ICT oddělení fakulty provozuje a poskytuje studentům servery pro provoz projektů souvisejících se studiem. Nazývá se CloudFIT a je to také prostředí, na kterém byla spuštěna a dále testována první verze webové aplikace pro spolehlivostní analýzu. Původně se jednalo o instanci OpenNebula, kde byl ICT oddělením vytvořen server s předem domluvenými parametry. V průběhu zpracování diplomové práce došlo k migraci na webové rozhraní VMWare. [35] [36] [37]

Migrace probíhala ve spolupráci s ICT oddělením, nejprve byl přesunuta veškerá data související s aplikací na nový server. Přesun se týkal především souborů aplikace (git repozitáře) a dat v databázi. V aplikaci bylo nutné změnit některé konfigurační proměnné prostředí a nakonec změnit DNS tak, aby URI - vratidan.stud.fit.cvut.cz - odkazovala na nový server.

5.2.2 Migrace dat

Další aktualizace proběhla až s aktualizací na novou verzi aplikace a tím přidáním několika nových funkcí popsaných v předchozích kapitolách. Změny přinesly několik změn, na které je nutné dát pozor při aktualizaci tak, aby aplikace fungovala správně a například neovlivnila dosavadní data.

Jedním z největších problémů byla migrace existujících dat. Databázové schéma muselo projít značně velkou úpravou, která se týkala všech elementů modelu (bloky i hrany) a parametrů. Parametry se v nové verzi změnily na matematické výrazy, proto jsem je pouze odstranil a všem blokům nastavil prázdné výrazy. Vstupní hodnoty tedy budou muset být zadány v nové formě.

Pro elementy modelu jsem nechtěl postupovat stejně, jelikož by se tím kompletně ztratily všechny rozpracované modely. Postupoval jsem tedy tak, že jsem získal všechny elementy z databáze a podle struktury tříd uložil data do správných tabulek. Obtížné bylo především dodržet všechna omezení cizích klíčů a synchronizace identifikátorů, jelikož jednou ze změn je spojení bloků a hran pod stejnou tabulku.

5.2.3 Aktualizace proměnných prostředí

Webová aplikace je typicky spouštěna v různých instancích, jedna z nich může být vývojová na lokálním stroji, další testovací pro širší základnu uživatelů hledajících chyby a nakonec i produkční pro použití v reálném prostředí. Tyto instance by si měli být co nejvíce podobné, v některých bodech se však lišit musí, například v napojení na testovací prostředí třetích stran nebo mají odlišné klíče pro šifrování citlivých dat. K tomu se používají proměnné prostředí. Jedná se o určité hodnoty, kterými se konfiguruje konkrétní instance, některé jsou stejné pro všechny instance, některé se liší.

V aplikaci se pro proměnné prostředí využívaní *.env* soubory. Jeden společný pro společné proměnné a jeden závislý na prostředí. Jelikož je aplikace dostupná pouze v lokálním a produkčním prostředí, používá se *.env.local* pro lokální a *.env.prod* pro produkční prostředí. Tyto dva soubory obsahují převážně závislé na prostředí, na kterém je konkrétní instance aplikace spuštěna. Mezi ně patří například: klíče pro generování JWT, napojení na *auth.fit.cvut.cz*, napojení na databázi a přístupové URI.

Kapitola 6

Ověření na praktickém příkladu

Tato kapitola volně navazuje na předchozí kapitolu, která se zabývala testováním a nasazením webové aplikace. Obsah této kapitoly je však zaměřen spíše na praktické použití; uživatelské prostředí, vypočtené výstupy, ale i celkové možnosti navržené a implementované webové aplikace jsou zde porovnávány a ověřovány s aplikacemi SHARPE a Zusim. Srovnání je popsáno na příkladu z železničního prostředí. Jedná se o modelovou situaci zabezpečeného vlakového přejezdu, cílem bylo určit střední dobu bezporuchového provozu a ukázat postup spolehlivostní analýzy. Vstupy (ať už číselné nebo formální) do této části poskytl vedoucí práce, který současně vymezil rozsah a upravil složitost analyzovaného zabezpečovacího zařízení.

6.1 Model vlakového zabezpečovacího zařízení

Na obrázku 6.1 se nachází model vlakového zabezpečovacího zařízení, jedná se o teoretický koncept železničního přejezdu se světelným signalizačním zařízením se závorami. Z důvodu velké složitosti tohoto zařízení, které se skládá z několika tisíců diskrétních součástek a které může být navrženo pro různá prostředí (dle použití konkrétního přejezdu, ale také dle národních požadavků) s různou škálou složitosti (přejezd může být řízen lokálně, na dálku či zcela autonomně) je pro začátek třeba upravit modelovou situaci tak, že bude celkový model přejezdu značně omezen a zjednodušen, aby i laik pochopil smysl funkce a byly zřejmé jednotlivé kroky uvedené v této kapitole.

Není cílem této kapitoly do detailu popisovat zabezpečovací techniku (existují různé druhy a možnosti zabezpečovacích zařízení), která se běžně používá na křížení kolejové dráhy s pozemní komunikací. Zároveň není cílem kapitoly popisovat detailní principy fungování vlakového zabezpečovacího zařízení, opět může existovat několik způsobů řízení přejezdového výstražníku, bližší informace a vysvětlení lze nalézt v těchto publikacích. [38] [39] [40]

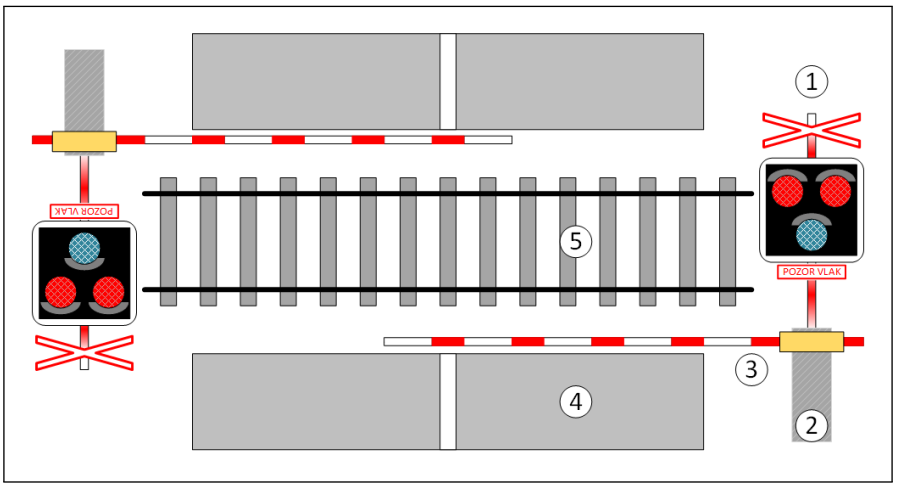
Cílem této části je demonstrovat, že implementovaná webová aplikace obstojí ve srovnání s již zavedenými nástroji SHARPE a Zusim. S vedoucím práce jsme pro toto srovnání připravili praktický příklad, ve kterém půjde o výpočet střední doby bezporuchového provozu (MTTF) a poté jednoduchou ukázku spolehlivostní analýzy. Ta bude zaměřena na kvalitativní analýzu, tedy půjde o návrh stromu poruch pomocí webové aplikace bez číselných hodnot.

Veškerá bloková schémata, spolehlivostní modely i číselné vstupy jsem připravoval společně s vedoucím práce a ty jako takové nejsou předmětem mého výstupu, ale slouží pro demonstraci možností mnou navržené a implementované webové aplikace.

6.1.1 Základní předpoklady

Jedním ze základních předpokladů je nastavení úrovně abstrakce. Funkční blok, který se může skládat z několika jednotek ale i tisíců diskrétních součástek je zde uvažován jako jedna dále nedělitelná součást (s výjimkou možnosti redundance, což bude vždy v textu uvedeno). Systém 2 ze 2 se tedy bude skládat pouze ze dvou identických bloků. Barvy a čísla bloků vzájemně korespondují s obrázky zde uvedenými, aby bylo pro čtenáře snazší se orientovat mezi obrázky v této kapitole.

Poruchy jsou uvažovány pouze takové, které jsou vyjmenovány pod jednotlivými částmi zařízení v dalších částech textu. Přejezdové zabezpečovací zařízení je zde uvažováno pouze v jednom směru, nicméně pokud se hovoří o selhání jedné části přejezdového výstražníku, předpokládá se, že přejde do bezpečného stavu (vypne se) i protistrana tohoto přejezdového zařízení.

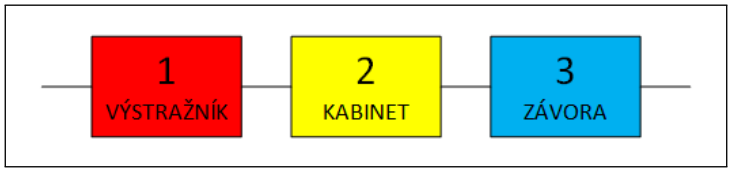


Obrázek 6.1 Modelová situace zabezpečeného přejezdu výstražným zabezpečovacím zařízením se závorami v místě křížení železniční tratě a pozemní komunikace.

Následují definice základních pojmů týkajících se přejezdového zabezpečovacího zařízení, které vychází z obrázku 6.1:

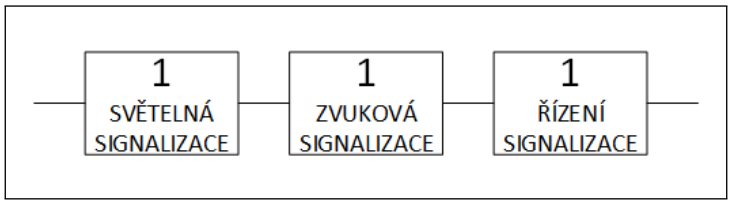
1. Přejezdový výstražník - případně jen *výstražník* obsahuje 2 výstražná červená světla a jedno bílé (pozitivní) světlo, dále obsahuje zvukovou signalizaci.
2. Kabinet – skříň na baterii s nutnou řídící elektronikou, která je situovaná v *noze výstražníku* jako součást přejezdového zabezpečovacího zařízení.
3. Závora ovládaná elektromotorem řízeným pomocí přejezdového zabezpečovacího zařízení, které se nachází v kabinetu.
4. Silnice – pozemní komunikace pro motorová i nemotorová vozidla.
5. Kolejiště – železniční trať.

Světelné přejezdové zařízení se závorami se nachází v každém směru provozu pozemní komunikace. Každé zabezpečovací zařízení má svou světelnou, zvukovou signalizaci a závoru. Obecně se tedy skládá z těchto tří bloků, které ilustruje obrázek 6.2. Porucha jakéhokoliv z těchto tří bloků by znamenala poruchu celého systému (jedná se tedy o klasický sériový model).



Obrázek 6.2 Základní části přejezdového zabezpečovacího zařízení: výstražník, kabinet a závora. Čísla jednotlivých bloků odpovídají vyobrazené modelové situaci.

Tyto tři bloky lze dále rozdělit na menší části podle funkčních prvků, které obsahují. Toto rozdělení je patrné na obrázcích 6.3, 6.4 a 6.5. Pozorný čtenář si zde jistě povšiml, že se jedná o velmi zjednodušené rozdělení zabezpečovacího zařízení, a to především způsobem, který by měl být pochopitelný i pro laika. Nicméně, pro účely demonstrace spolehlivostní analýzy by měla být tato úroveň detailů dostatečná.



Obrázek 6.3 Blokový model základních částí výstražníku.

Obrázek 6.3 popisuje rozdělení výstražníku na 3 základní bloky: světelnou a zvukovou signalizaci a řízení této signalizace.

Blok *světelná signalizace* tedy obsahuje součásti jako jsou dvě červená výstražná světla a bílé (pozitivní) světlo.

Blok *zvuková signalizace* je dále nedělitelná součást.

Blok *řízení signalizace* obsahuje dvě stejné redundantní části, které zajišťují řízení světelné a zvukové signalizace, tak jak ji známe z provozu.

Uvažované poruchy bloků výstražníku:

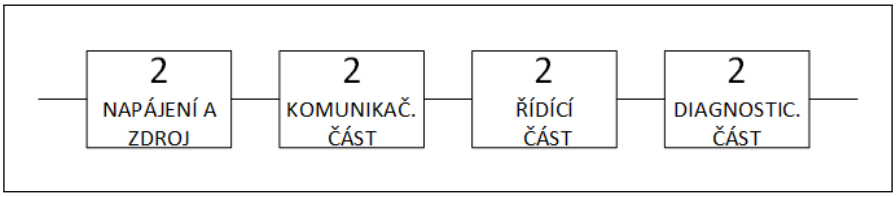
Porucha bílého světla – světlo přestane blikat, nicméně zabezpečovací zařízení může fungovat v omezeném režimu dál. Řidiči budou muset 50 m před přejezdem zpomalit na rychlost nejvýše 30 km/h a předpokládat, že zabezpečovací zařízení není v provozu.

Porucha červeného varovného světla – varovné světlo přestane blikat, z tohoto důvodu jsou na výstražníku 2 červená varovná světla, v případě poruchy obou červených varovných světel, přejde přejezdové zabezpečovací zařízení do bezpečného stavu, kdy žádným způsobem nereaguje na přijíždějící ani projíždějící vlak. Řidiči tak budou muset 50 m před přejezdem zpomalit na rychlost nejvýše 30 km/h a předpokládat, že zabezpečovací zařízení není v provozu.

Porucha zvukové signalizace – zabezpečovací zařízení přestane při obsazenosti kolejových úseků v oblasti přejezdu vydávat akustické varovné hlášení. Nicméně, zabezpečovací zařízení bude dále fungovat bez zvukové signalizace.

Porucha řízení signalizace – jedná se o systém 2 ze 2, a dokud bude v provozu, alespoň jedna část řízení signalizace, přejezdové zabezpečovací zařízení bude aktivní.

V případě výskytu jakékoliv výše uvedené poruchy, bude obeznámen nadřazený zabezpečovací systém.



Obrázek 6.4 Blokový model základních částí kabinetu.

Obrázek 6.4 ilustruje rozdělení kabinetu na funkční bloky: *napájení a zdroj*, *komunikační část*, *řídící část* a *diagnostická část*.

Blok *napájení a zdroj* obsahuje dva nezávislé zdroje, které zajišťují přívod elektrické energie do přejezdového zabezpečovacího zařízení. Jedná se o klasické napájení ze sítě a napájení z akumulátoru v případě výpadku dodávek energie v síti.

Blok *komunikační část* slouží ke komunikaci zařízení s nadřazeným zabezpečovacím zařízením. V případě ztráty spojení, je zabezpečovací přejezdové zařízení schopno pracovat omezenou dobu autonomně.

Blok *řídící část* je opět redundantním systémem, který zajišťuje funkční a bezproblémový chod celého zabezpečovacího zařízení.

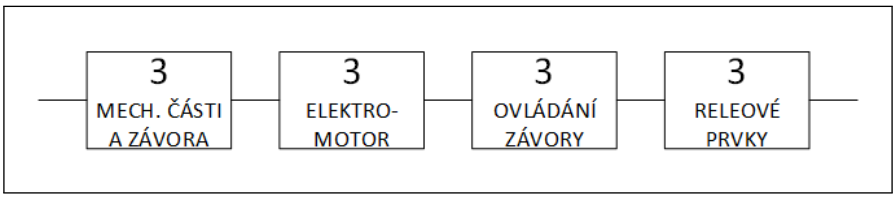
Blok *diagnostická část* slouží k vnitřnímu testování a sběru diagnostických dat, které se neustále vyhodnocují a posílají se přes komunikační část nadřazenému systému.

Uvažované poruchy boků kabinetu:

Porucha zdroje napájení – v případě poruchy síťového zdroje, bude zabezpečovací zařízení schopno v omezeném čase 36 hodin dále plnit funkci díly záložní baterii. V případě poruchy zdroje i baterie, přejde zabezpečovací zařízení do bezpečného stavu, kdy žádným způsobem nereaguje na přijíždějící ani projíždějící vlak. Tudíž řidiči tak budou muset 50 m před přejezdem zpomalit na rychlost nejvýše 30 km/h a předpokládat, že zabezpečovací zařízení není v provozu.

Porucha řídící části – jedná se opět o systém 2 ze 2, a dokud bude v provozu, alespoň jedna část řízení, přejezdové zabezpečovací zařízení bude aktivní.

Dojde-li k poruše komunikačního nebo diagnostického modulu nebude ovlivněn provoz zabezpečovacího zařízení. V případě výskytu jakékoliv výše uvedené poruchy, bude obeznámen nadřazený zabezpečovací systém.



Obrázek 6.5 Blokový model základních částí závory.

Obrázek 6.5 ilustruje rozdělení bloku závora na funkční bloky: *mechanické části a závora*, *elektromotor*, *ovládání závory*, *reléové prvky*.

Blok *závora a mechanické části* obsahuje vlastní závoru včetně různých táhel apod.

Blok *elektromotor* obsahuje motor, který pohybuje závorou, opět včetně všech dalších nutných částí.

Blok *ovládání závory* obsahuje dvě stejné redundantní části, které zajišťují ovládání mechanické závory.

Blok *reléové prvky* obsahuje opět dvě sady stejných relé, které umožňují ovládací elektronice ovládat motor a pohybovat tak závorou (tento typ prvku je použit v celém zařízení hned několikrát, ale z důvodu zamýšlené analýzy je uvažován pouze zde).

Uvažované poruchy bloků závory:

Porucha mechanických částí včetně závory – v případě poruchy závory (zlomení, či jiné deformace) se závora zvedne a železniční přejezd bude fungovat v omezeném režimu bez zabezpečení závorami.

Porucha elektromotoru – zabezpečovací zařízení zajistí zvednutí závory nahoru (protivahou) a železniční přejezd bude fungovat v omezeném režimu bez zabezpečení závorami. Pokud by z nějakého důvodu nebylo možné závoru zvednout, zhorší se dostupnost železničního přejezdu, přejezd se uvede do bezpečného stavu a nebude možné jej použít pro silniční dopravu, dokud závada nebude odstraněna.

Porucha reléových prvků – jedná se vždy o sadu prvků, tak se jednalo o systém 2 ze 2, čili v případě poruchy jedné sady relé, může závora dále fungovat, v případě poruchy obou sad relé bude funkce přejezdu omezena – nebude možno použít závory.

V případě výskytu jakékoliv výše uvedené poruchy, bude i v tomto případě obeznámen nadřazený zabezpečovací systém.

6.1.2 Vstupní hodnoty

Každý blok či podblok vlakového přejezdu má definovanou intenzitu poruch či střední dobu bezporuchového provozu. Jelikož zde není uvažovaná doba obnovy z poruchového stavu, není zde započítávaná ani doba opravy. Z toho důvodu je intenzita poruch odvozena z parametru střední doby do poruchy MTTF, který bude pro zjednodušení chápán jako střední doba bezporuchového provozu.

Na tomto místě je potřeba upozornit, že různé nástroje používají tento parametr různě, proto bude možné na některých obrázcích zejména z nástroje Zusim vidět parametr MTBF. Naopak nástroj SHARPE toto striktně rozlišuje, takže užívá ukazatel MTTF. Avšak v obou případech bude tento parametr chápán jako MTTF jak je uvedeno výše.

V tabulce 6.1 lze najít vstupní parametry pro bloky či pod bloky, pokud se jedná o nějakou redundanci uvedené na obrázcích 6.3 až 6.5.

Strom	Součást	MTTF [h]	Intenzita poruch [h^{-1}]
Výstražník	Červené světlo	75 000	0.00001333
Výstražník	Bílé světlo	75 000	0.00001333
Výstražník	Řízení signalizace	20 000	0.00005
Výstražník	Zvuková signalizace	50 000	0.00002
Kabinet	Zdroj	100 000	0.00001
Kabinet	Baterie	10 000	0.0001
Kabinet	Komunikace	20 000	0.00005
Kabinet	Řídící část	15 000	0.00006667
Kabinet	Diagnostika	35 000	0.00002857
Závora	Relé	30 000	0.00003333
Závora	Elektromotor	45 000	0.00002222
Závora	Mechanická závora	120 000	0.00000833
Závora	Ovládací závory	25 000	0.00004

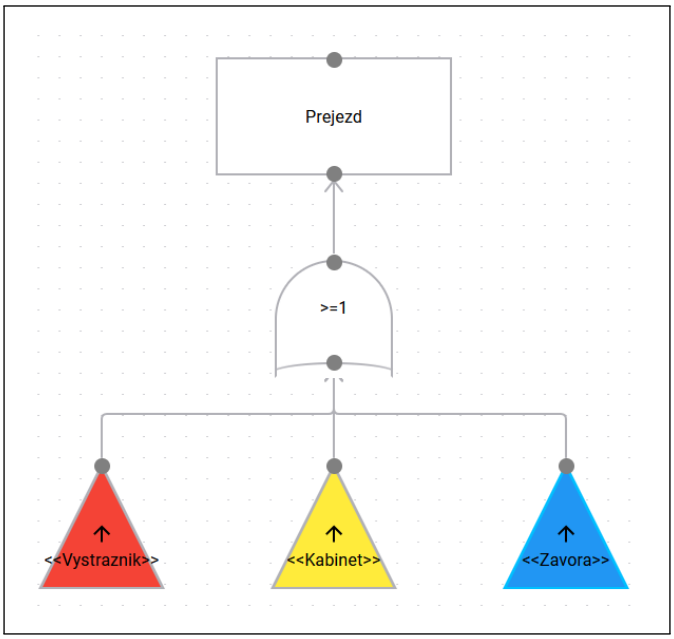
Tabulka 6.1 Tabulka hodnot spolehlivostních parametrů pro strom poruch železničního přejezdu.

6.1.3 Analýza metodou stromu poruch

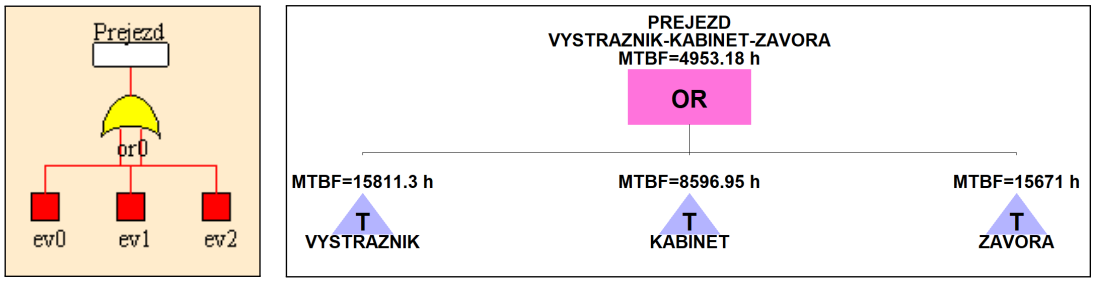
Výpočet střední doby bezporuchového provozu metodou stromů poruch. Nejedná se o obvyklou úlohu, na kterou by se tato metoda často používala ve smyslu [41]. Většinou se stromy poruch používají na výpočet pravděpodobnosti vzniku poruchy (vrcholové události). Nicméně, i tento spolehlivostní ukazatel stromy poruch umí vypočítat.

Nejprve je nutné vytvořit vrcholovou událost (v tomto případě lépe řečeno vrcholový stav), který se bude jmenovat *Přejezd*. Hradlo OR znamená, že selže-li kterýkoliv z podstromů dojde k selhání celého přejezdu. Přechody do podstromů jsou naznačeny barevně a korespondují s obrázkem 6.2.

Obrázek 6.6 ilustruje použití webové aplikace, obrázek 6.7 je společný pro výstup ze SHARPE (vlevo) a výstup z Zusimu (vpravo). SHARPE má všechny podstromy červené, bohužel jsou pojmenovány jako události generickým názvem *evX*, červený čtvereček znamená, že událost je přiřazená konkrétnímu podstromu. Zusim má naopak podstromy označeny konkrétním názvem a symbolem trojúhelníku s *T* jako *Transfer*.

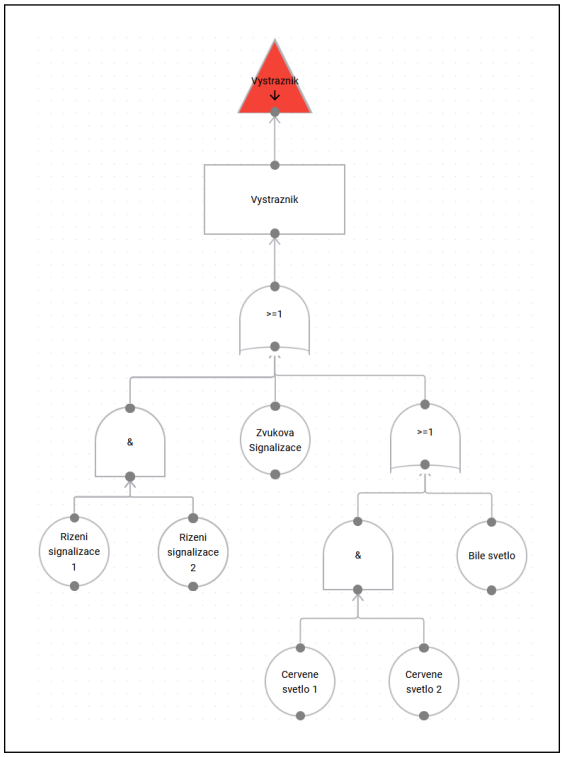


Obrázek 6.6 Hlavní strom poruch pro vlakový přejezd. Vrcholový stav zde představuje bezporuchovost celého systému. Jednotlivé podstromy jsou barevně rozlišeny pro lepší orientaci a jako prevence před špatným přiřazením.

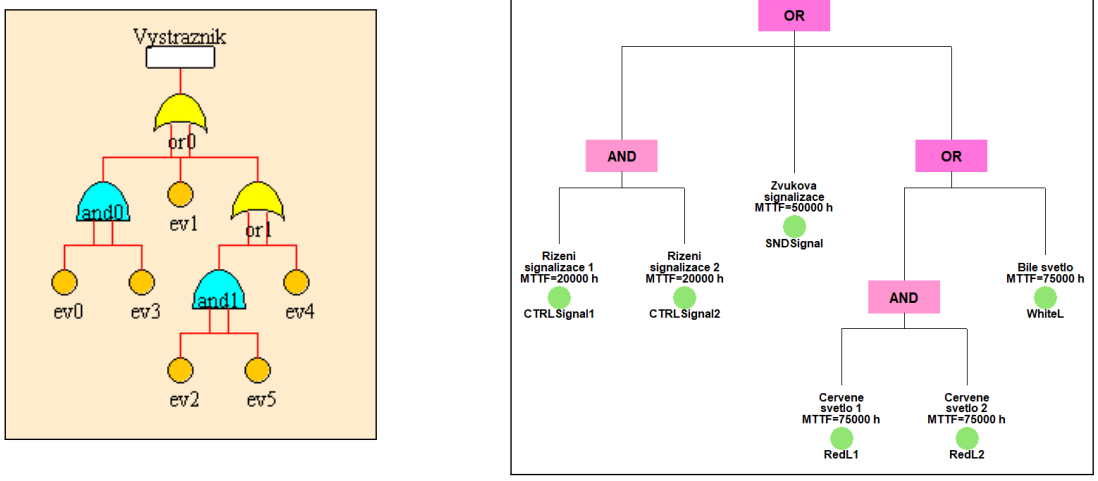


Obrázek 6.7 Hlavní strom poruch pro vlakový přejezd. Levá část je prostředí SHARPE a pravá část Zusim.

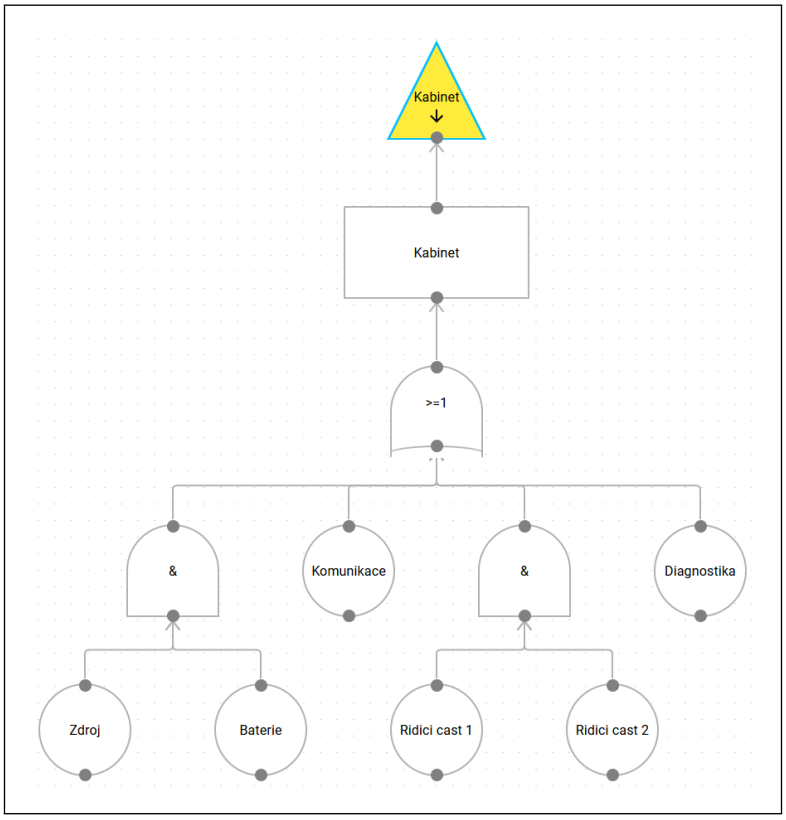
Dalšími kroky je vytvoření jednotlivých podstromů: výstražník, kabinet a závora. Nacházející se na obrázcích výstražník: 6.8, 6.9, kabinet: 6.10, 6.11, závora: 6.12, 6.13. Opět lze vidět výstupy všech nástrojů.



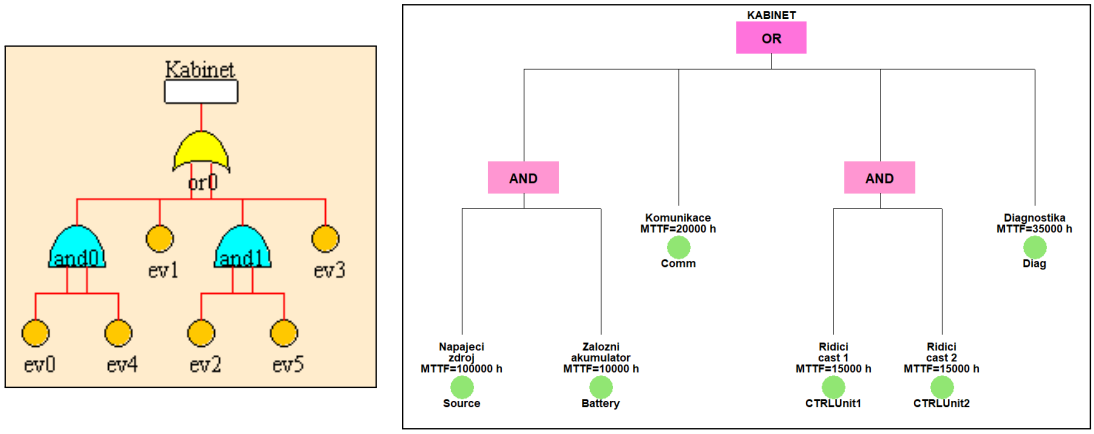
Obrázek 6.8 Podstrom výstražník, při poruše jakéhokoliv bloku z obrázku 6.3, dojde k selhání výstražníku, z toho důvodu je první hradlo OR. Vzhledem k tomu, že červené světlo a řízení signalizace jsou systémy 2 ze 2, byla pro tyto prvky použita hradla AND.



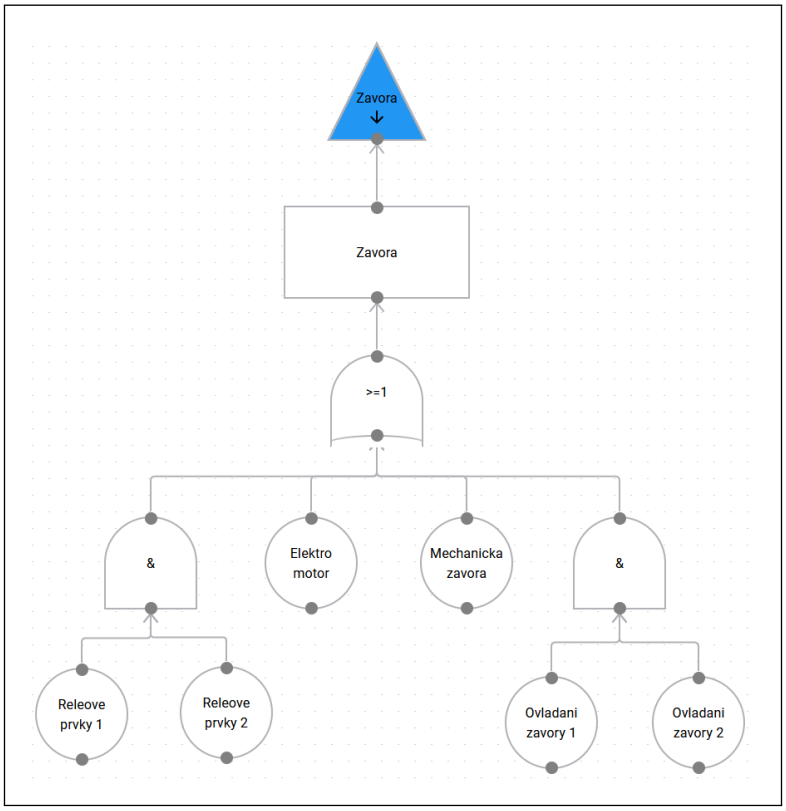
Obrázek 6.9 Podstrom popisující výstražník, výstup z aplikací SHARPE (vlevo) a Zusim (vpravo). SHARPE i když poskytuje barevný výstup nepůsobí moc přehledně, jelikož si uživatel musí pamatovat co jednotlivé události *evX* znamenají.



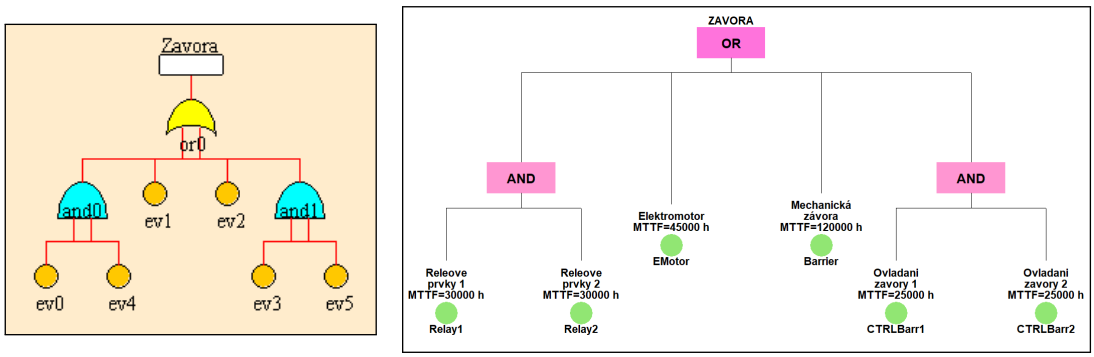
Obrázek 6.10 Podstrom kabinet, kdy při poruše jakéhokoliv bloku z obrázku 6.4, dojde k selhání kabinetu. Události komunikace a diagnostika jsou zde jako v předchozím případě navíc a je možné je ze stromu poruch odebrat, protože nezpůsobí selhání přejezdu.



Obrázek 6.11 Podstrom popisující kabinet, výstup z aplikací SHARPE (vlevo) a Zusim (vpravo).



Obrázek 6.12 Podstrom závora, při poruše jakéhokoliv bloku z obrázku 6.5, dojde k selhání závory, z toho důvodu je první hradlo opět OR. Vzhledem k tomu, že reléové prvky a ovládání závory jsou systémy 2 ze 2, byla pro tyto prvky použita hradla AND. Čili selhání jedné sekce ovládání či relé nepovede k vyvolání vrcholové události.



Obrázek 6.13 Podstrom popisující závoru, výstup z aplikací SHARPE (vlevo) a Zusim (vpravo).

6.1.3.1 Podstrom: výstražník

Obrázky 6.8 a 6.9 demonstrují strom poruch pro výstražník. Oproti předpokladům v předchozích sekcích této kapitoly je zde ten rozdíl, že porucha bílého světla vede na poruchu výstražníku. Nicméně, pokud by byly uvedené předpoklady splněny, znamenalo by to, že základní událost (stav) intenzita poruch bílého světla by se do tohoto stromu zahrnovat nemusela. Červená výstražná světla a řízení signalizace jsou vždy dva nezávislé systémy, jejichž porucha neznamená

selhání funkce přejezdu, lze v tomto případě použít hradla AND.

6.1.3.2 Podstrom: kabinet

Obrázky 6.10 a 6.11 demonstrují strom poruch pro kabinet. Zdroj elektrické energie je zálohovaný baterií, lze tedy opět tyto dva bloky projit hradlem AND. Řídící část celého zabezpečovacího zařízení je systém 2 ze 2 a proto je také tato část pod hradlem AND. Události komunikace a diagnostika, jsou ve stromu uvedeny navíc, protože jejich porucha nevede k selhání zabezpečovacího zařízení. V případě potřeby je zle odstranit.

6.1.3.3 Podstrom: závora

Obrázky 6.12 a 6.13 demonstrují podstrom poruch pro závoru. Analogicky jako v předchozích případech vše, co je redundantní a nezávislé se může zařadit pod hradlo AND. V tomto podstromu však není nic nadbytečné, není tedy co by bylo možné zanedbat.

Do všech uvedených podstromů byly nakonec započítány i nekritické části, protože původním záměrem výpočtu byla střední doba bezporuchového provozu.

Do všech uvedených podstromů byly nakonec započítány i nekritické části, protože původním záměrem výpočtu byla střední doba bezporuchového provozu, je tedy potřeba započít vše co svou poruchou může uvést systém do poruchového stavu (i když může pracovat v nouzovém režimu).

6.1.4 Výsledné vypočtené hodnoty MTTF

Jelikož výpočet probíhal ve třech různých programech, níže v tabulce jsou uvedeny výsledky.

Nástroj	MTTF [h]
Webová aplikace	4953.17
SHARPE	5055.91
Zusim	4953.18

Tabulka 6.2 Výsledné hodnoty střední doby bezporuchového provozu.

Výsledek webová aplikace se shoduje s vypočteným výsledkem programu Zusim, který beru jako etalon, protože je již lety prověřený a má v této oblasti udělanou validaci.

Důvod, proč je výsledek z programu SHARPE o cca 100 hodin rozdílný je způsoben zaokrouhlováním. Protože vstupní hodnoty do programu Zusim a webové aplikace byly ve formátu MTTF převzaté z tabulky 6.1. Vstupy do programu SHARPE byly ve formátu intenzity poruch zaokrouhlované na 8 desetinných míst, také z tabulky 6.1. V této tabulce je také mimo jiné vidět, že první 4 cifry za desetinnou čárkou jsou vždy 0, bohužel program SHARPE ne vždy podporuje práci se vstupními čísly v jiných formátech.

Závěrem k tomuto příkladu lze podotknout, že všechny tři aplikace umožňují vytvářet jednoduché stromy poruch s vnořenými podstromy a že webová aplikace obstála ve správnosti výpočtu.

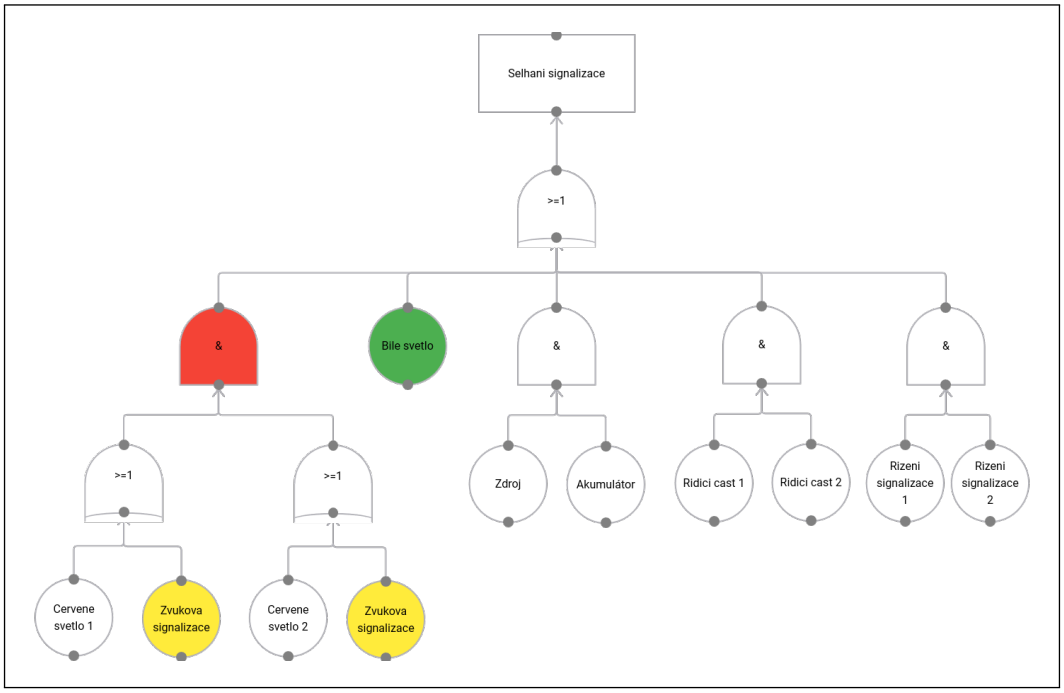
6.2 Ukázka spolehlivostní analýzy

Na stejné modelové situaci zabezpečovacího přejezdového zařízení lze ještě demonstrovat ukázku, jak postupovat při tvorbě kvalitativní analýzy. To znamená, že není nezbytně nutné ohodnocovat pravděpodobnostmi základní události a počítat pak celkovou pravděpodobnost míry selhání [38]. V tomto příkladu půjde pouze o tvorbu stromu. Také se již omezím pouze na použití webové aplikace, která je z mého pohledu pro tento účel nejvhodnější, protože výsledný strom je přehledný

a můžu si barevně označovat ty stavy/hradla/události, které mají speciální význam. Postup je následující:

Uživatel určí vrcholové události, pro které bude strom poruch postupně rozvíjet až po základní události. Zde se vyplatí již uvažovat a respektovat základní požadavky a typy poruch, které jsou definovány na začátku této kapitoly. Postup bude ukázán na selhání funkce signalizace přejezdu.

Vrcholová událost: *Selhání signalizace*



Obrázek 6.14 Strom poruch pro vrcholovou událost selhání signalizace.

K selhání signalizace vede jedno z těchto selhání:

Buď selžou světla na výstražníku, řízení signalizace nebo selže napájení nebo řídící část přejezdu. Při pohledu na strom poruch je však zřejmé, že všechny tyto části jsou redundantní a vzájemně nezávislé (což byl i původní předpoklad při návrhu zabezpečeného přejezdu). Pokud selže jakákoliv z výše uvedených částí, funkce signalizace by měla být dále zajištěna právě pomocí redundance těchto bloků.

Pokud selže bílé světlo, vlakový přejezd bude stále funkční, ale v omezeném režimu (proto je bílé světlo na obrázku 6.14 vyznačeno zeleně), jen budou muset řidiči silničních vozidel zpomalit rychlost na 30 km/h nejméně 50 m před přejezdem. Čili selhání bílého světla může omezit rychlost provozu, ale přejezd bude i tak stále zabezpečen. Otázkou je, zda se považuje selhání signalizace i selhání bílého světla. V případě, že ano, měl by výskyt poruchy bílého světla vést k vrcholové události (jak je naznačeno na obrázku 6.14), pokud ne, není důvod, aby bylo bílé světlo na obrázku uvedeno.

Pokud selže zvuková signalizace (ve stromu poruch vyznačena žlutě), přejezd bude dále funkční, ale bez akustického varování. Červená výstražná světla jsou nadřazena zvukové signalizaci. [39] [40] Zvuková signalizace stejně jako bílé světlo nemá žádnou redundanci, nicméně je ve stromu uvedena dvakrát kvůli nutné součinnosti s červenými výstražnými světly a také z toho důvodu, že její porucha neznamená selhání, ale pouze omezení.

Z obrázku 6.14 je tak patrné, jak je důležité využívat různá barevná označení a textový popis k dokreslení situace, pokud jde o kvalitativní analýzu, kdy si návrhář potřebuje ujasnit rozvoj kritických událostí vedoucích k vrcholové události.

6.3 Porovnání nástrojů

Porovnávané aplikace se liší mnoha věcmi, přístupem k problému, rozsahem, designem a dalším. V průběhu práce bylo několikrát uvedeno zaměření na cílovou skupinu a její případy užití. Jedná se především o malé školní či vědecké projekty. Tedy skupiny pár lidí nebo jednotlivců, kteří nyní nemají přístup ke kvalitním a podporovaným aplikacím. Porovnání bude brát toto v potaz, cílem je zjistit, zda může být webová aplikace dobrou alternativou k dosavadním nástrojům.

Jednou z prvních výhod webové aplikace, na kterou lze narazit ještě před spuštěním je dostupnost přímo v prohlížeči bez nutnosti instalace. Odpadá zde problém s kompatibilitou operačního systému a běžně používané prohlížeče s aplikací nemají problém. V tomto ohledu může být mínusem jen nutnost registrace a přihlášení do účtu. Naopak zbylé dvě aplikace je nutné instalovat a SHARPE mělo při práci na dvou příkladech velké problémy s kompatibilitou.

Na to navazuje uživatelské rozhraní a zkušenost s používáním aplikací. Webová aplikace se používá na první pohled velmi jednoduše. Vytvoření projektu a modelu je přímočaré a následná analýza cílí na jednoduchost. Využívá standardní webové prvky jako je drag-n-drop a standardní klávesové zkratky pro ovládání modelu. Objekty se propojují přímo pomocí handlerů, není potřeba vytvářet skupiny a propojovat ty, jako v případě Zusimu. SHARPE naopak používá pracuje s nabídkou, takže je pro vytvoření událostí a jejich propojení potřeba zbytečné množství akcí.

Rozsah aplikace může být jedním z nevýhod, protože například oproti SHARPE obsahuje mnohem méně spolehlivostních modelů a naopak Zusim dokáže pracovat i s dalšími událostmi a hranami, které webová aplikace zatím nemá. Obě tyto aplikace dokáží pracovat se samotnými pravděpodobnostmi poruchy na straně vstupní i výstupní hodnoty. Tento postup se používá v některých analýzách stromy poruch. Pro zisk této informace by musela v aplikaci proběhnout změna jak na straně zadávání vstupu, tak i na straně šablony WM notebooku. Menší rozsah však jistě může sloužit jako podklad pro další vývoj webové aplikace.

Kapitola 7
Závěr

Diplomová práce vychází z předešlé bakalářské práce, jejíž výsledkem byla základní webová aplikace pro práci se spolehlivostními modely, která obsahovala RBD a Markovské modely. Cílem této práce bylo rozšířit aplikaci o nové funkce, z nichž některé jsou stanoveny již v zadání a jiné vyšly najevo v průběhu analýzy a návrhu. Aplikace vznikla z důvodu absence dostupných nástrojů pro spolehlivostní analýzu. Proto bylo dalším hlavním cílem připravení aplikace pro použití v reálném produkčním prostředí.

V analytické části jsem vyzkoušel dostupné nástroje a sepsal jak pozitivní, tak i negativní zkušenosti. Ty jsem zakládal na modelování několika zkušebních modelů a později na finálním modelu železničního přejezdu. To stejné jsem udělal i s první verzí aplikace. Na základě práce s těmito modely a připomínek z provozu během cvičení předmětu *Testování a Spolehlivost* jsem definoval požadavky cílové aplikace, včetně několika případů užití. V důsledku toho jsem mohl analyzovat technologie použité s v první verzi aplikace, porovnat je s alternativami a rozhodnout, pro ponechání nebo změnu technologie.

V návrhu jsem se jednotlivě věnoval novým funkcím a jejich integraci do aplikace. Největší změnou bylo rozšíření sady spolehlivostních modelů o stromy poruch. Podrobně jsem popsal základní smysl a funkce modelu společně se základními bloky (události a hradla). Následně jsem postupně okomentoval složitější bloky a jejich vlastnosti. Ty mají větší důsledky pro implementaci, jako například propojení více modelů. Se stromy poruch souvisí kvalitativní analýza, která je odlišná od všech funkcí ostatních modelů v aplikaci. Při návrhu zavedení tohoto typu analýzy do aplikace jsem hledal algoritmy, které k tomu lze využít a následně jsem zvolil konkrétní algoritmus, který lze v aplikaci dobře implementovat.

Ve stejné kapitole jsem kromě stromu poruch rozebíral i ostatní přidané nebo změněné funkce. Jednou z nich je zavedení vstupních parametrů v podobě matematického výrazu zapsaného jako klasický text s možností zadávat předem specifikované funkce a operátory. Tato změna přinesla uživatelům velké zjednodušení, jelikož mohou zadávat hodnoty v libovolném formátu a s využitím sdílených proměnných. Další velkou funkci je lokální historie změn, uživatelé mohou jednoduše vrátit provedené kroky pomocí klávesové zkratky. V podkapitole věnující se právě historii jsem popsal původní chybný návrh s následným řešením. Studentům FIT ČVUT v Praze, pro které je původně aplikace určena, zjednoduší registraci a přihlášení do aplikace autentizace pomocí fakultního účtu. Nemusí tak vytvářet nový účet se všemi přihlašovací údaji, což zrychlí a zjednoduší přístup k jejich projektům.

Kapitola testování a nasazení je věnována základním testům na serverové části aplikace. Jejich vysvětlení a použití v aplikaci. Součástí nasazení je důležitá především migrace dat z předchozí verze na novou. Celá migrace je provedena tak, aby uživatelům zůstaly jejich účty a veškerá data, včetně modelů a dokumentací.

V poslední kapitole je představen zjednodušený praktický příklad železničního přejezdu,

který dodal vedoucí práce. Na tomto příkladu jsem vypracoval spolehlivostní analýzu ve všech zmíněných aplikacích, SHARPE, Zusim a nakonec výsledná verze webové aplikace. Po dokončení analýzy jsem aplikace porovnal, popsal jejich výhody a nevýhody. Toto porovnání navíc slouží jako verifikace správnosti výpočtů na tomto konkrétním příkladu.

Všechny části zadání byly popsány v jednotlivých kapitolách a ty, které se dotkly implementace byly integrovány do nové verze aplikace. Nová verze aplikace byla nasazena na fakultní server na adrese *vratidan.fit.cvut.cz*. Vzhledem k šířce zpracovávaného tématu je zde prostor pro mnoho dalších rozšíření, které se mohou týkat nových spolehlivostních modelů, nových způsobů, jak zobrazovat výsledek analýzy (např. zobrazení spolehlivostních parametr včetně grafů přímo v aplikaci) a další funkcí. Stromy poruch v některých případech mohou počítat pouze s pravděpodobností poruchy, to v tuto chvíli aplikace také neobsahuje. Tyto nové funkce mohou znovu posunout rychlost a jednoduchost spolehlivostní analýzy na další úroveň.

Příloha A

Uživatelská příručka

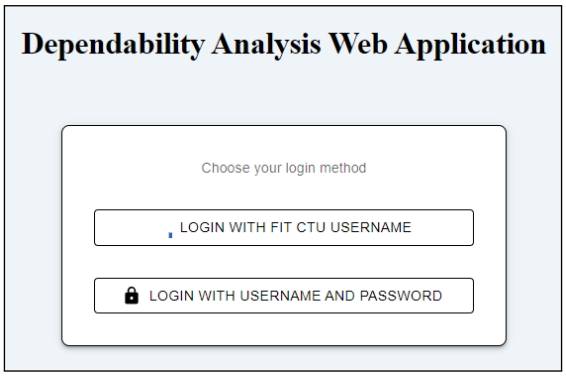
Příručka slouží uživatelům, kteří s aplikací ještě nemají zkušenosti a chtějí vytvořit první projekt nebo model. Všechny základní funkce jsou podrobně popsány a zobrazeny v obrázcích z aplikace. Součástí této práce bylo nasazení aplikace, která je dostupná na adrese vratidan.stud.fit.cvut.cz, ta je součástí odkazů použitých v příručce. Pokud je aplikace přístupna z jiné domény, stačí tuto doménu nahradit.

A.1 Přihlášení a registrace

Na domovské stránce *vratidan.stud.fit.cvut.cz* nabízí aplikace dva způsoby přihlášení. Jedním z nich je přihlášení pomocí účtu FIT ČVUT v Praze (první možnost na obrázku A.1). Pokud takový účet máte je tento způsob doporučen. V opačném případě lze využít přihlášení pomocí uživatelského jména a hesla (druhá nabízená možnost na obrázku A.1).

FIT ČVUT - Po výběru účtu FIT ČVUT v Praze aplikace přesměruje na autorizační stránku fakulty. Do formuláře zadáte přihlašovací údaje a potvrdíte. Následně jste přesměrování zpět do aplikace na úvodní stránku se seznamem projektů.

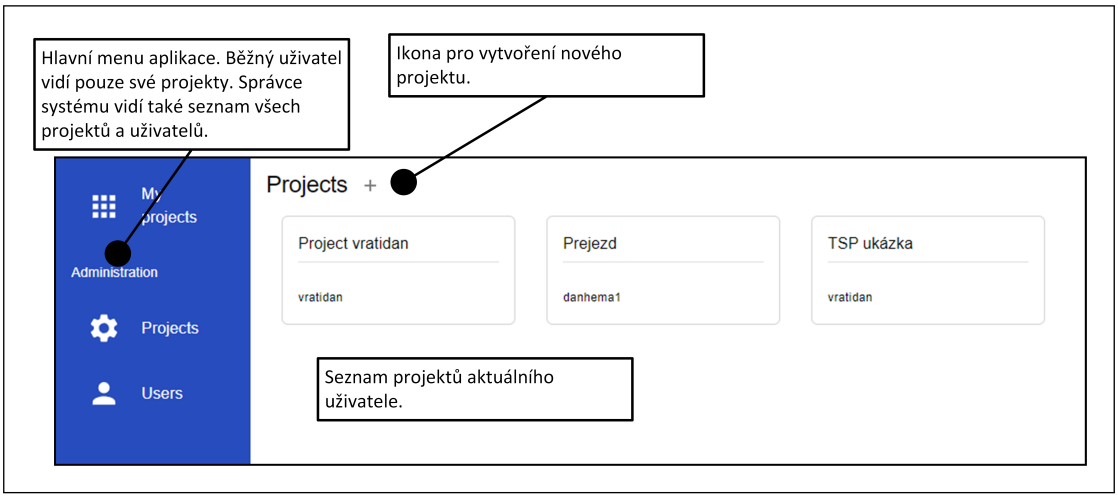
Uživatelské jméno - Po volbě uživatelského jména, aplikace zobrazí přihlašovací formulář. Pokud jste již zaregistrováni, zadejte údaje a pokračujte do aplikace. V opačném případě pokračujte k registraci. V dalším kroku si aplikace vyžádá nové uživatelské jméno a heslo. Po zadání těchto údajů aplikace přesměruje zpět na přihlašovací formulář, kam stačí zadat údaje a pokračovat do aplikace.



Obrázek A.1 Rozcestník s dvěma možnostmi přihlášení.

A.2 Správa projektů

Pokud je uživatel v aplikaci poprvé, pravděpodobně nemá přístup k žádnému projektu. Ten může získat dvěma způsoby. Může být přiřazen k projektu jiného uživatele nebo může vytvořit vlastní projekt. Nový projekt lze přidat pomocí tlačítka +, otevře se dialog, v něm se vyplní název projektu a vytvoří se projekt. V takto vytvořeném projektu je uživatel automaticky označen jako vlastník projektu a má tak plná práva k jeho správě.



Obrázek A.2 Stránka se seznamem projektů aktuálního uživatele. Aplikace zde zobrazuje menu, které má v případě, že se jedná o správce dvě skupiny. Uživatel zde může vybrat a přejít do projektu nebo vytvořit nový.

V detailu projektu je v hlavní části stránky zobrazen seznam modelů a dokumentací. Vedle názvu a autora projektu je ikona pro nastavení, která vede do dialogu, ve kterém může být projekt přejmenován nebo smazán. Druhá ikona zobrazí dialog se správou uživatelů, které jsou k projektu přiřazeni. Ten obsahuje formulář pro přidání nového uživatele pomocí jeho uživatelského jména a výběr rolí, která uživatel dostane. Níže je seznam přiřazených uživatelů a jejich role. Každého uživatele lze odebrat nebo mu roli změnit. Autor projektu má vždy plná práva, proto tu ani není uveden.

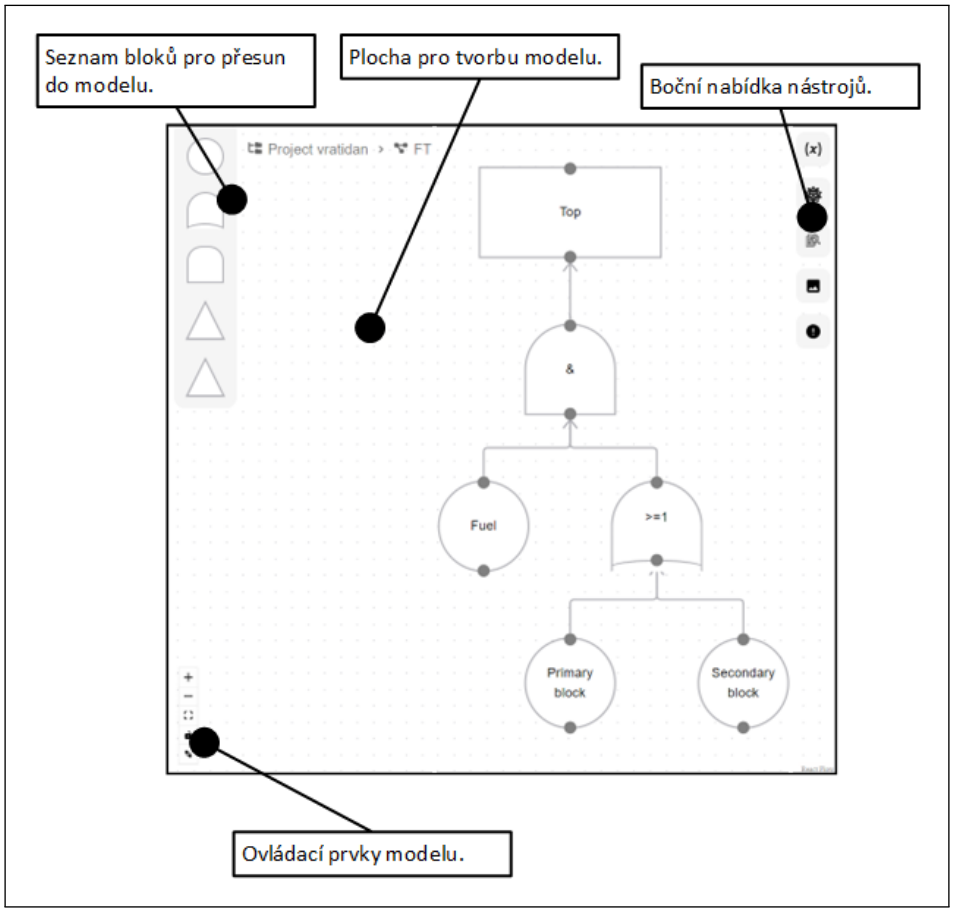
Pomocí ikony + nad seznamem modelů lze vytvořit nový model. Zadává se jeho název, typ modelu a popis. To stejné platí pro dokumentaci, s tím rozdílem, že se zadává pouze název a verze.

A.3 Detail modelu

Po vytvoření a otevření modelu se uživatel dostane na stránku, ve které probíhá většina spolehlivostní analýzy. Ta je zobrazena na obrázku A.3. Hlavní část okna je věnována samotnému modelu. Obsahuje bloky a hrany, se kterými může interagovat. Pro přidání nového bloku slouží tabulka bloků vlevo nahoře, které stačí přetáhnout do modelu a tím blok přidat. Většina bloků má spojovací body. Přetažením z jednoho do druhého se vytvoří mezi bloky hrana.

K většině bloků můžeme zadat doplňující informace a vlastnosti, jako je název, barva nebo vstupní hodnota (např. intenzita poruch). Tato nabídka se otevře po dvojkliku na daný blok. Zadávání vstupní hodnoty je možné ve třech jednotkách. Samotný matematický výraz pak může obsahovat kromě číslic a standardních binárních operací (+, -, *, /) také funkce (exp, abs, avg, ...) nebo proměnné. Proměnnou lze zapsat pomocí jejího názvu a měla by být definována v seznamu

proměnných v konkrétním modelu. Pokud není proměnná definována, aplikace vypíše varování. Ve výsledném WM notebooku bude tato proměnná obecná bez konkrétní hodnoty.



Obrázek A.3 Stránka s detailem projektu pro zpracování spolehlivostní analýzy. Na obrázku je uveden příklad stromu poruch, ostatní modely se liší pouze v nabídce bloků a bočné nabídce nástrojů.

Kromě bloků je v horní části obrazovky název projektu a model, druhý název je možné po kliku upravit. Na pravé straně se zobrazují aktuálně připojení uživatelé. Vedle nich je potom další nabídka několika ikon. První slouží k definici dříve zmíněných proměnných. Každá z nich má svůj název a hodnotu v podobě matematického výrazu. Proměnné na sebe mohou odkazovat. Níže je export do obrázkových formátů (PNG a SVG) s možností odebrání bílého pozadí nebo nastavení velikosti okraje. Další ikona představuje nabídku pro vygenerování WM notebooku. Zde je hlavní nastavení jednotek, ve kterých chce uživatel výsledek zobrazit. Poslední ikonou je seznam obecných chyb, která se v aktuálním modelu vyskytují.

Aplikace dokáže po provedení změn v modelu také tyto změny vrátit nebo znovu provést. Tato funkce se ovládá pomocí klávesových zkratek *CTRL+Z* a *CTRL+Y*. Pokud chceme některý blok nebo hranu smazat, můžeme využít klávesu *DELETE*.

Nakonec přejděme k nabídce vlevo dole, kde jsou pomocné funkce pro ovládání zobrazení modelu, jako je přiblížení, oddálení, vycentrování model nebo uspořádání všech objektů modelu.

A.4 Generování výsledku

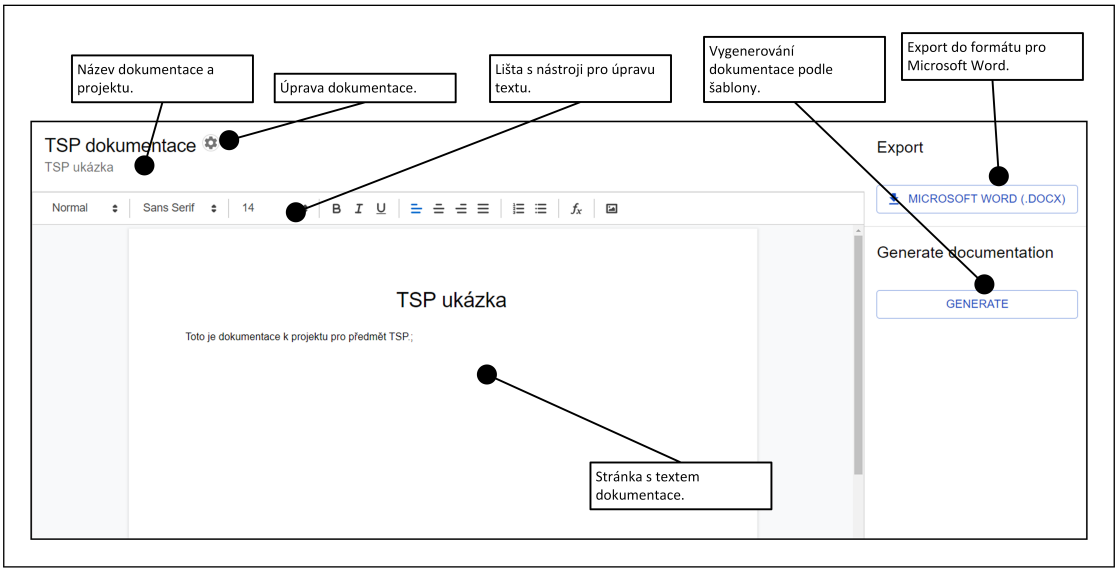
Pokud se uživatel nachází v situaci, kdy má vytvořený korektní stromu poruch a zadány vstupní spolehlivostní parametry. Dalším krokem je typicky vygenerování výpočtu v podobě WM notebooku. Uživatel nejprve otevře panel nástrojů pro generování (druhý odshora na obrázku A.3). Následně zvolí jednotky, do kterých budou všechny hodnoty převedeny a poté spustí generování. Po chvíli generování se spustí stažení souboru, uživatel si vybere místo pro uložení a dojde ke stažení souboru. Tento vygenerovaný soubor je ve formátu WSL (Wolfram Script Language). Jedná se o skript, po jehož spuštění je výstupem finální WM notebook. Skript se spouští v nástroji pro příkazovou řádku Wolfram Script, který se běžně nainstaluje společně s WM. Příklad tohoto příkazu je v následujícím bloku, kde `fault_tree_input_2024-05-08-09-10-27.wsl` je soubor vygenerovaný webovou aplikací.

```
$ wolframscript fault_tree_input_2024-05-08-09-10-27.wsl
```

A.5 Detail dokumentace

Podobně jako u modelu se i detail dokumentace uživatel dostane z detailu projektu. Pro její vytvoření potřebuje zadat pouze název a volitelně i verzi. Po otevření detailu se zobrazí obrazovka, která je na obrázku A.4 a její hlavní část tvoří dokument, do kterého může zapisovat text.

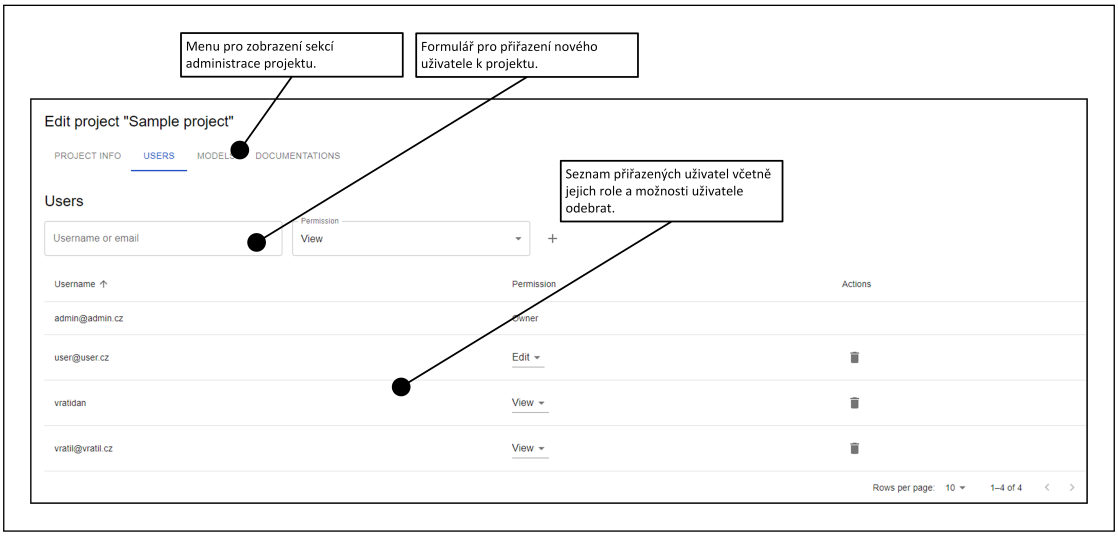
Tento dokument je rozšířen o panel nástrojů, který obsahuje standardní nástroje pro úpravu textu. Mezi ně patří například nastavení velikosti textu, fontu, zarovnání, vložení obrázku, atd. Pro tvorbu dokumentace je možné využít i automatické generování na základě vytvořeného projektu. Po dokončení nebo zálohu dokumentace obsahuje aplikace funkci pro export do formátu pro Microsoft Word. Celá dokumentace je podobně jako model živě sdílena mezi více připojenými uživateli.



Obrázek A.4 Stránka s detailem dokumentace. Na této stránce si uživatel může dokumentaci přečíst nebo ji sepsat. K tomu může využít lištu s nástroji pro úpravu textu. V pravém sloupci může uživatel dokumentaci vygenerovat podle připravené šablony nebo exportovat do formátu pro Microsoft Word.

A.6 Administrace

Při používání aplikace může nastat situace, ve které je potřeba spravovat projekty centrálně bez přihlášení za konkrétního uživatele. Například v případě, kdy je potřeba uživatele nebo projekty smazat. Pro takový případ existuje uživatelská role pro správce, tu může mít standardní uživatel, není potřeba pro přístup do správy vytvářet zvláštní účet. Aplikace takovým uživatelům zobrazí v menu sekci *Administration*, která obsahuje položky *Projects* a *Users*. Obě zobrazují seznam všech projektů, resp. uživatelů v aplikaci ve formě tabulky. Správce tak jednoduše vidí všechny projekty a zároveň je může upravit, smazat nebo vytvořit nové. Úprava projektu zahrnuje nejen jeho název, ale také seznam přiřazených uživatelů, který lze také upravovat. Nakonec seznam všech modelů a dokumentací, do kterých má zároveň plný přístup. Na obrázku A.5 je zobrazena administrace uživatelů přiřazených k projektu.



Obrázek A.5 Stránka s detailem projektu s názvem *Sample project*. Konkrétně se jedná o seznam přiřazených uživatelů k projektu. V tabulce můžeme vidět přihlašovací jméno uživatele, jejich oprávnění vztažená k tomuto projektu a možnost odebrání uživatele z projektu. Nad tabulkou je zobrazen malý formulář, který slouží k přiřazení nového uživatele.

Bibliografie

1. HLAVIČKA, Jan. *Číslicové systémy odolné proti poruchám*. 1. vydání. Praha: ČVUT, 1992. ISBN 80-010-0852-5.
2. STOREY, Neil. *Safety-critical computer systems*. Harlow: Addison-Wesley Longman, 1996. ISBN 02-014-2787-7.
3. STAPELBERG, Rudolph Frederick. *Handbook of reliability, availability, maintainability and safety in engineering design*. 1st edition. British Library Cataloguing in Publication Data, 2009. ISBN 978-1-84800-174-9.
4. DAŇHEL, Martin; KUBÁTOVÁ, Hana; DOBIÁŠ, Radek. Predictive Analysis of Mission Critical Systems Dependability. *16th Euromicro Conference on Digital System Design*. 2013, s. 561–566.
5. ANDERS, Enrico; THEEG, Gregor; VLASENKO, S. V. *Railway Signalling & Interlocking: international compendium*. 2nd edition. Hamburg: Eurailpress, 2018. ISBN 37-771-0394-2.
6. RUIJTERS, Enno; STOELINGA, Mariëlle. Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools. *Computer Science Review*. 2015, roč. 15-16, s. 29–62. ISSN 1574-0137. Dostupné z DOI: `https://doi.org/10.1016/j.cosrev.2015.03.001`.
7. LEE, W. S.; GROSH, D. L.; TILLMAN, F. A.; LIE, C. H. Fault Tree Analysis, Methods, and Applications A Review. *IEEE Transactions on Reliability*. 1985, roč. R-34, č. 3, s. 194–203. Dostupné z DOI: `10.1109/TR.1985.5222114`.
8. VRÁTIL, Daniel. *Webová aplikace pro spolehlivostní modely*. Praha, 2022.
9. *PHP: What is PHP?* [Online]. 2022. [cit. 2022-03-17]. Dostupné z: `https://www.php.net/manual/en/intro-whatis.php`.
10. TOAL, Rory. *What is PHP? Uses & Introduction - Code Institute Global*. 2024. Dostupné také z: `https://codeinstitute.net/global/blog/what-is-php-programming/`.
11. *Usage Statistics and Market Share of Server-side Programming Languages for Websites, March 2022* [online]. 2022. [cit. 2022-03-18]. Dostupné z: `https://w3techs.com/technologies/overview/programming_language`.
12. *What is Symfony*. 2024. Dostupné také z: `https://symfony.com/what-is-symfony`.
13. *Symfony explained to a Developer*. 2024. Dostupné také z: `https://symfony.com/explained-to-a-developer`.
14. *What Is Symfony? (Definition, How It Works, Benefits)*. 2022. Dostupné také z: `https://builtin.com/software-engineering-perspectives/symfony`.

15. *What is PHP Framework Symfony? Explained for executives — Accesto Blog* [online]. 2021. [cit. 2022-04-03]. Dostupné z: https://accesto.com/blog/what-is-php-framework-symfony-explained-for-executives/.
16. *PostgreSQL: About* [online]. 2022. [cit. 2022-04-03]. Dostupné z: https://www.postgresql.org/about/.
17. *Mercure.rocks: Real-time APIs Made Easy* [online]. [B.r.]. [cit. 2022-03-30]. Dostupné z: https://mercure.rocks/.
18. *The Mercure.rocks Hub is now based on Caddy Web Server*. 2020. Dostupné také z: https://medium.com/@dunglas/the-mercure-rocks-hub-is-now-based-on-caddy-web-server-5f400ac0a295.
19. *JSON Web Tokens - jwt.io* [online]. 2022. [cit. 2022-03-18]. Dostupné z: https://jwt.io/.
20. *TypeScript: JavaScript With Syntax For Types.* 2024. Dostupné také z: https://www.typescriptlang.org/.
21. *JavaScript Development Statistics and Facts*. 2023. Dostupné také z: https://kruschecompany.com/javascript-development-statistics-and-facts/.
22. *React – A JavaScript library for building user interfaces* [online]. 2022. [cit. 2022-03-18]. Dostupné z: https://legacy.reactjs.org/.
23. *React*. 2024. Dostupné také z: https://react.dev/.
24. *What is React.js? Uses, Examples, & More*. 2023. Dostupné také z: https://blog.hubspot.com/website/react-js.
25. *Redux - A JS library for predictable and maintainable global state management*. 2024. Dostupné také z: https://redux.js.org/.
26. *Getting Started with Redux*. 2024. Dostupné také z: https://redux.js.org/introduction/getting-started.
27. *All About Zustand!* 2023. Dostupné také z: https://blog.stackademic.com/all-about-zustand-807c2fd80e13.
28. *Zustand Documentation*. 2024. Dostupné také z: https://docs.pmnd.rs/zustand/getting-started/introduction.
29. *OAuth 2.0*. 2024. Dostupné také z: https://oauth.net/2/.
30. *What is OAuth 2.0 and what does it do for you?* 2024. Dostupné také z: https://auth0.com/intro-to-iam/what-is-oauth-2.
31. *An Introduction to OAuth 2*. 2021. Dostupné také z: https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2.
32. REMENYTE-PRESCOTT, R.; ANDREWS, J.D. An enhanced component connection method for conversion of fault trees to binary decision diagrams. *Reliability Engineering System Safety*. 2008, roč. 93, č. 10, s. 1543–1550. ISSN 0951-8320. Dostupné z DOI: https://doi.org/10.1016/j.ress.2007.09.001.
33. *What is Unit Testing?* [B.r.]. Dostupné také z: https://aws.amazon.com/what-is/unit-testing/.
34. *PHPUnit*. 2024. Dostupné také z: https://phpunit.de.
35. *CloudFIT - FIT ČVUT*. 2023. Dostupné také z: https://help.fit.cvut.cz/cloud-fit/index.html.
36. *WMware - FIT ČVUT*. 2023. Dostupné také z: https://help.fit.cvut.cz/cloud-fit/vmware/index.html.
37. *VMware vSphere — Virtualization Platform*. 2024. Dostupné také z: https://www.vmware.com/products/vsphere.html.

38. RÁSTOČNÝ, Karol; BALÁK, Josef. *Kvantitatívne hodnotenie integrity bezpečnosti elektronických systémov súvisiacich s bezpečnosťou*. 1. vydání. Žilina: Žilinská univerzita v Žiline, EDIS - vydavateľské centrum ŽU, 2020. ISBN 978-80-554-1646-5.

39. RÁSTOČNÝ, Karol. *Prvky zabezpečovacích systémov*. 1. vydání. V Žiline: EDIS - vydavateľstvo ŽU, 2012. ISBN 978-80-554-0593-3.

40. ZAHRADNÍK, Jiří; RÁSTOČNÝ, Karol. *Aplikácie zabezpečovacích systémov*. 1. vydání. Žilina: Žilinská univerzita, 2006. ISBN 80-807-0546-1.

41. IEC, 61025. *Fault tree analysis (FTA)*. 2. vyd. IEC, 2006.

Obsah příloh